

Software Engineering

Objektorientierte Analyse und Entwurf
Gero Wedemann

Software Engineering

Herausgeber:

Prof. Dr. Axel Buhl, Prof. Dr. Gero Wedemann

Hochschule Stralsund

Objektorientierte Analyse und Entwurf

Autor:

Prof. Dr. Gero Wedemann

Hochschule Stralsund, Fakultät für Elektrotechnik und Informatik

© 2009 Land Rheinland-Pfalz
Ministerium für Wissenschaft, Weiterbildung und Kultur
1. Auflage

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung des Ministeriums für Wissenschaft, Weiterbildung und Kultur des Landes Rheinland-Pfalz reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Text, Abbildungen und Programme wurden mit größter Sorgfalt erarbeitet. Das Fernstudium Informatik und die Autorinnen und Autoren können jedoch für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische noch irgendeine Haftung übernehmen.

Die in dieser Kurseinheit erwähnten Soft- und Hardwarebezeichnungen sind in den meisten Fällen auch eingetragene Warenzeichen und unterliegen als solche den gesetzlichen Bestimmungen.

Herausgeber: Fernstudium Informatik

Anschrift: Hochschule Trier
Fachbereich Informatik • Fernstudium Informatik
Postfach 18 26 • 54208 Trier
Telefon: (06 51) 81 03-57 6

Vertrieb: Zentralstelle für Fernstudien an Fachhochschulen – ZFH –

Anschrift: Konrad-Zuse-Str. 1 • 56075 Koblenz
Telefon: (02 61) 9 15 38-0

Titelbild: eberle & wollenweber • Koblenz

D8.17

Einleitung

In der vorangegangenen Kurseinheit „Grundlagen der Softwaretechnik und Requirements Engineering“ haben Sie die Inhalte und Tätigkeiten der ersten Phase des Wasserfallmodells kennen gelernt, in der die Anforderungen erarbeitet werden. Die vorliegende Kurseinheit behandelt die sich daran anschließenden Phasen Analyse und Entwurf. In diesen Phasen wird zunehmend auf die endgültige Lösung hingearbeitet, bis schließlich der Entwurf, eine Art Konstruktionszeichnung für die Software, entstanden ist. Für diese Tätigkeiten wurden in der Vergangenheit verschiedene Techniken entwickelt und erprobt. In der Praxis werden heute zumeist die Methoden der objektorientierten Analyse und des objektorientierten Entwurfs eingesetzt, die in dieser Kurseinheit ausführlich behandelt werden.

In der Analyse werden objektorientierte Konzepte, die Sie bereits aus dem Modul „Einführung in die Programmierung“ kennen, benutzt, um die Anforderungen systematisch zu untersuchen und eine Lösung des Problems aus rein fachlicher Sicht zu entwickeln. Dazu werden die fachlichen Zusammenhänge mit Hilfe von Klassen und ihren Beziehungen modelliert. Mögliche Abläufe wie z.B. Geschäftsprozesse können graphisch modelliert werden. Sie werden geeignete Hilfestellungen und Vorgehensweisen an die Hand bekommen, um anschließend selbst Analysen durchführen zu können. Ebenfalls zur Analyse gehört der Entwurf der Benutzeroberfläche. In vielen Fällen ist es wichtig, dass die Oberfläche hohe Ansprüche an die Benutzerfreundlichkeit erfüllt. Sie werden einige Grundregeln kennen lernen, um diesen Aspekt bei der Dialoggestaltung besser zu berücksichtigen.

Beim Entwurf werden aufbauend auf den Anforderungen und den Ergebnissen der Analyse die Strukturen und Abläufe der Software entworfen. Dazu wird zunächst ein Rahmen definiert, die so genannte Architektur. Dabei wird z.B. festgelegt, ob Informationen in Dateien oder in einer Datenbank abgelegt werden. Innerhalb dieses Rahmens werden dann Klassen, ihre Beziehungen zueinander und Abläufe entworfen. Sie werden allgemeine Prinzipien erlernen, die Ihnen helfen, im Entwurf Qualitätsziele wie Wartbarkeit zu erreichen. Dazu gehört im Wesentlichen die Aufteilung, die Unabhängigkeit der Teile voneinander und das Prinzip der Abstraktion.

Bei der Erarbeitung von Analyse, Architektur und Entwurf findet man bei verschiedenartigen Aufgabenstellungen häufig sehr ähnliche Lösungen. Lösungsstrategien für verallgemeinerte Probleme werden in Form von so genannten Mustern beschrieben. Sie helfen, effektiver bei Analyse und Entwurf vorzugehen. Aufgrund des hohen Bekanntheitsgrades von vielen Mustern erleichtern sie zudem die Kommunikation mit Mitarbeitern und

Kollegen. Sie werden in dieser Kurseinheit typische Muster kennen lernen und ihre Anwendung erproben können.

Lernziele

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- Anforderungen mit objektorientierten Methoden analysieren können,
- systematisch eine ergonomische Benutzeroberfläche entwerfen können,
- anhand der Anforderungen für einfache Systeme eine Architektur entwerfen können,
- Software anhand der Analyse objektorientiert entwerfen können, so dass sie den Anforderungen genügt,
- bei diesen Tätigkeiten Muster einsetzen können.

Inhalt

1 Objektorientierte Analyse	1
1.1 Modelle in Analyse und Entwurf.....	1
1.2 Klassen und Objekte.....	3
1.3 Beziehungen von Klassen.....	6
1.4 Domänenmodell	9
1.4.1 Beispiel Bibliothek.....	9
1.4.2 Erstellung eines Domänenmodells	12
1.5 Modellierung der Dynamik	18
1.6 Analysemuster	21
1.6.1 Exemplartyp (Abstraction – Occurence)	22
1.6.2 Wechselnde Rollen (Player – Role).....	24
1.6.3 Allgemeine Hierarchie (General Hierarchy, Kompositum)	26
1.7 Dialogentwurf.....	29
1.7.1 Vorgehen	29
1.7.2 Ergonomie	32
1.8 Zusammenfassung	36
2 Architektur	37
2.1 Softwarearchitektur.....	37
2.2 Architekturmuster.....	40
2.2.1 Client-Server	40
2.2.2 Mehrschichtenmodell	41
2.3 Beispiel Bibliothek.....	43
2.4 Zusammenfassung	46
3 Objektorientierter Entwurf	47
3.1 Von der Analyse zum Entwurf.....	47
3.2 Klassen im Entwurf.....	48
3.3 Strukturen und Verhalten	49

3.4 Vorgehen beim Entwerfen	51
3.5 Entwurfsmuster	52
3.5.1 Beobachter (Observer)	52
3.5.2 MVC	54
3.5.3 Immutable	58
3.5.4 Iterator	60
3.5.5 Fassade.....	61
3.6 Beispiel eines Entwurfs.....	63
3.7 Prinzipien guten Entwurfs.....	69
3.7.1 Was ist ein guter Entwurf	69
3.7.2 Teile und Herrsche.....	70
3.7.3 Hohe Kohäsion	71
3.7.4 Lose Kopplung	73
3.7.5 Grundlegende Prinzipien, die weiterhelfen.....	75
3.8 Verbesserung des Entwurfs durch Refactoring.....	77
3.9 Zusammenfassung	79
 Literatur	 81
 Lösungen zu den Aufgaben	 83
 Glossar	 91
 Stichwortverzeichnis	 93

1 Objektorientierte Analyse

Nach dem Durcharbeiten dieses Kapitels sollen Sie

- verstehen, wie und wozu Modelle in Analyse und Entwurf verwendet werden,
- wissen, wie Objekte und Klassen zur Analyse von Anforderungen der Software verwendet werden,
- Klassendiagramme der Analyse auf systematische Art und Weise erstellen können,
- Nutzen von Mustern beurteilen können,
- den Aufbau von Mustern verstehen,
- die wichtigsten Analysemuster kennen und verwenden können,
- Konzepte zum systematischen Entwurf und zur ergonomischen Gestaltung von Benutzeroberflächen kennen.

1.1 Modelle in Analyse und Entwurf

Wie Sie in der Kurseinheit „Grundlagen der Softwaretechnik und Requirements Engineering“ gelernt haben, werden in der ersten Phase einer Softwareentwicklung die Anforderungen erfasst. Man könnte denken, dass bereits auf Grundlage der Anforderungen die Software direkt codiert werden könnte, da ja nun bekannt ist, was die Kunden möchten. Bei diesem Vorgehen entstehen allerdings einige Probleme: Zum einen sind die auf die beschriebene Art erfassten Anforderungen zu ungenau. In der Praxis zeigt es sich, dass bei diesem Vorgehen dauernder Kontakt zu Kunden und späteren Endanwendern notwendig wäre, um die offenen Punkte zu klären. Als zweites Problem ergibt sich häufig, dass die Anforderungen in sich widersprüchlich sind, dies aber nicht ohne Weiteres erkennbar ist. Der manchmal beschrittene Weg, die Dokumente der Anforderungen formaler zu gestalten, hat sich als schwierig erwiesen, da diese Dokumente dann meist extrem umfangreich und schwer verständlich werden. Um diese Probleme zu vermeiden, wird in der Phase der Analyse ein formales Modell des Problems in seinem Umfeld und der Lösung erstellt. Die Analyse beschreibt wohlgerneht nur fachliche Zusammenhänge, keine technischen. In diesem Abschnitt werden Sie lernen, diese Modelle der Analyse zu erstellen.

Analyse

Entwurf	<p>Im Rahmen des Entwurfs wird zunächst die Architektur der Anwendung bestimmt, also z.B., welche Programmiersprache und welche Frameworks zum Einsatz kommen oder ob eine Datenbank verwendet werden soll. Mehr zur Softwarearchitektur werden Sie in Kapitel 2 erfahren. Sind die architektonischen Fragen geklärt, bleiben noch viele technische Fragen offen, z.B.: Welche Klassen aus der Analyse können wie übernommen werden? Wie werden Klassen in einem relationalen Datenbank-Schema gespeichert? Welche Steuerungsklassen für die graphische Benutzeroberfläche müssen implementiert werden? Diese Fragen werden im Entwurf beantwortet. Zu diesem Zweck wird ein Modell der geplanten Software erstellt. Gegenüber Ad-hoc-Lösungen besitzt dieses Vorgehen den Vorteil, dass Entwurfs-Alternativen einfacher und schneller durchdacht werden können. Zudem ist die Zusammenarbeit deutlich einfacher, wenn die Umsetzung genauer geplant wird.</p>
Definition Modell	<p>Nachdem der Begriff „Modell“ bei Analyse und Entwurf als entscheidender Begriff aufgetaucht ist, fragen Sie sich vielleicht, was denn darunter genau zu verstehen ist. Der Begriff „Modell“ ist in der Tat so schwierig zu definieren, dass die Definition ein nach wie vor ungeklärtes philosophisches Problem darstellt. Dies hat zur Folge, dass in der Literatur ein bunter Strauß zum Teil widersprüchlicher Definitionen zu finden ist. Als Arbeitsdefinition, also ohne den Anspruch auf vollständige Korrektheit, vereinbaren wir, dass ein Modell ein Produkt des Modellierungsvorgangs ist. Es beschreibt tatsächliche oder gedachte Gegenstände oder Konzepte und deren Beziehungen, also z.B. Bücher eines Versandhandels, Monster in einem Computerspiel oder Konten bei einer Bank. In einem Modell werden diese Konzepte nicht vollständig, sondern verkürzt und vereinfacht erfasst.</p>
Statische und dynamische Modelle	<p>Modelle der Analyse und des Entwurfs bestehen aus miteinander verzahnten statischen und dynamischen Modellen¹. Statische Modelle beschreiben die Bausteine eines Systems und wie sie zusammengesetzt sind. Diese Modelle sind vergleichbar mit technischen Zeichnungen. Dynamische Modelle beschreiben die Zusammenarbeit der Bausteine, also deren Verhalten und die Botschaften, die Verhalten auslösen. Dynamische Modelle sind eine Besonderheit der Informatik, die es in anderen Ingenieurdisziplinen nicht in dieser Form gibt². Sie sind notwendig, da das Zusammenspiel der Modellelemente aus dem statischen Modell nicht immer eindeutig abgeleitet werden kann. Sie stellen außerdem eine wichtige Verständnishilfe dar.</p>

1 Modelle können selbst wieder aus Modellen bestehen.

2 Dynamische Modelle sind am ehesten mit Gleichungen zu vergleichen, die z.B. die Bewegungen einer Tragfläche eines Flugzeugs beschreiben.

Modelle können in mathematisch-logischen oder graphischen Sprachen definiert werden. In der Praxis hat sich in den meisten Bereichen die graphische Modellierung durchgesetzt, da sie übersichtlich und auch für Ungeübte leicht verständlich ist. Als Grundlage der Modellierung wird in den meisten Fällen heute der objektorientierte Ansatz gewählt. Dies hat den Vorteil, dass für die Modellierung dieselben Techniken genutzt werden wie für die Umsetzung. Ausnahmen sind z.B. Entwicklungen für Embedded Systeme für Flugzeuge oder Automobile, für die speziell auf diese Bereiche ausgerichtete Modellierungstechniken entwickelt worden sind.

**Graphische,
objektorientierte
Modellierung**

Damit die Bedeutung der Beschreibung eines Modells eindeutig für alle Benutzer ist, wurde für die graphische, objektorientierte Modellierung der sehr verbreitete Standard UML entwickelt, den Sie bereits aus der Einheit „Einführung in die Programmierung“ kennen. UML steht für „Unified Modeling Language“. Der Standard steht zur Zeit der Erstellung dieser Kurseinheit in der Version 2.2.1 zur Verfügung. Als Referenz können Sie ihn von der Webseite der Object Management Group OMG herunterladen (<http://www.uml.org>). In den nächsten Abschnitten wird die Notation nach UML so weit wie nötig wiederholt oder eingeführt. Eine vollständigere Darstellung der UML werden Sie in der Kurseinheit „Systemmodellierung“ erhalten.

Standard UML



Übungsaufgabe

- 1.1 Welche zwei Eigenschaften zeichnen ein Modell gemäß unserer Arbeitsdefinition aus?

1.2 Klassen und Objekte

Objekte, Klassen und deren Beziehungen haben Sie bislang zur Beschreibung der Struktur von Programmcode kennen gelernt. Die gleichen Begriffe werden auch in der objektorientierten Analyse zur Beschreibung von fachlichen Zusammenhängen verwendet. In diesem Abschnitt werden die wichtigsten Begriffe der Objektorientierung und der UML in diesem Zusammenhang eingeführt.

Klassen und Objekte

Objekte sind konkrete tatsächliche oder gedachte Gegenstände oder Konzepte, also z.B. diese Kurseinheit, die Sie in Händen halten, das Buch, das Sie sich zu Weihnachten gewünscht haben, die Zahl 25 oder die Zeichenkette „ACGTAG“. Klassen sind Abstraktionen von Objekten, die gemeinsame Eigenschaften oder gleiches Verhalten von Objekten beschreiben³, also z.B. Kurseinheit, Buch, Zahl, String. Man sagt auch, dass ein Objekt eine Instanz oder ein Exemplar einer Klasse ist. Klassen und Objekte werden in der UML durch Rechtecke dargestellt. Bei Klassen enthalten die Rechtecke den Name der Klasse, bei Objekten einen Bezeichner der Form Objektname:Klassenname, wobei jeweils einer der beiden Namen weggelassen werden kann, wenn dies zur Klarheit der Darstellung beiträgt (Abb. 1.1)



Abb. 1.1: Beispiel UML-Darstellung einer Klasse (links) und eines Objekts (rechts)

Eigenschaften und Verhalten

Objekte besitzen Eigenschaften und Verhalten. Die Eigenschaften werden durch Attribute modelliert, das Verhalten durch Verantwortlichkeiten oder Operationen. Instanzen einer Klasse besitzen dasselbe Verhalten und dieselben Attribute, sie unterscheiden sich lediglich im Wert ihrer Attribute. Attribute sowie Verhalten und Operationen werden aus diesem Grund in den Klassen modelliert, in Objekten hingegen nur der Wert von Attributen. Beispielsweise hat eine Bestellung als Attribute ein Bestelldatum, ein Lieferdatum und eine Bestellnummer. Zu den Verantwortlichkeiten einer Bestellung gehört es, die Bestellsumme anzugeben und die Mehrwertsteuer berechnen zu können. In der UML werden die Attribute und Verantwortlichkeiten oder Operationen in so genannten Abteilen (engl. Compartment) dargestellt (Abb. 1.2).



Abb. 1.2: Beispiel UML-Darstellung einer Klasse (links) und eines Objekts (rechts) mit Attributen und Verantwortlichkeiten

3 Ähnlich zum Begriff „Modell“ ist die exakte Definition der Begriffe „Objekt“ und „Klasse“ ein ungelöstes philosophisches Problem. Das soll uns aber nicht davon abhalten, diese Begriffe zu benutzen.

Attribute besitzen einen Typ. Er kann in der Form `Attributname:Typ-bezeichnung` am Attributnamen angegeben werden. Dies geschieht in der Regel aber erst spät in der Analyse. Einschränkungen des Typs, z.B. bei Zahlen die Anzahl der Vor- und Nachkommastellen, können in geschweifte Klammern hinter dem Typ in der Object Constraint Language OCL, einem Teil der UML, angegeben werden. In den meisten Fällen kommen Sie mit der Syntax der Beispiele in Abb. 1.3 und Abb. 1.4 aus. Sollten Sie komplexere Beispiele benötigen, können Sie die Referenz zu OCL zu Rate ziehen.

Attributtypen

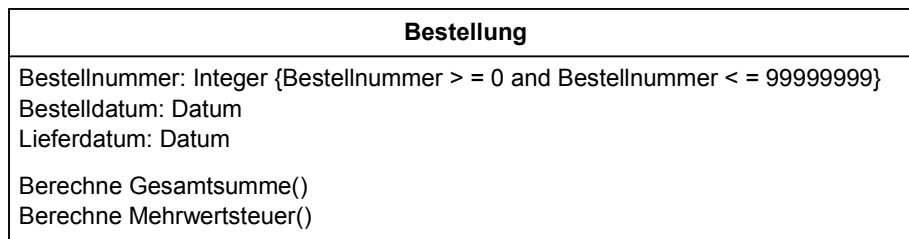


Abb. 1.3: Beispiel UML-Darstellung einer Klasse mit Attributen und Attributtypen

Generell ist es praktikabler, gemeinsame Datentypen zu vereinbaren, bei denen das Format einmal definiert wird, als bei jedem Auftreten dieses Typs einen allgemeinen Datentyp zu verwenden und die Einschränkung in OCL anzufügen (Abb. 1.4). Manchmal kann es auch praktisch sein, Einschränkungen in Textform in einem Kommentar festzulegen. Diese Angaben im Klassendiagramm der Analyse ersetzen das Datadictionary, das Sie in der ersten Kurseinheit „Grundlagen der Softwaretechnik und Requirements Engineering“ kennen gelernt haben.

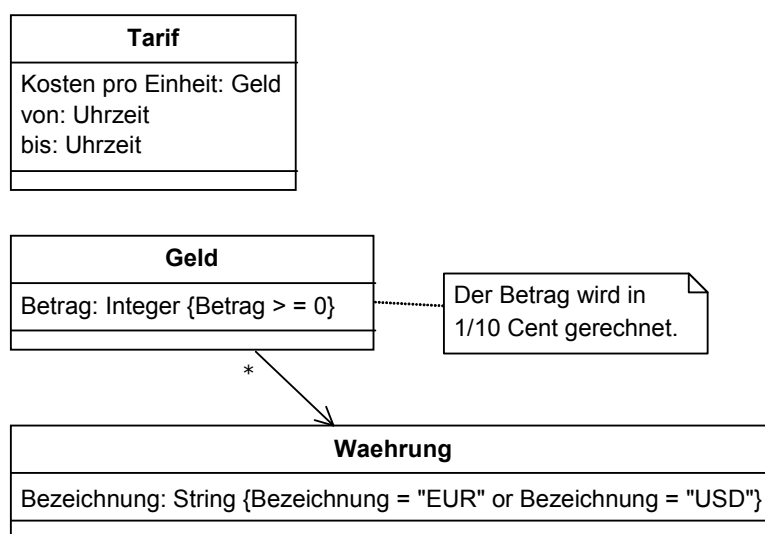


Abb. 1.4: Beispiel Definition mit Datentyp. Geld ist dabei der Datentyp, der von der Klasse Tarif benutzt wird.



Übungsaufgabe

- 1.2 Definieren Sie den Datentyp „Uhrzeit“ aus Abb. 1.4 in UML. Wählen Sie als Attribute Stunde und Minute.

1.3 Beziehungen von Klassen

Klassen können auf drei unterschiedliche Arten miteinander in Beziehung stehen:

- Abhängigkeit (engl. Dependency)
- Assoziation (engl. Association)
- Vererbung (engl. Inheritance)

Bei der Analyse ist herauszufinden, welche Art von Beziehung vorliegt, damit sie entsprechend modelliert werden kann.

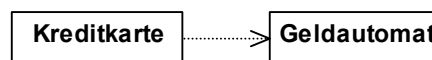


Abb. 1.5: Beispiel UML-Darstellung einer Abhängigkeit

Abhängigkeit

Eine Klasse ist von einer anderen Klasse abhängig, wenn sie diese benutzt, aber keine dauerhafte Beziehung zu ihr hat. Ändert sich etwas an der benutzten Klasse, muss auch die Nutzerklasse geändert werden. Ein Beispiel sind Kreditkarten und Geldautomaten. Eine Kreditkarte hat keine dauerhafte Beziehung zu einem Geldautomaten. Werden beispielsweise Geldautomaten so umgestellt, dass der Magnetstreifen nicht mehr gelesen werden kann, sondern nur noch ein integrierter Chip, benötigen die Benutzer eine neue Kreditkarte. Dieser Zusammenhang wird in UML durch einen Pfeil mit einer gestrichelten Linie ausgedrückt (Abb. 1.5). Abhängigkeiten werden in der Analyse selten modelliert.

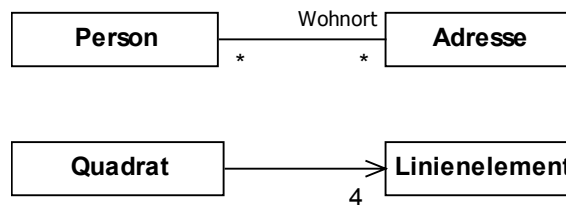


Abb. 1.6: Beispiel UML-Darstellung von Assoziationen. Eine Person kann an mehreren Adressen einen Wohnort besitzen. An einer Adresse können mehrere Personen leben. Ein Quadrat besteht aus 4 Linien.

Unter einer Assoziation versteht man, dass eine Instanz einer Klasse eine Instanz derselben oder einer anderen Klasse dauerhaft kennt. „Dauerhaft“ meint keinen speziellen Zeitraum, sondern dass die Beziehung über eine einzelne Benutzung hinaus dauern kann. Beispiele für Assoziationen sind die Beziehung zwischen einem Menschen und seiner Adresse oder zwischen einem Quadrat und den Linien, aus denen es gebildet wird. Eine Assoziation wird in UML als durchgehende Linie dargestellt. Richtungen in Beziehungen können durch Pfeile dargestellt werden. Multiplizitäten können direkt aufgezählt oder als Wertebereich angegeben werden (Abb. 1.6). Assoziationen sind in der Modellierung die am häufigsten verwendete Beziehung.

Assoziation

Vererbung ist ein Begriff, der aus mehreren Richtungen kommend definiert werden kann. Die verschiedenen Definitionen beleuchten unterschiedliche Aspekte dieses Begriffs. Für eine korrekte Verwendung der Vererbung ist es sehr wichtig, diese Aspekte zu kennen. Zum einen bezeichnet die Vererbung eine Spezialisierung. Als Beispiel dienen Verkäufer in einem gedachten Geschäft. Dort gibt es drei Typen von Verkäufern: Hilfskräfte, die Kunden bedienen und Regale einräumen, Angestellte, die auch an der Kasse arbeiten dürfen, und Abteilungsleiter, die ebenfalls dieselben Tätigkeiten durchführen können und zusätzlich auch Anweisungen geben dürfen. Hilfskräfte, Angestellte und Abteilungsleiter sind also spezielle Arten von Verkäufern. Sie besitzen die gleichen Eigenschaften, wie z.B. einen Namen, und die gleichen Verantwortlichkeiten, wie „verkaufen“ und „Regal einräumen“, sie haben aber noch zusätzliche Eigenschaften und Verantwortlichkeiten. Für diesen Zusammenhang gibt es verschiedene Formulierungen: Der Abteilungsleiter erbt Eigenschaften und Verantwortlichkeiten eines Verkäufers. Abteilungsleiter ist eine Spezialisierung des Verkäufers. Der Verkäufer ist eine Verallgemeinerung von Hilfskraft, Angestellter und Abteilungsleiter. Die Vererbung wird in der UML als durchgezogene Linie mit einem nicht ausgefüllten Dreieck als Pfeilspitze dargestellt. Der Pfeil zeigt in Richtung der Verallgemeinerung (Abb. 1.7).

Vererbung

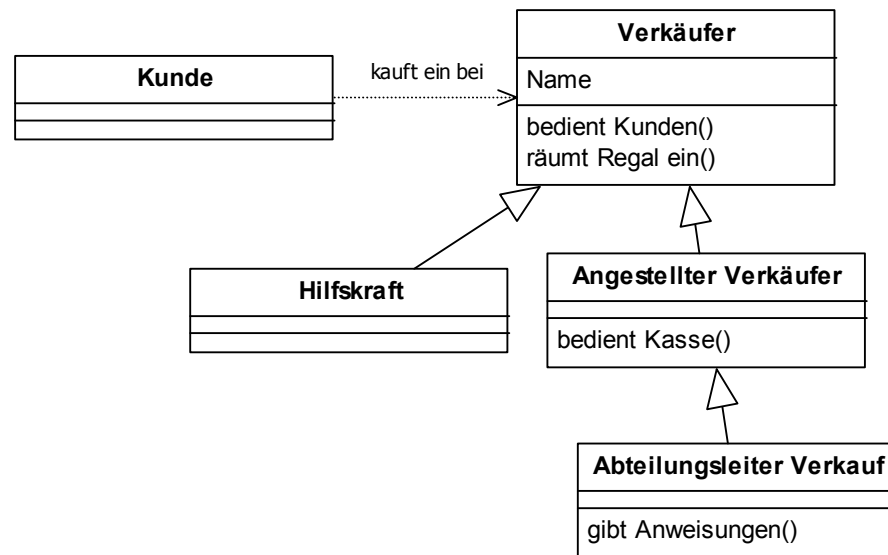


Abb. 1.7: Beispiel für Vererbung und Polymorphie. Ein Kunde kann von jedem Verkäufertyp bedient werden.

Polymorphie

Eine weitere Bedeutung der Vererbungsbeziehung ist, dass eine Instanz einer abgeleiteten Klasse genauso verwendet werden kann wie eine Instanz der übergeordneten Klasse. Man nennt dies Polymorphie. Sie ist durch das Liskovsche Substitutionsprinzip (s. S. 76) beschrieben: Jede Klasse kann durch eine abgeleitete Klasse ersetzt werden und genauso verwendet werden. Das Verhalten muss nicht notwendigerweise identisch sein. Als Beispiel für Polymorphie kann ein Kunde bei jedem Verkäufer einkaufen, egal, ob der Verkäufer Hilfskraft, Angestellter oder Abteilungsleiter ist. Wird über den Preis verhandelt, haben die verschiedenen Verkäufertypen ggf. unterschiedliche Befugnisse.



Übungsaufgaben

- 1.3 Welche Pfeilspitzen und Linientypen gibt es zur Darstellung von Beziehungen? Was bedeutet welche Kombination?
- 1.4 Um welche Beziehungen handelt es sich bei den folgenden Beschreibungen? Modellieren Sie den Zusammenhang in UML:
 - a) Ein Auto besitzt vier Räder.
 - b) Autos sind Fahrzeuge.

- c) An Tankstellen können Autos betankt werden.
 - d) Jedes Auto gehört einer Person.
- 1.5 Modellieren Sie folgenden Zusammenhang in einem UML-Diagramm. Es gibt zwei Arten geometrischer Objekte: Kreise mit einem Radius und Quadrate mit einer Seitenlänge. Von allen geometrischen Objekten kann der Flächeninhalt berechnet werden.

1.4 Domänenmodell

Der Begriff „Domäne“ bezeichnet das fachliche Umfeld der Aufgabe. Das Domänenmodell ist das statische Modell der Domäne. Es wird in der Analyse in Form von Klassendiagrammen erstellt. Das Domänenmodell beschreibt, welche Klassen es in einer Domäne gibt, wie sie aussehen und welche Beziehungen sie zueinander haben. Dabei wird vollständig auf die Darstellung von technischen Gesichtspunkten der späteren Lösung verzichtet. Da das Domänenmodell rein fachlich ist, kann es auch zur Kommunikation mit Kunden oder Anwendern verwendet werden, falls diese keine Probleme mit formaleren Methoden haben.

Klassenmodell der Domäne

Das Domänenmodell ist wichtig, da es in den weiteren Arbeiten sehr oft benutzt wird: In der Analyse ist es eine der Grundlagen der Entwicklung der graphischen Benutzeroberfläche (Abschnitt 1.7). In der Realisierung dient es als Basis der Implementierung der Datenhaltung der Software, der Datenbankmodelle, der Formate von Dateien oder auch der Schnittstellen zwischen Anwendungen (Kapitel 3). Aus diesem Grund entscheiden sich viele Praktiker, das Domänenmodell im Lauf einer Entwicklung immer weiterzuentwickeln und konsistent zu dokumentieren.

Grundlage für weitere Aktivitäten

Im nächsten Abschnitt lernen Sie zunächst ein etwas komplexeres Beispiel kennen. Anhand dieses Beispiels wird das typische Vorgehen bei der Domänenmodellierung Schritt für Schritt demonstriert.

Vorgehen in diesem Abschnitt

1.4.1 Beispiel Bibliothek

In diesem Abschnitt werden Sie als Beispiel die Anforderungen für die Entwicklung einer Software für eine Bibliothek kennen lernen. Dieses Beispiel wird auch in Abschnitt 2.3 verwendet. Selbstverständlich gibt es auf dem

Markt verschiedene ausgereifte Produkte für diese Aufgabe, so dass eine Eigenentwicklung nicht sinnvoll wäre. Die Aufgabe stellt aber ein gut geeignetes Beispiel dar. Auf die Beschreibung von Geschäftszielen wird bewusst verzichtet. Auch die Beschreibung der Anforderungen ist knapper, als man dies in der Praxis machen würde.

Ziel des Kunden: Eine Bibliothek, die bisher die Katalogisierung und die Verwaltung der Nutzer und Ausleihen manuell betrieben hat, möchte eine Software entwickeln lassen, die diese Vorgänge unterstützt.

Vision: Für Mitarbeiter der Bibliothek, die Bücher katalogisieren und Bücher verleihen, und für Benutzer, die Bücher im Katalog suchen und Bücher ausleihen, ist EasyBibo das neue Bibliothekssystem, das ihnen hilft, zu recherchieren und die Entleihungen zu verwalten. Anders als bisher wird es mit unserem Produkt nicht mehr nötig sein, in Karteikarten langwierig zu suchen. Weiterhin ist immer abfragbar, welche Bücher ausgeliehen und welche verfügbar sind.

Nicht-funktionale Anforderungen und Randbedingungen

- N1. Mehrbenutzerbetrieb
- N2. Die Daten sind lebensnotwendig für die Bibliothek. Sie dürfen nicht verloren gehen oder unbrauchbar werden.
- N3. Das System ist für den Betrieb der Bibliothek sehr wichtig. Bei Ausfällen kann die Bibliothek nicht mehr arbeiten.
- N4. Die Benutzeroberfläche muss ergonomisch sein, da die Benutzer der Bibliothek ungeschult und die Bibliotheksangestellten keine Computerexperten sind.
- N5. Antwortzeit unter 1s, bei der Suche unter 3s

User Storys:

- U1. Als Bibliotheksmitarbeiter melde ich einen neuen Benutzer an oder ändere seine Adresse.
- U2. Als Bibliotheksmitarbeiter trage ich ein Buch oder eine CD in den Katalog ein.
- U3. Als Benutzer oder Mitarbeiter suche ich ein Buch oder eine CD im Katalog.
- U4. Als Benutzer sehe ich mein Leihkonto an und verlängere meine Leihfristen.
- U5. Als Bibliotheksmitarbeiter bestelle ich ein von Benutzern gewünschtes Buch oder eine CD im Buchhandel.

Als Grundlage für die Analyse hier noch die wichtigsten verwendeten Formulare.

<u>Benutzeranmeldung</u>	
Nachname	
Vorname	
Strasse und Hausnummer	
Wohnort	
Postleitzahl	
Benutzernummer	<u>Von der Bibliothek auszufüllen</u>

Abb. 1.8: Formular für die Anmeldung eines neuen Benutzers

Signatur	ST 278 C776
Autor(en)	Cooper, Alan; Reimann, Robert M; Cronin, David
Titel	<u>About face 3.0: The essentials of interaction design</u>
Verlag	Wiley & Sons
Auflage	3
Jahr	2007
Exemplare	3

Abb. 1.9: Beispiel einer Karteikarte des Katalogs eines Buchs

Signatur	B000J0SNFK
Interpret(en)	Von Karajan, Herbert
Komponist	Mozart, Wolfgang Amadeus
Titel	<u>Eine kleine Nachtmusik</u>
Verlag	Deutsche G(Universal)
Jahr	2006
Exemplare	2

Abb. 1.10: Beispiel einer Karteikarte des Katalogs einer CD

Anmerkung: Gibt es mehrere Exemplare, erhalten diese eine eindeutige Nummer der Form Signatur-Nummer.

Benutzernummer	32170815	
Name	Gero Wedemann	
Signatur	Titel	Entliehen am
ST 230 L647(2)	Software Engineering	12.1.07
ST 230 W6455	Software Requirements	12.1.07 27.2.07
ST 278 C776	About Face	27.2.07

Abb. 1.11: Beispiel einer Entleihungskarte

Mengengerüst

Die Bibliothek hat ca. eine Million Bücher, fünfzigtausend CDs, zehntausend Benutzer. Jeden Tag nehmen etwa 250 Benutzer ungefähr 500 Entleihungen vor. Die Ausleihe ist jeden Tag von 9:00 bis 19:00 geöffnet. Es werden jedes Jahr ca. 5.000 Bücher beschafft. Die Bibliotheksverwaltung erwartet, dass sich die Nutzerzahl und die Anzahl der Ausleihen aufgrund des neuen Systems in den nächsten Jahren verdoppeln.

1.4.2 Erstellung eines Domänenmodells

Schritt für Schritt

Das Domänenmodell basiert auf den Anforderungen und den in der Anforderungsphase gesammelten Informationen. Es hat sich bewährt, dabei systematisch die einzelnen Teile des Modells nacheinander zu ermitteln:

1. Klassen
2. Assoziationen
3. Attribute
4. Vererbungsbeziehungen
5. Verantwortlichkeiten oder Operationen

Mehrere Durchläufe

In dieser Reihenfolge werden bei der Modellierung in mehreren Durchgängen nach und nach immer mehr Klassen hinzugenommen, bis alles zufriedenstellend abgebildet ist. Die vorgeschlagene Reihenfolge sollten Sie als

Hilfestellung betrachten, nicht als Regel, an die man sich sklavisch halten muss. Manche Autoren sind übrigens der Auffassung, dass zuerst die Verantwortlichkeiten und erst dann die Beziehungen und Attribute modelliert werden sollten. Ich persönlich empfehle Ihnen, die angegebene Reihenfolge einzuhalten und sie ggf. bei Bedarf anzupassen.

Im Folgenden wird dieses Vorgehen genauer beschrieben und anhand des Beispiels der Bibliothek demonstriert. Wir fangen mit Katalog, Büchern und CDs an und lassen die Modellierung der Benutzer für den zweiten Durchgang.

Sehen Sie sich zunächst die Hauptwörter (Substantive) in den Beschreibungen an. Sie stehen meistens für Klassen, Objekte oder Attribute. In unserem Beispiel sind die Klassen Katalog, Buch und CD sofort zu identifizieren. Bei der Durchsicht der beiden Karteikarten (Abb. 1.9 und Abb. 1.10) stellt sich die Frage, ob es sich bei den anderen Einträgen um Attribute oder Klassen handelt. Wir stellen fest, dass Personen als Autoren, Interpreten oder Komponisten vorkommen können. Es gibt viele Fälle, in denen Interpret und Komponist dieselbe Person ist, oder ein Interpret einer CD der Komponist einer anderen CD ist. Es gibt auch den Fall, dass ein Musiker als Autor eines Gedichtbands auftritt. Es ist also logisch, eine neue Klasse Künstler einzuführen. Jedes Exemplar eines Buchs oder einer CD erhält eine eigenständige Nummer. Aus diesem Grund ist auch Exemplar als Klasse zu sehen.

Klassen ermitteln

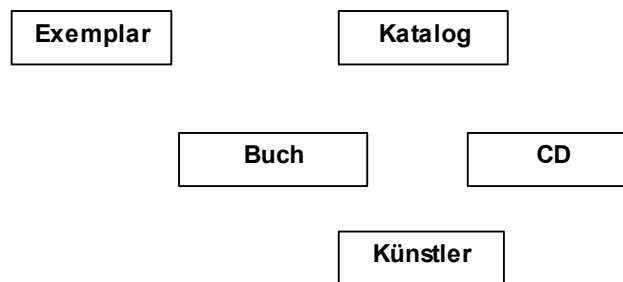


Abb. 1.12: Beispiel Analyse Schritt 1: Identifiziere Klassen

Assoziationen erkennt man an Beschreibungen wie

- hat
- besitzt
- kontrolliert
- ist verbunden mit

Assoziationen ermitteln

- ist Teil von
- hat Teil
- ist Mitglied von
- besitzt Mitglied

Hinter den Formulierungen „hat“ und „besitzt“ verbergen sich allerdings häufig auch Attribute.

In unserem Beispiel gibt es mehrere Assoziationen zwischen Künstler und Buch und CD. Dabei berücksichtigen wir, dass es möglich ist, dass der gleiche Künstler mehrere Bücher geschrieben oder CDs aufgenommen hat.

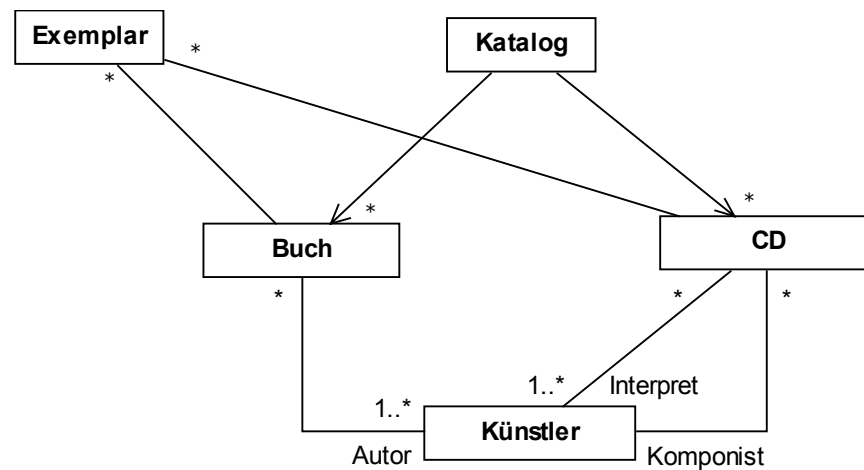


Abb. 1.13: Beispiel Analyse Schritt 2: Identifiziere Assoziationen

Attribute sind Informationen, die eine Klasse besitzt und pflegt. In unserem Beispiel können wir die Attribute den Karteikarten entnehmen. Jahreszahl wurde dabei als eigenständiger Datentyp definiert.

Attribute ermitteln

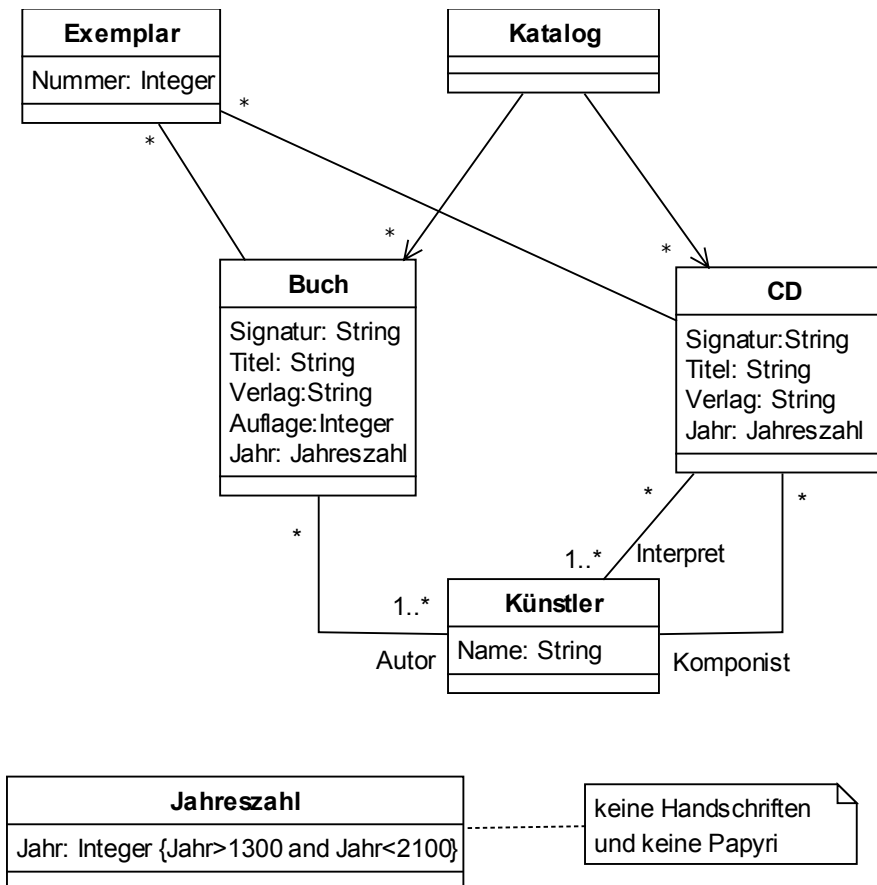


Abb. 1.14: Beispiel Analyse Schritt 3: Identifiziere Attribute

Vererbungsbeziehungen ermitteln

Vererbungsbeziehungen erkennt man daran, dass Daten oder Verhalten mehrerer Klassen identisch sind. Es ist ein wichtiges Merkmal einer Vererbungsbeziehung, dass eine Klasse anstatt einer anderen verwendet werden kann. Für die Beziehung zwischen einer abgeleiteten Klasse und ihrer Superklasse muss gelten „kann verwendet werden als“ und „ist ein“ (Abschnitt 1.3).

In unserem Beispiel haben Buch und CD viele identische Attribute. Wir definieren die neue Klasse Titel, die diese Attribute trägt und die Beziehung zum Exemplar und zum Katalog beinhaltet.

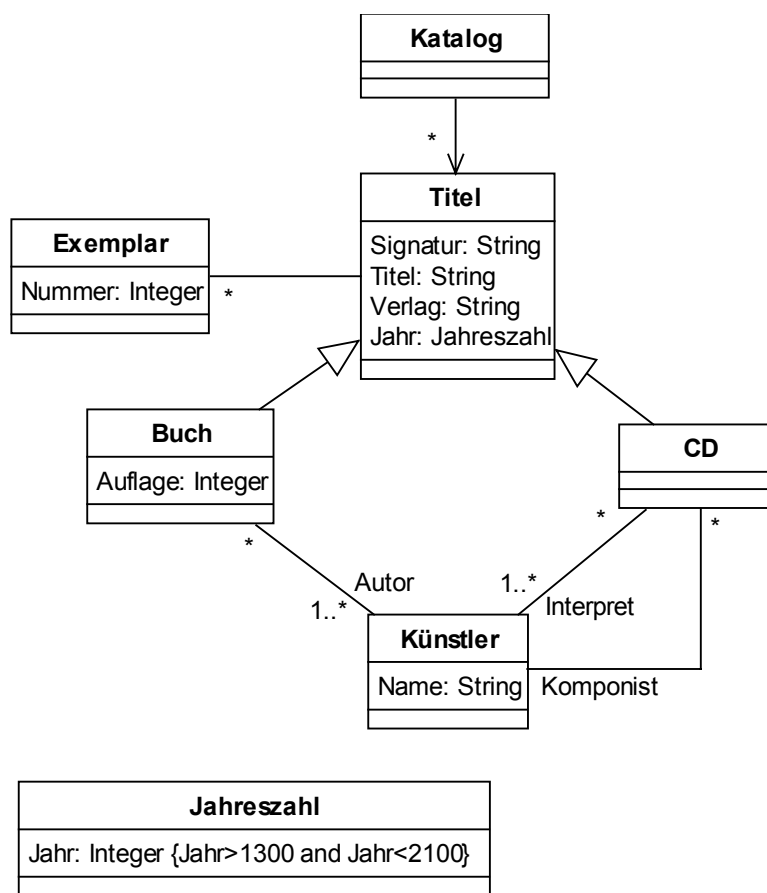


Abb. 1.15: Beispiel Analyse Schritt 4: Identifiziere Vererbungsbeziehungen

Unter dem Begriff Verantwortlichkeit fasst man die Summe des Verhaltens einer Klasse zusammen. Zum typischen Verhalten gehört z.B. lesen, schreiben, suchen, berechnen, entscheiden. In unserem Beispiel kann im Katalog nach Titeln gesucht werden. Die Klasse Katalog hat also die Verantwortlichkeit „Suche“. Bei der Suche müssen die Felder eines Titels durchsucht werden. Diesen Teil ihrer Verantwortlichkeit delegiert die Klasse Katalog an die Klasse Titel. Die Klasse Titel hat also die Verantwortlichkeit „Suche nach Begriff“.

Verantwortlichkeit

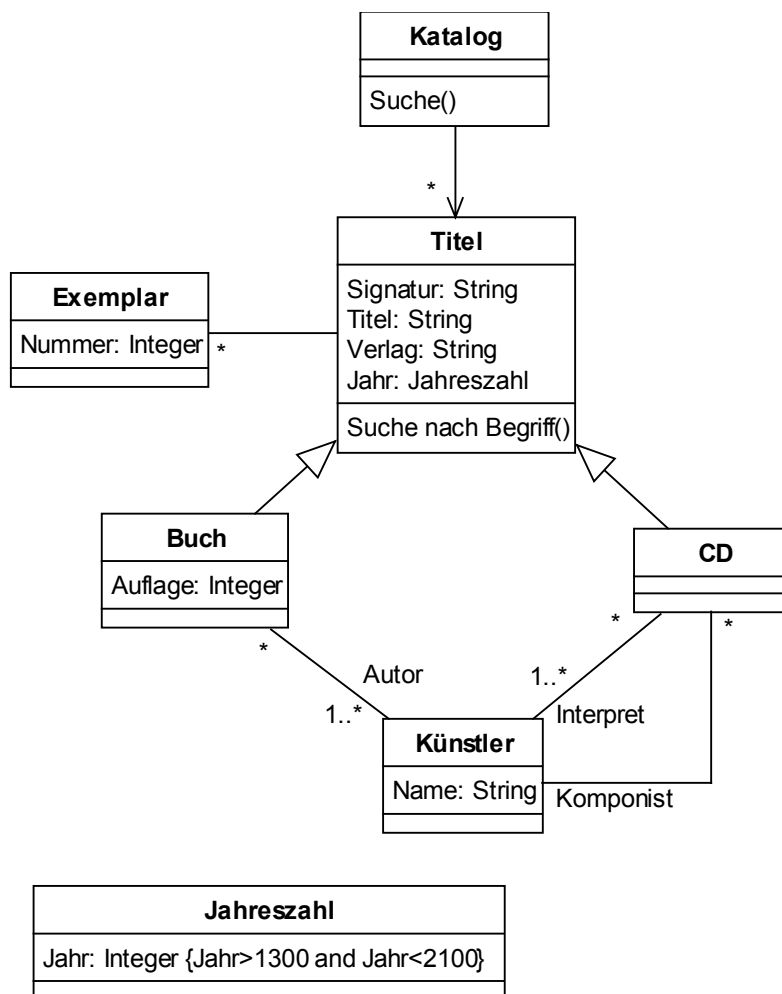


Abb. 1.16: Beispiel Analyse Schritt 5: Identifiziere Verantwortlichkeiten

Manchmal kann eine Verantwortlichkeit nicht eindeutig einer Klasse zugeordnet werden. In diesem Fall sollten Sie prüfen, ob entweder ein Fehler in der Modellierung vorliegt oder eine Klasse modelliert werden muss, die genau diese Verantwortung trägt.

**Eindeutige
Zuordnung der
Verantwortung**

Ein oder zwei Verantwortlichkeiten pro Klasse

Klassendiagramme sind übersichtlicher und einfacher zu verstehen, wenn die Klassen nur eine, maximal zwei Verantwortlichkeiten haben. Hat eine Klasse mehr Verantwortlichkeiten, sollten Sie darüber nachdenken, diese Klasse aufzuteilen.

Umgang mit Fragen an die Anforderungen

Vielleicht haben Sie nun nach der Analyse verschiedene Fragen zu den Anforderungen. So stellt sich z.B. im Beispiel die Frage, ob der Name eines Künstlers besser durch Vor- und Nachname modelliert werden sollte. Vielleicht könnten mit dem neuen System ja noch mehr Daten erfasst werden, was bisher technisch nicht möglich war, z.B. könnte das Inhaltsverzeichnis eines Buchs eingescannt und durchsuchbar gemacht werden. Solche Ideen sind wichtig. Sie sollten sie notieren und mit den Kunden besprechen. Auf keinen Fall sollten Sie sie ohne Rücksprache integrieren.



Übungsaufgabe

- 1.6 Nun sind Sie an der Reihe: Modellieren Sie zunächst den Benutzer und dann seine Entleihungen.

1.5 Modellierung der Dynamik

Modellierung des Zusammenspiels der Klassen

In der statischen Modellierung im Domänenmodell werden Klassen in ihrem Zusammenhang modelliert. Aus diesen statischen Zusammenhängen bleibt unklar, wie sich die Klassen oder Lösungen genau verhalten sollen. Es kann nicht dargestellt werden, wenn beispielsweise eine Instanz einer Klasse mehrere Zustände haben kann, in denen sie sich unterschiedlich verhält. Im statischen Modell wird auch nicht modelliert, welche Klasse welchen anderen Klassen welche Botschaften in welcher Reihenfolge tatsächlich sendet. Zur Beantwortung dieser Fragen dient das dynamische Modell. In der Analyse werden also Zustände und Zustandsübergänge eines Systems sowie Aktivitäten modelliert. Die Grundlage für die dynamische Modellierung sind Dokumente der Anforderungen, die Verhalten beschreiben, also insbesondere User Storys, Anwendungsfälle und Szenarien. Die dort beschriebenen Abläufe werden bei der Erstellung des dynamischen Modells konkretisiert und genauer definiert.

Beispiel Beschaffungsvorgang

Als Beispiel wird die User Story U5 des Beispiels aus Abschnitt 1.4.1 modelliert (Seite 10). Zunächst eine ausführlichere Beschreibung dieser Story: Bibliotheksmitarbeiter oder Benutzer können die Neuanschaffung eines

Buchs vorschlagen. Der oder die Verantwortliche des Bereichs (z.B. Naturwissenschaften, Wirtschaft oder Belletristik) prüft den Vorschlag und entscheidet. Er oder sie bestellt ein oder mehrere Exemplare in einer Buchhandlung, gibt das Buch zum Buchbinder und sortiert es ein. Sobald das Buch da ist, wird es katalogisiert. In der neuen Lösung soll dieser Vorgang von der Software unterstützt werden. Das Buch soll bereits im Katalog sichtbar sein, sobald es bestellt ist; die endgültige Katalogisierung erfolgt aber erst bei Eingang des Buchs.

Systeme, also Klassen, Subsysteme oder ganze Anwendungen, können Zustände besitzen. Diese Zustände können sich verändern. Meist sind nur Übergänge zwischen bestimmten Zuständen möglich. Zustände und Zustandsübergänge können in UML-Zustandsdiagrammen modelliert werden. Als Beispiel sind in Abb. 1.17 die Zustände eines Exemplars im oben beschriebenen Beschaffungsvorgang modelliert. Die Klasse Exemplar hat für diesen Zweck ein weiteres Attribut mit der Bezeichnung Bestellstatus erhalten. Diesen Status trägt die Klasse Exemplar und nicht Titel, da ja konkrete Buch- oder CD-Exemplare bestellt werden und nicht abstrakte Titelinformationen. Die Einzelheiten von Zustandsdiagrammen werden ausführlich in der Kurseinheit „Systemmodellierung“ erklärt.

Zustände und Zustandsänderungen

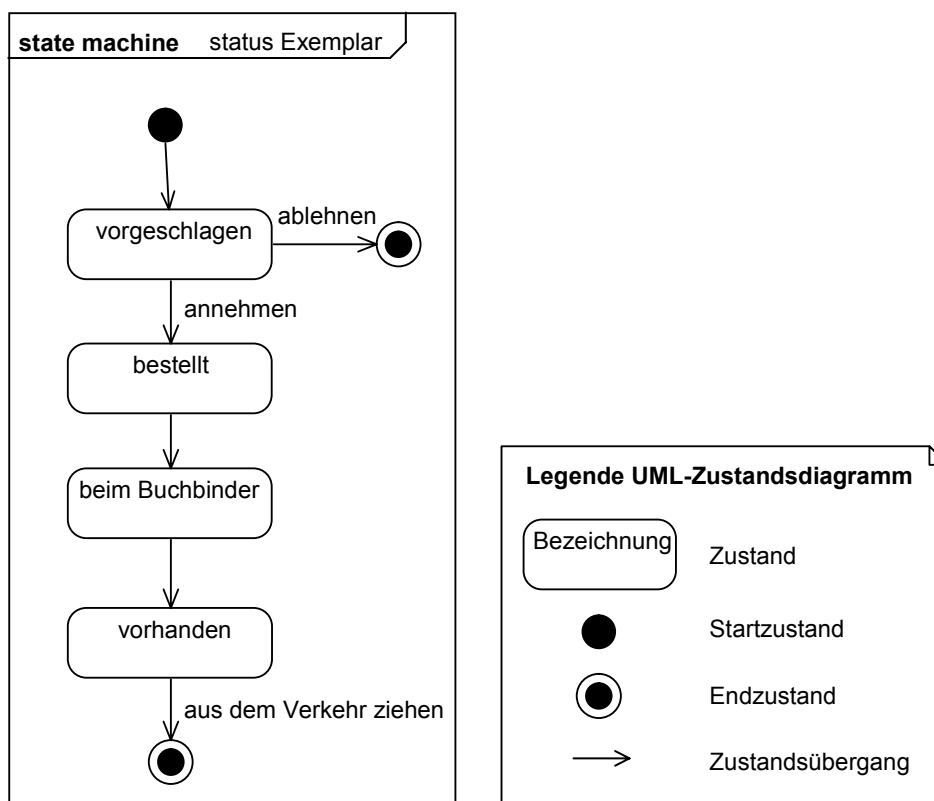


Abb. 1.17: Beispiel für ein UML-Zustandsdiagramm. Zustände und Zustandsänderungen des Bestellstatus eines Exemplars eines Buchs oder einer CD im Bestellprozess.

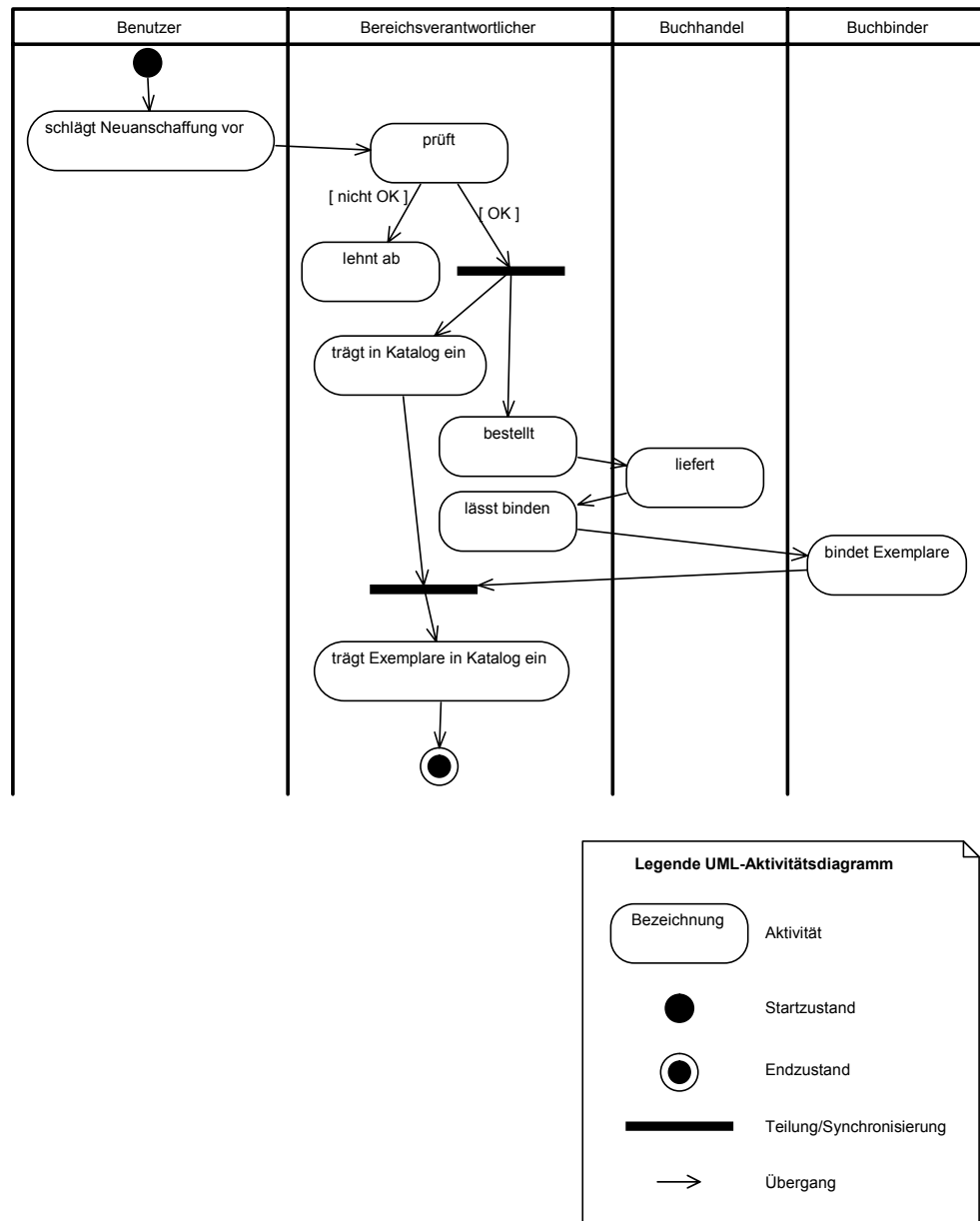


Abb. 1.18: Beispiel für ein UML-Aktivitätsdiagramm. Aktivitäten bei der Beschaffung von Exemplaren. Die Schwimmbahnen ordnen die Aktivitäten Akteuren zu.

Aktivitätsdiagramme

In Zustandsdiagrammen werden Aktivitäten als Übergänge zwischen Zuständen modelliert. Der Ablauf von Aktivitäten erschließt sich nur indirekt. Was mit Zustandsdiagrammen nicht modellierbar ist, sind Abläufe von Aktivitäten verschiedener Klassen, deren Instanzen als Handelnde (Akteure) auftreten. Zu diesem Zweck werden UML-Aktivitätsdiagramme eingesetzt. In einem Aktivitätsdiagramm werden also Aktivitäten in ihrer möglichen Abfolge und ihrer Zuordnung zu Akteuren modelliert. Als Beispiel sind in Abb. 1.18 die Aktivitäten der oben dargestellten Buchbestellung modelliert.

Aktivitätsdiagramme besitzen noch deutlich mehr Ausdrucksmittel. So kann beispielsweise auch der Fluss von Informationen und die Auswirkung auf Instanzen dargestellt werden. Sie sind zur Modellierung von User Storys, Szenarien oder Anwendungsfällen bestens geeignet. Aktivitätsdiagramme werden in der Kurseinheit „Systemmodellierung“ ausführlich dargestellt.



Übungsaufgabe

- 1.7 Wie werden Aktivitäten in Zustands- und wie in Aktivitätsdiagrammen dargestellt?

1.6 Analysemuster

Im Software Engineering versteht man unter dem Begriff Muster (engl. Pattern) vorgefertigte Lösungsschablonen für verallgemeinerte Probleme. Diesem Ansatz liegt die Beobachtung zugrunde, dass bei der Entwicklung ganz unterschiedlicher Software immer wieder sehr ähnliche Probleme auftreten und sich auch die Art der Lösung dieser ähnlichen Probleme sehr stark ähnelt. Die Beschreibung der verallgemeinerten Probleme und Lösungen auf eine formalisierte Weise nennt man Muster. Muster sind wohl-gemerkt verallgemeinerte Lösungen. Möchten Sie ein Muster also bei einem konkreten Problem einsetzen, müssen Sie das Muster für Ihren Zweck konkretisieren.

Muster

Muster sind aus vielen Gründen nützlich: Zum einen stehen Ihnen mit Mustern eine Reihe von bewährten Standardlösungen zur Verfügung. Die Verwendung dieser bewährten Standardlösungen ist meist schneller und besser als selbst neue Lösungen zu entwickeln. Zum anderen helfen Muster bei der Kommunikation. Kennt Ihr Gegenüber das verwendete Muster, hilft ihm oder ihr der Verweis auf das Muster zu verstehen, warum die gewählte Lösung so aussieht.

Vorteile

Muster gibt es für die Bereiche Analyse, Architektur und Entwurf. In diesem Abschnitt werden Sie eine Auswahl der wichtigsten Analysemuster kennen lernen, die erfahrungsgemäß bei fast jeder Analyse benutzt werden. Architektur- und Entwurfsmuster werden Sie in den entsprechenden Kapiteln kennen lernen. Weitere Analysemuster finden Sie z.B. in Lehrbüchern über Software Engineering wie [Let05] oder in speziellen Büchern über Analysemuster wie [Fow98].

Analysemuster

Standardisierte Beschreibung

Die Beschreibung der Muster erfolgt nach einem standardisierten Aufbau:

- Beschreibung des Problems in seinem Kontext
- Lösung
- Beispiele
- Gegebenenfalls Antimuster

Antimuster sind Beispiele für Lösungen, die ungeeignet sind.

Lesereihenfolge

Falls Ihnen die Problembeschreibung zu abstrakt ist, können Sie auch mit den Beispielen beginnen und erst dann die abstraktere Problem- und Lösungsbeschreibung lesen.

1.6.1 Exemplartyp (Abstraction – Occurence)

Problem in seinem Kontext

Bei der Modellierung fällt auf, dass Objekte einer Klasse sich untereinander ähneln. Sie tragen gemeinsame, gleiche Information oder besitzen gleiches Verhalten, unterscheiden sich aber trotzdem wesentlich. Dieser Zusammenhang zwischen den Klassen soll zum Ausdruck gebracht werden. Dabei soll vermieden werden, dass mehrere Instanzen die identischen Daten mehrfach beinhalten (Datenreplikation).

Lösung

Gemeinsame Daten beinhalten Objekte der Klasse Abstraction (engl. für Abstraktion), die individuellen Daten Objekte der Klasse Occurrence (Auftreten) (Abb. 1.19).

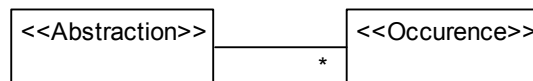


Abb. 1.19: Klassendiagramm des Musters Exemplartyp. Die spitzen Klammern im Diagramm („<<...>>“) bedeuten, dass der Name die Bezeichnung des Typs der Klasse ist und nicht der Klasse selbst.

Beispiel

Ein Beispiel sind z.B. Flüge, die die gleiche Flugnummer und denselben Abflug- und Zielflughafen besitzen, aber an unterschiedlichen Tagen fliegen. Der Flug ist dabei die Abstraction. Er trägt die gemeinsamen Attribute. Zu jedem Flug kann es mehrere Flugtermine geben, die sich im Datum unterscheiden (Abb. 1.20).

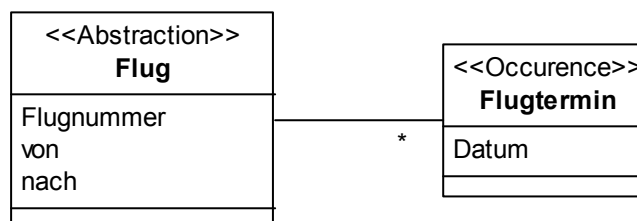


Abb. 1.20: Beispiel für den Einsatz des Musters Exemplartyp in einem Klassendiagramm

Dieser Zusammenhang wird häufig falsch modelliert. Abb. 1.21 zeigt drei Lösungen, die falsch sind. Bei Antimuster A werden alle Daten als Attribute einer Klasse modelliert. Der Zusammenhang zwischen den Flügen mit derselben Flugnummer wird hier nicht ausgedrückt. Die Instanzen dieser Klasse mit derselben Flugnummer tragen immer auch die gleichen Werte in den Attributen „von“ und „nach“. Antimuster B sieht auf den ersten Blick besser aus. Hier werden die Daten in zwei Klassen modelliert. Die „ist ein“-Beziehung zwischen beiden Klassen wurde allerdings falsch modelliert. So wie es hier modelliert wurde, würde die Klasse Flug nie instanziiert werden. Jede Instanz von Flugtermin würde genauso wie bei A alle Daten replizieren (Aufgabe 1.8). Bei Antimuster C müssen bei allen Instanzen ebenfalls die Daten repliziert werden. Es kommt hier noch schlimmer. Für jeden Flugtermin muss eine neue Klasse und nicht nur eine neue Instanz erzeugt werden.

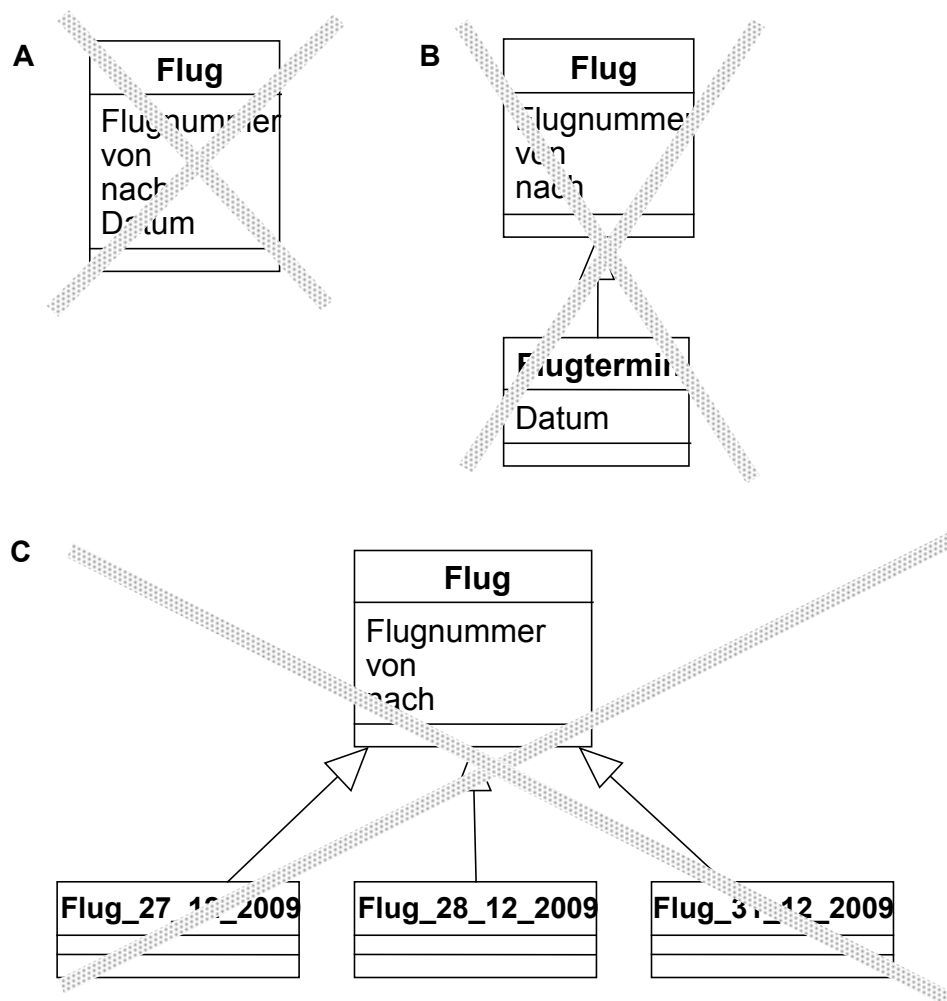
Antimuster

Abb. 1.21: Antimuster Exemplartyp für Beispiel Flug. Diese Lösungen sind ungeeignet.

Grenzen

Bitte beachten Sie, dass das Muster Exemplartyp nur sinnvoll eingesetzt werden kann, wenn die einzelnen Exemplare auch tatsächlich unterschieden werden können und sollen. So wird z.B. bei einem Lebensmittelhändler nicht jede Spaghetti-Packung als einzelnes Exemplar modelliert. Stattdessen ist hier die Angabe der Anzahl ausreichend.



Übungsaufgaben

- 1.8 Zeichnen Sie Beispiele für Instanzen von Flugtermin des Antimusters aus Abb. 1.21 B.
- 1.9 Identifizieren Sie Abstraction und Occurrence und zeichnen Sie ein Klassendiagramm für folgendes Beispiel: Die Seminare „Projektleitung“, „Objektorientierte Analyse und Entwurf“, „Konfliktmanagement“ und einige mehr finden an verschiedenen Tagen und Orten statt. Sie haben aber immer den gleichen Inhalt.

1.6.2 Wechselnde Rollen (Player – Role)

Problem in seinem Kontext

Ein Objekt kann in verschiedenem Kontext verschiedene Verantwortlichkeiten und Beziehungen haben. Diese Verantwortlichkeiten können sich ändern.

Lösung

Die Verantwortlichkeiten werden aus dem Objekt (engl. Player) in Rollenklassen (engl. Role) verlagert. Konkrete Rollen erben von einer Rollensuperklasse.

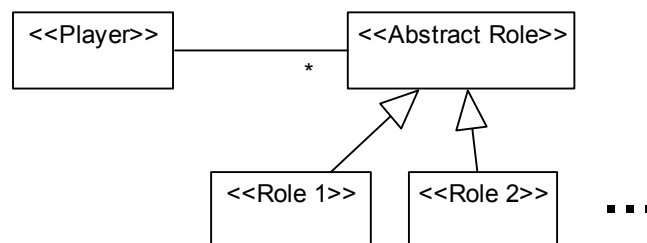


Abb. 1.22: Klassendiagramm des Musters Wechselnde Rollen

Mitarbeiter können in Unternehmen unterschiedliche Positionen einnehmen, z.B. Softwareentwickler, Projektleiter, Abteilungsleiter. Ein Mitarbeiter kann mehrere Rollen gleichzeitig einnehmen, z.B. kann ein Softwareentwickler oder ein Abteilungsleiter gleichzeitig Projektleiter sein. Die Mitarbeiter sind Player, ihre Positionen sind die Rollen (Abb. 1.23).

Beispiel

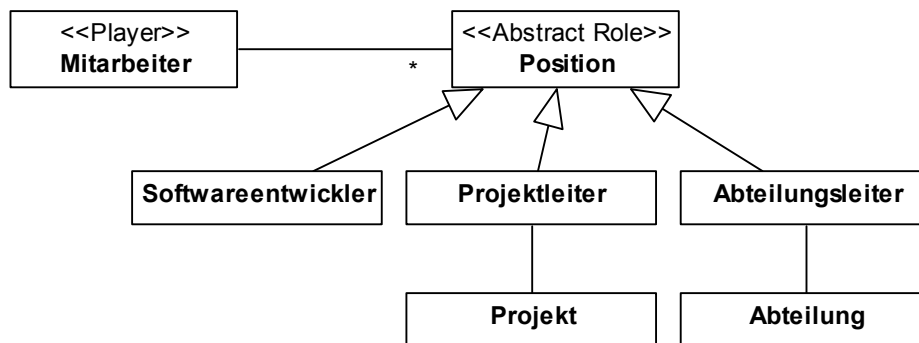


Abb. 1.23: Klassendiagramm des Beispiels für den Einsatz des Musters Wechselnde Rollen

Es wäre problematisch, auf Abstract Role zu verzichten und stattdessen die Rollen zu modellieren, indem sie von Player abgeleitet werden (Abb. 1.24). In diesem Fall wäre es nicht möglich, dass Mitarbeiter ihre Rolle wechseln oder mehrere Rollen annehmen, da sie in ihrer jeweiligen Rolle instanziiert werden.

Antimuster

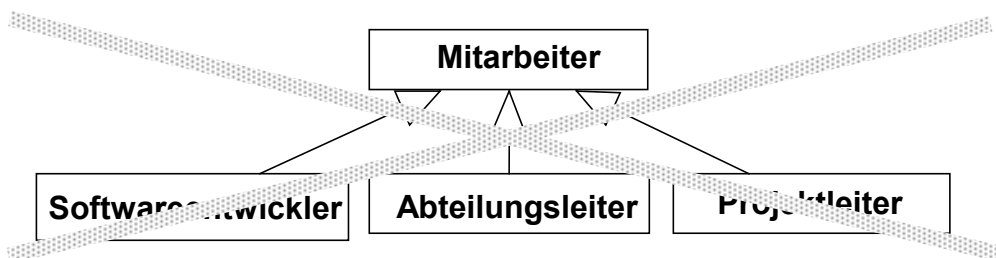


Abb. 1.24: Klassendiagramm des Antimusters zu Wechselnde Rollen anhand des Beispiels



Übungsaufgabe

- 1.10 Identifizieren Sie Player und Role und zeichnen Sie ein Klassendiagramm für folgendes Beispiel: Eine Person kann in einem Flugzeug als Passagier mitfliegen. Flugbegleiter sind ebenfalls Personen, die dann aber an Bord arbeiten. Personen, die als Flugbegleiter arbeiten, können manchmal als Passagier fliegen.

1.6.3 Allgemeine Hierarchie (General Hierarchy, Kompositum)

Problem in seinem Kontext

Objekte können hierarchisch geordnet sein. Das bedeutet, dass Objekte anderen Objekten über- oder untergeordnet sein können. Jedes Objekt kann maximal einem anderen Objekt untergeordnet sein. Manche Objekte können untergeordnete Objekte haben, andere nicht.

Lösung

Die Klassen werden von einer Superklasse „Node“ abgeleitet. Klassen, die andere Klassen referenzieren können, „Superior Nodes“, besitzen Referenzen auf „Nodes“. „Non Superior Nodes“ können keine anderen Klassen in der Hierarchie untergeordnet werden (Abb. 1.25).

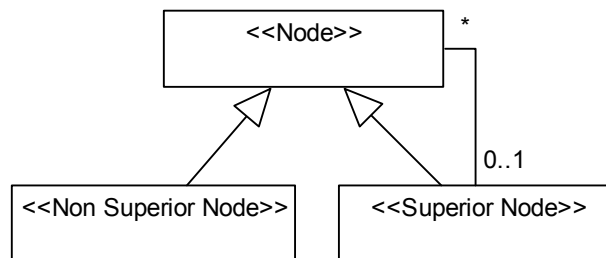


Abb. 1.25: Klassendiagramm des Musters Allgemeine Hierarchie

In einem Dateisystem können Verzeichnisse Dateien oder andere Verzeichnisse beinhalten. Verzeichnisse können also als „Superior Nodes“ und Files als „Non Superior Nodes“ modelliert werden (Abb. 1.26).

Beispiel

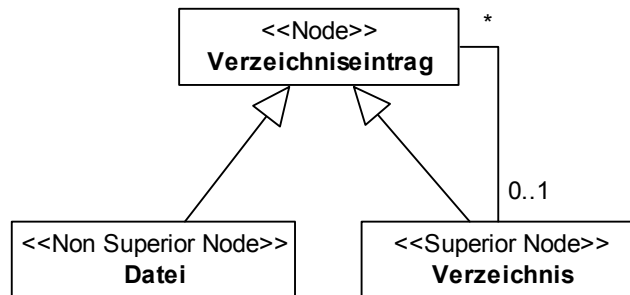


Abb. 1.26: Klassendiagramm des Beispiels Dateien in Verzeichnissen für das Muster Allgemeine Hierarchie

Alternativ zu diesem Muster wird manchmal darauf verzichtet, die Hierarchie explizit zu modellieren (Abb. 1.27). Die Hierarchie wird in diesem Fall nur durch eine einzige Assoziation modelliert. Der Vorteil dieser Lösung ist, dass sie besser verständlich ist. Gravierender Nachteil dieser Lösung ist allerdings, dass nicht modelliert wird, dass manche Nodes keine untergeordneten Nodes besitzen können. Diese Information könnte durch OCL-Ausdrücke modelliert werden, was aber unübersichtlicher ist, da es nicht graphisch modelliert ist.

Alternative

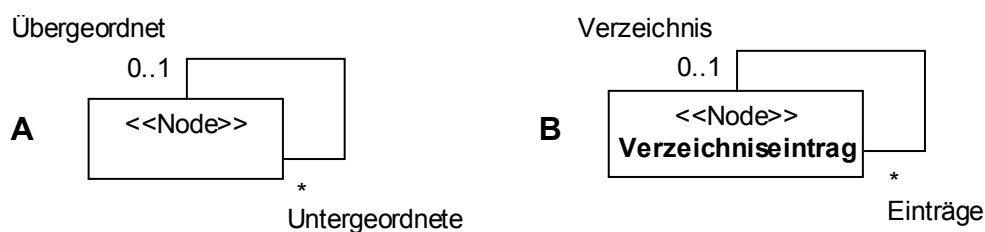


Abb. 1.27: Klassendiagramm einer Alternative für das Muster Allgemeine Hierarchie. A stellt die Alternative dar, B die Anwendung der Alternative am Beispiel.

Antimuster

Teilweise wird vorgeschlagen, die beschriebene Hierarchie durch Vererbungsbeziehungen zu modellieren (Abb. 1.28). Diese Modellierung stellt den Sachverhalt allerdings falsch dar. Die Vererbung beschreibt „kann-verwendet-werden-als“ oder auch „ist-ein“. Eine untergeordnete Klasse (z.B. das File `/etc/passwd`⁴) ist aber kein Verzeichnis (z.B. `/etc`) und kann auch nicht als solches verwendet werden. Beim Muster Allgemeine Hierarchie wird die Hierarchie von Objekten modelliert, nicht die Hierarchie von Klassen.

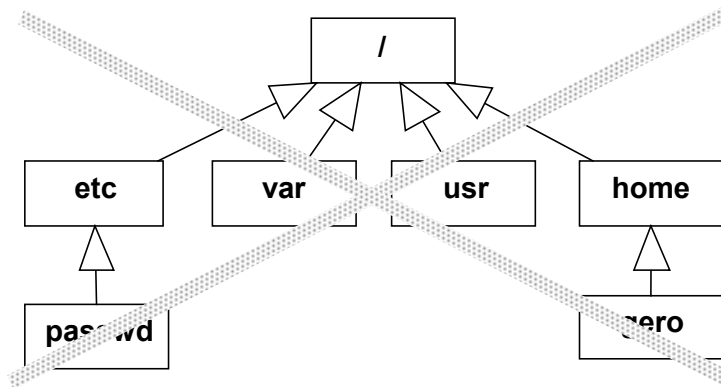


Abb. 1.28: Klassendiagramm des Beispiels zum Antimuster des Musters Allgemeine Hierarchie



Übungsaufgaben

- 1.11 Identifizieren Sie „Superior Node“ und „Non Superior Node“ und zeichnen Sie ein Klassendiagramm für folgendes Beispiel: Bei einer Softwareentwicklung werden Testfälle erstellt. Zur Übersicht werden diese in Test-Suiten organisiert. Test-Suiten können auch andere Test-Suiten enthalten.
- 1.12 Welches Analysemuster wurde im Beispiel Bibliothek (Abschnitt 1.4.2, Abb. 1.16) eingesetzt?

4 Die im Beispiel verwendeten Files und Verzeichnisse sind UNIX-artigen Betriebssystemen entnommen.

1.7 Dialogentwurf

Im Dialogentwurf wird die Schnittstelle zwischen Nutzern und einem System entworfen, die so genannte Benutzeroberfläche. Dabei wird geplant, welche Informationen an welcher Stelle angezeigt werden, welche Aktionen in welchem Zusammenhang ausgeführt werden können, wie die Abläufe für die Benutzer sein werden und wie die Oberfläche graphisch gestaltet wird. Eine Benutzeroberfläche kann sehr unterschiedlich sein, je nachdem, ob es sich um eine Desktopanwendung handelt, einen Webaufttritt oder einen Fahrkartenautomaten. Der Dialogentwurf ist wichtig für den Erfolg einer Software, da der Dialog das Gesicht der Anwendung ist und deswegen bei den meisten Anwendungen entscheidend für die Zufriedenheit der Endnutzer und den Kauf eines Produkts ist. Beim Entwurf der Benutzeroberfläche muss deswegen sichergestellt werden, dass Anwender das, was sie mit einer Softwarelösung machen wollen, auch tatsächlich tun können. Dabei müssen auch die technischen Randbedingungen und kommerziellen Interessen der Entwicklung berücksichtigt werden.

**Schnittstelle
zwischen Nutzer
und System**

In diesem Abschnitt werden Sie zuerst ein geeignetes Vorgehen für den Entwurf und anschließend die wichtigsten Grundzüge der Ergonomie kennen lernen. Möchten Sie mehr über die Gestaltung von Benutzeroberflächen lernen, empfehle ich Ihnen als Einstieg das interessante, instruktive und teilweise auch überraschende Buch von Alan Cooper [Coo07].

**Aufbau des
Abschnitts**

1.7.1 Vorgehen

Sicher haben Sie bereits die eine oder andere graphische Benutzeroberfläche oder Webseiten entworfen. Hat alles auf eine Seite gepasst und gab es nur wenige Elemente darzustellen, war das vermutlich recht einfach. Waren die Informationen und Aktionen aber auf mehrere Seiten und Dialoge zu verteilen, hat sich vermutlich die Frage gestellt, wie dies am besten zu tun ist. Erfahrene Interfacedesigner schlagen folgendes Vorgehen vor ([Gar02], [Tid05]): Zuerst muss entschieden werden, um welche prinzipielle Art der Anwendung es sich handelt. Anschließend sind die Inhalte und Verteilung der Inhalte zu überlegen, die so genannte Informationsarchitektur. Auf diesen Überlegungen aufbauend werden erst dann die äußere Struktur und das äußere Erscheinungsbild entworfen. Grundlage für diese Arbeiten sind die Anforderungen und die bisherigen Ergebnisse der Analyse.

Vorgehensweise

Informationsarchitektur	Die Informationsarchitektur einer Anwendung definiert, wie die Informationen und die Funktionalität einer Anwendung strukturiert sind, ohne bereits einen konkreten, graphischen Entwurf vor Augen zu haben. Sie definiert vorgezeichnete Wege für die Benutzer, also die Navigation. Dabei wird unterschieden zwischen Primärdialogen, die dem Benutzer zur Aufgabenerledigung zur Verfügung stehen, und Sekundärdialogen, in denen Hilfsarbeitsschritte durchgeführt werden. Bei den Bedienkonzepten unterscheidet man zwischen objektorientiertem und Anwendungsfall-orientiertem Bedienkonzept.
Objektorientiertes Bedienkonzept	Beim objektorientierten ⁵ Bedienkonzept werden in der Oberfläche vorhandene Objekte präsentiert und nach Auswahl eines Objekts Aktionen angeboten. In unserem Beispiel der Bibliothek würde dies z.B. bedeuten, dass zunächst ein Benutzer ausgewählt wird, und anschließend eine Aktion angeboten wird wie Adresse ändern oder Ausleihen einsehen. Beim Entwurf solcher Anwendungen können Sie sich am Domänenmodell orientieren. Als Faustregel wird für jede Domänenklasse ein Dialog zur Ansicht, Neuanlage und Änderung benötigt, und für jede 1-n-Assoziation ein Listen-Dialog, z.B. bei einem Benutzer die Liste aller seiner Ausleihen.
Anwendungsfall-orientiertes Bedienkonzept	Beim Anwendungsfall-orientierten Bedienkonzept gibt es je Anwendungsfall einen Dialog. Im unserem Beispiel gäbe es z.B. die Dialoge „Adresse von Benutzer ändern“ oder „Leihkonto einsehen“. In der Praxis sind meist beide Bedienkonzepte vorhanden, z.B. gibt es bei Microsoft Word eine Menüleiste, in der Aktionen ausgewählt werden können (Anwendungsfall-orientiert). Genauso ist es möglich, einen Teil des Texts zu markieren und dann den Text zu formatieren (objektorientiert).
Dialogablaufplan	Ein wichtiges Hilfsmittel ist der Dialogablaufplan. Leider gibt es für einen Dialogablaufplan keinen etablierten Standard. Am einfachsten ist es, die einzelnen Dialoge ähnlich wie Klassen durch Rechtecke darzustellen und Übergänge durch Pfeile (Abb. 1.29) [Amb04]. Zu jedem Dialog können in einem weiteren Dokument die angezeigten Daten, die Eingabefelder und mögliche Aktionen in tabellarischer Form dargestellt werden (s. Kurseinheit „Grundlagen der Softwaretechnik und Requirements Engineering“, Abschnitt 2.2.3).

5 Objektorientiert bezieht sich ausschließlich auf das Bedienkonzept und nicht auf die Art der Programmierung.

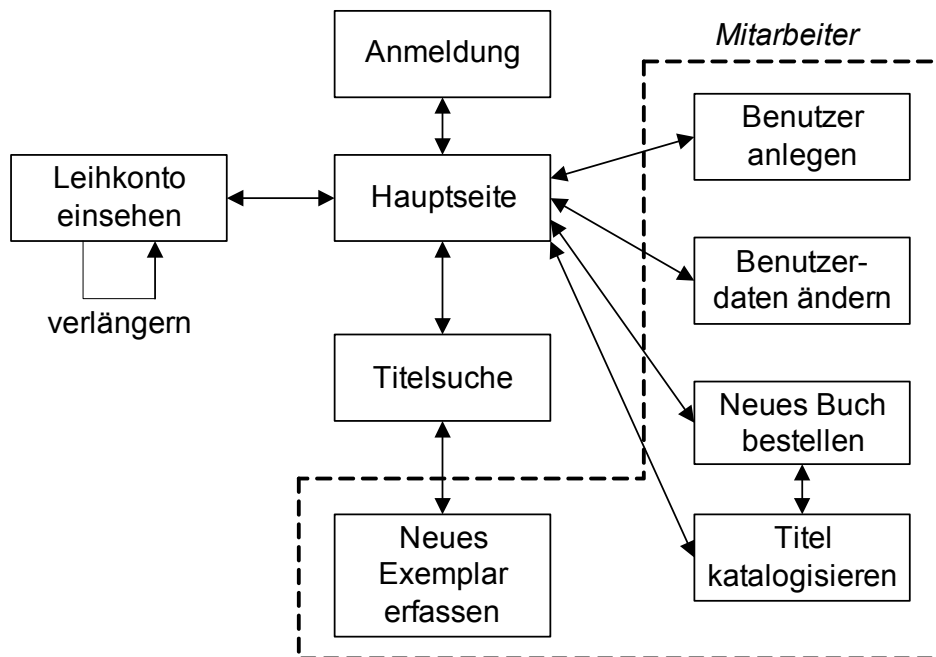


Abb. 1.29: Beispiel für ein Dialogablaufdiagramm

Neben der Informationsarchitektur, der logischen Gestalt, ist die äußere Gestalt der Anwendung zu entwerfen, die physische Form. Die grundlegende Entscheidung ist dabei, ob die Anwendung aus einem komplexen Fenster besteht, in dem sich Teile ändern („gekachelt“ eng. tiled), wie z.B. bei Microsoft Outlook, ob sich der Inhalt des Fensters beim Blättern komplett ändert, wie bei den so genannten „Wizards“ oder ob es mehrere Fenster geben soll (Abb. 1.30). Es gibt auch Mischlösungen, bei denen der Primärdialog z.B. in einem gekachelten Fenster implementiert ist und manche Sekundärdialoge wie die Auswahl von Files, in anderen Fenstern. Anschließend können dann der genaue Inhalt und das graphische Aussehen bestimmt werden.

Physische Form

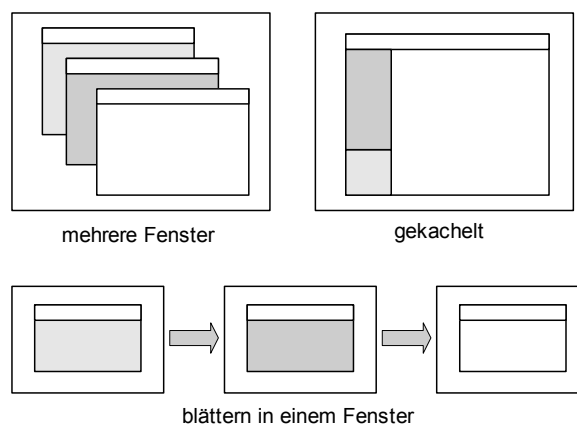


Abb. 1.30: Verschiedene physische Strukturen einer Anwendung (nach [Tid05])

Prototyping Der graphische Entwurf einer Oberfläche kann auf verschiedene Art und Weise erfolgen. Am einfachsten und schnellsten ist es, einen Prototyp mit Bleistift und Papier zu entwerfen (z.B. Abb. 3.13). Das hat den großen Vorteil, dass dies sehr schnell und einfach durchgeführt werden kann und Änderungen in einem Kundengespräch schnell und unkompliziert durchgeführt werden können. Prototypen können aber auch mit Computerprogrammen als Strichzeichnung (z.B. Abb. 3.7) oder auch mit vorgefertigten graphischen Elementen gezeichnet werden (z.B. Abb. 3.5), wenn dies zur Präsentation Erfolg versprechender erscheint. Sollen komplizierte Entwürfe einer Oberfläche samt möglichen Interaktionen demonstriert werden, können dazu auch Präsentationsprogramme wie PowerPoint oder Keynote oder auch Multi-Mediawerkzeuge wie Flash eingesetzt werden.

1.7.2 Ergonomie

Software-Ergonomie Mit dem Begriff Software-Ergonomie (engl. Usability) wird die Benutzbarkeit bzw. die Gebrauchstauglichkeit von Software bezeichnet. Die Ergonomie einer Software-Lösung ist wichtig, damit sie zum einen von den Benutzern akzeptiert wird, und zum anderen, damit die Benutzer auch längere Zeit mit einer Software-Lösung ohne körperliche und psychische Ermüdungserscheinungen arbeiten können. Für Software, die in Firmen kommerziell eingesetzt wird, ist dies in Deutschland sogar gesetzlich durch die Bildschirmarbeitsverordnung vorgeschrieben. In bestimmten Bereichen wie z.B. der Luftfahrt- oder Automobiltechnik kann die Ergonomie lebenswichtig sein, da eine Ermüdung der Nutzer fatale Folgen haben kann.

User Experience Seit einiger Zeit tritt bei Software für den Endverbrauchermarkt der Begriff User Experience (UX, engl. für Nutzungserlebnis) in den Vordergrund. Ziel der UX ist ein möglichst angenehmes Nutzungserlebnis. Dies ist nicht notwendigerweise mit einer guten Ergonomie verbunden. So kann z.B. ein Spiel kompliziert zu benutzen sein, trotzdem aber Spaß machen.

Einflüsse auf die Ergonomie Die Ergonomie einer Software wird durch die logische Aufteilung der Inhalte, die graphische Gestaltung und die Führung der Benutzer beeinflusst. Im vorangegangenen Abschnitt wurde bereits die Aufteilung der Inhalte behandelt. Graphische Gestaltung und Benutzerführung sind Inhalt dieses Abschnitts.

Graphische Gestaltung Die graphische Gestaltung ist für die Ergonomie einer Software nicht nur für das Erleben, sondern auch für die stressfreie Nutzung wichtig. Für die Gestaltung gibt es verschiedene Grundregeln. Die Benutzeroberfläche sollte übersichtlich gestaltet werden, indem genügend Platz gelassen, Zusammen-

gehöriges gruppiert und voneinander abgesetzt wird. Es sollten nicht zu viele verschiedene Schriftarten verwendet werden. Farben sollten sparsam und konsistent ohne zu großen Kontrast eingesetzt werden. Nach Möglichkeit sollten Farben und Schriftarten konfigurierbar sein, damit verschiedene Nutzer die Software an ihre Bedürfnisse anpassen können. Denken Sie daran, dass 9% aller Männer eine Rot-Grün-Sehschwäche haben.

Neben diesen rein graphischen Gesichtspunkten sollten gleiche Elemente einer Benutzeroberfläche gleich aussehen und nach einer durchgängigen Logik angeordnet sein, damit Benutzer nicht verwirrt werden. So sollte z.B. die „Weiter“-Schaltfläche immer gleich aussehen (z.B. Beschriftung „→“) und an derselben Stelle angeordnet sein (z.B. rechts unten). Solche Richtlinien werden am besten in einem Styleguide (engl. für Gestaltungsrichtlinie) zusammengefasst. Große Softwarehäuser wie z.B. Microsoft oder Unternehmen wie die Deutsche Telekom besitzen eigene Styleguides, damit die Benutzeroberflächen unternehmensweit gleich aussehen und sich gleich verhalten. Soll kein eigener Styleguide verwendet werden, bietet es sich an, den Styleguide der jeweiligen Plattform (z.B. Mac OS X⁶ oder GNOME⁷) zu verwenden, da potentielle Benutzer mit diesem Stil bereits vertraut sind und sich dadurch leichter zurechtfinden. Ein gute Zusammenstellung von Style Guides finden Sie auch in der Wikipedia unter dem Stichwort „Human Interface Guideline“.

Styleguide

Die für die Ergonomie von Bildschirmarbeitsplätzen wichtigste Norm ist die DIN EN ISO 9241 „Ergonomie der Mensch-System-Interaktion“⁸. Diese Norm besteht aus mehreren Teilen, die die Ergonomie von sowohl Software als auch Hardware definieren. Für die Gestaltung von Benutzeroberflächen ist besonders der Teil 110 „Grundsätze der Dialoggestaltung“ relevant. Die Norm enthält 7 Gestaltungsgrundsätze:

DIN EN ISO 9241

- **Aufgabenangemessenheit:** Ein Dialog ist den Aufgaben angemessen, wenn er die Benutzer unterstützt, ihre Arbeitsaufgabe effektiv und effizient zu erledigen.

Beispielsweise ist bei Fahrkartenautomaten der Bahn der Standortbahnhof des Automaten als Abfahrtbahnhof standardmäßig voreingestellt.

6 Apple Human Interface Guidelines:
Suchen Sie in Google nach „Apple Human Interface Guidelines“

7 <http://library.gnome.org/devel/hig-book/stable/>

8 DIN = Deutsche Industrienorm, EN = Europäische Norm, ISO = International Organization for Standardization. Es handelt sich also um eine verbindliche nationale und internationale Norm.

- **Selbstbeschreibungsfähigkeit:** Ein Dialog ist selbstbeschreibungsfähig, wenn für Benutzer zu jeder Zeit offensichtlich ist, in welchem Dialog, an welcher Stelle im Dialog sie sich befinden, welche Handlungen unternommen werden können und wie diese ausgeführt werden können.

Tooltips zeigen zum Beispiel Informationen zu Schaltflächen an, wenn die Maus über ihnen ruht.

- **Steuerbarkeit:** Ein Dialog ist steuerbar, wenn der Benutzer in der Lage ist, den Dialogablauf zu starten sowie seine Richtung und Geschwindigkeit zu beeinflussen, bis das Ziel erreicht ist.

Beispiele hierfür sind z.B. eine Rückgängigmachen-Funktion oder bei mehrseitigen Dialogen, dass zum vorangegangenen Fenster zurücknavigiert werden kann.

- **Erwartungskonformität:** Ein Dialog ist erwartungskonform, wenn er konsistent ist und den Merkmalen des Benutzers entspricht, z.B. seinen Kenntnissen aus dem Arbeitsgebiet, seiner Ausbildung und seiner Erfahrung sowie allgemein anerkannten Konventionen.

Dazu gehört z.B. die gleiche Verwendung von Funktionscodes und Funktionstasten in allen Dialogen.

- **Fehlertoleranz:** Ein Dialog ist fehlertolerant, wenn das beabsichtigte Arbeitsergebnis trotz erkennbar fehlerhafter Eingaben entweder mit keinem oder mit minimalem Korrekturaufwand seitens des Benutzers erreicht werden kann.

Dabei ist es wichtig, dass nicht nur eine Fehlermeldung ausgegeben wird, sondern dass auch der Benutzer darüber informiert wird, was genau an der Eingabe falsch ist.

- **Individualisierbarkeit:** Ein Dialog ist individualisierbar, wenn das Dialogsystem Anpassungen an die Erfordernisse der Arbeitsaufgabe sowie an die individuellen Fähigkeiten und Vorlieben des Benutzers zulässt.

Zu den Beispielen gehören dockbare Menüleisten oder die Möglichkeit, die Schriftart oder die Farben verändern zu können.

- **Lernförderlichkeit:** Ein Dialog ist lernförderlich, wenn er den Benutzer beim Erlernen des Dialogsystems unterstützt und anleitet.

Ein Beispiel für die Lernförderlichkeit ist z.B., dass in Menüs die Tastatur-Kürzel angegeben werden.

Unter dem Begriff Barrierefreiheit (engl. Accessibility) versteht man, dass Software so gestaltet ist, dass auch Menschen mit Behinderungen mit dieser Software arbeiten können. Dies ist besonders wichtig bei Software, die öffentlich genutzt wird, wie Fahrkartenautomaten oder Internetanwendungen. Aber auch im beruflichen oder privaten Bereich ist Barrierefreiheit wichtig, um behinderten Menschen die Möglichkeit zu geben, eine Software zu benutzen. Insbesondere die Gestaltung von Software für sehbehinderte Menschen wird inzwischen von den meisten Entwicklungsplattformen unterstützt. So gibt es z.B. für Java die Java Accessibility API (JAAPI). Dadurch wird der Einsatz von nicht-visuellen Werkzeugen wie die Sprachausgabe oder die Ausgabe in Brailleschrift ermöglicht. Spastiker haben meist Probleme mit der Bedienung von Mäusen. Für sie ist es wichtig, dass alle Elemente einer Oberfläche über die Tastatur erreichbar sind.

Barrierefreiheit

Übungsaufgaben

- 1.13 Was ist bei der Beschriftung von Schaltflächen in GNOME-Anwendungen zu beachten?
- 1.14 Was ist am folgenden Beispiel nicht bedienungsfreundlich? Gegen welchen Gestaltungsgrundsatz der DIN EN ISO 9241-110 wurde verstoßen?



1.8 Zusammenfassung

In der Analyse werden Modelle genutzt, um die Anforderungen besser zu verstehen und eine Lösung aus fachlicher Sicht zu entwickeln. Dazu werden Modelle des Umfelds und der Lösung auf formalisierte Art erstellt. Sie dienen als Grundlage von Modellen der Umsetzung, dem Entwurf. Modelle bestehen aus zwei Typen von Modellteilen: statischen Modellen, die die Teile und ihre Beziehungen modellieren, und dynamischen Modellen, die das Zusammenspiel der Teile beschreiben. In der objektorientierten Analyse mit UML werden zur statischen Modellierung Klassendiagramme eingesetzt. Dynamische Modelle beschreiben Zustände und Übergänge von Zuständen mit Zustandsdiagrammen und Abläufe von Aktivitäten mit Aktivitätsdiagrammen. Bei der Analyse kommen Analysemuster zum Einsatz. Muster beschreiben Lösungsskizzen verallgemeinerter Probleme. Sie helfen, indem sie bewährte Lösungen beschreiben und ein Kommunikationsmittel darstellen. Die wichtigsten Analysemuster sind Exemplartyp, Wechselnde Rollen und Allgemeine Hierarchie. Die Gestaltung der Benutzeroberfläche ist ein Bestandteil der Analyse. Dabei werden die logische Struktur, die Informationsarchitektur und die physische Gestalt entworfen. Überlegungen zur Ergonomie und zur Barrierefreiheit spielen hierbei eine wichtige Rolle.

2 Architektur

In Anforderungen und Analyse wird geklärt, was eine Softwarelösung leisten soll. Bevor Details der Lösung entworfen werden können, müssen grundlegende Entscheidungen über die Softwareentwicklung getroffen werden, also z.B. wie die Software prinzipiell aufgebaut ist oder ob eine Datenbank eingesetzt werden soll. Diese Entscheidungen sind Teil der Softwarearchitektur, die in diesem Kapitel dargestellt wird.

Nach dem Durcharbeiten dieses Kapitels sollen Sie

- verstehen, was Architektur ist und welchen Einflüssen sie unterliegt,
- die wichtigsten Architekturmuster kennen und einordnen können.

2.1 Softwarearchitektur

Die Softwarearchitektur beschreibt den grundlegenden Aufbau einer Softwarelösung. Typische Fragen, die eine Softwarearchitektur beantwortet, sind z.B.: Wird die Software nur auf einem Computer ausgeführt oder auf mehreren? Wie ist die Software verteilt, wenn sie auf mehreren Computern ausgeführt wird? Wie kommunizieren die Komponenten? Kommt eine Datenbank zum Einsatz oder werden die Informationen in Dateien geschrieben? Wie sieht der Aufbau der Software in der Gesamtsicht aus? Welche Programmiersprache und welche Frameworks kommen zum Einsatz? Wie sehen die Strategien bei Ausfall von Komponenten aus? Die sorgfältige Beantwortung dieser Fragen ist wichtig, da diese Entscheidungen Auswirkungen auf die gesamte weitere Entwicklung haben und Änderungen an diesen Punkten später großen Aufwand verursachen.

Softwarearchitektur beschreibt den grundlegenden Aufbau

Die Softwarearchitektur sollte nicht erst erstellt werden, wenn Anforderungen und Analyse durchgeführt worden sind. Softwarearchitekten sollten bereits bei der Klärung der Geschäftsanforderungen beteiligt sein. Zum einen liefern Architekten Aufwandsschätzungen, die die Grundlage für Überlegungen zur Wirtschaftlichkeit darstellen. Zum anderen können Softwarearchitekten die Machbarkeit von Ideen überprüfen. Dabei gibt es eine Rückkopplungsschleife: Ideen zum Produkt führen zu Fragen und Aussagen zur Softwarearchitektur, was wiederum die Ideen zum Produkt beeinflusst. Erst mit dieser Rückkopplung entstehen realistische Anforderungen und Softwarearchitekturen.

Architektur am Anfang

**Grundlage
Anforderungen und
Randbedingungen**

In der Regel gibt es immer mehrere Möglichkeiten, wie eine Softwarelösung aussehen könnte. Die Grundlage für die Auswahl einer Softwarearchitektur für eine Aufgabe sind die Anforderungen und Randbedingungen (Kurseinheit „Grundlagen der Softwaretechnik und Requirements Engineering“). Diese Informationen wurden als Anforderungen gesammelt. Die Architektur muss so ausgestaltet werden, dass sie die Erfüllung der Anforderungen unter Einhaltung der Randbedingungen ermöglicht. In fast allen Fällen bestimmen nicht die funktionalen sondern die nicht-funktionalen Anforderungen die Systemarchitektur. Bei der Entwicklung einer Softwarearchitektur wird also zunächst festgestellt, welche Randbedingungen welche Einschränkungen oder Vorgaben machen. Anschließend wird in der Reihenfolge ihrer Priorität dafür gesorgt, dass die Architektur die nicht-funktionalen Anforderungen erfüllt.

**Vorgefertigte
Lösungen**

Für die Entwicklung einer Architektur gibt es ganz ähnlich zu Analysemustern viele Standardlösungen, die helfen, eine Architektur schneller und einfacher zu entwickeln und anschließend umzusetzen. In den folgenden Abschnitten werden Sie die verschiedenen Typen von Standardlösungen kennen lernen.

Taktiken

Taktiken sind Standardlösungen, wie eine Architektur auf grundlegende Art und Weise nicht-funktionale Anforderungen erfüllbar machen kann. Einen Katalog von Taktiken finden Sie im Standardwerk „Software Architecture in Practice“ [Bas03]. Zum Beispiel werden dort Taktiken angegeben, wie hochverfügbare Systeme gebaut werden können. Beispielsweise ist eine viel verwendete Taktik zur „Verhinderung von Fehlfunktion“ die „Aktive Redundanz“, bei der redundante Systeme alle Anfragen gleichzeitig bearbeiten und nur die Antwort des schnelleren Systems verwendet wird. Diese Taktik wird z.B. bei hochverfügbaren Datenbanken eingesetzt.

**Referenz-
architekturen,
Frameworks**

Architekturen entstehen nicht auf der grünen Wiese. Für die meisten Typen von Problemen gibt es bereits mehr oder weniger ausgereifte Lösungen, die direkt oder als Vorlage verwendet werden können. Referenzarchitekturen sind Referenzmodelle, die bei der Entwicklung einer Architektur hinzugezogen werden können. Insbesondere für die Architektur von Anwendungen für das Internet gibt es mehrere Referenzarchitekturen. Die Anwendung von Referenzarchitekturen wird häufig durch vorgefertigte Frameworks unterstützt. Frameworks (engl. für Rahmenstruktur, Fachwerk) sind vorgefertigte Komponenten, die an genau definierten Punkten erweitert werden müssen, damit sie eingesetzt werden können. Das Framework verwendet anschließend diese selbst entwickelten Teile. Bekannte Frameworks sind z.B. Struts2 für Webapplikationen oder Spring für die Verteilung von Komponenten.

Architekturen werden in Sichten dokumentiert. Die Kontext-Sicht wurde bereits bei der Ermittlung der Anforderungen in der Kurseinheit „Grundlagen der Softwaretechnik und Requirements Engineering“ im Abschnitt „Kontextdiagramme“ erklärt. Weiterhin gibt es genau wie bei Analyse und Entwurf ein statisches Modell, die Baustein-Sicht, und ein dynamisches Modell, die Laufzeitsicht. Zusätzlich wird meistens noch die Verteilungssicht benötigt, die beschreibt, wie die Komponenten auf verschiedene Systeme verteilt sind. Mehr zu den Sichten erfahren Sie z.B. im Lehrbuch von G. Starke [Sta08].

Dokumentation in Sichten

Die Architektur soll nicht-funktionale Anforderungen erfüllen und beschreibt grundlegende Strukturen. Es wäre also ein enormes Problem, sollte sich erst nach Fertigstellung der Software herausstellen, dass die Architektur ungeeignet ist. Um dieses Risiko zu vermeiden, empfiehlt es sich, die Architektur mit einem zumindest sehr einfachen Prototyp zu erproben. Dies hat die nützlichen Nebeneffekte, dass im Sinne der Qualitätssicherung der Architekturansatz tatsächlich ausprobiert worden ist, und dass im Sinne der Dokumentation die Softwareentwickler einen Prototyp als Musterimplementierung zur Verfügung haben.

Prove it with code

Im Folgenden werden Sie zunächst die wichtigsten Architekturmuster kennen lernen. Am Beispiel des Bibliothekssystems aus Abschnitt 1.4.1 werden Sie anschließend exemplarisch einen Einblick in das Vorgehen bei der Erstellung einer Architektur erhalten.

Weiteres Vorgehen



Übungsaufgaben

- 2.1 Wann werden in einem Projekt Softwarearchitekten hinzugezogen und warum?
- 2.2 Welche Hilfsmittel stehen bei der Erstellung einer Softwarearchitektur zur Verfügung?
- 2.3 Welche Sichten werden zur Dokumentation einer Softwarearchitektur typischerweise verwendet?

2.2 Architekturmuster

Architekturmuster sind wie Analysemuster (s. Abschnitt 1.6) Lösungsskizzen verallgemeinerter Architekturprobleme. Alle in Abschnitt 1.6 genannten Vorteile der Verwendung von Mustern gelten auch für Architekturmuster. Im folgenden Abschnitt lernen Sie die wichtigsten Architekturmuster kennen, die in sehr vielen Anwendungen verwendet werden. Weitere Architekturmuster finden Sie in der Literatur (z.B. [Let05], [Sta08]). Einen Überblick finden Sie auf der Webseite der GI-Arbeitsgruppe „Architekturmuster“ (<http://www.architekturmuster.de>). Der Aufbau der Beschreibung folgt dem Aufbau der Beschreibung von Analysemustern. Auch hier können Sie je nach Vorliebe zuerst das Beispiel lesen und dann die abstrakten Beschreibungen oder umgekehrt.

2.2.1 Client-Server

Problem in seinem Kontext	Mehrere Benutzer arbeiten gleichzeitig und verteilt an mehreren Arbeitsplätzen. Sie greifen auf dieselben Daten zu, tauschen Daten aus oder synchronisieren ihre Arbeits- oder Handlungsabläufe. Eine direkte Kommunikation zwischen den Arbeitsplätzen ist z.B. aus Sicherheitsgründen nicht erwünscht oder aufgrund der Anzahl der Nutzer nicht möglich.
Lösung	Die Arbeitsplätze (Clients, engl. für Kunde, Klient) greifen auf ein zentrales System zu (Server, engl. für Diener, Zusteller) (Abb. 2.1). Im einfachsten Fall ist der Server eine Datenbank und die Kommunikation zwischen den Clients erfolgt durch die gemeinsamen Daten („Fat Client“). Als anderes Extrem besitzen die Clients keinerlei Logik, sondern sind nur für die Anzeige der Daten verantwortlich („Thin Client“). Diese Lösung wird dann verwendet, wenn sich die Logik häufig ändert, die Client-Geräte nur technischen Minimalanforderungen genügen oder den Client-Geräten nicht vertraut werden kann.
Beispiel	Webanwendungen sind ein weitverbreitetes Beispiel für diese Architektur. Die Server sind die Webserver, die Browser können als Thin Clients betrachtet werden. Die Logik muss von den Servern und nicht von den Browsern überprüft werden, da die Browser und der Datenverkehr leicht manipuliert werden können.

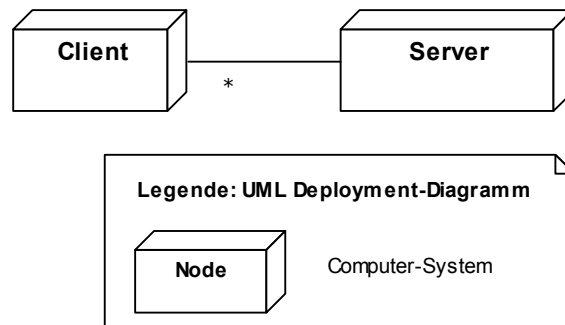


Abb. 2.1: Skizze des Architekturmusters Client-Server

Steht bei einer Anwendung die Kommunikation zwischen einzelnen Clients im Vordergrund, z.B. bei Internet-Telefonie oder Spielen, können alternativ Peer-to-Peer-Lösungen eingesetzt werden. In diesem Fall stellt ein Server den Kontakt zwischen den Peers her, die danach direkt miteinander kommunizieren. Peer-to-Peer-Lösungen besitzen aber alle oben genannten Probleme. Theoretisch gibt es noch die Möglichkeit einer reinen Serverlösung. In diesem Fall gibt es nur ein einziges System, das die Benutzer über ein Terminal nutzen. Diese Lösung war besonders bei Mainframes verbreitet, hat aber so starke Einschränkungen in Bezug auf die Nutzerfreundlichkeit, dass sie in der Praxis nur noch selten verwendet wird.

Alternativen

2.2.2 Mehrschichtenmodell

Eine Anwendung besteht aus vielen Teilen, die unterschiedliche Verantwortlichkeiten besitzen. Es ist schwer, die Übersicht zu behalten. Wenn jedes Teil jedes andere benutzt, wird es schwierig, Teile zu ändern. Manchmal sollen die Teile auch auf mehreren Systemen ausgeführt werden.

Problem in seinem Kontext

Es werden Schichten (engl. Layers oder Tiers) gebildet. Innerhalb einer Schicht befinden sich Teile mit ähnlicher Verantwortlichkeit. Jede Schicht benutzt nur Dienste von darunterliegenden Schichten, nie in umgekehrter Richtung (Abb. 2.2). Die Dienste jeder Schicht können als Abstraktion ihrer Verantwortlichkeiten betrachtet werden. Durch die Schichtenbildung können einzelne Schichten problemlos ausgetauscht werden.

Lösung

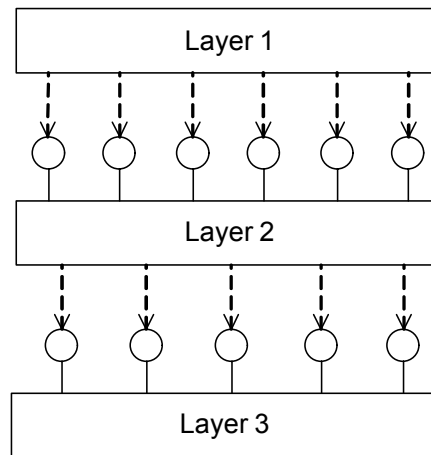


Abb. 2.2: Skizze der Aufteilung eines Systems in Schichten. Eine Schicht verwendet nur die Dienste der direkt darunterliegenden Schicht.

Beispiel

Große Geschäftsanwendungen, so genannte Enterprise-Systeme, und andere anspruchsvolle Anwendungen z.B. im Internet sind heute durchgehend in mehreren Schichten aufgebaut. Zum einen gibt es eine so genannte vertikale Schichtenbildung als Abstraktion von der Hardware („Layers“). Typische vertikale Schichten sind Hardware, Betriebssystem, Virtuelle Maschine (Java oder C#) und Anwendung. Orthogonal dazu gibt es eine so genannte horizontale Schichtenbildung von Schnittstellen zu den Datenhaltungssystemen („Tiers“)⁹. Typische horizontale Schichten sind Ressource (z.B. Datenbanken), Integration (z.B. Einbindung der Datenbank), Geschäftslogik und Client. Die Bildung dieser Schichten wird durch Technologien wie Java Enterprise Edition (JEE) oder Microsoft .NET unterstützt und vereinfacht. Schichtenbildung ist aber auch bei der Entwicklung kleinerer Systeme sinnvoll (s. Abschnitt 3.7).

	Client	Geschäftslogik	Integration	Ressource
Applikation	HTML	EJB	JPA	Mysql
Plattform	Webbrowser	Applikationsserver	Applikationsserver	keine

Abb. 2.3: Beispiel für eine Schichtenbildung. Horizontal wurden Layers, vertikal Tiers gebildet

⁹ Zu den Begriffen „Layer“ und „Tier“ gibt es in der Literatur unterschiedliche Verwendungen. Zum Teil wird definiert, Tiers seien physische Schichten und Layers logische Schichten. Alternativ werden teilweise zur größeren Klarheit logische und physische Tiers unterschieden, und Layer als Überbegriff benutzt. Die hier verwendete Terminologie folgt den viel verwendeten Core J2EE Pattern.



Übungsaufgaben

- 2.4 Ist es bei Verwendung des Client-Server-Musters sinnvoll, dass Clients unter Umgehung des Servers direkt miteinander kommunizieren? Was ist daran schwierig?
- 2.5 Welche Art von Schichten können beim Mehrschichtenmodell unterschieden werden?

2.3 Beispiel Bibliothek

In diesem Abschnitt werden Sie beispielhaft den Weg der Entwicklung einer Architektur am Beispiel des Bibliothekssystems aus Abschnitt 1.4.1 kennen lernen. Dazu wird zunächst eine Systemidee beschrieben und dann die Auswirkungen der verschiedenen nicht-funktionalen Anforderungen. Aus Platzgründen wird jedoch die Architektur nur skizziert.

Das System soll den Katalog der Bücher, die Daten der Benutzer und deren Entleihungen verwalten. Aus dieser Systemidee ist bereits klar, dass eine Datenbank zum Einsatz kommen muss, da große Datenmengen in strukturierter, durchsuchbarer Form gespeichert werden müssen.

Systemidee

Die Anforderung nach Mehrbenutzerbetrieb stellt die Frage, ob eine Client-Server- oder Peer-to-Peer-Lösung gewählt wird (Architekturmuster Client-Server, Abschnitt 2.2.1). Da für eine Peer-to-Peer-Lösung zur Suche im Katalog alle Daten auf jeden Client kopiert werden müssten, kommt hier nur eine Client-Server-Lösung in Frage.

**N1.
Mehrbenutzer-
betrieb**

Die Architekten fragen nach, ob denn Benutzer auch über das Internet auf die Bibliothek zugreifen sollen. Daraufhin wird die nicht-funktionale Anforderung N6 formuliert: Die Benutzer sollen auch über das Internet auf den Katalog und ihre Ausleihdaten zugreifen können. Aus diesem Grund wird entschieden, dass alle Benutzerschnittstellen als Webanwendung umgesetzt werden, da eine gleichzeitige Entwicklung mehrerer Typen von Oberflächen nicht ökonomisch ist. Mit neuer Webtechnologie stellt dies kein Widerspruch zur Anforderung nach Benutzerfreundlichkeit (N4) dar.

**N6.
Webanwendung**

Aufgrund des Angebots im Internet nehmen die Architekten an, dass das System sicher gestaltet werden muss. Insbesondere bedeutet dies die Forderung nach ausgereifter Authentifizierung. Die verschiedenen Nutzerrollen

**N7.
Integrität**

(Anwender aus dem Internet, Benutzer, Mitarbeiter) erfordern ein Konzept zur Autorisierung.

**N2.
Robustheit**

Die Daten der Bibliothek dürfen nicht verloren gehen. Dies bedeutet für die Architektur neben der Forderung nach Integrität, dass ein Backup der Daten möglich sein muss.

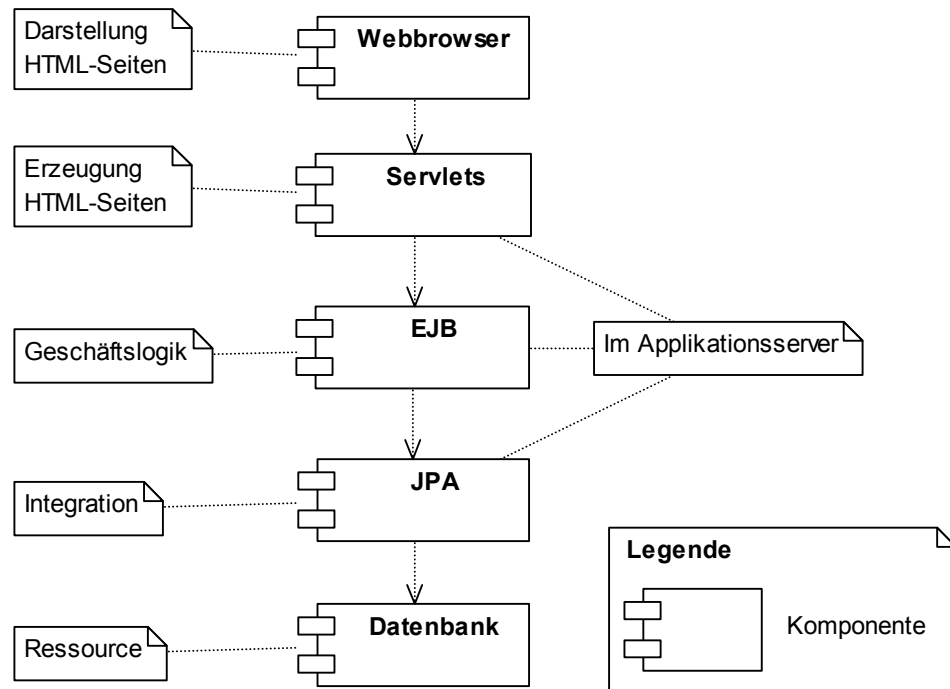


Abb. 2.4: Top-level-Bausteinsicht des Bibliothekssystems auf Grundlage der JEE in Form eines UML- Komponentendiagramms

Skalierbarkeit

Aus dem Mengengerüst kann entnommen werden, dass zum einen die Anzahl der ausleihbaren Gegenstände wächst und zum anderen mit einem Anstieg der Benutzerzahlen und damit auch der Anzahl der Ausleihen zu rechnen ist. Bei Angeboten für das Internet (N6) ist die Anzahl der Nutzer von vornherein nur mit großer Unsicherheit schätzbar. Bereits jetzt muss ca. eine Ausleihe pro Minute bearbeitet werden. Die Anzahl der Suchanfragen ist unbekannt und vermutlich größer. Die Antwortzeit soll aber nicht zu groß werden (N5). Dies bedeutet, dass das System skalierbar sein muss, dass also bei größeren Anforderungen das System leicht mit zusätzlicher Hardware erweitert werden kann. Das Architektenteam hat gute Erfahrung mit der Java Enterprise Edition (JEE) und weiß daher, dass diese Architektur für skalierbare Webapplikationen gut geeignet ist und auch Mechanismen für Autorisierung und Authentifizierung mitbringt. (Wahl einer Referenzarchitektur, s.

Abschnitt 2.1). Es wird also eine mehrschichtige Architektur (Multi-Tier) mit relationaler Datenbank (Ressource), Java Persistence API JPA (Integration der relationalen Datenbank), Enterprise Java Beans EJB (Geschäftslogik), Servlets (Generierung der Webseiten) und Standardinternetbrowser (Darstellung der Webseiten) gewählt (Abb. 2.4). EJBs mit JPA und Servlets werden in so genannten Applikationsservern ausgeführt. Ein Lastverteiler verteilt die Anfragen auf die Applikationsserver.

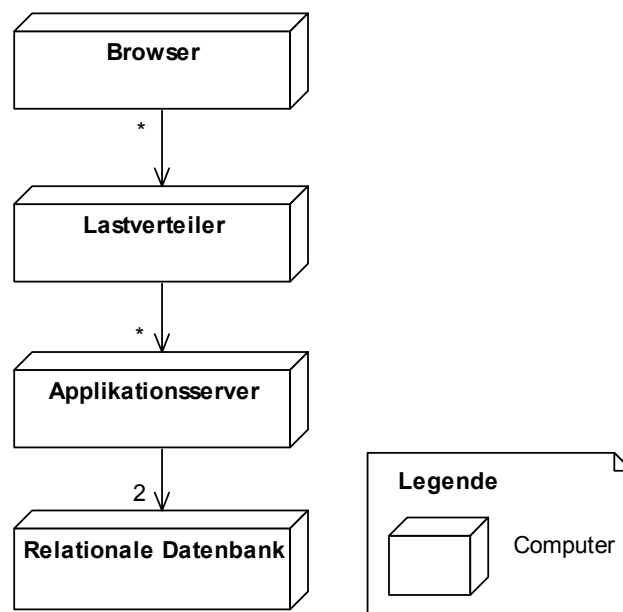


Abb. 2.5: Verteilungssicht des Bibliothekssystems in Form eines UML- Deploymentdiagramms

Da das System für den Betrieb der Bibliothek sehr wichtig ist, müssen Ausfälle nach Möglichkeit vermieden werden. Dies bedeutet, dass alle Komponenten auf Hardwareseite redundant sein müssen. Dazu prüfen die Architekten, welche Taktiken geeignet sind (Abschnitt 2.1). Da im Betrieb keine Computerexperten vor Ort sind, die notfalls schnell Server austauschen könnten, wird für die Datenbank die Taktik „Aktive Redundanz“ gewählt (Abschnitt 2.1). Bereits aus Gründen der Skalierbarkeit können Applikationsserver mehrfach vorhanden sein, was gleichzeitig auch die Verfügbarkeit erhöht (Abb. 2.5).

N3.
Verfügbarkeit

Sicherlich haben Sie noch viele Fragen zur konkreten Ausgestaltung dieser Architektur, da sie nur ein verkürztes Beispiel darstellt. Es wird aber klar, dass die grundlegende Architektur praktisch ausschließlich von den nicht-funktionalen Anforderungen und den Randbedingungen diktiert wird. Aus diesem Grund konnten auch Architekturmuster, Taktiken und Referenzarchitekturen eingesetzt werden.

Resümee



Übungsaufgaben

- 2.6 Welche Architekturmuster, Taktiken und Referenzarchitekturen wurden im Beispiel verwendet?
- 2.7 Welche Anforderungen ergaben sich bei der Entwicklung der Architektur?

2.4 Zusammenfassung

Softwarearchitektur beschreibt den grundlegenden Aufbau einer Software. Wegen der grundsätzlichen Bedeutung sollte die Architektur der Software bei einer Softwareentwicklung von Anfang an berücksichtigt werden. Grundlage für die Entwicklung einer Architektur sind Anforderungen und Randbedingungen, wobei die nicht-funktionalen Anforderungen und die Randbedingungen den entscheidenden Einfluss haben. Hilfsmittel zur Entwicklung einer Architektur sind Taktiken, Muster und Referenzarchitekturen. Wichtige Muster sind z.B. Client-Server und Schichtenbildung. Architekturen werden in Sichten dokumentiert.

3 Objektorientierter Entwurf

In diesem Kapitel werden Sie lernen, wie auf der Grundlage der Analyse und der Softwarearchitektur ein Softwaresystem mit objektorientierten Mitteln entworfen wird. Sie werden lernen, verschiedene Entwurfsmuster anzuwenden. An einem längeren Beispiel werden Sie den Entwurfsprozess modellhaft kennen lernen.

Nach dem Durcharbeiten dieses Kapitels sollen Sie

- verstehen, wie und wozu Modelle im Entwurf verwendet werden,
- den Zusammenhang zwischen Modellen der Analyse und des Entwurfs verstehen,
- Klassendiagramme des Entwurfs auf systematische Art und Weise erstellen können,
- bei der Erstellung von Entwürfen Entwurfsmuster verwenden können,
- die Qualität von Entwürfen beurteilen und systematisch verbessern können.

3.1 Von der Analyse zum Entwurf

Im Entwurf beschreiben Softwareentwickler, aus welchen Klassen, Methoden oder Dateien das zu entwerfende System bestehen soll, wie diese Bauteile aussehen sollen, in welcher Beziehung sie zueinander stehen und wie sie zusammenarbeiten. Beim objektorientierten Entwurf bedeutet dies festzulegen, welche Klassen es gibt, deren Attribute und Methoden zu spezifizieren, die Assoziationen und Vererbungsbeziehungen festzulegen und das Zusammenspiel der Instanzen zu beschreiben.

Entwurf

Der Entwurf geschieht auf Grundlage der Anforderungen, der Ergebnisse der Analyse und innerhalb der gewählten Architektur. Aus den Anforderungen sind vor allem die funktionalen Anforderungen für den Entwurf wichtig — die nicht-funktionalen Anforderungen sind bereits in der Architektur berücksichtigt worden. Die Architektur liefert die Information, welche Bauteile der Software prinzipiell benötigt werden. Im Entwurf werden dann die konkreten Bauteile innerhalb der Architektur anhand der funktionalen Anforderungen entwickelt. Genauso grundlegend sind die Ergebnisse der Analyse: Das Domänenmodell ist Grundlage für Datenhaltung in der Applikation und Datenbankschemas. Die Definitionen der Schnittstellen, insbesondere der

**Basis
Anforderungen,
Analyse und
Architektur**

GUI, bilden die Basis für den Entwurf der technischen Umsetzung der Schnittstelle.

Vor- und Nachteile

Vielleicht fragen Sie sich, welchen Vorteil es für Sie hat, zunächst zu entwerfen und nicht auf Grundlage von Anforderungen, Analyse und Architektur nun endlich loszuprogrammieren. Könnte es nicht die Arbeit beschleunigen, darauf zu verzichten? Erfahrungsgemäß sind nur bei gut bekannter Technologie und einfachen Problemen manche sehr erfahrenen Softwareentwickler in der Lage, die Software im Kopf zu entwerfen. Aber auch sie entwerfen – im Kopf. In allen anderen Fällen ist es effektiver, mit Bleistift und Papier wenigstens einen groben Entwurf zu skizzieren, da die Probleme eines Entwurfs meist erst beim Modellieren klar werden. Probleme sind im Entwurf viel einfacher als in der halbfertigen Software zu beseitigen. Sind mehrere Menschen zusammen an einer Aufgabe beteiligt, ist ein Entwurf wichtig, um eindeutige Absprachen zu ermöglichen. Sie sollten allerdings bei größeren und komplizierteren Problemen vermeiden, den Entwurf im ersten Anlauf zu perfekt machen zu wollen. Aller Erfahrung nach gelingt dies nur selten und nur sehr erfahrenen Softwareentwicklern unter günstigen Umständen. In vielen Fällen ist es günstiger, iterativ zu arbeiten, also zunächst Teile zu entwerfen, diese umzusetzen und anschließend mit den gemachten Erfahrungen die Entwürfe anzupassen (s. Abschnitt 3.4).



Übungsaufgabe

3.1 Welche Vorteile hat es zu entwerfen? Welche Gefahren drohen?

3.2 Klassen im Entwurf**Klassen im Entwurf**

Wie in der objektorientierten Analyse nutzen Softwareentwickler auch im objektorientierten Entwurf Klassen und Objekte. Als großer Unterschied zur Analyse beschreiben Klassen des Entwurfs nicht fachliche Zusammenhänge, sondern Klassen, die in der gewählten objektorientierten Programmiersprache umgesetzt werden. Die Klassen des Entwurfs besitzen entsprechend technische Verantwortlichkeiten, z.B.

- Datenhaltung („Entity“)
- Durchführung von Berechnungen
- Überprüfung von Regeln
- Interaktion mit dem Nutzer („User Interface“)

- Ablaufsteuerung („Control“)
- Schnittstellen zu anderen Systemen („Boundary“)
- Services für andere Klassen („Utility“), wie z.B. die Berechnung mathematischer Funktionen (z.B. Sinus, Kosinus) oder die Bereitstellung von Konstanten (z.B. Pi).

Wie in der Analyse gilt besonders auch im Entwurf, dass jede Klasse nach Möglichkeit nur eine, maximal zwei Verantwortlichkeiten besitzen soll, da dadurch die Klassen leichter verständlich und änderbar werden. Klassen mit ähnlichen Verantwortlichkeiten werden gemeinsam in Schichten oder Paketen verteilt (s. Abschnitt 2.2.2). Durch dieses Vorgehen entstehen viele Klassen, die wenig Funktionalität beinhalten. Dadurch ist es einfacher sie zu verstehen, als wenn wenige, unübersichtliche Klassen entworfen werden würden. Die Übersicht über die Vielzahl der Klassen wird durch einen sauberen Entwurf gewährleistet.

**Eine
Verantwortlichkeit
pro Klasse**

In den folgenden Abschnitten werden nacheinander die Modelle des Entwurfs, Vorgehensweisen beim Entwurf und Entwurfsmuster erklärt. Abschließend werden diese Konzepte an einem etwas längeren Beispiel demonstriert. Je nachdem, wie Sie Stoff besser verstehen, können Sie dem Text folgen und zunächst die abstrakte Darstellung und dann das Beispiel lesen. Alternativ können Sie auch zuerst das Beispiel (Abschnitt 3.6) lesen und sich dann anhand des Beispiels den Inhalt der vorangegangenen Abschnitte (3.3 – 3.5) klarmachen. Sie können aber auch erst mit den Entwurfsmustern beginnen (Abschnitt 3.5) und dann die anderen Abschnitte lesen.

Lesereihenfolge

3.3 Strukturen und Verhalten

Wie bei der Analyse bestehen auch Entwürfe aus statischen und dynamischen Modellen. Zur Beschreibung statischer Modelle kommen Paket- und Klassendiagramme zum Einsatz. Paketdiagramme, die auch zur Dokumentation der Architektur eingesetzt werden können, werden Sie in der Kursinheit „Systemmodellierung“ kennen lernen. In dynamischen Modellen werden meistens Zustandsdiagramme, die Sie bereits aus der Analyse kennen (s. Abschnitt 1.5), und Sequenzdiagramme eingesetzt, die in diesem Abschnitt eingeführt werden. Prinzipiell können auch Aktivitätsdiagramme genutzt werden, allerdings sind nach Meinung vieler Praktiker Sequenzdiagramme im Entwurf sinnvoller.

Klassendiagramme Klassendiagramme werden genauso wie in der Analyse eingesetzt. Im Entwurf werden allerdings üblicherweise auch Sichtbarkeiten (public/private) und die Typen von Attributen sowie von Argumenten und Rückgabewerten von Methoden spezifiziert. Um Diagramme leichter lesbar zu machen, empfiehlt es sich, Gerüstmethoden wie Zugriffsmethoden von Attributen (get/set) und Konstruktoren wegzulassen.

Sequenzdiagramme UML-Sequenzdiagramme können Sie im Entwurf zur Modellierung der Dynamik dann einsetzen, wenn Sie den zeitlichen Ablauf der Botschaften zwischen Instanzen modellieren wollen. Abb. 3.1 zeigt ein Beispiel eines Sequenzdiagramms. In Sequenzdiagrammen werden Instanzen in der gleichen Form wie in Instanzdiagrammen dargestellt (Abschnitt 1.2). Sie werden im Diagramm oben angeordnet. Die Zeitachse ist nach unten gerichtet. Jede Instanz besitzt eine Lebenslinie. Pfeile zwischen den Lebenslinien beschreiben Nachrichten, die eine Instanz an eine andere sendet. Aktivierungsbalken stellen die Zeit der Bearbeitung einer einkommenden Nachricht dar. Rechteckige Kästen, so genannte Fragmente, bestimmen Verzweigungen oder Schleifen. Die Bezeichnung definiert, was ein Fragment bedeutet, z.B. „loop“ für Schleife oder „alt“ für Verzweigung. Mehr Details zur Syntax von Sequenzdiagrammen werden Sie in der Kurseinheit „Systemmodellierung“ kennen lernen.

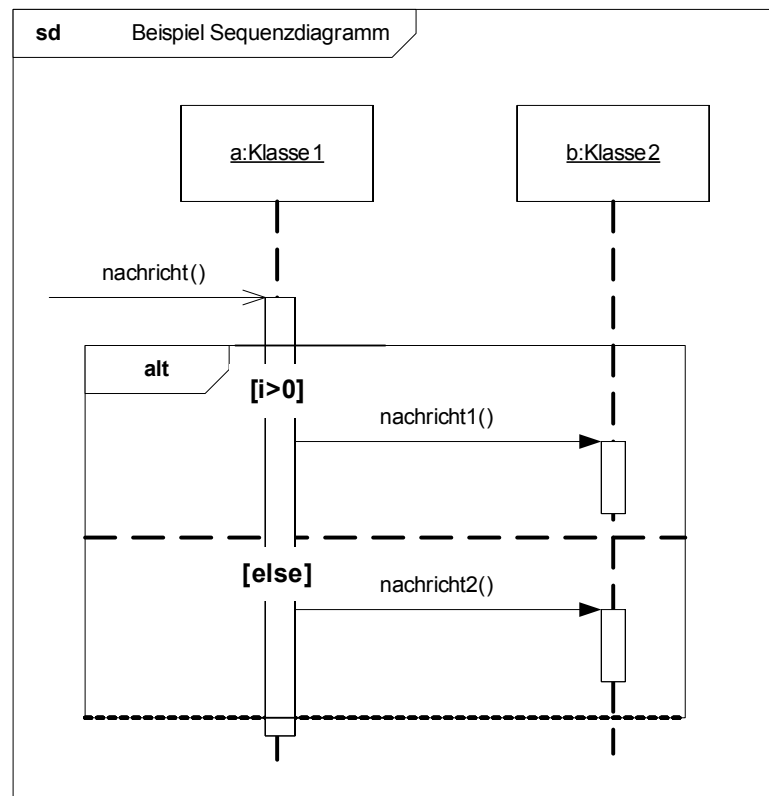


Abb. 3.1: Beispiel eines UML-Sequenzdiagramms

3.4 Vorgehen beim Entwerfen

Die Art und Weise, wie Klassen und ihr Zusammenspiel modelliert werden, hängt von der Aufgabenstellung und der gewählten Architektur ab. Der Entwurf von Webanwendungen mit Datenbanken, Spielen oder zustandslosen Anwendungen, z.B. Steuerungssoftware für Scheibenwischer, läuft anders ab. Es gibt allerdings ein typisches Vorgehen für den Entwurf, das in diesem Abschnitt beschrieben werden soll.

Vorgehen bei Entwurf abhängig vom Projekt

Typischerweise wird mit dem Entwurf der Klassen der Datenhaltung begonnen, da sie von vielen anderen Klassen benutzt werden. Dazu werden die Klassen des Domänenmodells als Grundlage genutzt. Schritt für Schritt werden Klassen entworfen, die für die Erfüllung der funktionalen Anforderungen verantwortlich sind, und anschließend in Quelltext umgesetzt. Zuerst werden die Klassen und ihre Beziehungen in einem Klassenmodell entworfen. Sequenzdiagramme werden meist nur für ausgewählte Abläufe modelliert, um zum einen zu prüfen, ob mit den entworfenen Klassen die Abläufe machbar sind, und zum anderen schwierige oder ansonsten unklare Abläufe zu dokumentieren. Beim Entwurf kommen Entwurfsmuster zum Einsatz (Abschnitt 3.5). Wichtig ist es dabei, dass nicht versucht wird, alles auf einmal zu modellieren („Model in small increments“), da es nur in den wenigsten Fällen gelingt, alles im ersten Wurf richtig zu entwerfen. Entwürfe werden deswegen durch die Implementierung im Quelltext erst erprobt („Prove it with code“), bevor weitergearbeitet wird.

Typisches Vorgehen

Im Rahmen des agilen Modellierens wurden einige weitere Prinzipien entdeckt, die helfen, effektiver zu entwerfen [Amb04]:

Bewährte Prinzipien

- **Travel light** („Reise mit leichtem Gepäck“). Im Laufe der Softwareentwicklung ändern sich immer wieder die Anforderungen und das Verständnis der Anforderungen und der Software. Diagramme müssen also immer wieder angepasst werden. Um einen zu hohen Arbeitsaufwand zu vermeiden, ist es deswegen sinnvoll, bei jedem Diagramm kritisch zu prüfen, ob es genügend Wert für die Dokumentation oder die künftige Entwicklung hat, um weiterhin aktualisiert zu werden, oder ob es unter Verzicht auf weitere Aktualisierungen archiviert oder verworfen wird.
- **Detailtiefe je nach Problemstellung**. Es ist nicht immer sinnvoll, bei jedem Entwurf alle Details darzustellen. In einem Diagramm sollten nur die Dinge dargestellt werden, die tatsächlich für das Verständnis wichtig sind. In einem Klassendiagramm können häufig viele Methoden und Attribute weggelassen werden. Auch die Sichtbarkeit von Attributen und Methoden ist meist aus dem Kontext klar.

- **Iterate to another artifact.** Sollten Sie an einem Diagramm nicht weiterkommen, kann es eine gute Idee sein, mit einem anderen Diagrammtyp weiterzumachen. Gibt es z.B. Probleme bei der Erstellung eines Klassendiagramms, könnte es hilfreich sein, sich die Zusammenhänge mit einem Instanzdiagramm oder sich die dynamischen Abläufe mit einem Sequenzdiagramm klarzumachen.
- **Model with others.** Modellieren zusammen mit anderen Menschen ist effektiver und macht mehr Spaß.

3.5 Entwurfsmuster

Entwurfsmuster (Design Patterns) sind ähnlich wie Analyse- (Abschnitt 1.6) und Architekturmuster (Abschnitt 2.2) vorgefertigte Lösungsschablonen für verallgemeinerte Probleme des Entwurfs. Entwurfsmuster begegnen einem heute in praktisch allen Klassenbibliotheken und Frameworks. Ein grundlegendes Verständnis dieser Muster ist also nicht nur wichtig, um besser und effektiver entwerfen und kommunizieren zu können, sondern auch, um moderne Klassenbibliotheken und Frameworks verwenden zu können. Wie alle Muster sind auch Entwurfsmuster verallgemeinerte Lösungen, die Sie zur Lösung konkreter Aufgaben anpassen müssen.

Die erste und bis heute wichtigste Sammlung von Mustern wurde von der so genannten „Gang of Four“ (Gamma, Helm, Johnson, Vlissides) publiziert [Gam96]. Seitdem ist eine unübersehbare Vielzahl an Entwurfsmustern in Büchern, Zeitschriften und Webseiten beschrieben worden. In diesem Abschnitt werden Sie die aus meiner Sicht wichtigsten Muster kennen lernen. Der Aufbau der Beschreibung erfolgt analog zu den anderen Mustern. Für einen effektiven und guten Entwurf ist es wichtig, dass Sie im Lauf der Zeit die für Ihren Bereich wichtigen Muster aus den bekannten Quellen finden und sich aneignen.

3.5.1 Beobachter (Observer)

Problem in seinem
Kontext

Eine Klasse soll die Möglichkeit erhalten, anderen Klassen Botschaften zu senden, ohne dass der Typ der anderen Klassen vorher festgelegt werden muss. Ein typisches Beispiel ist z.B. das Element für eine Schaltfläche („Button“) in einem GUI-Framework. Bei diesem Element ist es naturgemäß unmöglich vorher zu wissen, welche Klassen bei Verwendung des Frameworks auf das Ereignis des Auslösens der Schaltfläche in einer Anwendung reagieren sollen.

Eine Klasse Observable (engl. für beobachtbar) besitzt Referenzen auf Objekte, die das Interface Observer (engl. für Beobachter) implementieren. Konkrete Observablen werden von der Klasse Observable abgeleitet. (Abb. 3.2). Jeder Beobachter registriert sich bei einer Observablen als Beobachter durch Aufruf der Methode addObserver(). Ein Beobachter kann sich auch bei mehreren Observablen registrieren. Möchte eine konkrete Observable ihre Beobachter benachrichtigen, wenn z.B. ein Nutzer den Button gedrückt hat, nutzt sie die Methode notifyObservers(), die die Methode update() bei allen registrierten Beobachtern aufruft (Abb. 3.3). Aufgrund dieses Verhaltens ist das Muster auch unter der Bezeichnung „publish–subscribe“ (engl. für veröffentlichen–abonnieren) bekannt.

Lösung

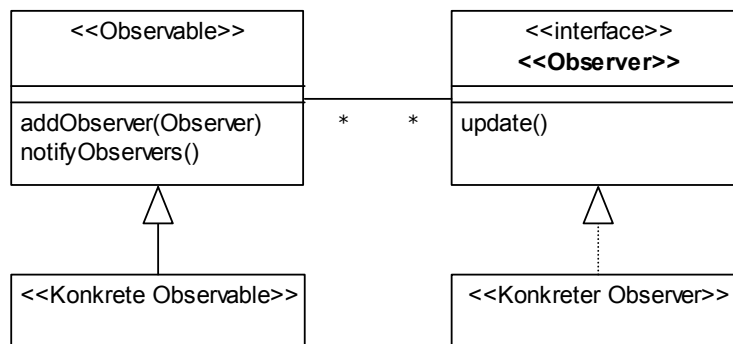


Abb. 3.2: Klassendiagramm des Musters Beobachter

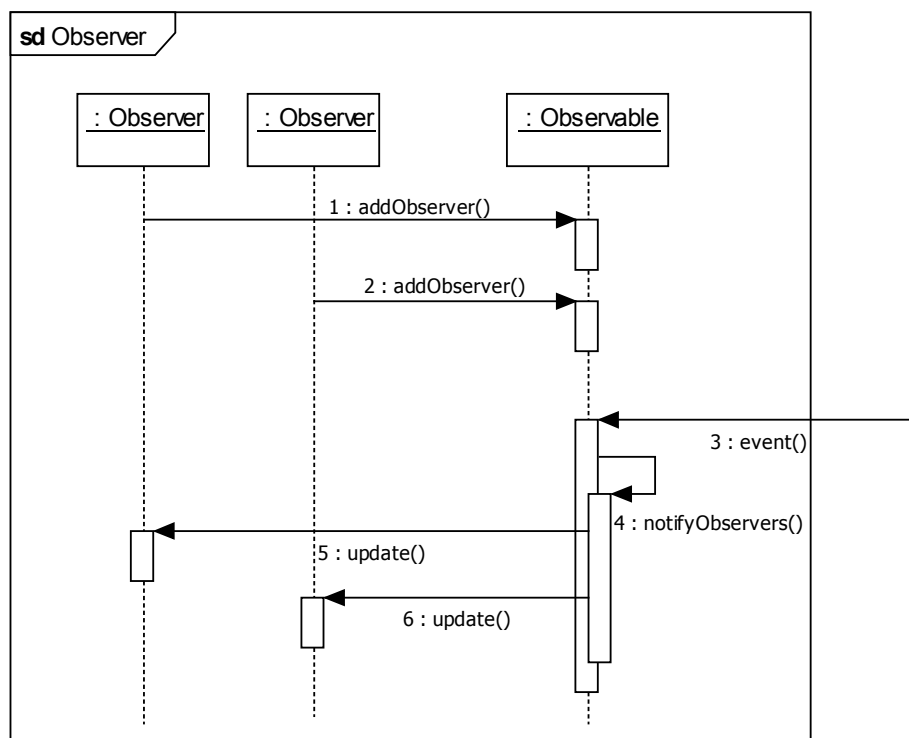


Abb. 3.3: Sequenzdiagramm des Musters Beobachter

Beispiele

Das Observer-Konzept findet man heute in fast allen Frameworks, insbesondere in Frameworks für GUI-Komponenten. In Java beispielsweise gibt es die Klasse `Observable` und das Interface `Observer` im Paket `java.util` genau so wie in diesem Muster beschrieben. Das Konzept spielt auch bei der Ereignisverwaltung der GUI-Komponenten von Java („Swing“) eine zentrale Rolle. Soll beispielsweise eine Klasse benachrichtigt werden, wenn eine Schaltfläche vom Typ `javax.swing.JButton` ausgelöst wird, implementiert sie das Interface `java.awt.event.ActionListener` und registriert sich bei der Schaltfläche durch Aufruf der Methode `addActionListener()`. Wird die Schaltfläche ausgelöst, ruft sie bei allen registrierten Listnern die Methode `actionPerformed(ActionEvent e)` auf. Die Klasse `JButton` entspricht also der Observablen und das Interface `ActionListener` dem Observer. Das Objekt vom Typ `ActionEvent` trägt dabei Informationen über das ausgelöste Ereignis, was zum Beispiel hilfreich ist, wenn ein Listener bei mehreren Observablen registriert ist.

3.5.2 MVC

Die Abkürzung MVC steht für Model–View–Controller (engl. für Modell–Ansicht–Steuerung). Dieses Muster kann als Architektur- und als Entwurfsmuster eingesetzt werden.

Problem in seinem Kontext

Eine Anwendung verarbeitet Eingabedaten und reagiert interaktiv mit Ausgabedaten. Ein typisches Beispiel ist eine graphische Benutzeroberfläche, bei der Benutzer Aktionen durchführen können wie Mausbewegungen, Mausklicks oder Tastatureingaben und Reaktionen der Software in Form von graphischen Ausgaben auf dem Bildschirm erscheinen. Es stellt sich die Frage, wie die Verantwortlichkeiten sinnvoll auf Klassen verteilt werden können.

Lösung

Die Verantwortlichkeiten werden in drei Typen unterschieden (Abb. 3.4):

- **Model:** Die Klassen des Models beinhalten Daten und fachliche Funktionalitäten
- **View:** Die Klassen der View zeigen Daten des Models an
- **Controller:** Die Klassen des Controllers nehmen Eingaben entgegen, speichern ggf. Daten im Modell oder steuern die Dialoge, wenn z.B. ein neuer Dialog angezeigt werden soll.

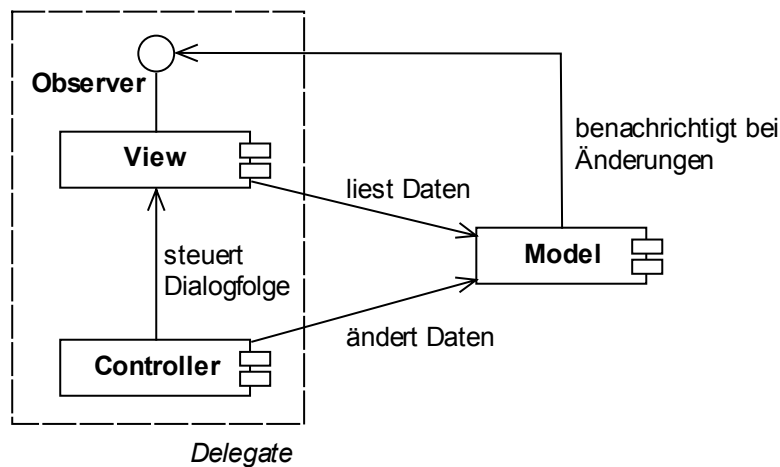



Abb. 3.4: Prinzipskizze des Muster Model–View–Controller. Das Symbol des Rechtecks mit zwei kleinen Rechtecken am rechten Rand  (Komponente) symbolisiert, dass es sich dabei nicht um eine einzelne Klasse handeln muss, sondern es sich um mehrere Klassen handeln kann. Bei mehreren Klassen fungiert nicht die Komponente als Observer, sondern die Klassen der View können sich einzeln bei Klassen des Models als Observer registrieren.

Die Klassen der View registrieren sich als Observer an den entsprechenden Klassen des Models. Ändern sich Inhalte des Models durch Aktionen des Controllers, erhält die View Nachricht und kann die Ansicht aktualisieren. Der Controller kommuniziert also die Daten nicht direkt an die View.

**View ist Observer
des Models**

Seine volle Stärke spielt MVC aus, wenn mehrere Views und Controller auf den gleichen Daten arbeiten, wie z.B. bei einem Auswahldialog für Farben (Abb. 3.5). Hier wird der Farbton gleichzeitig im Farbfenster, als HSB und RGB angezeigt, er kann als HSB direkt eingegeben werden oder direkt aus der Ansicht ausgewählt werden. Hier arbeiten mehrere Views und Controller des Dialogs auf einem gemeinsamen Model. View und Controller arbeiten in der Regel als Paar zusammen. Dieses Paar wird in der Literatur vielfach als Delegate (engl. für Delegierter) bezeichnet. Bei einfachen Aufgabestellungen werden manchmal View und Controller in einer gemeinsamen Delegate-Klasse implementiert.

**Mehrere Views und
Controller**

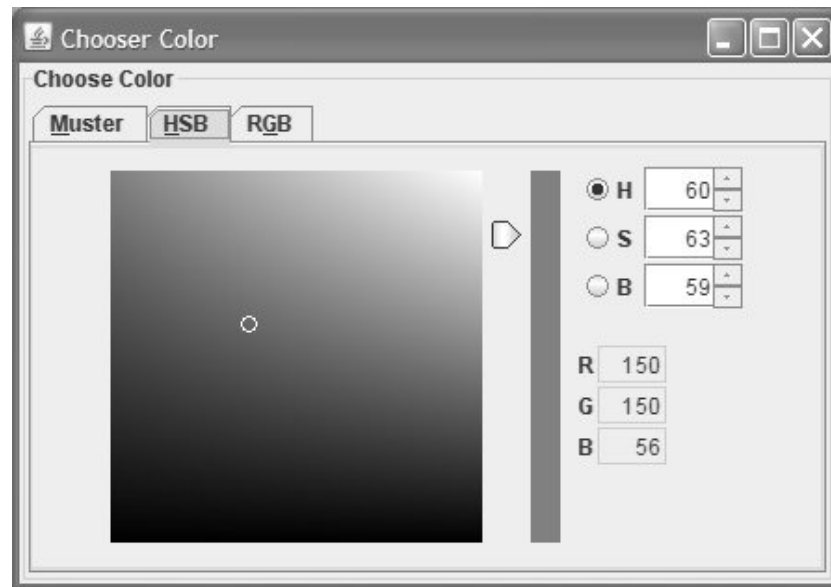


Abb. 3.5: Farbauswahldialog als Beispiel für einen Dialog, in dem mehrere Elemente dieselben Daten anzeigen und mit denen sie sich manipulieren lassen

Vorteile

Der Hauptvorteil von MVC ist die klare Verteilung der Verantwortlichkeiten auf die Klassen und die definierten Kommunikationswege zwischen den Klassen. Dies macht es möglich, dass zuerst die Klassen des Models einzeln entwickelt und getestet werden, anschließend die Klassen der View und abschließend die des Controllers.

HMVC

Bei komplexeren Anwendungen bestehen Oberflächen aus mehreren Teilen, die bei vielen heutigen Anwendungen auch dockbar gestaltet sein können, d.h. sie können innerhalb des Fensters verschoben, ein- oder ausgeblendet werden. In diesem Fall werden die Teile einer GUI einzeln nach dem Muster MVC entwickelt. Die Teile sind in übergeordnete Views eingebettet und kommunizieren miteinander über übergeordnete Models und Controller. Man nennt dieses Muster „Hierarchisches MVC“ (HMVC) [Cai00].

Beispiele

Das Muster MVC wurde ursprünglich für die Programmiersprache Smalltalk entwickelt. Es wird inzwischen sehr oft verwendet, z.B. in Java Swing, dem Webframework Struts oder dem SAP Webapplication Server.

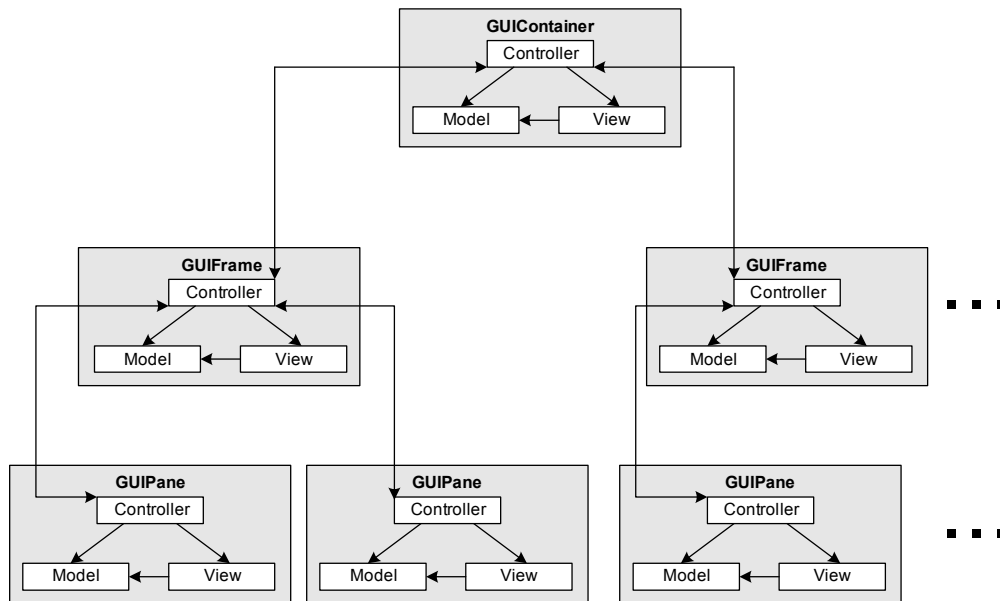


Abb. 3.6: Beispielskizze für Hierarchisches MVC abgewandelt aus [Cai00]



Übungsaufgabe

3.2 Identifizieren Sie Model, View und Controller im folgenden Beispiel.

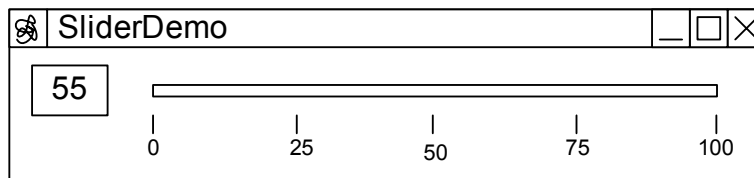


Abb. 3.7: GUI eines einfachen Dialogs für Übungsaufgabe

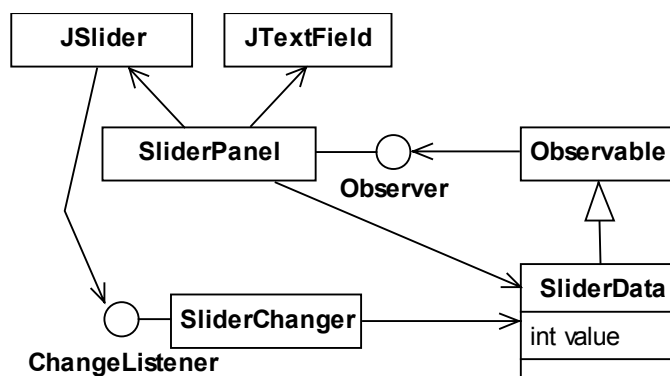


Abb. 3.8: Klassendiagramm der Übungsaufgabe

3.5.3 Immutable

Problem in seinem Kontext

Werden Referenzen auf Instanzen von einem Objekt an ein anderes weitergegeben, so besteht die Gefahr von unerwünschten Querwirkungen. Betrachten Sie folgendes Beispiel:

Beispiel

```
public class Point {
    private double x;
    private double y;

    public Point(double aX, double aY){
        x = aX;
        y = aY;
    }

    public double getX() { return x; }
    public double getY() { return y; }

    public void setX(double aX) { x = aX; }
    public void setY(double aY) { y = aY; }
}

public class Line {
    private Point start;
    private Point end;

    public Line(Point aStart, Point anEnd){
        start = aStart;
        end = anEnd;
    }

    public Point getStart(){ return start; }
    public Point getEnd(){ return end; }
}

public class Fatal {
    public static void main(String[] args){
        Line line = new Line( new Point(1., 2.),
                             new Point (-1., 1.));
        Point point = line.getStart();
        point.setX(7); // Problem: Zugriff auf private
                       // Daten von linie
    }
}
```

Unerwünschter Zugriff auf private Elemente einer anderen Klasse

In diesem Beispiel besitzt die Klasse `Line` zwei Assoziationen auf ihren Anfangs- und Endpunkt vom Typ `Point`. In der Methode `main()` erhält die Klasse `Fatal` über eine Zugriffsmethode Zugriff auf den privaten Anfangspunkt der Instanz `line` und ändert diese Daten unbemerkt von der Instanz. Dies führt zu unerwünscht versteckten Querbeziehungen zwischen Klassen, die Softwareentwicklern typischerweise die Entwicklung und Fehlersuche erschweren. Man nennt diesen unerwünschten Effekt **Kopplung**.

Klassen, deren Instanzen an andere Instanzen weitergegeben werden, werden so gestaltet, dass sie nach der Erzeugung nicht mehr verändert werden können:

Lösung

- Die Attribute werden im Konstruktor definiert.
- Es werden nur get-Methoden und keine set-Methoden implementiert.
- Es wird keine Methode implementiert, die den internen Zustand der Instanz verändert.
- Wird eine Änderung des internen Zustands benötigt, wird eine neue Instanz der Klasse mit den neuen Werten erzeugt, was die ursprüngliche Instanz unverändert lässt.
- Die Klasse sollte final sein, damit nicht in abgeleiteten Klassen veränderbare Elemente hinzugefügt werden können.

Diese Klassen werden als Immutable (engl. für unveränderbar) bezeichnet.

Immutable ist dann gut geeignet, wenn die Daten bei der Erzeugung bekannt sind und die Daten nur selten geändert werden müssen. Müssen die Daten allerdings häufig geändert werden, sollte in diesem Fall überlegt werden, ob auf Immutable verzichtet wird. In diesem Fall sind die Klassen veränderbar (engl. Mutable) und in get-Methoden werden Kopien der privaten Instanz zurückgegeben. Dies hat jedoch wiederum den Nachteil, dass das Kopieren von Klassen zeit- und speicherintensiv ist.

Probleme und Alternativen

In Java sind beispielsweise die Klassen `Integer` für Ganzzahlen und `String` für Zeichenketten Immutable. Es gibt in Java für Zeichenketten auch eine änderbare Klasse `StringBuffer`, die verwendet werden kann, wenn Zeichenketten veränderbar sein sollen.

Beispiele



Übungsaufgaben

3.3 Wie muss die Klasse `Point` des Beispiels verändert werden, damit sie Immutable ist?

3.4 Untersuchen Sie die folgenden Klassen der Java API daraufhin, ob es sich um Mutable- oder Immutable-Klassen handelt:

```
java.lang.Math
java.lang.Integer
java.awt.Color
java.awt.event.ActionEvent.
```

3.5.4 Iterator

Problem in seinem Kontext

Objekte können in verschiedenen Datenstrukturen wie verkettete Liste, Vektor oder Baum, so genannten Aggregaten, organisiert sein. Sollen Methoden sequentiell auf alle Objekte zugreifen, die ein Aggregat beinhaltet („traversieren“) muss bei jeder Datenstruktur anders vorgegangen werden. Diese Struktur sollte aber verborgen sein, da sie für den sequentiellen Zugriff unerheblich ist und die Datenstrukturen leichter austauschbar macht.

Lösung:

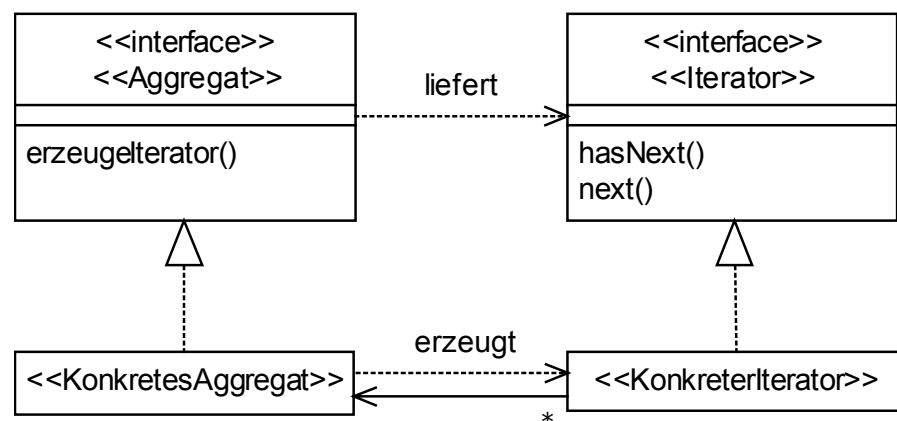


Abb. 3.9: Klassendiagramm des Musters Iterator. Die gestrichelte Linie zwischen Klasse und Interface bedeutet, dass die jeweilige Klasse das Interface implementiert.

Beispiel

Die so genannte Collection API von Java beinhaltet Iteratoren unter Verwendung von Generics. Die Funktionsweise können Sie sich am folgenden Beispiel klarmachen:

```

public class Wald {
    private ArrayList<Baum> baeume;

    public Wald() {
        baeume = new ArrayList<Baum>();
        baeume.add(new Baum("Buche", 1300.));
        // ...
    }

    public double hoechsterBaum() {
        double hoechster = 0.;
        Iterator<Baum> iterator = baeume.iterator();
        while (iterator.hasNext()) {
            Baum baum = iterator.next();
            if (baum.getHoehe() > hoechster) {
                hoechster = baum.getHoehe();
            }
        }
        return hoechster;
    }
}

```


Beispiel

Als Beispiel dient eine GUI, bei der sich ein Controller, der Person-Controller, bei einem Panel, dem PersonPanel, als ActionListener anmeldet (Abb. 3.11). An welchen konkreten GUI-Elementen er sich tatsächlich anmeldet oder wie dieses Panel intern aufgebaut ist, spielt so für den Controller keine Rolle.

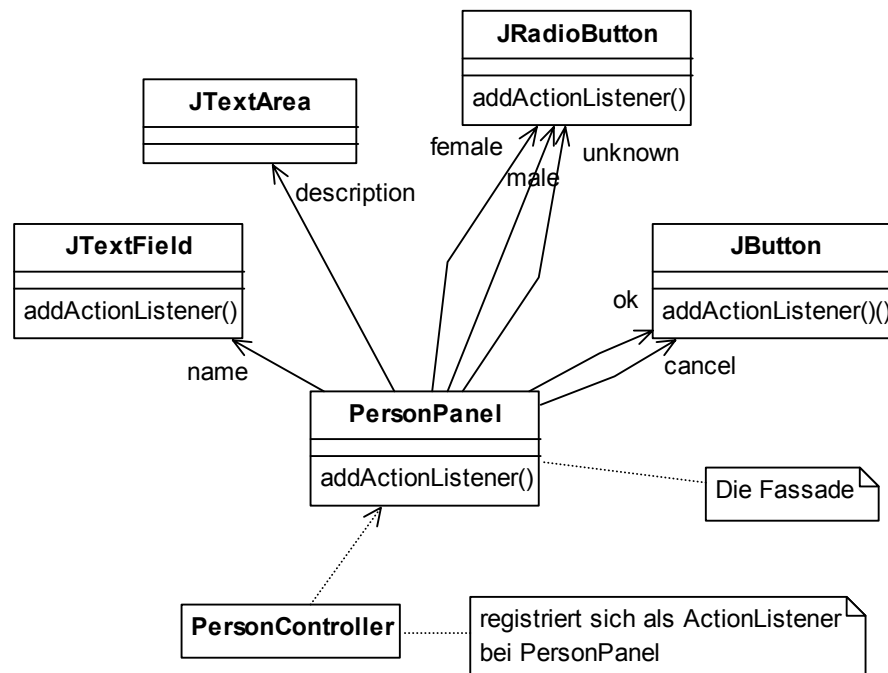


Abb. 3.11: Klassendiagramm eines Beispiels für das Muster Fassade



Übungsaufgabe

- 3.5 Welche Vorteile hat die Verwendung einer Fassade? Welche Nachteile können Sie sich vorstellen?

3.6 Beispiel eines Entwurfs

In diesem Kapitel haben Sie gelernt, wie auf Grundlage der Anforderungen und der Analyseergebnisse der Entwurf entwickelt wird. In diesem Abschnitt werden Sie den praktischen Einsatz dieser Konzepte an einem zusammenhängenden Beispiel kennen lernen. Als Beispiel habe ich ein vereinfachtes Zeichenprogramm gewählt.

Bei den Anforderungen habe ich auf ein Vision&Scope-Dokument verzichtet, da es sich hier nur um ein Lehrbeispiel handelt. Insofern sind auch die nicht-funktionalen und die funktionalen Anforderungen einfach und typisch für ein Lehrbeispiel:

Anforderungen

1. Verwendung von Java und Swing
2. Einsatz des MVC-Musters, da dieses demonstriert werden soll.

Nicht-funktionale Anforderungen und Randbedingungen

User Story: Eine Anwenderin legt ein neues Diagramm an. Sie zeichnet geometrische Objekte (Kreis, Quadrat). Sie speichert abschließend das Diagramm, um es später wieder zu laden.

User Story

Für die erste Version sollen alle geometrischen Objekte die gleiche Farbe und Größe besitzen; dies soll in der nächsten Version aber geändert werden.

In der User Story lassen sich folgende Anwendungsfälle identifizieren:

Anwendungsfälle

1. Neues Diagramm anlegen
2. Diagramm speichern
3. Diagramm laden
4. Objekt zeichnen

Für eine vollständigere Anwendung fehlen z.B. die Anwendungsfälle „Objekte verschieben“, „Objekte löschen“ oder „Objekte als Bild exportieren“.

Analyseergebnisse

Aus den funktionalen Anforderungen ergibt sich direkt das Domänenmodell (Abb. 3.12). Die Attribute von Geometrisches Objekt, Kreis und Quadrat wurden als Vorgriff auf die nächste Version modelliert. Für den fachlichen Dialogentwurf (Abb. 3.13) habe ich existierende Zeichenprogramme als Vorlage genutzt. Den Dialogentwurf habe ich bewusst nicht mit einem Zeichenprogramm erstellt, um zu demonstrieren, dass auch handschriftliche Entwürfe durchaus für die Arbeit ausreichend sein können.

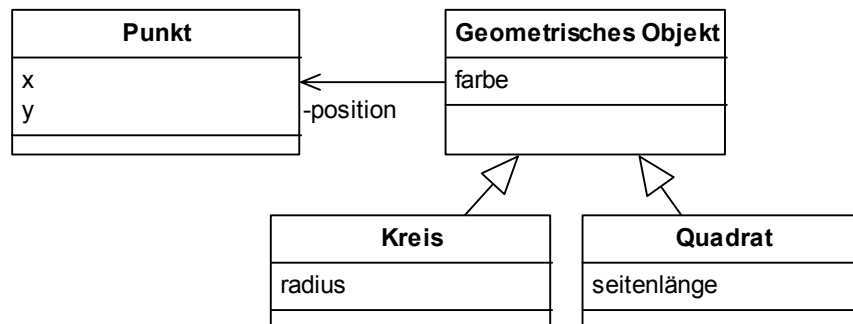


Abb. 3.12: Domänenmodell der Elemente des einfachen Zeichenprogramms

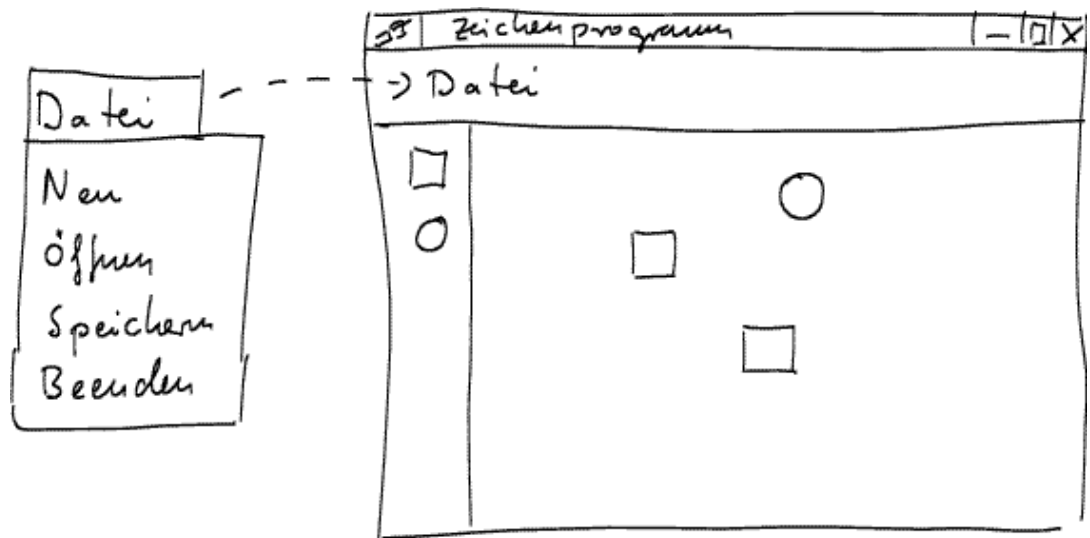


Abb. 3.13: Fachlicher Dialogentwurf des einfachen Zeichenprogramms

Da die Anwendung sehr stark auf die graphische Benutzeroberfläche ausgerichtet ist, bietet es sich an, vor dem Entwurf der Klassen zunächst die Oberfläche technisch zu entwerfen, um einen Überblick über die zu verwendenden Komponenten und die Ereignisse (engl. Events) zu erhalten (Abb. 3.14).

Entwurf

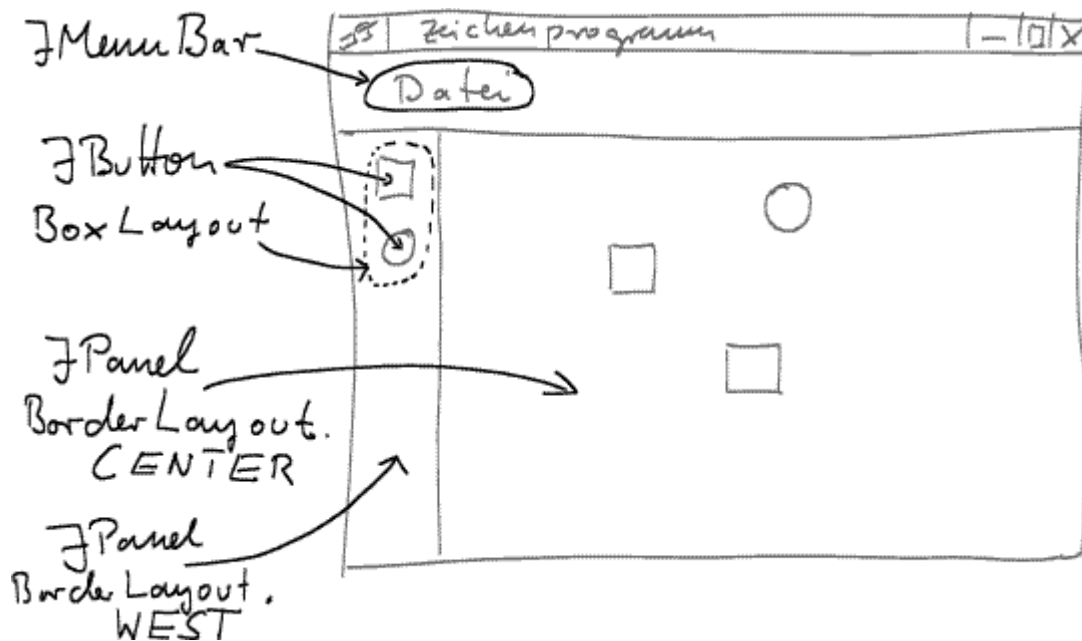


Abb. 3.14: Technischer Dialogentwurf des einfachen Zeichenprogramms. Der Typ der Elemente (JButton, JPanel, JMenuBar) und die Verteilung in verschachtelte Layout-Manager (BorderLayout enthält die Schaltflächen im BoxLayout) werden genau definiert.

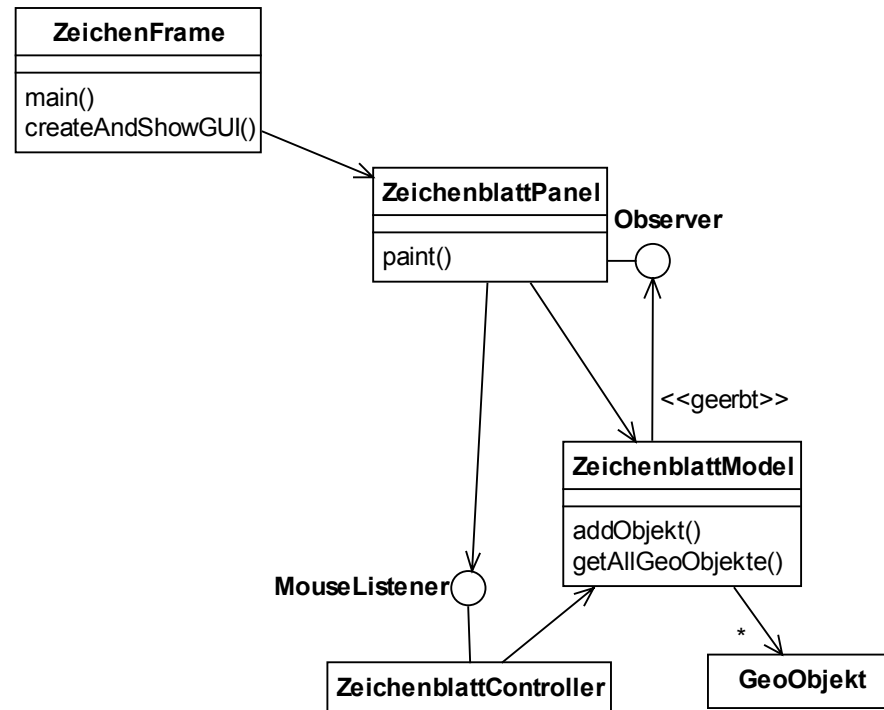


Abb. 3.15: Klassendiagramm: Entwurf des Zeichenblatts

Entwurf der Klassen des Zeichenblatts

Beim Entwurf der Klassen gehe ich nun Schritt für Schritt vor. Zunächst wird die zentrale Zeichenfläche entworfen (Abb. 3.15). Das `ZeichenFrame` beinhaltet das `ZeichenblattPanel`, auf dem gezeichnet wird. Streng nach dem MVC-Muster gehören zum Panel als View die Klassen `ZeichenblattModel` und `ZeichenblattController`. Für das `ZeichenblattModel` können die Klassen des Domänenmodells direkt ohne Änderungen verwendet werden. Bitte beachten Sie die Namenskonvention, nach der alle Klassen, deren Bezeichnung mit `Panel` endet, von `javax.swing.JPanel` abgeleitet sind. Auf die explizite Darstellung dieser Ableitungsbeziehungen habe ich bewusst verzichtet, um das Klassendiagramm übersichtlicher zu gestalten. An dieser Stelle wäre es nun für Sie als gute Übung möglich, den Entwurf durch die schrittweise Implementierung dieser Klassen zu erproben. Beginnen Sie bei der Implementierung beim Model, fahren Sie dann mit der View und abschließend mit dem Controller fort. Die Zwischenergebnisse können Sie jeweils testen¹¹.

¹¹ Hilfestellung: Achten Sie darauf, dass im Model vor `notifyObserver()` die Methode `setChanged()` aufgerufen wird, da ansonsten `notifyObserver()` nicht funktioniert.

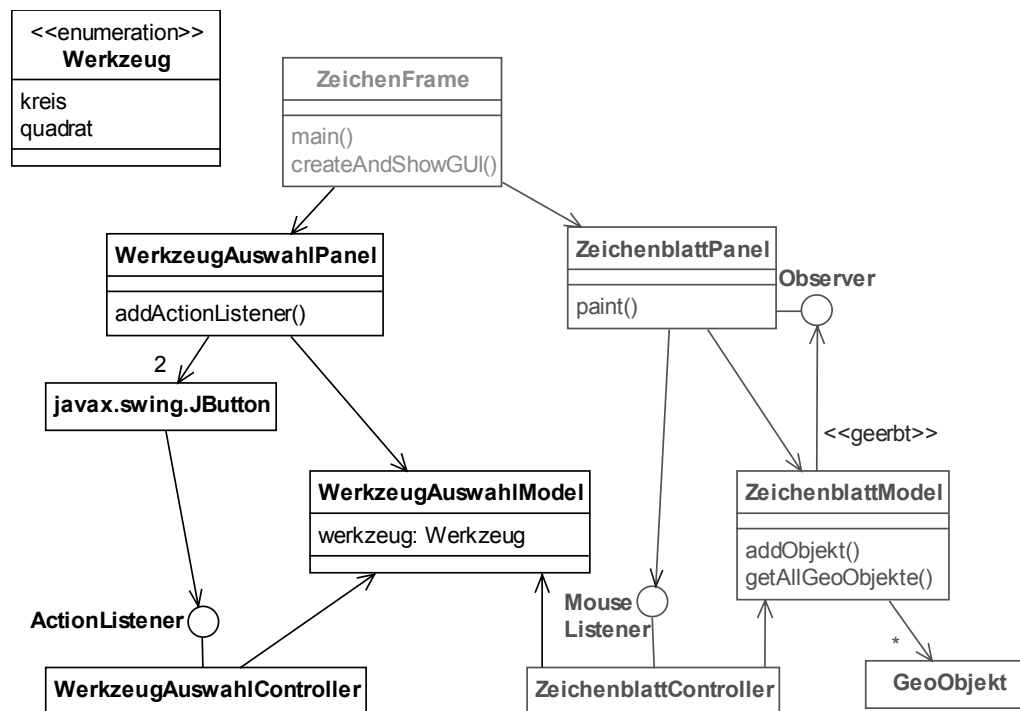


Abb. 3.16: Klassendiagramm: Entwurf der Werkzeugauswahl

Im nächsten Schritt werden die Klassen entworfen, die für die Werkzeugauswahl verantwortlich sind (Abb. 3.16). Zu den bisherigen Klassen kommt das **WerkzeugAuswahlPanel** als weitere View sowie **WerkzeugAuswahlModel** und **WerkzeugAuswahlController** hinzu. Das **WerkzeugAuswahlPanel** dient als Fassade für die Schaltflächen vom Typ **JButton**. Die **JButtons** sind mit so genannten **ActionCommands** versehen, anhand derer der Controller die Events identifizieren kann. Der **ZeichenblattController** erhält ebenfalls eine Referenz auf das **WerkzeugAuswahlModel**, da der Controller Informationen über das aktuell ausgewählte Werkzeug benötigt. Auch diesen Schritt können Sie zur Übung implementieren.

Entwurf
Werkzeugauswahl



Übungsaufgabe

3.6 Jetzt sind Sie dran. Entwerfen Sie die Klassen für den Menübalken. Hinweis: Nutzen Sie die Klassen **JMenuBar** und **JMenu**.

Zu kompliziert?

Möglicherweise finden Sie die Aufteilung und die Vorgehensweise logisch und nachvollziehbar. Vielleicht sind Sie aber auch irritiert, dass für eine Software, die bislang relativ wenig Funktionalität besitzt, so viele Klassen mit so vielen Beziehungen erstellt werden. Als denkbare Alternative könnte ja stattdessen alle Funktionalität direkt in die Panels codiert werden. Bei sehr einfachen Anwendungen ist dies durchaus ein denkbare Weg. Allerdings wären in diesem Fall die Panels für Darstellung, die Verarbeitung von Events, für das Speichern der Statusinformation und die Zusammenarbeit mit anderen Panels (z.B. bei der Werkzeugauswahl) verantwortlich. Dies würde gegen die Regel verstoßen, dass jede Klasse eine, maximal zwei Verantwortlichkeiten besitzen soll, und es würde aller Erfahrung nach dazu führen, dass der Code unübersichtlich und schlecht wartbar wird. Im Gegensatz dazu ist die Funktionalität im hier entwickelten Entwurf nicht auf Methoden, sondern auf Klassen mit unterschiedlichen Verantwortlichkeiten anhand eines durchgehenden Konstruktionsprinzips verteilt. Dieses Vorgehen verspricht, dass das System auch später noch leicht verständlich und änderbar ist. Um sich davon zu überzeugen, könnten Sie das hier entworfene System weiterentwickeln, so dass z.B. geometrische Objekte auch gelöscht oder verschoben werden können und ihre Größe und Farbe geändert werden kann.

Faktor Erfahrung

Möglicherweise haben Sie sich beim Lesen des Entwurfs gefragt, ob Sie diesen Entwurf tatsächlich bereits so detailliert vollständig vorher erstellen können. Meiner Erfahrung nach ist dies eine Frage der Kenntnis der Technologie, der verwendeten Architektur und der eingesetzten Muster. In unserem Beispiel ist auf technischer Seite Erfahrung mit dem Eventmechanismus von Swing und auf der Seite des Entwurfs Erfahrung mit MVC und Verständnis des Observer-Musters notwendig. Bei wenig Erfahrung mit Mustern oder der Technologie ist es praktisch unmöglich, ein komplexes System von Anfang an komplett zu entwerfen. In diesem Fall ist es sinnvoll, wie in Abschnitt 3.4 empfohlen, zunächst schrittweise und kleinteilig zu entwerfen und den Entwurf nach jedem Schritt zu implementieren („Prove it with code“, „Model in small increments“). Dieses Vorgehen hat jedoch den großen Nachteil, dass Probleme in späteren Teilen entstehen können, da an diese Funktionalität vorher nicht gedacht wurde und dadurch größere Änderungen notwendig werden. Aus dieser Logik heraus ist es also generell erstrebenswert, bei komplizierteren Aufgaben, für deren Lösung noch wenig Erfahrung vorliegt, zunächst Erfahrung wie beschrieben zu sammeln, die ersten Schritte ggf. als Prototyp zur Sammlung von Erfahrung zu nutzen, diesen Prototyp wegzuerwerfen und das System auf der Grundlage des gesammelten Wissens zu entwerfen. Generell ist es aber kaum möglich, einen perfekten Entwurf des gesamten System von Anfang an zu erstellen, da Softwaresysteme dazu zu komplex sind und sich eigentlich immer die Anforderungen im Laufe einer Entwicklung ändern. Um die Qualität des Entwurfs

trotzdem im Lauf einer Softwareentwicklung auf akzeptablem Niveau zu halten, wurde die Technik des Refactoring entwickelt, die Sie im Abschnitt 3.8 kennen lernen werden.

3.7 Prinzipien guten Entwurfs

Bei der Entwicklung einer Architektur oder eines Entwurfs eines Software-systems gibt es meist verschiedene Möglichkeiten, zum Ziel zu kommen. In diesem Abschnitt gehen wir der Frage nach, welche Kriterien und Prinzipien bei der Wahl der besten Alternative helfen.

Nach dem Durcharbeiten dieses Abschnitts sollten Sie

- Kriterien kennen, anhand derer Sie gute und weniger gute Entwürfe unterscheiden können,
- die Güte eines Entwurfs beurteilen können,
- bessere Entwürfe erstellen können.

3.7.1 Was ist ein guter Entwurf

Die Frage, was ein guter oder ein schlechter Entwurf ist, ist nicht generell zu beantworten. Die Antwort auf diese Frage hängt von den Anforderungen einer konkreten Aufgabe ab. Beispielsweise werden viele Strategien für den Entwurf einer komplizierten Internetanwendung nicht auf den Entwurf einer Embedded Anwendung zur Steuerung eines ABS eines Autos übertragbar sein.

Guter und schlechter Entwurf

Trotzdem gibt es aber generelle Prinzipien, die bei allen Entwürfen berücksichtigt werden sollten, da die grundlegenden Qualitätsziele „Funktionsfähigkeit“ und „Änderbarkeit“ immer wichtig sind. Solche Grundprinzipien werden Sie in diesem Abschnitt kennen lernen. Sollten Sie in einem konkreten Fall doch gegen eins der Prinzipien verstoßen, sollten Sie sich genau überlegen, warum Sie das tun.

Generelle Prinzipien

3.7.2 Teile und Herrsche

Nur sehr kleine Aufgaben können in einer einzelnen Methode oder einer einzelnen Klasse gelöst werden. Lösungen, die aus einem großen Block bestehen, sind nicht nur schwer zu verstehen, sondern auch schwer zu entwickeln. Es bietet sich also an, Lösungen in verschiedene Teillösungen aufzuteilen.

Vorteile der Zerlegung

Eine gute Zerlegung hat viele Vorteile: Kleine Teile sind deutlich einfacher zu verstehen als große Teile. Dies macht es leichter, sie zu entwickeln. Sie sind auch leichter zu ändern, da für eine Modifikation nur der zu ändernde Teil und nicht das gesamte System verstanden werden muss. Auch die Zuverlässigkeit wird höher, da Fehler auf die Teile begrenzt und mögliche Fehler leichter im Test zu finden sind, da die einzelnen Teile separat getestet werden können. Eine Zerlegung hilft auch zu vermeiden, dass die gleichen Aufgaben mehrfach in einer Software gelöst werden, indem gleiche Aufgaben in gemeinsamen Teilen gelöst werden. Nicht zuletzt ermöglicht eine Zerlegung in Teile erst, dass mehrere Softwareentwickler gleichzeitig arbeiten können.

Nachteile

Teilweise wird argumentiert, dass ein System aus vielen kleinen Teilen schwieriger zu verstehen und damit auch zu entwickeln sei, als ein System aus wenigen großen Teilen. Bei vielen Teilen ist es in der Tat schwieriger, die Übersicht zu behalten. Zwei Argumente sprechen aber trotzdem für eine Zerlegung: Zum einen ist die Übersicht über viele Teile gut zu halten, wenn sie einem durchgehenden, logischen Entwurf folgen und durch geeignete Entwurfsdokumentation wie Klassendiagramme beschrieben sind. Zum andern sind große Teile komplizierter aufgebaut, wenn sie weiter zerlegt werden könnten. Dies macht sie, wie bereits argumentiert, schwerer verstehbar und änderbar. Zusätzlich wird das System häufig verkompliziert, da komplexe Bausteine aufgrund ihrer internen Komplexität auch komplexe Beziehungen haben.

Beispiel für Zerlegung

Verschiedene Softwaresysteme werden auf verschiedene Art und Weise am besten zerlegt. Eine typische Art der Zerlegung von objektorientierten Systemen ist die Zerlegung von

- System in Subsysteme
- Subsystemen in Pakete
- Paketen in Klassen
- Klassen in Methoden.



Übungsaufgabe

3.7 Welche Vorteile hat die Aufteilung einer Software-Lösung in einzelne Teile?

3.7.3 Hohe Kohäsion

Eine Zerlegung in einzelne Teile wie Pakete oder Klassen erscheint ziemlich einfach. Die einzelnen Teile müssen aber in einem funktionierenden System so gestaltet werden, dass sie miteinander zusammenarbeiten. Im Umkehrschluss bedeutet dies, dass die einzelnen Teile miteinander zusammenhängen: Soll also ein Teil verstanden oder geändert werden, müssen auch die damit zusammenhängenden Teile verstanden bzw. geändert werden, zumindest soweit sie dafür relevant sind. Diese Abhängigkeit zwischen den Teilen wird als **Kopplung** bezeichnet.

Kopplung

Es stellt sich nun die Frage, inwieweit bei stark gekoppelten Teilen überhaupt noch von einzelnen Teilen gesprochen werden kann, oder ob es sich in diesem Fall nicht in Wirklichkeit um ein einziges großes Teil handelt, das nur auf eine eigenartige Art und Weise konstruiert worden ist. Damit würden dann alle im letzten Abschnitt beschriebenen Vorteile der Zerlegung verloren gehen. Dies bedeutet also, dass **Kopplung** nach Möglichkeit zu vermeiden ist.

Kopplung vermeiden

Das wichtigste Prinzip, um **Kopplung** zu vermeiden, ist es, den **Zusammenhalt** innerhalb der Teile möglichst hoch zu halten und Abhängigkeiten nach außen zu minimieren (**Zusammenhalt** = engl. Cohesion). Dies gelingt dadurch, dass Dinge, die zusammengehören, innerhalb eines Teils erledigt werden, und andere Dinge herausgehalten werden. Das kann z.B. bedeuten, dass Operationen die auf denselben Daten arbeiten, zu einer Klasse gehören, oder dass Klassen, die ähnliche Aufgaben haben, in einem Paket liegen. Weitere Prinzipien zur Vermeidung von **Kopplung** werden Sie im nächsten Abschnitt kennen lernen.

Erhöhe Zusammenhalt, wo möglich

Im Software Engineering unterscheidet man verschiedene Typen des **Zusammenhalts**, die im Folgenden ausführlicher diskutiert werden: Funktionaler, Schichten-, kommunikativer und Utility-Zusammenhalt. Der **Zusammenhalt** wird in dieser Reihenfolge schwächer.

Typen des Zusammenhalts

Funktionaler Zusammenhalt bedeutet, dass ähnliche Funktionalität in gleichen Softwareteilen implementiert wird. Nach Möglichkeit ist diese Funktionalität nicht von anderer Funktionalität abhängig und besitzt keine Nebeneffekte. Idealerweise führt dies dazu, dass eine bestimmte Operation ein

Funktionaler Zusammenhalt

Ergebnis liefert und dabei unabhängig ist von eventuell vorangegangenen Aufrufen, inneren Zuständen oder anderen Teilen des Systems. Ein Beispiel dafür wäre, dass ein Modul Unbekannte eines Gleichungssystems berechnet. Systeme, die so strukturiert sind, sind zwar einfach zu verstehen, die Forderung nach der Unabhängigkeit ist aber meist nicht vollständig möglich. Bei objektorientierten Systemen besitzen Klassen in der Regel über ihre Attribute innere Zustände, Klassen nutzen sich gegenseitig oder nutzen Datenbanken (Abschnitt 3.7.4).

Schichten-Zusammenhalt

Teile, die ähnliche Services für andere Teile bereitstellen, werden sinnvollerweise in Schichten gebündelt. Schichten wurden bereits im Kapitel Architektur ausführlich diskutiert (Abschnitt 2.2.2).

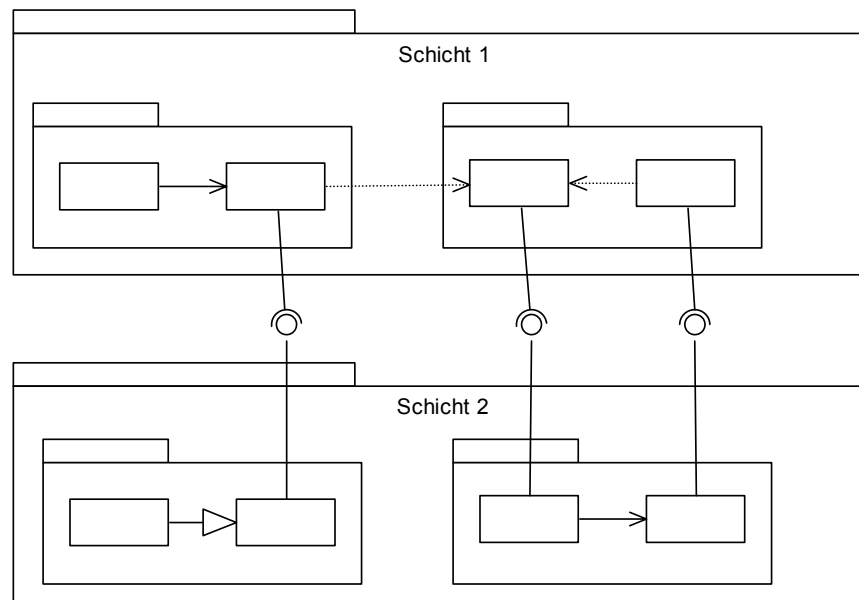


Abb. 3.17: Beispiel zum kommunikativen Zusammenhalt. Die Klassen sind in Schichten und Pakete aufgeteilt. Innerhalb der Schichten und innerhalb der Pakete einer Schicht herrscht kommunikativer Zusammenhalt.

Kommunikativer Zusammenhalt

Teile, die auf gleichen Daten operieren, gehören zusammen. Dieser Zusammenhalt ist allerdings schwächer als der Schichten-Zusammenhalt. Es ist z.B. generell keine gute Idee, die Geschäftslogik und den Zugriff auf eine Datenbank in denselben Klassen zu implementieren, obwohl sie natürlich auf denselben Daten arbeiten. Dadurch würden die Klassen aufgrund der Größe schnell unlesbar und durch die Mischung mit datenbankspezifischem Code wäre die Datenbank auch nur schwer auswechselbar.

Utilitys (engl. für Dienstprogramm) sind Teile der Software, die ähnliche Funktionalität bereitstellen und sich sonst aber nicht logisch z.B. in Schichten ordnen lassen, weil sie beispielsweise von mehreren Teilen genutzt werden. Ein Beispiel dafür ist in Java die Klasse `java.lang.Math`, in der mathematische Funktionen und Konstanten gebündelt sind.

Utility-
Zusammenhalt

3.7.4 Lose Kopplung

Wie bereits im letzten Abschnitt beschrieben, sind Kopplungen zwischen Teilen einer Software unvermeidlich, da ja die Teile einer Software zusammenarbeiten sollen und müssen. Diese Kopplung sollte aber möglichst schwach sein, da ansonsten die einzelnen Teile nicht wirklich voneinander unabhängig sind.

Kopplung möglichst
schwach

Im Software Engineering wird die Kopplung von Teilen nach Typen klassifiziert. Die Stärke der Kopplung wird je nach Typ unterschiedlich bewertet. In diesem Abschnitt werden Sie verschiedene Typen von Kopplung kennen lernen und weitere Strategien, wie Sie Kopplung möglichst schwach gestalten können (Abb. 3.18).

Typen von
Kopplung

Content → Common → Stamp → Data → Routine Call
--

Abb. 3.18: 6 Typen von Kopplung in der Reihenfolge ihrer Stärke (links stark, rechts schwach). Kopplung sollte möglichst schwach sein.

Bei der Content-Kopplung ändert ein Teil interne Daten eines anderen Teils (Content = engl. für Inhalt). Damit ist die ändernde Klasse völlig von der internen Struktur der geänderten Klasse abhängig. Generell sollte deswegen diese Kopplung vermieden werden. Die erste Maßnahme ist der Schutz von internen Daten über Zugriffsrechte (`private` – `public`) und die Verwendung von Zugriffsmethoden. Dabei sollten auch Querverwirkungen vermieden werden, indem z.B. das Immutable-Muster eingesetzt wird (Abschnitt 3.5.3).

Content-Kopplung

Common-Kopplung bezeichnet die Kopplung über die gemeinsame Verwendung von globalen Variablen¹². Auch in modernen Programmiersprachen sind globale Variablen quasi durch die Hintertür möglich. In Java beispiels-

Common-Kopplung

¹² Common = engl. für allgemein. Dieser Begriff bezieht sich auf COMMON-Blocks in FORTRAN, einer alten Programmiersprache für numerische Berechnungen. Im COMMON-Block werden globale Variablen definiert.

weise kann dies durch unsachgemäße Verwendung von „static“-Variablen geschehen. Die Erfahrung zeigt, dass Common-Kopplung zu unlesbarem und nur schwer änderbarem Code führt und deswegen vollständig vermieden werden sollte.

Stamp-Kopplung

Unter Stamp-Kopplung versteht man, wenn ein Objekt als Argument einer Methode verwendet wird (to stamp = engl. für prägen, kennzeichnen). Wird der öffentliche Teil der übergebenen Klasse verändert, muss in der Regel auch die Klasse geändert werden, der die Klasse übergeben wird. Um dies zu vermeiden, gibt es mehrere Strategien: Zum einen könnten statt der Klasse einfache Variablen übergeben werden. In diesem Fall wird allerdings die Data-Kopplung größer (s. nächster Absatz). Eine andere Strategie ist der Einsatz von Interfaces bzw. Klassen, die weiter oben in der Ableitungshierarchie stehen. Eine Leitschnur für die Entscheidung zwischen beiden Alternativen ist, dass die Anzahl der einfachen Variablen nicht zu groß sein darf.

Dazu folgendes Beispiel: In der Methode `Emler.sendEmail()` soll die Stamp-Kopplung mit der Klasse `Employee` vermieden werden, damit die Funktionalität auch für andere Adressaten verwendet werden kann:

```
public class Emler {
    public void sendEmail(Employee e, String text){
        ...
    }
    ...
}
```

Die kann durch den Einsatz einfacher Datentypen geschehen:

```
public class Emler {
    public void sendEmail(
        String name, String email, String text){
        ...
    }
    ...
}
```

Alternativ kann auch ein Interface für adressierbare Klassen deklariert werden:

```
public interface Adressee {
    public abstract String getName();
    public abstract String getEmail();
}

public class Employee implements Adressee{...}

public class Emler{
    public void sendEmail(Adressee e, String text) {...}
    ...
}
```

Je mehr Argumente eine Methode hat, desto größer ist die Kopplung mit der benutzenden Komponente. Dies wird als Data-Kopplung bezeichnet. Wie bereits im letzten Absatz erwähnt, bedingen Data-Kopplung und Stamp-Kopplung einander: Wird Stamp-Kopplung durch Einsatz von einfachen Datentypen erniedrigt, erhöht sich Data-Kopplung und umgekehrt.

Data-Kopplung

Eine Routine Call-Kopplung liegt vor, wenn eine Methode eine andere Methode aufruft. Diese Kopplung ist in jedem System vorhanden und nicht vermeidbar. Werden allerdings von einem Teil sehr viele Methoden eines anderen Teils aufgerufen, ist dies ein Indiz, dass die Zerlegung nicht in Ordnung sein könnte, da beide Teile stark voneinander abhängig sind. Werden diese Methoden immer in gleichen Sequenzen aufgerufen, könnte es auch sein, dass diese Sequenz im benutzten Teil als eigenständige Methode zusammengefasst werden sollte.

Routine Call-Kopplung



Übungsaufgabe

3.8 Welche Arten von Kopplung sollten unbedingt vermieden werden und warum?

3.7.5 Grundlegende Prinzipien, die weiterhelfen

Neben den in diesem Abschnitt genannten Regeln und Prinzipien gibt es weitere Prinzipien, die Ihnen helfen können, Systeme zu entwerfen, die nicht stärker als notwendig gekoppelt sind.

Eins der grundlegenden Prinzipien ist das Prinzip der Abstraktion, dessen Bedeutung für das Software Engineering von John Gutttag und Barbara Liskov entdeckt wurde¹³. Unter Abstraktion versteht man das Herausheben von Wesentlichem, Charakteristischem oder Gesetzmäßigem. Das Prinzip der Abstraktion beruht auf der Beobachtung, dass ein Teil leichter zu verstehen ist, wenn nicht alle Details verstanden werden müssen, sondern nur das Wesentliche. Es besagt, dass Bauteile und Abläufe nach außen immer möglichst abstrakt sein sollen, also Details weggelassen werden sollen.

Abstraktion

13 Von Barbara Liskov und John Gutttag gibt es ein sehr anspruchsvolles Buch zum Software Engineering, das fast ausschließlich auf dem Prinzip der Abstraktion aufgebaut ist.

Beispiele für Abstraktion

Beispielsweise sind vertikale Schichten schrittweise Abstraktionen von der Hardware. Das Geheimnisprinzip, das Content-Kopplung vermeidet, kann so betrachtet werden, dass die öffentlichen Methoden einer Klasse ihre Abstraktion sind. Wird z.B. zur Vermeidung von Stamp-Kopplung ein Interface genutzt, ist das Interface eine Abstraktion der tatsächlichen Klasse. Ein Iterator (Abschnitt 3.5.4) ist eine Abstraktion des Zugriffs auf eine Datenstruktur. Man könnte sogar die Ergebnisse von Anforderungen, Analyse und Entwurf als Abstraktionen des zu erstellenden Systems betrachten.

Gesetz von Demeter

Ein weiteres wichtiges Gesetz ist das Gesetz von Demeter. Es besagt, dass eine Methode nur auf Daten zugreifen (Methodenargumente, Assoziationen) oder beim Aufruf von anderen Methoden Daten erhalten sollte, die Nachbarn im Klassendiagramm sind. Dadurch wird weitreichende Kopplung vermieden. Der Name bezieht sich übrigens nicht auf den Namen der griechischen Göttin der Fruchtbarkeit, sondern auf ein Projekt namens Demeter, in dessen Rahmen dieses Prinzip formuliert wurde.

Vermeidung zirkulärer Abhängigkeiten

Eine zirkuläre Abhängigkeit liegt vor, wenn Beziehungen zwischen Klassen kreisförmig sind (Abb. 3.19 a). Sie hat zur Folge, dass die zyklisch abhängigen Klassen nicht einzeln entwickelt oder getestet werden können. Dies ist eine sehr starke Kopplung, die unbedingt vermieden werden sollte. Häufig ist eine zyklische Abhängigkeit ein Hinweis auf einen Fehler im Entwurf. Wird dieser behoben, verschwindet auch die Abhängigkeit. Beruht die zyklische Abhängigkeit nicht auf einem Fehler, so kann sie durch Einsatz eines Interfaces aufgelöst werden (Abb. 3.19 b).

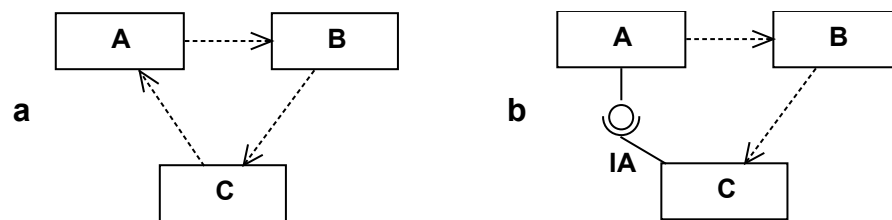


Abb. 3.19: Beispiel für eine zirkuläre Abhängigkeit (a) und eine mögliche Auflösung durch den Einsatz eines Interfaces (b).

Liskovsches Substitutionsprinzip

Das Liskovsche Substitutionsprinzip besagt, dass Klassen in jedem Fall durch ihre Unterklassen ersetzbar sein sollten. Dieses Gesetz wurde bereits in Abschnitt 1.3 eingeführt und wird noch einmal ausführlicher in der Kurseinheit „Systemmodellierung“ diskutiert.



Übungsaufgaben

- 3.9 Welche Regel zur Anzahl von Verantwortlichkeiten wurde im Kapitel „Objektorientierte Analyse“ genannt? Ist diese Regel auch im Entwurf wichtig?
- 3.10 Betrachten Sie das MVC-Muster (Abschnitt 3.5.2). Aufgrund welchen Prinzips aus diesem Abschnitt hat das Model keine direkte Assoziation zur View? Warum darf das Model den Controller nicht kennen?
- 3.11 Welches Prinzip guten Entwurfs wird im folgenden Codebeispiel verletzt? Wie könnte das Problem gelöst werden? Die Klasse Punkt (Point) wurde in Abschnitt 3.5.3 definiert.

```
public class Kreuz {
    private Linie linieVertikal;
    private Linie linieHorizontal;

    ...

    public double berechneHoehe() {
        double distx = linieVertikal.getAnfang().getX() -
            linieVertikal.getEnde().getX();
        double disty = linieVertikal.getAnfang().getY() -
            linieVertikal.getEnde().getY();
        double laengeLinieVertikal =
            Math.sqrt(distx*distx + disty*disty);
        return laengeLinieVertikal;
    }
}
```

3.8 Verbesserung des Entwurfs durch Refactoring

Es gelingt nur selten in besonderen Fällen, ein neues System von Anfang an so zu entwerfen, dass der Entwurf für das gesamte System durchgängig geeignet und zweckmäßig ist. Wie in Abschnitt 3.4 diskutiert, ist es in der Regel notwendig, iterativ vorzugehen und den Entwurf immer wieder anzupassen. Dieses Vorgehen macht es möglich, Schwächen im Entwurf frühzeitig zu entdecken und zu entfernen. Ansonsten häufen sich im Lauf der Entwicklungszeit kleinere und größere Fehler im Entwurf an, führen zu starken Kopplungen und machen den Code schnell nicht mehr wartbar. Die Erfahrung zeigt aber, dass es nicht einfach ist, bereits kleinere Mengen existierenden Codes zu ändern. Mit „Refactoring“ bezeichnet man den systemati-

schen Weg, den Entwurf bestehender Software zu verbessern, ohne dabei die Funktionalität zu verändern. Es wurde besonders durch das Buch von Martin Fowler populär, das dabei auch als Arbeitsbuch sehr hilfreich ist [Fow05].

Vorgehen

Refactoring besteht aus mehreren Schritten. Im ersten Schritt werden zunächst problematische Stellen identifiziert. Martin Fowler schlägt vor, nach „Gerüchen“ zu suchen. Beispiele für Gerüche sind „Duplizierter Code“, „Lange Parameterliste“ oder „Abgelehntes Erbe“. Für jeden Geruch schlägt Fowler als zweiten Schritt Refactorings vor, um diesen Geruch zu beseitigen. Für „Duplizierter Code“ wird z.B. u.a. „Extrahiere Methode“ vorgeschlagen oder für „Lange Parameterliste“ ist ein mögliches Refactoring „Führe Parameterobjekt ein“. Ist ein geeignetes Refactoring identifiziert, kann es im dritten Schritt durchgeführt werden. Dazu müssen nach Möglichkeit genügend automatische Testfälle vorliegen, damit sichergestellt werden kann, dass nach dem Refactoring das Programm noch so funktioniert wie vorher. Moderne Entwicklungsumgebungen wie Eclipse erleichtern die Durchführung der meisten auch komplexen Refactorings. Am besten experimentieren Sie mit den Möglichkeiten des Refactorings in Eclipse. Als letzter Schritt wird mit den vorhandenen Tests überprüft, ob der geänderte Code noch genauso funktioniert wie zuvor.

**Refactoring
verursacht Aufwand**

Refactoring ist ein praktisches, effizientes und bewährtes Verfahren, die Qualität von Code zu verbessern. In der Praxis zeigt sich aber, dass fast jeder größere Code eine Vielzahl kleinerer und größerer Probleme besitzt, an denen immer weiter verbessert werden kann. Da nicht alle Refactorings vollständig per Knopfdruck zu erledigen sind, sind sie teilweise mit hohem Aufwand verbunden. Auf der anderen Seite wird durch Refactoring keine neue Funktionalität erstellt, sondern der Aufwand für zukünftige Änderungen und neue Funktionalität verringert. Um den Aufwand in Schach zu halten ist es deswegen notwendig, die Dringlichkeit, den Aufwand und die Auswirkungen für ein Refactoring abzuwägen.



Übungsaufgabe

3.12 In welchen Schritten wird beim Refactoring zweckmäßigerweise vorgegangen?

3.9 Zusammenfassung

Im Entwurf werden die Klassen und Dateien eines zu entwerfenden Systems geplant. Der Entwurf beruht auf den Anforderungen, den Ergebnissen der Analyse und der Architektur. Die Klassen des Entwurfs entsprechen den Klassen in der verwendeten objektorientierten Sprache. Die Struktur der Klassen wird mit Hilfe von UML-Klassendiagrammen definiert. Mit Hilfe von UML-Sequenzdiagrammen werden Abläufe und das Zusammenspiel der Klassen modelliert. Der Entwurf wird sinnvollerweise in Schritten iterativ entwickelt. Dabei helfen grundlegende Prinzipien wie z.B. „Reise mit leichtem Gepäck“. Entwurfsmuster sind wichtige Hilfen beim Entwurf. Wichtige Entwurfsmuster sind Beobachter, Model-View-Controller, Immutable, Iterator und Fassade. Es gibt viele andere wichtige Muster, die in verschiedenen Musterkatalogen verzeichnet sind. Die Qualität eines Entwurfs kann anhand von Prinzipien guten Entwurfs beurteilt werden. Das wichtigste Grundprinzip ist die Aufteilung der Gesamtaufgabe in kleinere Teile. Damit diese Teile handhabbar sind, ist eine lose Kopplung der Teile essentiell. Dies wird erreicht durch hohen Zusammenhalt der Teile, Vermeidung hoher Kopplung, z.B. durch Verzicht auf globale Variablen, und die Beachtung von Grundregeln guten Entwurfs wie z.B. die Vermeidung zirkulärer Abhängigkeiten. Die Qualität eines Entwurfs bleibt gut, indem der Code durch Refactoring stetig verbessert wird.

Literatur

Lehrbücher

- [Amb04] Ambler, S. *The Object Primer. Agile Model-Driven Development with UML 2.0.: Agile Model-driven Development with UML 2.0.* Cambridge University Press; 3. Auflage, 2004.

Ein sehr durchdachtes und anspruchsvolles Lehrbuch zur objekt-orientierten Entwicklung mit einer Einführung in das Agile Modellieren. Auch die zugehörige Webseite <http://www.agilemodeling.com/> ist sehr empfehlenswert.

- [Let05] Lethbridge, T.; Laganieri, R.: *Object-Oriented Software Engineering.* 2. Auflage. McGraw-Hill, 2005.

Meines Erachtens der beste einführende Text in Software Engineering.

Standardliteratur zu einzelnen Themen

- [Bas03] Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice.* 2. Auflage. Addison-Wesley Longman, 2003.

Das Standardwerk zu Softwarearchitektur. Sehr angenehm zu lesen mit vielen Praxisbeispielen. Für Praktiker besonders wichtig ist das Verzeichnis der Architektur-Taktiken.

- [Coo07] Cooper, A.; Reimann, R, Cronin, D.: *About face 3.0: The essentials of interaction design.* Wiley, 2007.

Ein Schmöcker vom Erfinder von Visual Basic mit vielen Aspekten, wie Benutzeroberflächen verständlicher werden können. Leicht und unterhaltsam zu lesen. Unbedingt empfehlenswert.

- [Fow98] Fowler, M: *Analysemuster.* Addison-Wesley, 1998.

Die wichtigste und größte Quelle für Analysemuster.

- [Fow05] Fowler, M: *Refactoring.* Addison-Wesley, 2005.

Das grundlegende und nach wie vor wichtigste Buch über Refactoring. Es enthält einen großen Katalog mit Gerüchen und passenden Refactorings.

- [Gam96] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Entwurfsmuster*. Addison-Wesley, 1996.
- Das grundlegende Standardwerk zu Entwurfsmustern. Leider etwas schwer zu lesen. Für die Entwürfe wird ein Vorläufer der UML verwendet.
- [Gar02] Garrett, J.J.: *The Elements of User Experience*. New Riders Press, 2002.
- Das Buch überhaupt zu User Experience.
- [Sta08] Starke, G.: *Software-Architekturen*. 3. Auflage. Hanser, 2008.
- Eine knappe, gut verständliche Einführung in die Softwarearchitektur für Praktiker. Auf der zugehörigen Webseite <http://www.arc42.de> finden Sie nützliche Vorlagen für die Dokumentation der Sichten einer Architektur.
- [Tid05] Tidwell, J.: *Designing Interfaces*. O'Reilly, 2005.
- Entwurf von Benutzeroberflächen für Fortgeschrittene. Viele Praxisbeispiele mit Lösungsmustern.

Originalartikel

- [Cai00] Cai, J.; Kapila, R.; Pal, G.: HMVC: The layered pattern for developing strong client tiers. JavaWorld.com, 07/21/00. <http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc.html>

Lösungen zu den Aufgaben

Aufgabe 1.1

- 1) Abbildung tatsächlicher oder gedachter Gegenstände oder Konzepte und deren Beziehungen.
- 2) Verkürzte Darstellung.

Aufgabe 1.2

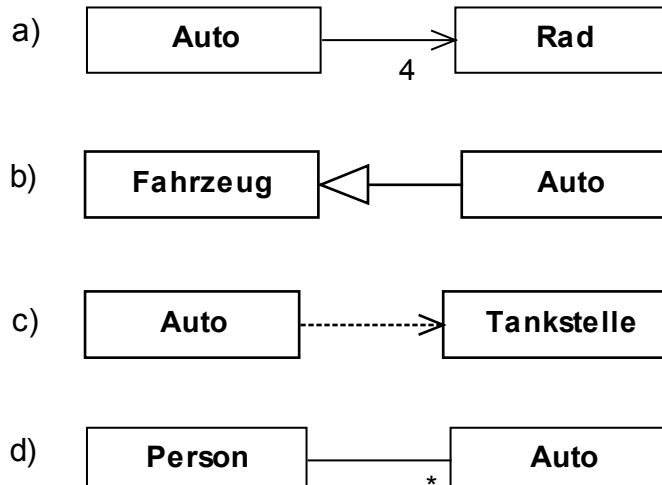
Uhrzeit
Stunde: Integer {Stunde >= 0 and Stunde < 24}
Minute: Integer {Minute >= 0 and Minute < 60}

Aufgabe 1.3

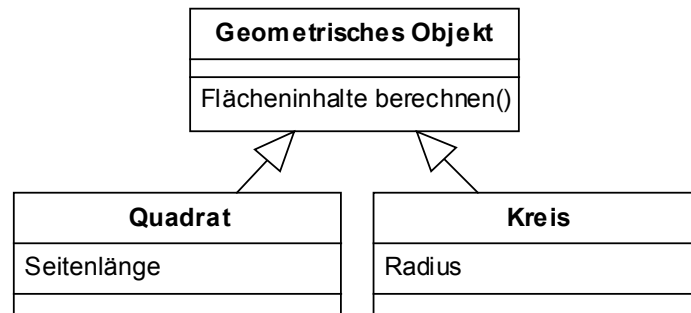
Durchgezogene Linie und gestrichelte Linie. Kein Ende, Pfeilspitze als Ende und offenes Dreieck als Ende. Durchgezogene Linie ohne oder mit Pfeilspitze steht für Assoziation, durchgezogene Linie mit offenem Dreieck als Pfeilspitze steht für Vererbung, gestrichelte Linie mit Pfeilspitze steht für Abhängigkeit.

Aufgabe 1.4

- a) Assoziation
- b) Vererbung
- c) Abhängigkeit
- d) Assoziation



Aufgabe 1.5

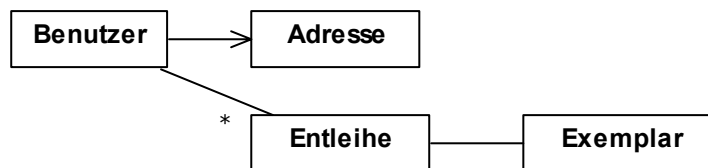


Aufgabe 1.6

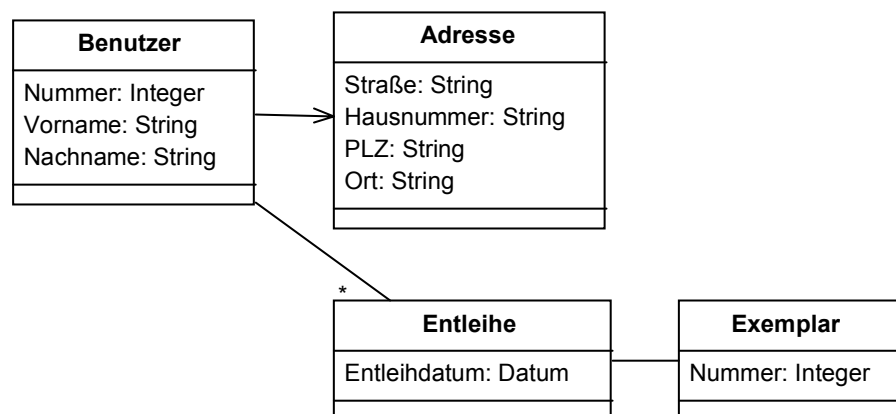
Schritt 1:



Schritt 2:



Schritt 3:



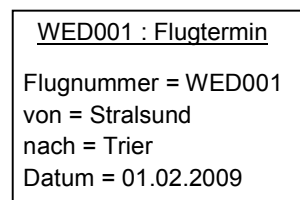
Man könnte argumentieren, dass die Adressinformationen Attribute des Benutzers sind und die Attribute der Adresse in die Klasse des Benutzers integrieren. Dies ist aus zwei Gründen nicht überzeugend: Erstens ändern sich Vor- und Nachname als Daten des Benutzers nicht oder nur sehr selten. Diese Daten machen also die Identität des Benutzers aus. Die Adressdaten

nicht, da sie sich häufiger ändern können. Zweitens besitzt eine Adresse insgesamt vier Attribute, die direkt zusammenhängen und sich bei einem Umzug in der Regel auch zusammen ändern, während die anderen Daten des Benutzers gleich bleiben. Außerdem ist es übersichtlicher, zwei wohldefinierte Klassen zu modellieren, als eine große Klasse mit mehreren, unzusammenhängenden Feldern.

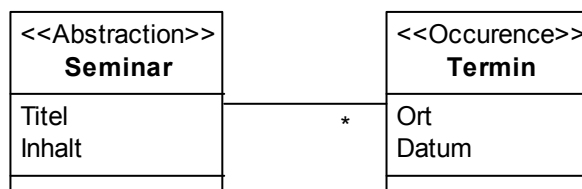
Aufgabe 1.7

Aktivitäten sind in Zustandsdiagrammen meistens mit dem Übergang von Zuständen verknüpft. Nicht jeder Zustandsänderung entspricht aber eine Aktivität. Bei Aktivitätsdiagrammen werden Aktivitäten direkt abgebildet als Rechtecke mit abgerundeten Ecken.

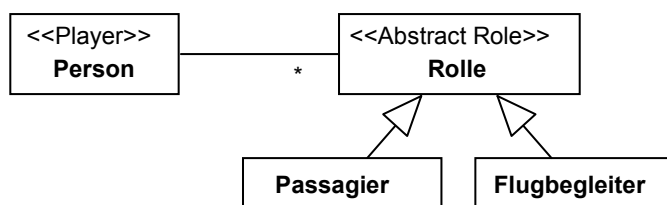
Aufgabe 1.8

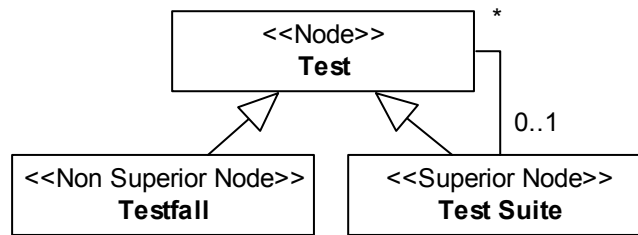


Aufgabe 1.9



Aufgabe 1.10



Aufgabe 1.11**Aufgabe 1.12**

Exemplartyp: „Titel“ ist Abstraktion, und „Exemplar“ ist Auftreten.

Aufgabe 1.13

GNOME Human Interface Guidelines 2.2 Kapitel 6.7 Buttons:

Label all buttons with imperative verbs, using header capitalization. For example, Save, Sort or Update Now.

Deutsch: Alle Schaltflächen sollen mit einem Verb im Imperativ beschriftet sein und mit einem Großbuchstaben beginnen. Zum Beispiel: Speichern, Sortieren, Aktualisieren.

Aufgabe 1.14

Der Dialog ist nicht selbstbeschreibend: Es fehlt die Angabe des Originalnamens des umzubenennenden Files. Hat ein Benutzer oder eine Benutzerin aus Versehen das falsche File ausgewählt, gibt es keine Möglichkeit, dies zu bemerken. Erinnert er oder sie sich nicht an die Auswahl, muss der Dialog geschlossen werden und die Funktionalität erneut ausgeführt werden.

Aufgabe 2.1

Softwarearchitekten sollten gleich zu Beginn hinzugezogen werden:

1. Architekten liefern Aufwandsschätzungen.
2. Architekten beurteilen die Machbarkeit.
3. Die Aussagen zu Aufwand und Machbarkeit beeinflussen die Anforderungen.

Aufgabe 2.2

Taktiken, Architekturmuster, Referenzarchitekturen, Frameworks

Aufgabe 2.3

Kontextsicht, Bausteinsicht, Laufzeitsicht, Verteilungssicht

Aufgabe 2.4

Normalerweise sollten die Muster Client-Server und Peer-to-Peer nicht gemischt werden, da beide Konzepte ganz unterschiedliche Probleme adressieren. Ausnahme sind Peer-to-Peer-Lösungen, die häufig von Servern unterstützt werden, die z.B. bei Anwendungen zur Internettelefonie Adressdienste anbieten. Das Problem ist dabei, dass die Peers dann direkt miteinander kommunizieren müssen, was zu Sicherheitsproblemen führen kann. Ein weiteres, im Text nicht erwähntes, Problem ist, dass bei Verwendung beider Muster Softwareentwickler durcheinanderkommen können, wie denn nun Informationen ausgetauscht werden sollen, da es zwei alternative Kommunikationswege gibt.

Aufgabe 2.5

Vertikale Schichten (Layers) abstrahieren von der Hardware, horizontale Schichten (Tiers) abstrahieren nach logischen Gesichtspunkten, in der Regel nach Funktionalität.

Aufgabe 2.6

Architekturmuster: Client-Server, Mehrschichtenmodell

Taktiken: Aktive Redundanz (s. Seite 38)

Referenzarchitektur: JEE

Aufgabe 2.7

N6. Webapplikation, N7. Integrität

Aufgabe 3.1

Vorteile:

- Bessere Möglichkeiten, den Entwurf zu durchdenken
- Probleme sind im Entwurf leichter zu beheben als in fertiger Software
- Leichtere Absprachen mit anderen Entwicklern

Gefahr: Zu viel Zeit an Modellen verbringen, ohne die Machbarkeit zu prüfen.

Aufgabe 3.2

Model: SliderData

View: SliderPanel, JSlider und JTextField

Controller: SliderChanger

Aufgabe 3.3

Die Klasse muss final sein. Die Methoden `setX()` und `setY()` müssen entfernt werden.

Aufgabe 3.4

`java.lang.Math`: Diese Klasse ist weder Mutable noch Immutable, sondern hat nur statische, unveränderbare Elemente.

`java.lang.Integer`: Immutable

`java.awt.Color`: Auch dies ist eine Immutable-Klasse, obwohl sie nicht final ist. Hier wurde bei der Implementierung der Unterklassen darauf geachtet, dass sie keine veränderbaren Elemente enthalten.

`java.awt.event.ActionEvent`: Mutable, da sie von der Klasse `java.awt.AWTEvent` die Methode `setSource()` erbt.

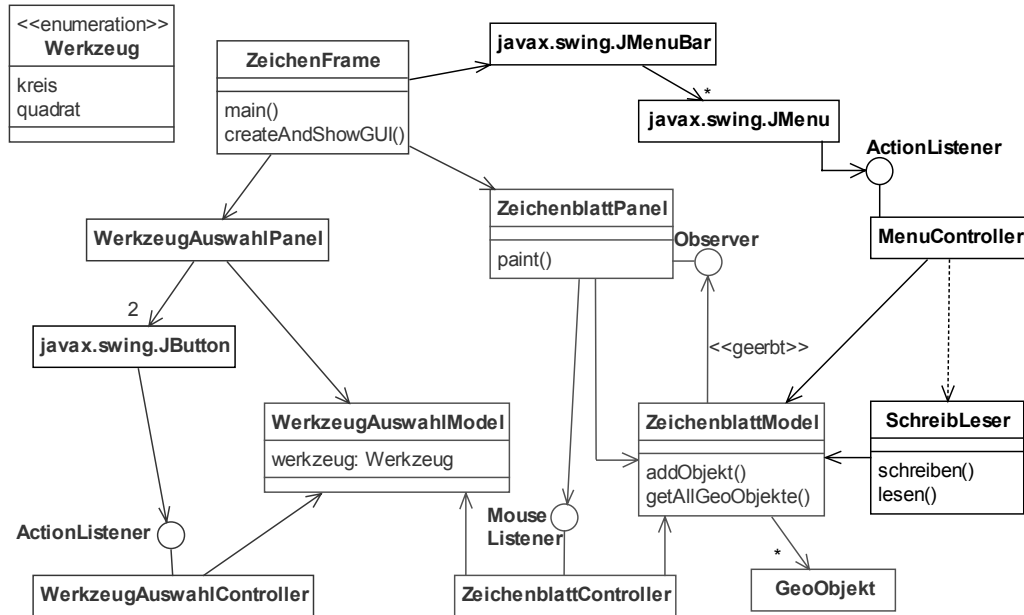
Aufgabe 3.5

Vorteile:

- Klassen, die die Fassade nutzen, werden nicht durch Änderungen hinter der Fassade beeinflusst.
- Einfaches Verstehen, da die Komplexität durch die Fassade abstrahiert wird.

Möglicher Nachteil: Wenn die Aufgaben der Klassen hinter der Fassade sehr komplex sind und die gesamte Funktionalität benötigt wird, wird die Fassade sehr umfangreich. In diesem Fall sollte keine Fassade verwendet werden, da so nicht abstrahiert werden kann. Es sind dann andere Maßnahmen zu bedenken (s. Abschnitt 3.7).

Aufgabe 3.6



Die View besteht hier aus Objekten der Klassen `JMenuBar` und `JMenu` aus der Java Standard Bibliothek. Da dieses Menü zustandslos entworfen wurde, können diese Klassen direkt und ohne Model verwendet werden. Der `MenuController` nutzt eine neue Klasse `SchreibLeser`, die die Funktionalität zum Speichern und Lesen des aktuellen Zeichenblatts bereitstellt.

Aufgabe 3.7

Einfacher zu verstehen, leichter zu ändern, Änderungen wirken nur lokal, höhere Zuverlässigkeit, modulare Testbarkeit, bessere Wiederverwendbarkeit, Arbeitsteilung wird möglich.

Aufgabe 3.8

- Content-Kopplung, da dann ein Teil völlig von der inneren Struktur eines anderen abhängig ist.
- Common-Kopplung, da dann unübersehbare Querwirkungen zwischen Teilen entstehen.

Aufgabe 3.9

Eine Klasse sollte nur eine, maximal zwei Verantwortlichkeiten besitzen (s. Schluss von Abschnitt 1.4.2). Dies gilt natürlich genauso auch für den Entwurf.

Aufgabe 3.10

Vermeidung zirkulärer Abhängigkeiten. Würde das Model die View kennen, wäre es eine bidirektionale Assoziation, ein Spezialfall der zirkulären Abhängigkeit. Würde das Model den Controller kennen, würde ebenfalls eine zirkuläre Abhängigkeit zwischen allen drei Komponenten entstehen.

Aufgabe 3.11

Die Klasse Kreuz arbeitet mit Daten der Klasse Punkt, auf die allerdings nur die Klasse Linie Assoziationen besitzt. Damit wird das Gesetz von Demeter verletzt. Die beste Lösung ist es, dass die Klasse Linie eine Methode erhält, die die Länge berechnet. Dann verwendet die Klasse Kreuz nur noch diese Methode der Klasse Linie. Dadurch arbeitet die Klasse Kreuz also nur noch mit einer Abstraktion der Klasse Linie. Sollte es notwendig sein, z.B. die Darstellung der Linie zu ändern, z.B. durch Angabe von Anfangspunkt, Länge und Winkel, ist dies dann ohne Auswirkung auf die Klasse Kreuz.

Aufgabe 3.12

1. Geruch wahrnehmen
2. Geeignetes Refactoring mit Hilfe eines Katalogs finden
3. Testüberdeckung prüfen
4. Refactoring durchführen
5. Mit Tests unveränderte Funktion überprüfen

Glossar

Applikationsserver

Ein Applikationsserver stellt für Entwickler Dienste wie Mehrbenutzerfähigkeit, Autorisierung und Authentifizierung bereit. Dafür müssen bei der Entwicklung bestimmte Standards eingehalten werden. Bekannte Applikationsserver gibt es z.B. von JBoss, Bea oder IBM.

Authentifizierung

Nutzer identifizieren sich zweifelsfrei z.B. mittels Nutzerkennung und Passwort gegenüber einem System.

Autorisierung

Nutzer bekommen unterschiedliche Rechte für den Zugriff auf Funktionalität eines Systems.

Benutzeroberfläche

Die Schnittstelle zwischen System und Nutzer, das „Gesicht“ einer Anwendung. In der Regel handelt es sich um eine graphische Benutzeroberfläche (GUI).

Domäne

Der Begriff Domäne bezeichnet das fachliche Umfeld, für das ein Softwaresystem entwickelt wird.

Embedded System

Eingebettete (engl. Embedded) Systeme sind spezialisierte Computersysteme, die in technischen Geräten integriert sind und dort zweckbestimmte Funktionen übernehmen, zum Beispiel die elektronische Steuerung einer Bremse in einem Auto. Meistens gelten hier Echtzeit-Anforderungen. Sie sind für die Endnutzer meist unsichtbar.

Generics

Generics ist die Bezeichnung für parametrisierte Klassen in Java. Das Pendant in C++ heißt Template-Klasse. Bei Generics können eine oder mehrere andere Klassen als Parameter einer Klasse angegeben werden. Dies

kommt z.B. bei Container-Klassen vor. Beispielsweise bedeutet in Java die Deklaration `LinkedList<Integer> eineListe`, dass `eineListe` eine Klasse vom Typ `LinkedList` ist, die `Integer`-Objekte enthält.

GUI

Abkürzung für Graphical User Interface. Englisch für Graphische Benutzeroberfläche.

Integration

In der Softwarearchitektur bezeichnet Integration die Einbindung von anderen Systemen, z.B. relationalen Datenbanken oder Host-Systemen. Dies kann in einem eigenen Tier, dem Integration-Tier geschehen. Für relationale Datenbanken werden heute in der Regel objektrelationale Mapper wie z.B. die Java Persistence API eingesetzt.

OCL

Abkürzung für Object Constraint Language. Es handelt sich um einen Teil der UML. In OCL können formalisiert in Textform Randbedingungen für Elemente von UML-Diagrammen beschrieben werden.

Persistenz

Unter Persistenz versteht man, dass Instanzen von Klassen dauerhaft gespeichert werden, so dass sie auch nach einem Programmneustart wiederhergestellt werden können. Dies geschieht typischerweise in Dateien oder in Datenbanken.

Schnittstelle

Grenze zwischen einem System und seiner Umwelt. Dies kann eine graphische Benutzeroberfläche sein, es kann sich aber auch z.B. um Webservices oder den Export von Files handeln.

Skalierbarkeit

Ein System ist dann skalierbar, wenn es durch zusätzliche Computer ohne Implementierungsaufwand mehr Leistung erbringt.

Stichwortverzeichnis

A

Abhängigkeit 6, 7
Abstraction 23
Abstraktion 43, 79
Aggregat 63
Aktivität 21
Aktivitätsdiagramm 21, 52
Allgemeine Hierarchie 28
Analyse 1, 23, 39, 49
Analysemuster 23
Änderbarkeit 73
Anforderung 1, 19, 39, 40, 49
Antimuster 23
Anwendungsfall 19, 32
Anwendungsfall-orientiertes
 Bedienkonzept 32
Applikationsserver 47
Architektur 23, 49, 53
Architekturmuster 42
Assoziation 6, 7, 13, 14, 49
Attribut 4, 13, 15, 49, 52
Aufgabenangemessenheit 36
Aufwandsschätzung 39

B

Barrierefreiheit 37
Baustein-Sicht 41
Benutzerführung 35
Benutzeroberfläche 10, 31
Beobachter 55
Beziehung 6
Bibliothek 10, 45
Bildschirmarbeitsverordnung 34

C

Client 44
Client-Server 42, 45
Content-Kopplung 77

D

Datadictionary 5
Data-Kopplung 77, 78
Datenbank 39, 44, 76
Datenhaltung 53
Datenstruktur 63
Datentyp 5, 15
Demeter 80
Design Patterns 55
Detailtiefe 54
Dialogablaufplan 32
Dialogentwurf 31, 67
DIN EN ISO 9241 36
dockbar 59
Domäne 9
Domänenmodell 9, 13, 32, 50, 53, 67
Dynamisches Modell 2, 19, 51

E

Embedded Anwendung 73
Embedded Systeme 3
Entwurf 2, 23
Entwurfsmuster 55
Ereignis 55
Erfahrung 72
Ergonomie 34, 36
Erwartungskonformität 36
Exemplartyp 23

F

Farbe 35
Fassade 64
Fehlertoleranz 36
Fenster 33
Framework 39, 40, 55
funktionale Anforderungen 40, 49, 53
Funktionsfähigkeit 73

G

Gang of Four 55

Geruch 81
Geschäftslogik 44, 76
globale Variable 77
graphische Benutzeroberfläche 31
Graphische Gestaltung 35
Guttag 79

H

handschriftlich 67
Hierarchie 29
HMVC 59
horizontale Schicht 44

I

Immutable 61, 77
Individualisierbarkeit 37
Informationsarchitektur 31
Instanz 4
Iterate to another artifact 54
Iterator 63, 79

J

Java Enterprise Edition 44

K

Klasse 4, 13, 50
Klassenbibliothek 55
Klassendiagramm 9, 52
Klassenmodell 54
Kontextdiagramm 41
Kontext-Sicht 40
Kopplung 74

L

Laufzeitsicht 41
Layer 43, 44
Lernförderlichkeit 37
Liskov 79
Liskovsches Substitutionsprinzip 9, 80

M

Machbarkeit 39
Mengengerüst 46

Methode 49, 52
Microsoft .NET 44
Model in small increments 54, 72
Model with others 54
Modell 2
Model–View–Controller 57
Muster 22
Mutable 62
MVC 57, 70, 72

N

Namenskonvention 70
Navigation 32
nicht-funktionale Anforderungen 40, 49

O

Object Constraint Language 5
Objekt 4, 50
Objektorientierte Modellierung 3
objektorientierter Entwurf 49
objektorientiertes Bedienkonzept 32
Observable 55
Observer 55, 72
Occurrence 23
OCL 5, 29
Operation 4, 13

P

Paket-Diagramm 51
Pattern 22
Peer-to-Peer 43, 45
philosophisches Problem 2, 4
Player – Role 26
Polymorphie 9
Primärdialog 32, 33
Priorität 40
Programmiersprache 39
Prototyp 34, 72
Prove it with code 41, 54, 72

Q

Quelltext 53

Querwirkung 77

R

Randbedingung 40

Refactoring 72, 81

Referenzarchitektur 40, 47

Rot-Grün-Sehschwäche 35

Routine Call-Kopplung 78

S

Schicht 43, 75

Schnittstelle 31

Schriftart 35

Schwächen im Entwurf 81

Sekundärdialog 32, 33

Selbstbeschreibungsfähigkeit 36

Sequenzdiagramm 52, 54

Sicht 40

Softwarearchitekt 39

Softwarearchitektur 39

Software-Ergonomie 34

Spezialisierung 8

Spiel 53

Stamp-Kopplung 77, 78, 79

Statisches Modell 2, 51

Steuerbarkeit 36

Styleguide 35

Swing 57

Systemidee 45

Szenario 19

T

Taktik 40

Tier 43, 44

Travel light 54

U

UML 3

Usability 34

User Experience 35

User Story 19, 67

Utility 76

V

Verantwortlichkeit 4, 13, 17, 18, 26, 43,
51

Vererbung 6, 8, 13, 16, 49

Verhalten 17

Verteilungssicht 41

vertikale Schicht 44

Vorgehen 53

W

Webanwendung 45, 53

Wechselnde Rollen 26

Wizard 33

Z

Zeichenprogramm 66

Zerlegung 74

zirkuläre Abhängigkeit 80

Zugriffsmethode 77

Zusammenhalt 75

Zustand 19, 20

Zustandsdiagramm 20

