

SOFTWARE ENGINEERING- KOMPAKT

anja
METZNER

HANSER

Anja Metzner
Software-Engineering - kompakt



Bleiben Sie auf dem Laufenden!



Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter:
www.hanser-fachbuch.de/newsletter

Anja Metzner

Software-Engineering

– kompakt

HANSER

Autorin:

Prof. Dr. Anja Metzner

Hochschule für angewandte Wissenschaften Augsburg
Fakultät für Informatik



Alle in diesem Buch enthaltenen Informationen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt geprüft und getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor(en), Herausgeber) und Verlag übernehmen infolgedessen keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Weise aus der Benutzung dieser Informationen – oder Teilen davon – entsteht. Ebenso wenig übernehmen Autor(en), Herausgeber) und Verlag die Gewähr dafür, dass die beschriebenen Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2020 Carl Hanser Verlag München;
Internet: www.hanser-fachbuch.de

Lektorat: Frank Katzenmayer

Herstellung: Anne Kurth

Satz: Anja Metzner

Titelmotiv: © istockphoto.com/Artush

Umschlagrealisierung: Max Kostopoulos

Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München

Druck und Binden: Friedrich Pustet GmbH & Co. KG, Regensburg

Printed in Germany

Print-ISBN: 978-3-446-45949-6

E-Book-ISBN: 978-3-446-46365-3

Vorwort



– *Guide to the Labyrinth of Software-Engineering* –

Das vorliegende Buch ist aus diesem Grund anders. Anstatt Anspruch auf Vollständigkeit zu erheben, werden hier die wichtigsten Themen rund um Software-Engineering erklärt, zusammengefasst und anhand von kleinen Praxisbeispielen vertieft.

Software-Engineering ist kurz umrissen das Wissensgebiet, wie qualitativ hochwertige Software erstellt und dieser Erzeugungsvorgang in Projekten erfolgreich durchgeführt werden kann. Dafür sind umfassende Kenntnisse über Standards, Methoden und Werkzeuge des Software-Engineerings notwendig. Die wichtigsten und anwendungsorientiertesten Kenntnisse will dieses Buch vermitteln.

Es richtet sich sowohl an Anfänger als auch an fortgeschrittene Leser. Die erste Zielgruppe erlangt durch die Lektüre schnelle Orientierung und kompaktes Grundwissen. Der zweiten Lesergruppe mag dieses Buch als strukturiertes Nachschlagewerk



und verdichteter Überblick über den aktuellen Stand dieses Fachgebiets dienen. Verwenden Sie zur schnelleren Recherche doch auch einfach gerne das umfangreiche Stichwortverzeichnis.

Verbesserungsvorschläge

Über Ihre Gedanken und Verbesserungsvorschläge zu diesem Buch freue ich mich.

Richten Sie Anfragen und Anregungen gerne an meine Email-Adresse:

anja.metzner@hs-augsburg.de

Lernfortschritt erkennen

Ich lade Sie ein, Ihren eigenen Lernfortschritt zu erforschen. Aus diesem Grund endet jedes Kapitel mit einem Abschnitt „Aufgabensammlung“. Die Antworten zu den Fragen sind im Anhang -A- zu finden.

Weiteres Übungsmaterial finden Sie auch auf meiner Webseite:

<https://www.hs-augsburg.de/Informatik/Anja-Metzner.html>

Informationen zum Buch

Ergänzungen und weitere Informationen zu diesem Buch können Sie ebenfalls unter *<https://www.hs-augsburg.de/Informatik/Anja-Metzner.html>* finden.

Danksagung

Dieses Buch ist meiner Familie gewidmet. Ohne sie wäre dieses Buchprojekt nicht entstanden. Besonderer Dank auch an meinen Kollegen Prof. Dr.-Ing. Christian Martin für die vielen sehr hilfreichen Tipps und die umfangreiche Unterstützung. Danke auch für die gute Zusammenarbeit mit den Mitarbeitern des Verlags. Prof. Dr. Alfred Holl verdanke ich viele fachliche Einsichten. Zuletzt danke ich sehr meinen Studenten der Hochschule Augsburg für die vielen inspirierenden Momente, durch die dieses Buch überhaupt erst möglich wurde.

Und nun wünsche ich allen Lesern viel Vergnügen und viele hilfreiche Informationen über das spannende Gebiet des Software-Engineerings!

Anja Metzner

Augsburg, im Dezember 2019

Inhalt

Vorwort	V
1 Einführung	1
1.1 Aufteilung dieses Buches	1
1.2 Überblick und Terminologie	2
1.2.1 Der Software-Lebenszyklus	5
1.2.2 Komplexität der Softwareentwicklung	6
1.3 Geschichtlicher Überblick und die Folgen der Software-Krise	8
1.4 Modellbildung zur Erstellung von Softwarearchitekturen	12
1.5 Der Software-Engineering-Spezialist	14
1.6 Zusammenfassung	15
1.7 Aufgabensammlung	15
2 Phasenübergreifende Verfahren	17
2.1 Vorgehensmodelle	17
2.1.1 Wasserfallmodell	19
2.1.2 Verbessertes Wasserfallmodell	20
2.1.3 V-Modell	22
2.1.4 Spiralmodell	24
2.1.5 Agiles Modell	26
2.2 Klassisches Projektmanagement	32
2.2.1 Projektplanung	33
2.2.2 Projektmanagement (Zeit-, Kosten- und Ressourcenplanung)	34

2.2.2.1	Zeitmanagement.....	36
2.2.2.2	Ressourcenplan	40
2.2.2.3	Kalkulation.....	41
2.2.2.4	Pufferzeiten, Ressourcenauslastung und Schätzung der Dauer von Tätigkeiten.....	41
2.3	Zusammenfassung	42
2.4	Aufgabensammlung	42
3	Planungsphase	45
3.1	Übersicht Planungsphase	45
3.2	Lastenheft.....	47
3.3	Aufwandsschätzung	48
3.4	Risikomanagement	55
3.5	Zusammenfassung	56
3.6	Aufgabensammlung	57
4	Definitionsphase	59
4.1	Überblick Definitionsphase	60
4.2	Pflichtenheft.....	62
4.3	Requirements-Engineering.....	64
4.3.1	Anforderungen	64
4.3.2	Anforderungsarten	64
4.3.2.1	Funktionale Anforderung.....	64
4.3.2.2	Nicht-funktionale Anforderung.....	65
4.3.2.3	Problembereichsanforderung.....	65
4.3.2.4	Benutzeranforderung	66
4.3.2.5	Systemanforderung.....	66
4.3.3	Qualitätsmerkmale für Anforderungen	66
4.3.4	Beschreibung von Anforderungen	67
4.3.4.1	Natürliche Sprache	67
4.3.4.2	Anforderungen in strukturierter Sprache	69
4.3.4.3	Anforderungen in grafischer Notation	70
4.3.5	Erhebung von Anforderungen.....	71
4.3.6	Anforderungsanalyse – Notation im Überblick	73

4.3.6.1	Anwendungsfalldiagramm (engl. Use-Case-Diagramm)	73
4.3.6.2	Diskussion von Use-Case-Diagrammen	79
4.3.6.3	Anwendungsfälle im Zusammenhang mit dem Software-Lebenszyklus.....	79
4.3.7	Validation von Anforderungen	82
4.3.7.1	Ziele guter Anforderungen	82
4.3.7.2	Prüfung von Anforderungen	82
4.3.8	Anforderungsmanagement	83
4.4	Zusammenfassung	84
4.5	Aufgabensammlung	86
5	Software-Design-Phase	87
5.1	Überblick	87
5.2	Ein durchgängiges Beispiel	89
5.3	Notationen.....	94
5.3.1	Strukturdigramme	96
5.3.1.1	UML-Klassendiagramm	96
5.3.1.2	UML-Komponentendiagramm	101
5.3.2	Verhaltensdiagramme	103
5.3.2.1	Struktogramme	103
5.3.2.2	UML-Aktivitätsdiagramme	105
5.3.2.3	UML-Sequenzdiagramm	113
5.3.2.4	UML-Zustandsdiagramm.....	118
5.4	Softwarearchitekturen	119
5.4.1	Subsysteme und Komponenten.....	122
5.4.2	Makroarchitekturen.....	124
5.4.2.1	Allgemeine Architekturen	124
5.4.2.2	Verteilte Architekturen	125
5.4.2.3	Adaptive Systeme	127
5.4.2.4	Andere Architekturen	128
5.4.3	Mikroarchitekturen	128
5.5	Strategien und Methoden.....	131
5.6	Software-Wiederverwendung.....	133
5.7	Zusammenfassung	136
5.8	Aufgabensammlung	137

6 Testphase – Verifikation und Validation.....	139
6.1 Grundlagen	139
6.2 Software-Testverfahren	143
6.2.1 Statische Testverfahren	144
6.2.2 Dynamische Testverfahren	146
6.2.2.1 White-Box-Techniken	146
6.2.2.2 Black-Box-Techniken.....	149
6.2.3 Diversifizierende Tests.....	152
6.3 Zusammenfassung	154
6.4 Aufgabensammlung	154
7 Wartungsphase	157
7.1 Grundlagen	157
7.2 Wartungsprozess.....	162
7.3 Wartungstechniken.....	164
7.3.1 Re-Engineering	164
7.3.2 Reverse-Engineering.....	166
7.4 Zusammenfassung	168
7.5 Aufgabensammlung	168
A Lösungen zur Aufgabensammlung	171
Literatur	181
Stichwortverzeichnis.....	187

1

Einführung

■ 1.1 Aufteilung dieses Buches

Dieses Buch ist, nach einer kurzen Einführung in diesem Kapitel, in die für das Software-Engineering wichtigen **Software-Lebenszyklusphasen** untergliedert. So enthält jedes Kapitel den Titel einer dieser Phasen. Neben einer sinnvollen Strukturierung des Buches bekommt der Leser dadurch auch einen zeitlichen Plan, wie ein Softwareentwicklungsprojekt in der Regel abläuft.

Zum besseren Überblick werden in Kapitel 2 vorab phasenübergreifende Verfahren besprochen und eingeschoben.

In Kapitel 3 wird Software-Engineering in der **Planungsphase** erörtert.

Das sogenannte „Requirements Engineering“ (auf Deutsch: Anforderungsanalyse) wird in Kapitel 4 – der **Definitionsphase** – genauer beleuchtet.

In Kapitel 5 folgt die Besprechung der Verfahren für die **Designphase** des Software-Lebenszyklus. Wie gute Softwarearchitekturen erdacht und gebaut werden können, wird hinterfragt.

Die nachfolgende Phase im Software-Lebenszyklus, die **Implementierung**, ist nicht Gegenstandes dieser Arbeit und kann in einschlägiger Literatur über Programmierung nachgelesen werden. Je nach Programmiersprache eignen sich hier unterschiedliche Bücher. Wer jedoch eine Übersicht über mehrere gängige Programmiersprachen sucht, der findet im „Taschenbuch Programmiersprachen“ ([HV07]) ein gutes Nachschlagewerk. Auch bei der Implementierung gibt es natürlich Themengebiete, die in den Bereich Software-Engineering fallen. Beispiele sind hier Versionsverwaltung, Konfigurationsmanagement oder auch die Methoden, um Softwarearchitekturen in sauberen Programmcode umzusetzen. Gute Literatur dazu kann gefunden werden bei Sommerville [Som18] oder beispielsweise zum Thema „Clean Code“ bei Robert C. Martin [Mar08].

Kapitel 6 widmet sich der **Test- und Abnahmephase** und damit den Qualitätssicherungsfragen eines Softwareprojektes.

Zuletzt wird in Kapitel 7 noch die **Wartung**, und damit die wichtigsten Wartungstechniken und Verfahren aus dem Software-Engineering-Bereich, besprochen.

■ 1.2 Überblick und Terminologie

Was ist Software-Engineering?

Dieses Buch widmet sich bekanntlich dem Thema **Software-Engineering**. Dabei handelt es sich um die Sammlung und Beschreibung von Erfolg versprechenden Verfahren, Methoden und Werkzeugen rund um den Prozess von qualitativ hochwertiger Software-Erstellung.



Definition

„The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.“

- IEEE Standard Glossary of Software-Engineering -[IEEE18]

Gegenstand des Software-Engineering ist die **ingenieurmäßige** Entwicklung **komplexer Softwaresysteme** hoher **Qualität** unter Berücksichtigung der einzusetzenden Arbeits- und Zeitressourcen.

Gebietseinordnung

Nach Broy gehört Software-Engineering zu den Ingenieurwissenschaften (siehe Abbildung 1.1) wie beispielsweise auch Elektrotechnik oder Maschinenbau. Diese Ingenieurwissenschaft bedient sich unter anderen der Grundlagenwissenschaften der Informatik, der Betriebswirtschaft und der Psychologie. Die Informatik liefert dabei alle technologischen Grundlagen für Softwaresysteme. Betriebswirtschaftliche Grundlagen sind notwendig, da Projekte betriebswirtschaftlichen Regeln (z. B. ein finanzieller Rahmen) unterliegen und Projekte meist auch in Unternehmen initiiert werden. Grundlagen aus der Psychologie sind erforderlich, um Teamarbeit im Rahmen von Software-Engineering überhaupt erst zu ermöglichen. Ein erfolgreicher Projektverlauf ist erst durch den Erwerb der sogenannten „Soft Skills“ (gemeint sind Fähigkeiten bezüglich Problemlösungen, Konfliktmanagement, Führungsmanagement, etc.) möglich.

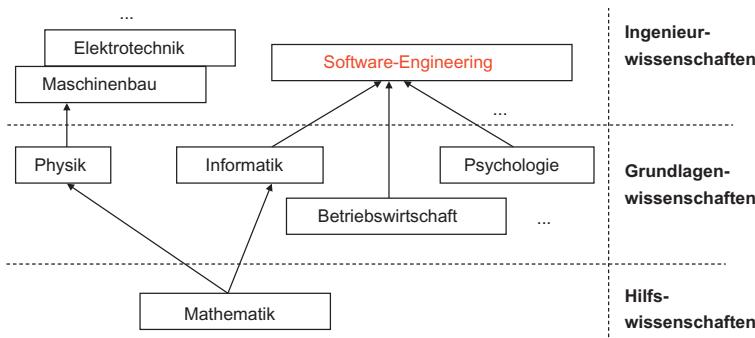


Abbildung 1.1 Gebietseinordnung nach Broy und Rombach [BR02, pp. 438–451]

Die hier verwendete Gebietseinordnung nach Broy wird in Fachkreisen dennoch kontrovers diskutiert, denn Viele sehen die Informatik selbst auch als Ingenieurwissenschaft an und Software-Engineering als Teil der Informatik. Auch verwendet ein Großteil der deutschsprachigen Fachwelt im Deutschen die Übersetzung „Softwaretechnik“ für Software-Engineering (z. B. Informatik-Handbuch [ZR06]).

Terminologie

In dieser Arbeit werden die nun genannten Begrifflichkeiten und Artefakte wie in Tabelle 1.1 verwendet.

Tabelle 1.1 Terminologie dieses Werkes

Begriff	Erklärung	Abbildung
Software	„Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.“ IEEE Standard Glossary of Software-Engineering [IEE18]	
Software-System	Ein System, dessen Subsysteme und Komponenten aus Software bestehen, ist ein Software-System.	
Software-Produkt	Ein Produkt ist ein in sich abgeschlossenes, i. A. für einen Auftraggeber bestimmtes Ergebnis eines erfolgreich durchgeföhrten Projekts (oder Herstellungsprozesses). Ein Software-Produkt ist ein Produkt, das aus Software besteht.	

Begriff	Erklärung	Abbildung
CASE	<p>Dies ist eine Abkürzung (engl.) für „Computer Aided Software-Engineering“ und bezeichnet einen Fachbegriff aus dem Software-Engineering. Gemeint sind hier meist die Werkzeuge, welche eingesetzt werden können, um Software-Engineering zu unterstützen. Beispiele sind UML-Tools wie Visual Paradigm (von Visual Paradigm International [Int19]), Rational Rhapsody Developer (von IBM [IBM19b]) oder Enterprise Architekt (von Sparx [Eur19]). Inhaltlich steht die Modellierung von Projektlösungen im Mittelpunkt. Beispiele hierfür sind:</p> <ul style="list-style-type: none"> ▪ UML ▪ Transformation in verschiedene Programmiersprachen ▪ Datenstrukturen ▪ Reverse Engineering ▪ Generierung von Prototypen. <p>Aber auch Projektmanagement, Prozessmanagement und viele weitere Entwicklungswerkzeuge sind oft zusätzlich mit im Paket.</p>	
UML [OMG19b]	<p>Es handelt sich um eine Abkürzung (engl.) für „Unified Modeling Language“. Die UML ist eine Modellierungssprache. „Die UML dient zur Modellierung, Dokumentation, Spezifizierung und Visualisierung komplexer Systeme, unabhängig von deren Fach- und Spezialisierungsgebiet. Sie liefert die Notationselemente gleichermaßen für die statischen und dynamischen Modelle von Analyse, Design und Architektur und unterstützt insbesondere objektorientierte Vorgehensweisen. ... Die 1989 gegründete OMG (Objekt Management Group) – ein Gremium mit heute 800 Mitgliedern – ist Hüterin dieses Standards. Das es sich bei Werken der OMG um herstellerneutrale Industriestandards handelt, gewährleistet die Teilnahme aller relevanten Marktvertreter (zum Beispiel Rational Software [IBM], Hewlett-Packard, Daimler AG, I-Logix, Telelogic, Oracle, Microsoft, ...).“ eine breite Unterstützung in der Industrie. (aus [RQSG12, S. 4])</p>	

Hinweis

Da dieses Buch eine kompakte Übersicht über Software-Engineering bietet und absichtlich viele Themen daher lediglich anschneidet, aber nicht vollständig diskutiert, wird hier und im Verlauf des Buches auf passende ausführlichere Literatur verwiesen. Ein Nachschlagewerk der ausführlichen Natur ist das Buch von Sommerville [Som18] und eine etwas unbürokratische, amerikanische Variante das Buch von Pressman [Pre14]. Zum guten Studium der Modellierungssprache UML sind die „UML Kurzreferenz“ [ÖS14] und „UML glasklar“ [RQSG12] hilfreich.

1.2.1 Der Software-Lebenszyklus

Im Fokus von Software-Engineering steht der sogenannte **Software-Lebenszyklus**. Gemeint ist damit der gesamte Prozess, der zur Erstellung und Erhaltung eines Softwaresystems führt. Bei jeder Art von Softwareerstellung läuft dieser Lebenszyklus ab. Er ist zur einfacheren Unterscheidung der einzelnen Tätigkeiten in sogenannte Phasen unterteilt (siehe Abbildung 1.2).

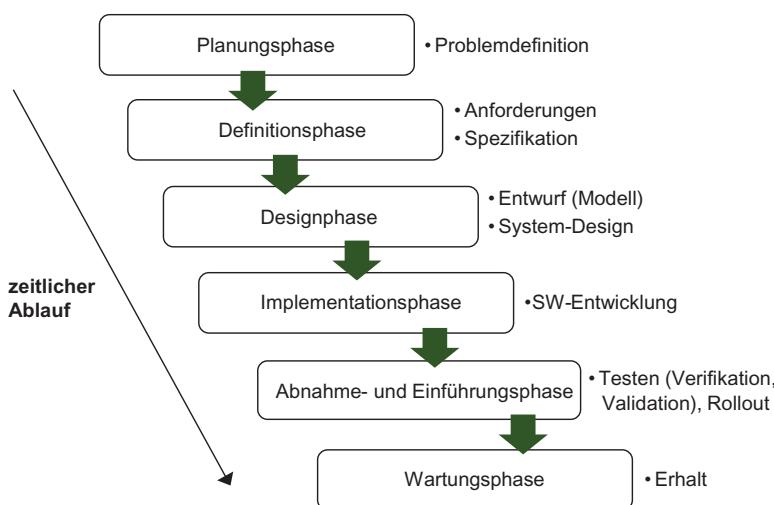


Abbildung 1.2 Software-Lebenszyklus

- **Planungsphase** (s. Kap. 3):
In dieser Phase wird ein Softwareprojekt neu aufgesetzt, Projektmanagement begonnen, das Lastenheft geschrieben und eine Problemdefinition erarbeitet.
- **Definitionsphase** (s. Kap. 4):
Nun werden die Anforderungen an eine Software eruiert und in einer Spezifikation dokumentiert.
- **Designphase** (s. Kap. 5):
Danach erfolgt der Entwurf und die Modellbildung für das Systemdesign der Software. Es entsteht die Softwarearchitektur.
- **Implementationsphase** (vgl. [HV07]):
In dieser Phase findet die eigentliche Programmierung der Software statt. Zugrunde liegt die in der vorherigen Phase gefundene Softwarearchitektur.
- **Abnahme-/Einführungsphase** (s. Kap. 6):
Um hohe Qualität zu gewährleisten, finden nun das Testen, die Verifikation, die Validation und die Markteinführung (engl. **Rollout**) der Software statt.
- **Wartungsphase** (s. Kap. 7):
Die Software wird in dieser Phase vor Software-Alterung (engl. Aging) bewahrt; in der Regel werden Fehler der Software gesammelt und wenn möglich behoben. Auch Änderungsanforderungen werden erhoben, um für die aktuelle oder die Folgeversion der Software Verbesserungen erzielen zu können.

Nach der Wartungsphase findet in der Regel entweder die Entwicklung von **Folgeversionen** der Software, manchmal auch deren **Neuentwicklung** oder die **Stilllegung** statt. Bei einer Stilllegung wird fallweise ein sogenanntes **End-of-Life-Management** durchgeführt.

1.2.2 Komplexität der Softwareentwicklung

Benutzer von Software erwarten heute einwandfrei funktionierende, effiziente Programme. Software, die nicht den Erwartungen entspricht, wird daher schnell verworfen und Konkurrenzprodukte werden eingesetzt. Softwareproduzenten haben daher großes Interesse daran, qualitativ hochwertige Programme herzustellen.

Warum fällt es schwer, gute Software zu entwickeln?

Grundsätzlich sind oft bereits kleine Softwareprojekte schwer beherrschbar. In der Regel sind es jedoch insbesondere die großen bzw. komplexen, qualitativ hochwertigen Softwareprojekte, die eine Vielzahl von Teilnehmern, angefangen vom Kunden über ein Team von Software-Engineering-Spezialisten, Entwicklern, Testpersonal

und Wartungsspezialisten, benötigen. Jeder Projektteilnehmer bringt unterschiedliche Erfahrungen, Verfahren und Methoden mit, die miteinander abgeglichen und verwendet werden müssen. Unterschiedliche wirtschaftliche Faktoren genauso wie gesetzliche Aspekte müssen Berücksichtigung finden. In internationalen Projekten müssen beispielsweise oft Ländergrenzen, unterschiedliche Kulturen und Sprachen überbrückt werden.

Auch haben verschiedenartige Projektteilnehmer auch unterschiedliche Ziele. Kunden- und Benutzerwünsche liegen oft weit auseinander. Ein Projektleiter hat eine andere Sichtweise auf sein Projekt als ein Entwickler.



Merke

Software-Engineering ist demzufolge eine Wissensquelle, von der alle Projektteilnehmer profitieren können, um ihre Ziele auch erreichen zu können.

Software-Engineering ist somit eine Art **Baukastensystem**, in dem **Standards**, **Verfahren** und **Methoden** angeboten werden, um einen möglichst erfolgreichen Projektverlauf zu gewährleisten. In dieses Baukastensystem sind das Wissen und die Projekt erfahrungen aus zahlreichen, erfolgreichen und nicht-erfolgreichen Vorprojekten eingeflossen. Es bietet erfolgsorientierte Lösungsmöglichkeiten für den gesamten Ablauf eines Softwareprojektes an.

Eigenschaften von Software

Welche Eigenschaften von Software erschweren die Erstellung von qualitativ hochwertiger Software?

Software ...

- ist **immateriell**
- wird nicht durch physikalische Gesetze begrenzt
- eine zugrunde liegende Modellbildung ist schwierig
- Anforderungen sind oft unzureichend geklärt, dokumentiert oder spezifiziert
- kann Defekte enthalten, Beispiele sind: Konstruktionsfehler, Spezifikationsfehler oder Portierungsfehler
- Ersatzteile gibt es nicht (nur evtl. sogenannte (engl.) „Patches“ (d. h. ein Austausch von Teilstücken des Softwarecodes))
- ist schwer zu vermessen (Was ist gute und was ist schlechte Software?)
- ist leicht änderbar
- hat keinen Verschleiß, aber Software-„Aging“ (auf Deutsch: Alterung)
- Veränderungen sind fortlaufend nötig, um Software lauffähig zu halten.

Schwierigkeiten bei der Software-Erstellung

Schwierigkeiten bei der Software-Entwicklung gibt es aufgrund deren Komplexität viele! Deshalb folgt hier keine vollständige Liste der zu bewältigenden Problematiken, sondern ein exemplarischer Ausschnitt möglicher Komplikationen:

- Kommunikationsprobleme mit Projektbeteiligten:
Wenig Wissen über die Anwendung bei Software-Engineering-Spezialisten und Entwicklern; Anwender hat unklare Vorstellungen des Systems
- Arbeitsabläufe werden durch Software oft verändert:
Akzeptanz- und Integrationsprobleme
- Software-Varianten:
Konfiguration und Versionierung gestaltet Software viel komplexer
- Software Einsatz in verschiedenen Umgebungen:
Portabilitätsprobleme.

Was kann man nun tun, um die genannten Probleme zu minimieren?

Die Antwort, die Software-Engineering-Spezialisten geben, lautet in etwa wie folgt:

Abhilfe schafft die Verwendung von:

- **Standards**
Beispiel: Planen der Software durch Modellbildung
- **Methoden**
Beispiel: Einsatz von Projektmanagement
- **Werkzeuge**
Beispiel: Verwendung von UML-Tools, welche die Zeichenarbeit bei der Erstellung der Softwarearchitektur vereinfachen.

Genau diese Sammlung und Beschreibung der Standards, Methoden und Werkzeuge stehen im Fokus von Software-Engineering und damit auch im Mittelpunkt dieser Arbeit. Aus diesem Grund enthalten die weiteren Kapitel die kompakte Übersicht über die gebräuchlichsten und praxisrelevanten Verfahren dieses Fachgebiets.

■ 1.3 Geschichtlicher Überblick und die Folgen der Software-Krise

In den fünfziger Jahren entstanden die ersten benutzbaren, jedoch teuren Computer, die Makros und bald Prozeduren niederer Programmiersprachen verstanden

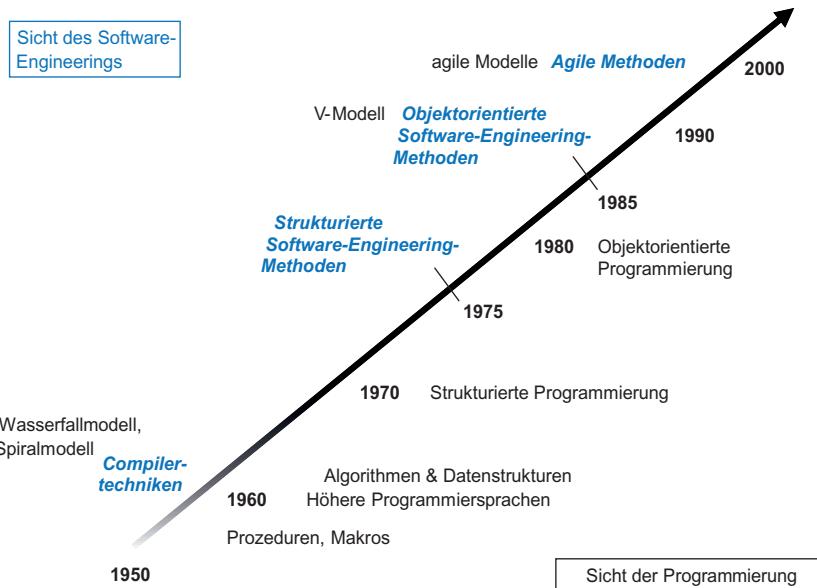


Abbildung 1.3 Software-Engineering-Zeitlinie

(siehe Abbildung 1.3). Diese funktionierten zunächst mit Lochkartentechnologie. Später folgten in den sechziger Jahren höhere Programmiersprachen, die mit ersten Compilern übersetzt wurden. In dieser Zeit überholten erstmals die **Kosten für Software**, durch gestiegene Komplexität und steigenden Umfang, die **Kosten für die Hardware**. Erste Software-Engineering-Methoden kamen auf. In dieser Zeit entstanden beispielsweise Vorgehensmodelle wie das Wasserfallmodell (publiziert von Royce 1970) (siehe Details in Kapitel 2.1.1). Algorithmen und Datenstrukturen wurden stetig komplexer und die strukturierte prozedurale Programmierung wurde zum Standard. Dies erforderte auch strukturierte Software-Engineering-Methoden wie beispielsweise das V-Modell (beschrieben in Kapitel 2.1.3) oder das Spiralmodell (publiziert von Böhm, 1988, IEEE) (siehe Kapitel 2.1.4). Als in den achtziger Jahren die objektorientierte Programmierung populär wurde, schlug sich das im Software-Engineering durch die Erfindung von grafischen Notationen, wie der UML (engl. Unified Modelling Language) zur Modellierung der Software, nieder.

Moderne und heute vielseitig eingesetzte Software-Engineering-Standards sind agile Methoden wie „Scrum“, „Extreme Programming“, agil-unterstützende Varianten wie „V-Modell XT“ und ähnliche (siehe Kapitel 2.1.5 – Agiles Modell). Agil heißt hier, dass Anforderungen für eine Software heutzutage jederzeit Änderungen unterworfen sein

können und das auf geänderte Kundenwünsche sehr viel schneller reagiert wird als mit anderen früher üblichen Vorgehensweisen. Mit den wichtigsten Anforderungen wird begonnen und die Lagerhaltung von Anforderungen wird möglichst minimiert. Dadurch können Anforderungen leichter später spezifiziert oder verändert werden.

Die Software-Krise

Als ein Phänomen der Zeit entstand in den sechziger Jahren die sogenannte „Software-Krise“. Erstmals erkannte man hier, dass die Kosten für die Software die Kosten für die Hardware übersteigen und dass dieser Trend durch steigende Komplexität und Umfang der Software anhalten wird. Erste große Softwareprojekte scheiterten und verursachten große monetäre Schäden.

Der Grund dafür war, dass die bisher genutzten undefinierten, undokumentierten Software-Engineering-Techniken nicht mit dem Umfang und der Komplexität der Software mithalten konnten.



Merke

Auf einer NATO-Tagung (1968) wurde das Phänomen „Software-Krise“ diskutiert und als Reaktion der Begriff des **Software-Engineering** eingeführt.

Beispiele für sicherheitskritische Softwareprojekte mit Problemen gibt es viele; gestern wie heute. Genannt seien hier drei bekannte Vertreter zur Verdeutlichung der Relevanz des Einsatzes von Software-Engineering:

- Verstrahlung von Patienten durch Therac 25 (1985–87) u. a. durch fehlerhafte Prozesssynchronisation bei Messwerterfassungen
- Explosion der Ariane 5-Rakete (1996) u. a. durch den Speicherüberlauf eines Zählers
- Scheitern der Mars-Spirit-Mission (1999) durch einen Einheitenfehler im Navigationssystem.

Eine bekannte Publikation, die gerne zur Untermalung der Notwendigkeit für Software-Engineering herangezogen wird, ist die Standish Group CHAOS-Studie (siehe Abbildung 1.4), bei der in der letzten Version über 10.000 IT-Projekte im Zeitraum zwischen 2011–2015 betrachtet wurden [SG15]).

Es zeigte sich, dass die Komplexität von Software-Projekten oft schwierig zu meistern ist. Die Gründe des Scheiterns von Softwareprojekten sind vielfältig. So gelin-

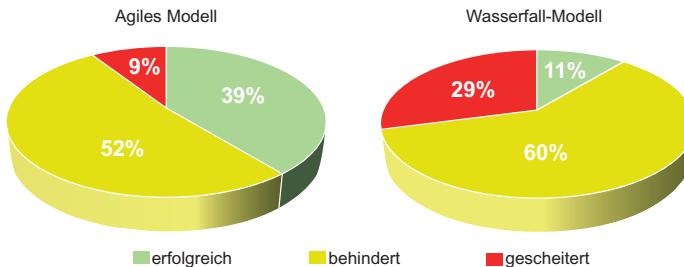


Abbildung 1.4 Standish Group CHAOS-Studie 2011–2015 (>10.000 Software-Projekte jeder Größe)

gen auch heute im Mittel nur 11–39 % der IT-Projekte. Schon 52–60 % werden zumindest als „herausfordernd“ beschrieben und 9–29 % gelten gar als gescheitert. Dabei schneiden Projekte mit zugrunde liegender agiler Vorgehensweise (siehe Kap. 2.1.5) etwas besser ab als Projekte, die dem Wasserfallmodell (siehe Kap. 2.1.1) unterliegen. Dies ist gewiss darauf zurückzuführen, dass Änderungsanforderungen bei der agilen Vorgehensweise schnell und einfach integrierbar sind. Agiles Vorgehen wird daher derzeit als das modernste Vorgehensmodell angesehen.



Merke

Weniger als die Hälfte aller Software-Projekte wird, durch Studien belegt, als „erfolgreich“ charakterisiert.

Dies rechtfertigt in hohem Maße den Einsatz von Software-Engineering!

Aus diesem Grunde werden in der Fachwelt folgende **Schlussfolgerungen** aus der Software-Krise gezogen:

- Früher war Software-Entwicklung ähnlich wie der Bau von Häusern **ohne** Architekten, Pläne und Maschinen.
- Software-Entwicklung ist keine kreative Kunst.
(Die essenziellen **Ideen** für **gute Produkte und deren Anforderungen** zu haben ist jedoch sehr wohl **kreativ!**).
- Software-Entwicklung ist demnach hauptsächlich eine **ingenieurmäßige Wissenschaft mit wohldefinierter Vorgehensweise**.

■ 1.4 Modellbildung zur Erstellung von Softwarearchitekturen

Beim Software-Engineering geht es darum, gute Modelle **vor** der eigentlichen Implementierung von Software zu erstellen. Daraus ergeben sich für Interessierte meist die folgenden Fragen:

- **Warum überhaupt sollen Modelle erstellt werden?**

Antwort: Um die reale Welt besser zu verstehen.

Forschung auf den Gebieten der Neurowissenschaften, der Hirnforschung und der Psychologie ergaben, dass wir Menschen die Komplexität der Realität durch Abstraktion und Vereinfachung in Form von Modellen besser verstehen.

- **Was für Modelle werden erstellt?**

Antwort: In diesem Fall handelt es sich um Modelle rund um die zu erstellende Software. In einem ersten Modell handelt es sich häufig um Darstellungen von bestehenden Geschäftsprozessen in Unternehmen. Die Beschreibung dieser Geschäftsprozesse wird wiederum dem Gebiet der Wirtschaftsinformatik zugeschrieben. Es folgen schließlich technischere Modelle wie Anwendungsfalldiagramme (siehe Kapitel 4.3.6.1), Aktivitätsdiagramme (siehe Kapitel 5.3.2.2) oder Klassendiagramme (siehe Kapitel 5.3.1.1).

- **Wie werden die Modelle erstellt?**

Antwort: Mit Methoden des Software-Engineerings werden die Modelle erstellt. Heutzutage ist die Modellierung mithilfe der UML ein Standard auf diesem Gebiet (siehe auch Kapitel 5 – Designphase).

Poppers drei Welten

Sir Karl Raimund Popper (*1902; †1994) war ein österreichisch-britischer Philosoph, der mit seinen Arbeiten u. a. zur Wissenschaft der Erkenntnistheorie beitrug. Er beschrieb, dass sich ein Mensch (d. h. hier ein Informatiker) in drei „Welten“ bewegt (siehe Abbildung 1.5).

Die erste Welt ist die Realität, wie sie wirklich existiert. In Abbildung 1.5 ist beispielsweise ein Unternehmen in der Realität abgebildet. Durch Sinneswahrnehmungen werden wir uns als Informatiker der Realität bewusst. Wir speichern ein Bild des Wissens über das Unternehmen in unserem Bewusstsein ab. Dadurch, dass wir uns über die Realität bewusst geworden sind, müssen wir uns aber auch im Klaren darüber sein, dass wir die Realität im Gegenzug auch manipulieren können. Das Bild, das

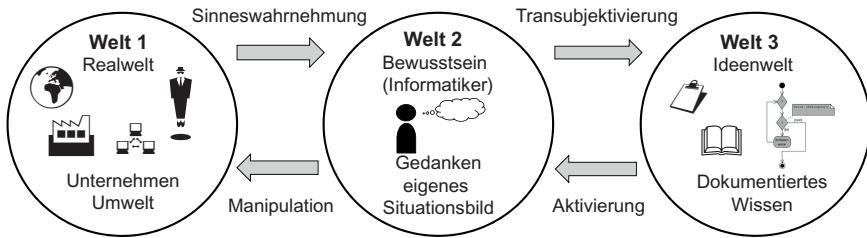


Abbildung 1.5 Poppers drei Welten [HS99]

wir uns in Welt 2 von der Realität aus Welt 1 erstellt haben, transferieren wir nun als dokumentiertes Wissen in die Welt 3, die sogenannte Ideenwelt. Welt 3 repräsentiert somit alles aufgeschriebene, dokumentierte Wissen über die Realität der Welt 1. Durch die Dokumentation unseres Wissens in Welt 3 aktivieren wir wiederum unser Bewusstsein in Welt 2 und arbeiten an der Vervollkommnung unseres Wissens in Welt 2.



Merke

Popper lehrt uns mit seinem 3-Welten-Kreislauf wichtige Prozesse und Effekte für die Modellierung von Software. Derjenige, der sich dieses Kreislaufs bewusst ist, wird bessere Modelle im Software-Engineering erstellen können.

Bedeutung von Modellen für das Software-Engineering

Aus den bereits genannten Fragestellungen ergibt sich folgende Bedeutung von Modellen im Software-Engineering:

- Terminologische **Unterscheidung** zwischen **Realität** und **Modellen** ist wichtig
 - **Reales Objekt** und **Modell-Objekt** (z. B. repräsentiert durch Objektorientierte Klasse) werden unterschieden
- Modelle sind für das Verständnis komplexer Abläufe notwendig
 - Es werden hier Methoden wie „**Vereinfachung**“ und/oder „**Abstraktion**“ angewendet
- Je **ähnlicher** ein Modell einem realen Objekt wird, **desto besser ist das Modell**
 - **Abwägungen** zwischen Kosten/Zeit und Nutzen sind notwendig.

■ 1.5 Der Software-Engineering-Spezialist

Software-Engineering-Spezialist ist die Bezeichnung eines Menschen, der **systematisch** Software entwirft und implementiert. Experte zu werden, erfordert sicherlich einen lebenslangen Lernprozess. In der Industrie können hierfür verschiedene Zertifikate erworben werden.

Die **Aufgaben** und die benötigten **Kenntnisse** eines Software-Engineering-Spezialisten werden der Übersichtlichkeit halber in vier Teilbereiche untergliedert:

1. Software-Produkt

- Erfahrungen im Anwendungsgebiet
- Kenntnisse über „Usability“ (engl.: Benutzbarkeit) und Ergonomie
- Anfertigung der Softwarearchitektur, von Modellen und Plänen

2. Software-Prozess

- Beherrschung und Kenntnisse des Prozessablaufs
- Kenntnisse einer Vielzahl von Methoden zur Meisterung von Software-Prozessen
- Beherrschung der Software-Entwicklung
- Kenntnisse der Wiederverwendung von Software

3. Ressourcen

- Teamfähigkeit, Team-Management
- Kenntnisse über Software, Hardware und Plattformen
- Kenntnisse über Hilfsmittel (z. B. Werkzeuge)

4. Projekt

- Beherrschung von Projektformen
- Projektmanagementfähigkeiten
- Kenntnisse über Schätzverfahren.

Durch diese Vielfältigkeit ist der Beruf des Software-Engineering-Spezialisten sicherlich einer der interessantesten im IT-Bereich.

■ 1.6 Zusammenfassung

In diesem Kapitel wird ein Überblick über Software-Engineering gegeben und die verwendete Terminologie besprochen. Außerdem wird eine Gebietseinordnung für Software-Engineering diskutiert und der Software-Lebenszyklus erläutert. Der Software-Lebenszyklus wird durch Software-Engineering-Verfahren unterstützt und verbessert.

Es wird erörtert, warum es schwer ist, gute Software zu schreiben und auch die Schwierigkeiten bei der Software-Erstellung werden thematisiert. Zusammenfassend kann gesagt werden, dass Abhilfe gegen ein chaotisches oder eher zufälliges Vorgehen bei einem Software-Entwicklungsprozess durch Verwendung von Standards, Methoden und Werkzeuge aus dem Software-Engineering geschaffen werden kann.

Ein geschichtlicher Überblick über die Entstehung des Begriffs „Software-Engineering“ und ein Einblick in den Ursprung der sogenannten Software-Krise wird gegeben. Eine Schlussfolgerung aus der Software-Krise ist, dass qualitativ hochwertige Software-Entwicklung durch Modellbildung unterstützt wird. Heutiger Standard zur Modellbildung in der Informatik ist die Verwendung der Modellierungssprache UML. Zuletzt werden Aufgaben und benötigte Kenntnisse eines Software-Engineering-Spezialisten beleuchtet, um Kerngebiete des Arbeitsfeldes dieses spannenden Berufes aufzuzeigen.

■ 1.7 Aufgabensammlung

In jedem Unterkapitel werden Fragen und Aufgaben vorgestellt, welche nach dem Lesen des jeweiligen Kapitels beantwortet werden können. Dies soll dem Leser eine eigenständige Lernzielkontrolle ermöglichen.

1. Was ist Software-Engineering und warum ist die Erstellung von qualitativ hochwertiger Software so schwer?
2. Wie heißen die Software-Lebenszyklus-Phasen?
3. Nennen Sie einige wichtige und nützliche Verfahren, die dazu beitragen, Software mit Qualität zu produzieren

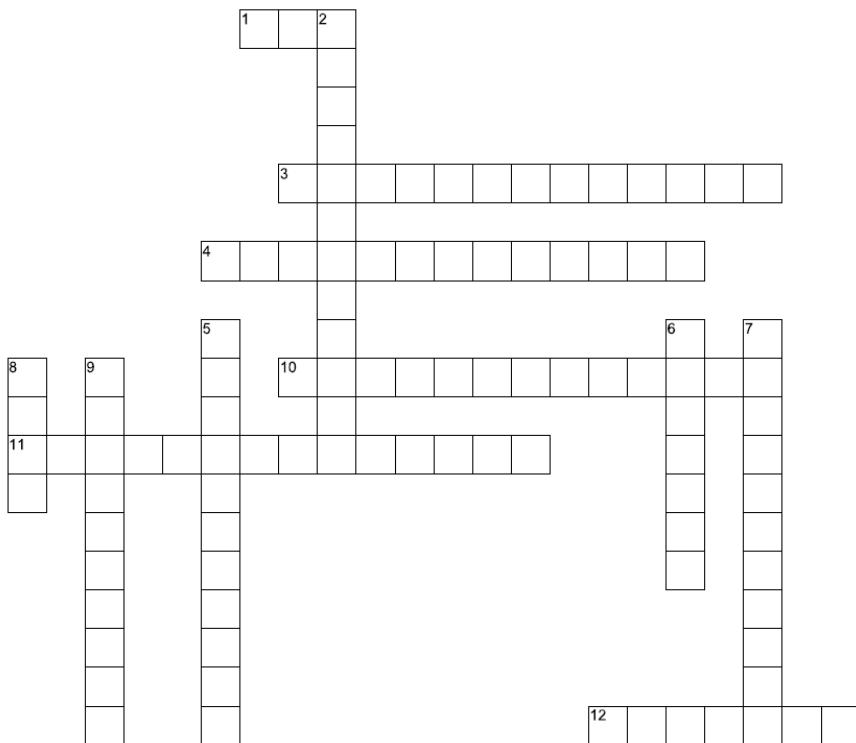
4. Und hier für das Training von Fachbegriffen ein Kreuzworträtsel:

▪ **Horizontal:**

1. Grafische Modellierungssprache zur Spezifikation, Konstruktion und Dokumentation von Software
3. Was ist der Hauptgegenstand im Requirements-Engineering?
4. Wichtiges Ergebnis der Definitionsphase
10. Einheit im Softwarelebenszyklus
11. Ein System, dessen Subsysteme und Komponenten aus Software bestehen
12. Form, in der Softwareerstellung stattfindet.

▪ **Vertikal:**

2. Was durchläuft jede Software?
5. Haupteigenschaft von Software
6. Einheit im Softwarelebenszyklus
7. Einheit im Softwarelebenszyklus
8. Abkürzung für Computerunterstützung im Software-Engineering
9. Grundlagenwissenschaft für Software-Engineering.



2

Phasenübergreifende Verfahren

In diesem Kapitel werden phasenübergreifende Verfahren des Software-Engineerings diskutiert. Darunter fällt beispielsweise die Entscheidung in einem Projekt für ein sogenanntes Vorgehensmodell (siehe Kapitel 2.1) bei der Software-Erstellung. Auch Techniken des Projektmanagements sind in heutigen IT-Projekten nicht mehr wegzudenken (siehe Kapitel 2.2). Die wichtigsten Zeit-, Kosten- und Ressourcenmanagement-Techniken werden daher erklärt.

■ 2.1 Vorgehensmodelle

In den Jahren nach 1960, als erkannt wurde, dass Prozessmodelle für die Erstellung von Software-Projekten benötigt werden, beherrschte man die Produktion von anderen Gütern (wie z. B. Bleistifte oder Autos) schon perfekt. Naheliegend war daher, trotz der fundamental anderen Eigenschaften von Software (siehe Seite 7), einen Vergleich von Software als eine Art von „Produktion“ anzustellen (siehe Abbildung 2.1).

Unter dem Begriff des „Vorgehensmodells“ wurden im Lauf der Zeit viele solcher Prozessmodelle vorgeschlagen. Die bekanntesten werden in diesem Buch nun behandelt.

Zu Beginn eines IT-Projektes entscheidet die Projektleitung, welches Vorgehensmodell für das bevorstehende Projekt eingesetzt wird.

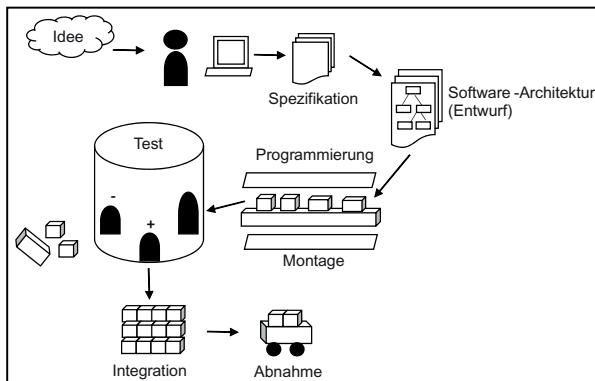


Abbildung 2.1 Vorstellung der „Software-Fabrik“ als Fließbandfabrik (vgl.[Wie90])



Merk

Ein Vorgehensmodell ...

- ist die abstrakte Darstellung – das Modell – für das Vorgehen während des Prozesses des Software-Engineerings
- stellt die Lebenszyklen von Software dar
- legt Aktivitäten fest
- gibt eine Reihenfolge der Aktivitäten vor
- die Aktivitäten werden zu Phasen zusammengefasst
- jede Phase hat wiederum ...
 - Phasenziele
 - Aktivitäten und Rollenzuordnungen
 - Dokumentation
 - Methoden, Richtlinien, Konventionen, Werkzeuge und Sprachen
 - vorgesehene/erlaubte Phasenübergänge.

Der Einsatz eines Vorgehensmodells gilt auch heute noch als obligatorisch und wird auch aus moderner Sicht nur als vorteilhaft betrachtet.

Vorteile

Wird ein Vorgehensmodell eingesetzt, hat dies folgende positive Aspekte:

- Vorhandensein eines Leitfadens für die Systementwicklung
- Gemeinsame und verbindliche Sicht der logischen und zeitlichen Struktur eines Projekts
- Anleitung für projektbegleitende Dokumentation

- Verbesserte Planbarkeit
- Möglichkeit der Zertifizierung
Software-Hersteller lassen sich falls möglich von unabhängigen Zertifizierungsstellen (z. B. TÜV) zertifizieren und haben dadurch gegenüber Kunden ein gutes Verkaufsargument mehr
- Höhere Personenunabhängigkeit
Personalausfall in Projekten kann somit einfacher vom Management ausgeglichen werden
- frühzeitige Fehlererkennung durch festgeschriebene Testaktivitäten.

Unterschiedliche Arten von Vorgehensmodellen

Alle Vorgehensmodelle beinhalten **immer** sämtliche **Lebenszyklusphasen** (siehe Abbildung 1.2 auf Seite 5). Jedoch wurden im Laufe der Jahre mehrere, auch heute noch im Einsatz befindliche Vorgehensmodelle entworfen, deren Unterschiede und Gemeinsamkeiten im Folgenden diskutiert werden.

2.1.1 Wasserfallmodell

Beschreibung

Das Wasserfallmodell ist das älteste und bekannteste Vorgehensmodell, das gleichwohl auch heute noch vielfältig im Einsatz ist. Dabei wird jede Phase des Software-Lebenszyklus (siehe Abbildung 2.2) strikt nacheinander abgearbeitet.

Aus jeder Phase gehen definierte Dokumente hervor, die begutachtet und abgenommen werden (beispielsweise durch „Reviews“ (siehe Kapitel 6.2.1)). Sind alle

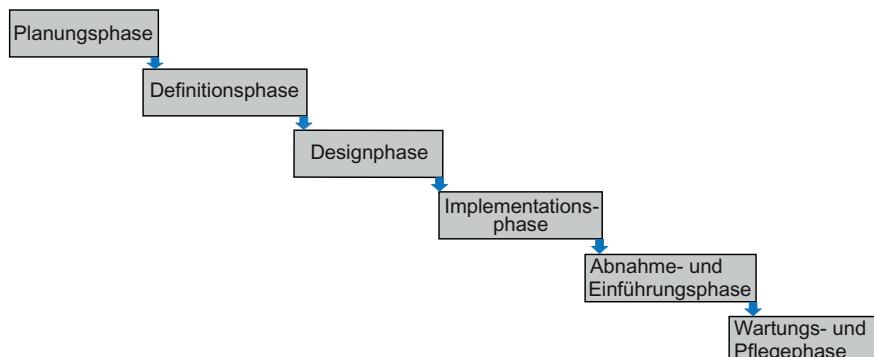


Abbildung 2.2 Wasserfallmodell nach Royce (1970)

Dokumente einer Phase abgenommen, so wird in die nächste Phase gewechselt. Vorgängerphasen müssen somit abgeschlossen sein. Projektteilnehmer haben in der Regel Rollen (z. B. Projektleiter, Software-Engineering-Spezialist oder Entwickler).

Praxis

- Einsatz in eher kleinen Projekten mit eher wenigen Teilnehmern
- Projekte, in denen wenige Anforderungsänderungen zu erwarten sind (vorab gut definierbare, überschaubare Projekte)
- Projekte, in denen keine Zyklen durch die verschiedenen Lebenszyklusphasen zu erwarten sind
- Projekte, in denen eine strikte Vorgehensweise (d. h. rigorose Einhaltung der Reihenfolge des Entwicklungsprozesses) bevorzugt wird
- Klar abgrenzbare Projekte mit festen Budgetgrenzen, eher fixer Anzahl an Teilnehmern und damit gut planbarem Ablauf.

Vorteile

- Intuitiver, linearer Prozess: einfach zu verstehen, klarer Ablauf
- Nicht unterbrechbarer Prozess
- Top-down-Vorgang
- Gute Planbarkeit.

Nachteile

- Starre Aufteilung in die Phasen
- Keine Rückkopplung auf frühere Phasen möglich
- Verpflichtungen zu frühen Zeitpunkten
- Neu dazu kommende Anforderungen nicht integrierbar
- Entspricht häufig nicht den Notwendigkeiten in der Praxis.

2.1.2 Verbessertes Wasserfallmodell

Beschreibung

Verbesserte Wasserfallmodelle traten erstmals kurz nach der Verwendung des traditionellen Wasserfallmodells auf. Die Praxis zeigte, dass Kunden in den meisten Fällen nachträglich mit neuen Anforderungen oder Prozessänderungswünschen an die Software-Hersteller herantraten. So ist der Wunsch hin zu der Wiederholbarkeit der

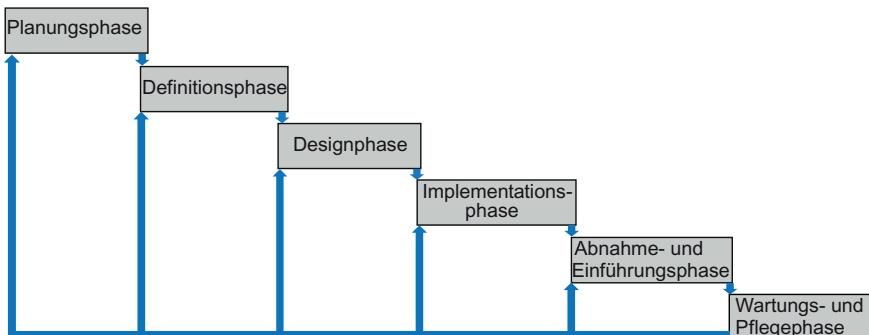


Abbildung 2.3 Verbessertes Wasserfallmodell – iterativ

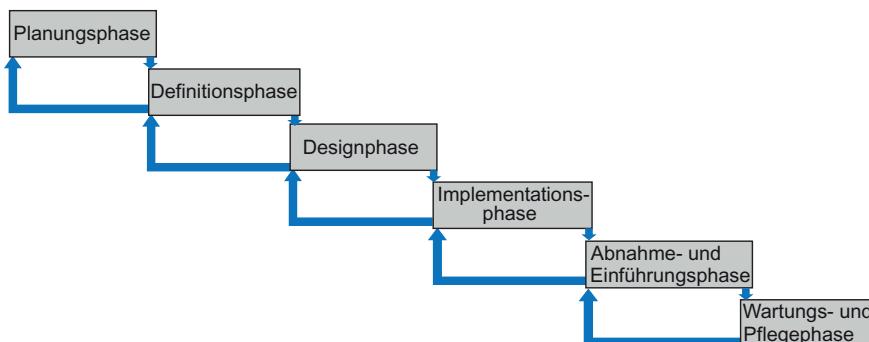


Abbildung 2.4 Verbessertes Wasserfallmodell – inkrementell

einzelnen Lebenszyklusphasen schnell zu verstehen und zu erklären. Deshalb funktionieren die verbesserten Wasserfallmodelle grundsätzlich zwar nach den Grundsätzen des traditionellen Wasserfallmodells, einzelne Phasen sind bei Bedarf jedoch entweder inkrementell oder iterativ wiederholbar (siehe Abbildung 2.3 und Abbildung 2.4). **Iterativ** bedeutet, dass ein Projekt zunächst alle Phasen des Lebenszykluses durchlaufen muss, anschließend jedoch in eine beliebige Phase zurückgesprungen werden kann.

Bei einer **inkrementellen** Vorgehensweise ist es möglich, nach Ablauf einer Phase in beliebige Vorgängerphasen zurückzuspringen. Ein Projektverantwortlicher sollte sich jedoch bei beiden Varianten im Klaren sein, dass jede Wiederholung von Phasen im Budget berücksichtigt werden muss.

Praxis

- Wie bei dem traditionellen Wasserfallmodell können Änderungswünsche integriert werden, wenn Budget und Ressourcen die Wiederholung und Überarbeitung ganzer Phasen zulassen.
- Zu berücksichtigen ist, dass viele Wiederholungen von Phasen häufig teuer werden.

Vorteile

- Intuitiver, linearer Prozess: einfach zu verstehen, klarer Ablauf
- Wiederholbarer Prozess
- Top-down-Vorgang
- Gute Planbarkeit
- Ausbau des Produkts in mehreren Versionen möglich.

Nachteile

- Starre Aufteilung in die Phasen
- Verpflichtungen müssen zu einem frühen Zeitpunkt eingegangen werden (neu dazukommende Anforderungen sind aber integrierbar).

2.1.3 V-Modell

Beschreibung

Das V-Modell (s. [We18]) verwendet als Grundlage ein verbessertes Wasserfallmodell, in dem einzelne Lebenszyklusphasen grundsätzlich wiederholbar sind. Gleichzeitig wird in jeder Phase zwingend auch die zugehörige Testphase definiert, die zu gegebener Zeit abgearbeitet werden muss. Vorteilhaft ist daran, dass dadurch in der Regel qualitativ hochwertigere Software entsteht. Diese Regelung lässt im grafischen Überblick des V-Modells (siehe Abbildung 2.5) die typische namensgebende V-Form entstehen.

Aktivitäten und Ergebnisse sind definiert. An den Phasenenden findet keine strikte Phasenabnahme statt. Teilnehmern werden vorgegebene Rollen zugeordnet.

Das V-Modell ist häufig in administrativer Umgebung und bei großen Unternehmen vorzufinden. Verwendet werden heutzutage eher die neueren Varianten *V-Model 97* und *V-Model XT*, in denen agile Vorgehensweisen ermöglicht werden.

Dokumente werden im V-Modell als Produkte bezeichnet. Jedes Produkt durchläuft wiederum definierte Zustände (Abbildung 2.6). Als Aktivitäten werden die Tätigkeiten bezeichnet, welche die Produkte verändern.

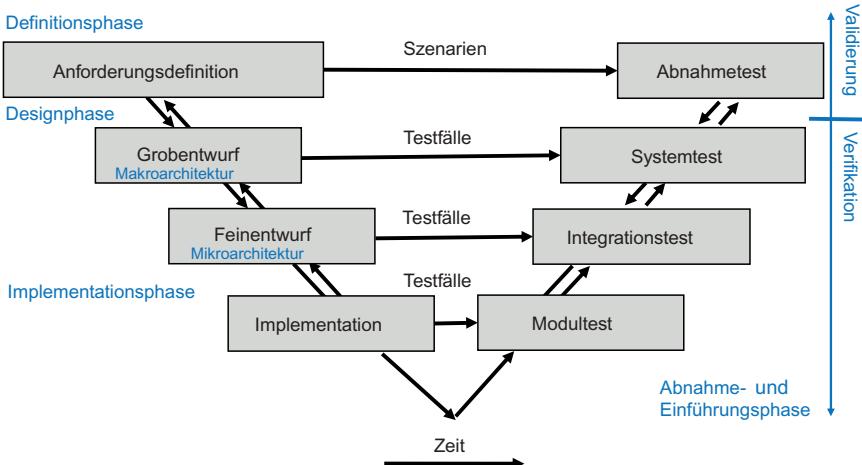


Abbildung 2.5 V-Modell

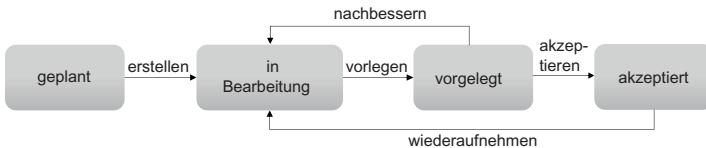


Abbildung 2.6 Produktzustände im V-Modell

Tailoring

Ein Merkmal für moderne Formen des V-Modells ist das sogenannte Tailoring. So wird die Eigenschaft genannt, dass dieses Vorgehensmodell anpassbar ist (s. Abbildung 2.7). Der modulare Aufbau von Produktmodell, Rollenmodell und Prozessabläufen erlaubt den Ein- und Ausbau von Vorgehensbausteinen für ein spezifisches Projekt. Der Vorteil besteht daher darin, dass wie aus einem kompletten Baustein-Kasten (Vorgehensmodell) die relevanten Bausteine (d. h. die Aktivitäten) ausgewählt werden können.

Praxis

- Das Vorgehensmodell wird häufig verwendet für die Softwareentwicklung in staatlichen Organisationen und Behörden.
- Gut für sicherheitsrelevante Projekte wegen der integrierten Testaktivitäten
- Bausteinartiger Aufbau ist praxisrelevant.

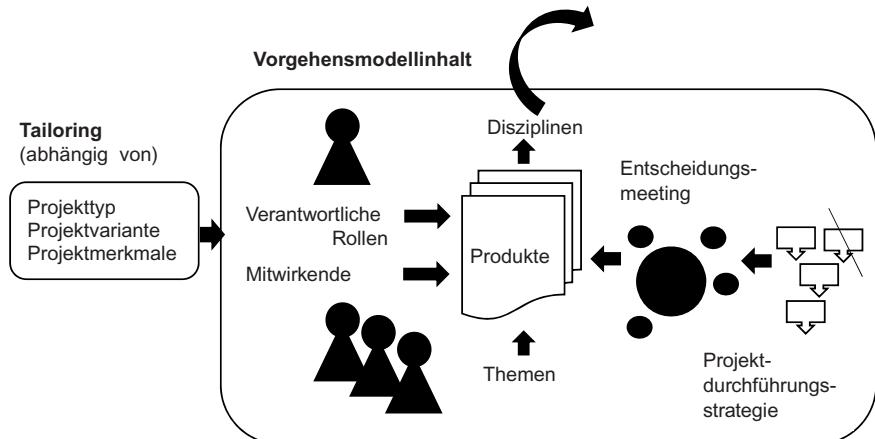


Abbildung 2.7 V-Modell XT Tailoring aus [We18]

Vorteile

- Geeignet für große Systeme und Komplexität
- Detaillierte Vorgaben, Ergebnismuster, Rollendefinitionen
- Qualitätsorientiertes Modell.

Nachteile

- Für kleine Projekte oft zu viel Overhead
- Testaktivitäten finden erst recht spät statt
- Zu strikter Phasenablauf
- Durch Bausteinprinzip und Komplexität ist ein eher hoher Schulungsaufwand zu erwarten.

2.1.4 Spiralmodell

Beschreibung

Im Spiralmodell wird der Softwarelebenszyklus häufiger nacheinander wiederholt. Beginnend in der Mitte einer Spirale laufen in den einzelnen Quadranten die Lebenszyklusphasen ab (siehe Abbildung 2.8). Dabei werden in der Implementationsphase keine ganzen Produkte erzeugt, sondern eher **Prototypen** gebaut. Diese werden evaluiert. Anschließend wird entschieden, ob ein neuer Spiralzyklus begonnen werden soll und ob der Prototyp behalten wird oder eine Neuentwicklung vorgegeben wird.

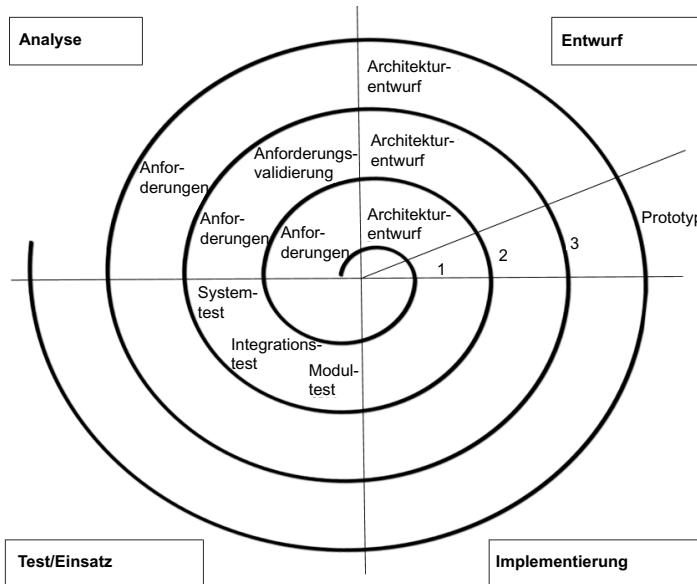


Abbildung 2.8 Spiralmodell (Böhm, 1988 [Boe88])

Die Phasen werden auch hier iterativ behandelt. Man sagt das Spiralmodell ist ein **flexibles Modell**, da das Vorgehensmodell für jeden Zyklus und jedes Teilprodukt theoretisch separat festlegbar ist.

Praxis

- Einsatz gut geeignet für Projekte mit **Prototyping**
- Viele Spiralzyklen sind teuer
- Für große Projekte geeignet.

Vorteile

- inkrementeller Ansatz
- Jeder Spiralzyklus führt zu Verbesserungen, Änderungen und Erweiterungen
- In kurzer Zeit einsatzfähige Produkte
- Flexibles Modell.

Nachteile

- Phasen werden immer wieder von neuem durchlaufen

- Prototypen können komplett neue Programmierung erfordern
- Hoher Managementaufwand, da weitreichende Entscheidungen (z. B. Prototyp behalten oder Neuentwicklung) getroffen werden müssen
- Je mehr **Spiralzyklen** durchlaufen werden, desto teurer wird das Projekt.

2.1.5 Agiles Modell

Beschreibung

Es handelt sich hier um die moderne Weiterentwicklung des iterativen Paradigmas, bei der die Planung der Iterationen dynamischer erfolgt als bei anderen Vorgehensmodellen. Eine stetige Anpassung an Änderungswünsche des Kunden ist dabei obligatorisch (siehe Abbildung 2.9). Im Fall der agilen Modelle wird absichtlich mit fairen Prinzipien gearbeitet (siehe „agiles Manifesto“ [BGM⁺⁰¹]), wie beispielsweise:

- „Das Individuum und die Interaktion ist wichtiger als der Prozess und das Werkzeug.“
- „Die Zusammenarbeit mit dem Kunden ist wichtiger als Vertragsverhandlung.“
- „Die Berücksichtigung von Änderungen ist wichtiger als das Bestehen auf Plänen.“

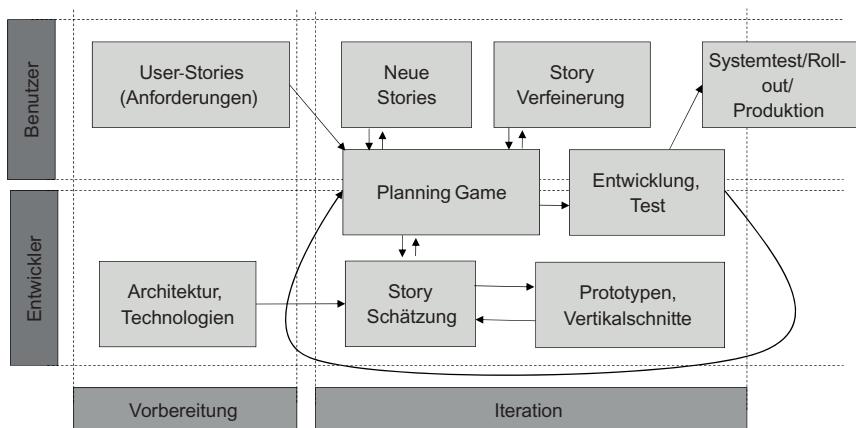


Abbildung 2.9 Agiles Modell aus [HHMS09]

Für den Kunden wichtig zu wissen ist, dass **ein Projektergebnis in diesen Modellen in der Regel nicht voraussagbar ist** und sich erst im Laufe der Entwicklungsarbeiten herausstellt.

Beispiele für verbreitete Methoden: **SCRUM** (engl. für „Gedränge“) (s. [SS18]), **XP** (engl. eXtreme Programming) (s. [Bec03]) oder auch **Kanban** (s. [LK18])

Es handelt sich demnach um ein sogenanntes **leichtgewichtiges Modell** und kann jeweils auf einem der anderen Vorgehensmodelle basieren (wie alle Vorgehensmodelle basiert dieses natürlich auch generell auf dem Wasserfallmodell). Überwiegend wird derzeit in der Industrie SCRUM eingesetzt. SCRUM ist ursprünglich eine Umsetzung von sogenanntem „Lean-Development“ (auf Deutsch: schlanke Softwareentwicklung) für das Projektmanagement, wird heute aber als vollständiges Vorgehensmodell in der Softwaretechnik angesehen.

In Abbildung 2.9 sind die Rollen Benutzer und Entwickler zu sehen. Benutzer können Kunden, aber auch Software-Engineering-Spezialisten sein, die neue „User-Stories“ (d. h. neue Anforderungen) definieren. Die Aufgabe des Entwicklers ist es, zu den User-Stories Arbeitsaufwandsschätzungen abzugeben und sich die Technologie und die Art und Weise der Implementierung zu überlegen. User-Stories werden in ein sogenanntes „Planning Game“ gegeben. Der Kunde priorisiert dabei, welche User-Stories im nächsten Durchgang zu erledigen sind. Ein Projektleiter entscheidet für den Durchgang dann, welche Ressourcen für die geplanten User-Stories zur Verfügung stehen.

Ein Grundelement ist agiles Projektmanagement (siehe Abbildung 2.10), das sich durch kurze Iterationszyklen (z. B. ca. 2 Wochen) und hohe Kundenbeteiligung mit hohem Anforderungsänderungspotenzial auszeichnet.

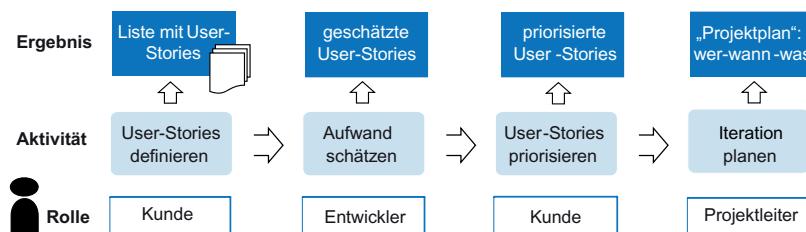


Abbildung 2.10 Agiles Projektmanagement

Für einen besseren Überblick werden nun einige Artefakte aus dem agilen Modell beschrieben.

User-Story

Eine „User-Story“ ist eine in natürlicher Sprache formulierte Software-Anforderung. Sie wird absichtlich kurz gehalten. Meist reichen als Umfang wenige, prägnante Sätze aus. Der Arbeitsumfang einer User-Story sollte in der Regel durch einen Mitarbeiter

in wenigen Tagen bearbeitet werden können, damit die Arbeitspakete planbar bleiben. Dabei ist die Verwendung einer Sprachschablone hilfreich (s. Tabelle 2.1).

Tabelle 2.1 SCRUM Sprachschablone

Sprachschablone	„Als {ROLLE} möchte ich {ZIEL/WUNSCH ERREICHEN}, um {NUTZEN ZU ERHALTEN}“
Beispiel	„Als User möchte ich einen neuen Warenkorb anlegen können, um beliebig viele Produkte hineinlegen zu können, an denen ich Kaufinteresse habe.“

Rollen

- **Product-Owner:** Der „Product-Owner“ ist für die Eigenschaften und den wirtschaftlichen Erfolg des Produkts verantwortlich.
- **Scrum-Master:** Der „Scrum-Master“ ist dafür verantwortlich, dass der Einsatz der Methode SCRUM gelingt.
- **Entwicklungs team:** Beteiligte Entwickler, aber auch Software-Engineering-Spezialisten und Tester.

Story-Map

Die „Story-Map“ (siehe Abbildung 2.11) ist eine grafische Übersicht über vorhandene User-Stories eines Projektes. Benutzeraktivitäten werden dabei horizontal als einzelne User-Stories dargestellt. Die vertikale Aufteilung wird organisatorisch genutzt, beispielsweise vom Kundenziel angefangen bis hin zu den Entwicklerzielen.

Sprint

„Das Herz von Scrum ist der Sprint, ein Zeitraum (Time-Box) von maximal einem Monat, innerhalb dessen ein fertiges („Done“), nutzbares und potenziell auslieferbares Produktinkrement hergestellt wird. Alle Sprints innerhalb eines Entwicklungs vorhabens sollten die gleiche Dauer haben. Der neue Sprint startet sofort nach Abschluss des vorherigen Sprints. Ein Sprint beinhaltet und umfasst das **Sprint-Planning**, die **Daily-Scrums**, die Entwicklungsarbeit, das **Sprint-Review** und die **Sprint-Retrospektive**.“ [SS18, S.9]. Während ein Sprint abläuft sind aber keine Änderungen erlaubt, die das Entwicklungsziel beeinflussen würden.“

Sprint-Backlog

Das „Sprint-Backlog“ (siehe Abbildung 2.12) ist eine Planungsübersicht über ausgewählte Arbeitsaufträge (Dies sind in der Regel User-Stories aus dem sogenannten „Product-Backlog“). Es dient auch als Prognoseübersicht des Entwicklungsteams, welche User-Stories im nächsten Sprint abgearbeitet werden können.

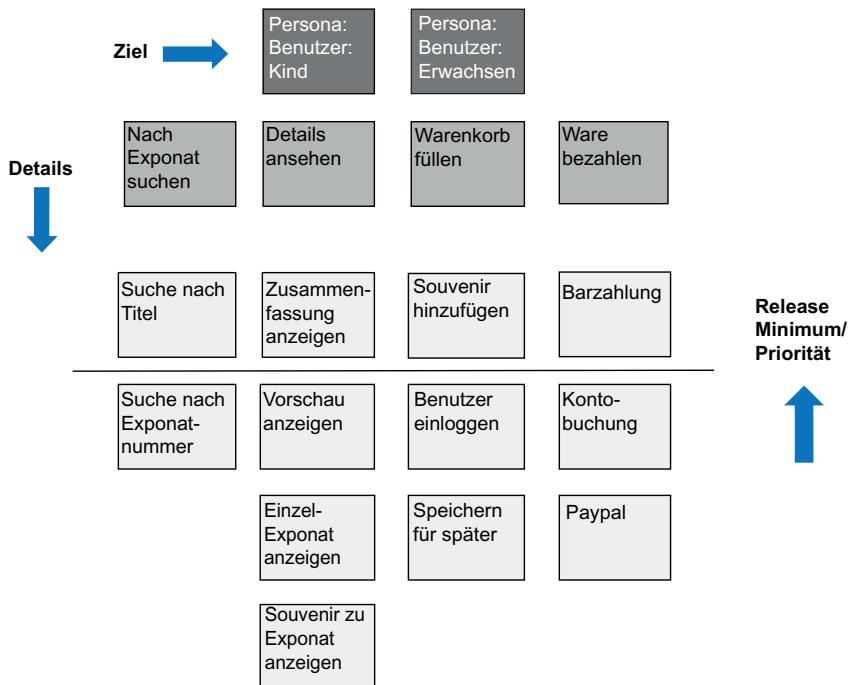


Abbildung 2.11 Beispiel einer Story-Map für einen Museumsführer mit der Besichtigung von Exponaten und dem Verkauf von Souvenirs

Berichte

Es gibt beispielsweise bei Scrum eine Reihe von speziellen Berichten, die über den aktuellen Stand des Projektes Auskunft geben. Dazu gehört beispielsweise unter anderem ein sogenannter **Sprint-Burn-Down-Chart** (siehe Abbildung 2.13).

Wenn alle Aufgaben in einem Sprint geschätzt sind, erhält man die geschätzte, verbleibende Gesamtarbeit der Projektmitglieder in Gesamtarbeitsstunden.

Das Projektteam errechnet sich nun jeden Tag die verbleibenden Gesamtarbeitsstunden und erhält somit eine Übersicht über die täglich abnehmende Anzahl noch zu leistender Arbeitstunden.

Werte agiler Projekte

Der Mensch steht im Mittelpunkt, daher werden in agilen Projekten folgende Wertvorstellungen verfolgt:

- Bescheidenheit

Sprint	Offen	In Bearbeitung	Im Test	Fertig
KW 2/3		Zusammenfassung anzeigen Benutzer einloggen		Suche nach Titel
KW 4/5	Suche nach Exponatnummer		Vorschau anzeigen	
KW 6/7	Speichern für später Paypal	Kontobuchung		

Abbildung 2.12 Beispiel eines Sprint-Backlogs für den Museumsführer aus Abbildung 2.11

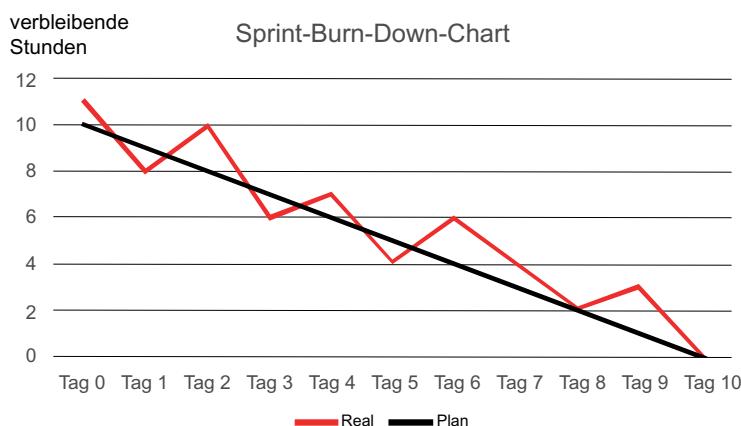


Abbildung 2.13 Beispiel eines Sprint-Burn-Down-Charts

- Feedback
- Mut zur Wahrheit
- Kommunikation
- Innovation
- Respekt.

Unterstützte Prinzipien in agilen Projekten

In agilen Modellen setzt man auf faire **Prinzipien** für alle.

- Collective-Code-Ownership (CCO): Jedem Entwickler „gehört“ der Quellcode, d. h. jeder darf alles ohne weitere Nachfrage ändern.
- Mehrfache Verwendung von Ressourcen (Wirtschaftlichkeit)
- Einfachheit (z. B. Konzepte, Sprache)
- Totale Qualitätsorientierung (ähnlich zu Ansätzen wie *Total Quality Management*[MHS98])
- Dokumentation nur wo nötig
- Anforderungen werden erst umgesetzt, wenn sie benötigt werden.
- Customer-on-Site: In der Regel ist ein Kundenvertreter vor Ort bei der Entwicklung mit integriert.
- Arbeitszeiten respektieren (engl.: sustained pace): Jeder Mitarbeiter des Projektes wird nur fair eingeplant und entsprechend seiner verfügbaren Arbeitszeit eingesetzt.

Methoden

In agilen Modellen wird zur Zielerreichung häufig mit folgenden **Methoden** gearbeitet.

- Programmierrichtlinien (d. h. Vorgaben, wie Quellcode geschrieben wird; eng.: Code-Conventions)
- (engl.) Pair-Programming (in der Regel werden schwierige Quellcodestücke oder Architekturartefakte gemeinsam besprochen)
- (engl.) Rapid-Code-Reviews (im Entwicklerteam werden häufig kurze Sitzungen zur Qualitätsüberprüfung vorgenommen)
- Automatisierte sogenannte Buildprozesse
- (engl.) Story-Cards (d. h. man arbeitet mit User-Stories, die auf Zetteln (virtuell) aufgeschrieben sind)
- (engl.) Refactoring (Anstatt komplexe, unüberschaubare Architektur- und Software-Artefakte weiter mitzunehmen, wird durch häufige Überarbeitung eine Reduzierung der Komplexität erreicht)

- Testgetriebene-Entwicklung (d. h. erst werden bei der Softwareentwicklung Tests geschrieben und danach erst programmiert; engl. Testdriven-Development)
- Ständige Integration (d. h. neue Programmstücke werden möglichst sofort nach Erstellung mit in das Gesamtprojekt integriert, engl. Continuous-Integration).

Nachdem nun agile Modelle in ihren groben Grundzügen beschrieben wurden, folgen nun wieder einige Gedanken zu den Einsatzmöglichkeiten in der Praxis und zu den Vorteilen und Nachteilen des Einsatzes.

Praxis

- relevant für große und komplexe, genauso wie für kleine Systeme und Projekte
- Lösung für Projekte, in denen Prototyping eingesetzt werden soll und kann.

Vorteile

- geringer bürokratischer Aufwand
- wenige und flexible Regeln
- nur soviel Dokumentation wie nötig
- verspricht besseres Kosten/Nutzen-Verhältnis
- verspricht eine durchschnittlich höhere Code-Qualität.

Nachteile

- das gesamte Team muss sich an Regeln halten
- Projektergebnis ist nicht vorhersagbar.

Das agile Vorgehensmodell gilt derzeit als das modernste seiner Art und erfreut sich in der Praxis in großen und kleinen IT-Projekten großer Beliebtheit.

■ 2.2 Klassisches Projektmanagement

Neben der Festlegung der Vorgehensweise durch ein Vorgehensmodell muss das Management vor allem die Übersicht über ein IT-Projekt gewährleisten. Dafür werden Methoden aus dem so genannten Projektmanagement eingesetzt. Darunter werden im Allgemeinen alle Managementaufgaben zur Steuerung von Projekten verstanden.

Nach einer ersten Phase der Projektplanung folgt die kontinuierliche Verfolgung des Projektes durch das Projektmanagement. Beide Phasen werden nun betrachtet.

2.2.1 Projektplanung

In der Planungsphase des Software-Lebenszyklus wird in der Regel das neue Projekt aufgesetzt. Diese Tätigkeit nennt man Projektplanung. In der Projektplanung ist die Kenntnis über Projektmanagement und der damit verbundenen Methoden erforderlich. Die drei zu betrachtenden Hauptaspekte sind: **Kosten, Zeit und Ressourcen**. Daher sind die folgenden Aufgaben zu erledigen.

Managementaufgaben

- Erstellung des Angebots
- Projekt- und Zeitplanung
- Projektkostenkalkulation
- Projektüberwachung und Reviews
- Auswahl und Beurteilung von Ressourcen (Personal, Sachmittel)
- Präsentation und Erstellen von Berichten.

Grundsätzlich ist Projektplanung ein iterativer Vorgang, bei dem während der gesamten Projektlaufzeit wiederholt der gleiche Ablauf stattfindet.

Ablauf Projektplanung

1. Randbedingungen aufstellen
2. erste Einschätzungen der Projektparameter treffen
3. Meilensteine und Lieferschritte des Projekts definieren
4. Randbedingungen verhandeln
5. Projektzeitplan aufstellen
6. Aktivitäten gemäß Zeitplan beginnen
7. Projektfortschritt prüfen
8. technische Reviews und mögliche Überarbeitung
9. Projektparameter überarbeiten
10. iterativ weiter mit Wiederholung ab Schritt 4.

Nachdem die initiale Phase der Projektplanung stattgefunden hat, beginnt die kontinuierliche Projektverfolgung mit den Verfahren des Projektmanagements.

2.2.2 Projektmanagement (Zeit-, Kosten- und Ressourcenplanung)

In diesem Abschnitt wird die traditionelle, klassische Variante des Projektmanagements beschrieben. Für die Vertiefung der agilen Variante sei auf Literatur zu SCRUM [SS18] und Kapitel 2.1.5 verwiesen. Zum Verstehen der agilen Variante ist die Kenntnis der klassischen Variante jedoch äußerst hilfreich. (Weiterführende Literatur zu Software-Management im Allgemeinen kann auch in [Bal08] gefunden werden.) Klassisches Projektmanagement kann beispielsweise in [Ker08] weiter studiert werden.



Definition

Projektmanagement ist die Gesamtheit von Führungsaufgaben, -organisation, -techniken und -mitteln für die Abwicklung eines Projektes

(nach DIN-Norm (DIN 69901))

Projektmanager

Der Projektmanager hat eine leitende Funktion innerhalb des Projekts. Seine Hauptaufgaben sind Terminmanagement, Kostenmanagement, Projektinhalt und Umfang der Aufgaben zu definieren, zu koordinieren und zu steuern. Außerdem ist er in der Regel für das Personalmanagement zuständig. Grundsätzlich fungiert der Projektmanager als Bindeglied zwischen den Erwartungen der Geschäftsleitung, bzw. Kunden und den Entwicklern, Testern und des Supports.

Wissensgebiete des Projektmanagements

Projektmanager müssen zunehmend in der Lage sein, eine Vielzahl von Wissensgebieten zu verstehen und zu beherrschen. Eine Auflistung ist in Abbildung 2.14 zu sehen. In dieser Abhandlung wird hauptsächlich das Zeitmanagement (Terminmanagement) vertieft, ein kurzer Einblick in die Ressourcenplanung (u. a. Personalplanung) gegeben und nur grob das Kostenmanagement betrachtet (blau markiert).

Um besser zu verstehen, wie Projektmanagement funktioniert, ist in Abbildung 2.15 am Beispiel ein typischer zeitlicher Ablauf eines Projektes zu sehen.

Projektmanagement muss also kontinuierlich während des gesamten Projektverlaufs stattfinden, wobei die klassische Software-Engineering-Tätigkeit in die einzelnen Phasen des Software-Lebenszyklus aufgeteilt werden kann.

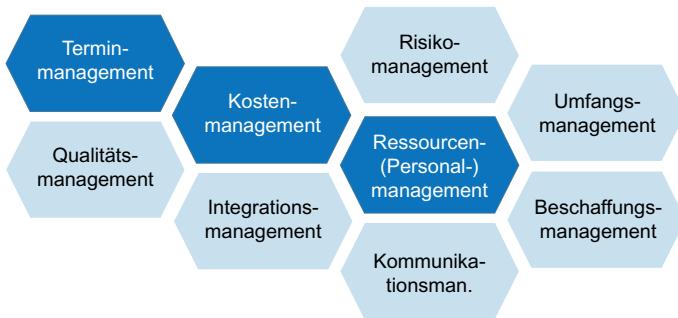
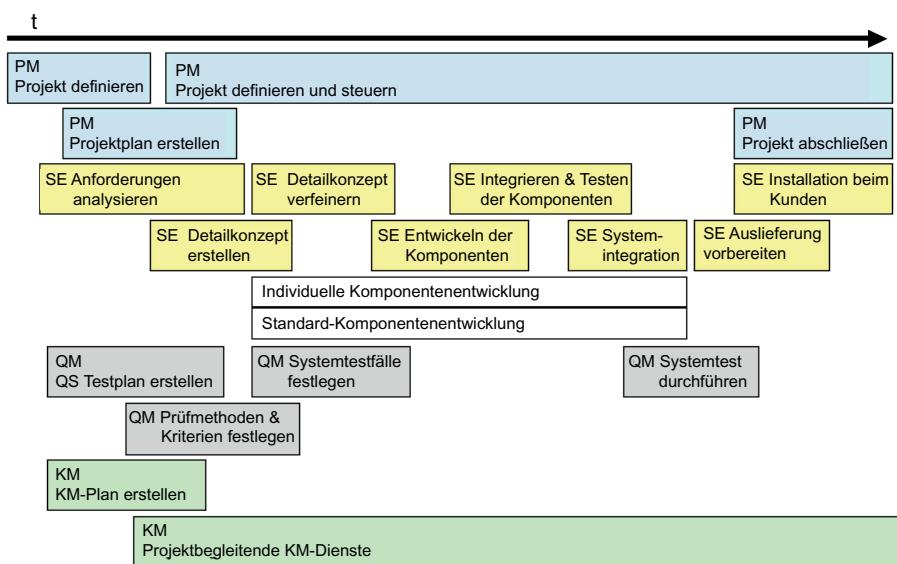


Abbildung 2.14 Einige wichtige Wissensgebiete des Projektmanagements



(PM: Projektmanagement, SE: Software-Engineering, QM: Qualitätsmanagement, KM: Kundenmanagement)

Abbildung 2.15 Typischer Ablauf eines IT-Projekts

2.2.2.1 Zeitmanagement

Im Rahmen des Zeitmanagements werden im Folgenden die Begriffe und Funktionsweisen von Ablaufplänen und Vorgangsknotennetzen erklärt.

Ablaufplan (GANTT-Diagramm)

Ein Ablaufplan stellt ein Projekt grafisch dar. In Abbildung 2.16 ist ein Beispiel eines GANTT-Diagramms mit acht voneinander abhängigen Vorgängen zu sehen. Durch umfangreiche Symbolik und Farbcodierung werden Abhängigkeiten, Terminierungen und Pufferzeiten sichtbar dargestellt.

Definition

Ein **Gantt-Diagramm** oder **Balkenplan** ist ein nach dem Unternehmensberater Henry L. Gantt (1861–1919) benanntes Instrument des Projektmanagements, das die zeitliche Abfolge von Aktivitäten grafisch in Form von Balken auf einer Zeitachse darstellt.

In einem Gantt-Diagramm werden die Projektaktivitäten in eine Spalte des Diagramms eingetragen. Eine Zeitachse wird in die oberste Zeile des Diagrammes eingetragen. Jede Aktivität wird in ihrer Zeile anschließend mit einem waagrechten Balken auf der Zeitachse dargestellt. Dies ergibt eine gute Übersicht, wie lange Aktivitäten dauern. Mit Pfeilen werden Abhängigkeiten zwischen den Aktivitäten dargestellt. Auch ist durch die Angabe der Abhängigkeiten die Darstellung des kritischen Pfades durch farbliche Markierung möglich.

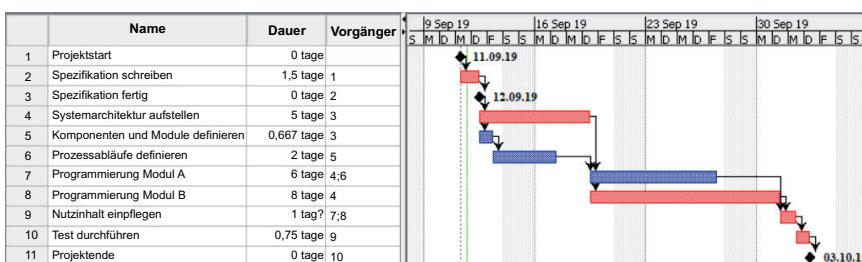


Abbildung 2.16 Beispiel eines GANTT-Diagramms (mit [Bru19] erstellt)

Legende zu Abbildung 2.16:

- blauer Balken: Vorgang
- roter Balken: kritischer Vorgang
- schwarze Raute: Meilenstein
- schwarze Pfeile: Abhängigkeiten der Vorgänge.

Netzplan (Pert-Diagramm, hier: Vorgangsknotennetz)

Definition

Netzpläne stellen Aufgaben und Arbeitsschritte verknüpft dar, so dass parallele Abläufe und bei abhängigen Aufgaben die Vorleistungen erkennbar sind. Indem Angaben zur Dauer von einzelnen Aufgaben berücksichtigt werden, kann ein Endtermin kalkuliert werden.

Es gibt grundsätzlich zwei Arten von Netzplänen, die Vorgangspfeilnetzpläne und die Vorgangsknotennetze.

- **Vorgangspfeilnetz, Ereignisknotennetz** (siehe Abbildung 2.17)

Beim Vorgangspfeilnetz stellen die Pfeile Aktivitäten dar, die zwischen Ereignissen stehen. Ein Ereignis kann mehrere Aktivitäten auslösen oder erst nach Abschluss mehrerer Aktivitäten eintreffen. Ereignisse haben früheste und späteste Termine, je nach dazwischen liegenden Aktivitäten und deren Kalkulierbarkeit.

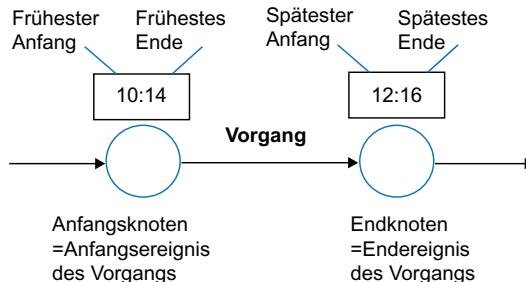


Abbildung 2.17 Beispiel eines Vorgangspfeilnetzplans mit Pufferzeiten nach [Ker08]

Der wohl bekannteste Repräsentant eines Vorgangspfeil-Netzplans ist das „CPM-Diagramm“ (engl. Critical-Path-Method). Diese Netzplanart hat sich jedoch in der Praxis nicht durchgesetzt. Verwendet wird hingegen üblicherweise das Vorgangsknotennetz.

- **Vorgangsknotennetz** (siehe Abbildung 2.18)

Beim Vorgangsknotennetz stellen die Kästchen Aktivitäten dar, die über Pfeile, welche Vorgänger- und Nachfolgerbeziehungen darstellen, verbunden sind.

Ein Knoten enthält:

- Bezeichnung der Aktion
- Dauer der Aktion

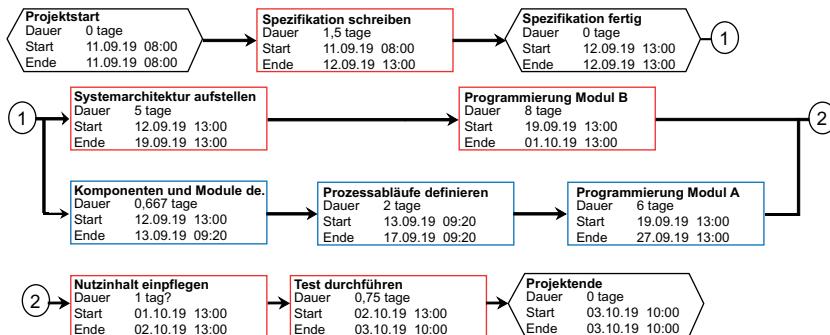


Abbildung 2.18 Beispiel eines Vorgangsknotennetzes (mit [Bru19] erstellt)

- optional: Start, Ende, Pufferzeiten, Kostenstelle, benötigte Ressourcen, Verantwortlichkeit, u. v. m.

Das Beispiel mit den Vorgängen des GANTT-Diagramms aus Abbildung 2.16 wird in Abbildung 2.18 nun als Pert-Diagramm dargestellt. GANTT-Diagramme können demnach in die Darstellungsform des Pert-Diagramms umgewandelt werden und umgekehrt.

Nach dem Aufstellen des Netzplans werden in der Regel (mindestens) folgende Felder ausgefüllt (siehe Abbildung 2.19).

FA	Dauer	SA
Vorgangsnname, Nr.		
FE	GP	SE

Abbildung 2.19 Legende eines Vorgangsknotens

- FA: Frühester Anfangstermin
- SA: Spätester Anfangstermin
- FE: Frühester Endtermin
- SE: Spätester Endtermin
- GP: Gesamtpufferzeit.

Knoten können unterschiedlich verknüpft werden (meistens reicht die Variante „FS“ aus).

- **FS Finish-Start** Ablauf nacheinander
„Beginn der nächsten Phase erst nach Review zur vorigen Phase“
- **SS Start-Start** gleichzeitiger Beginn
„Protokollierung startet bei Beginn Testphase“
- **FF Finish-Finish** gleichzeitiges Ende
„Abschalten erst bei erfolgreicher Datensicherung“
- **SF Start-Finish** Ende erst nach erfolgreichem Start
„Anmelde-Vorgang endet nur, wenn Bediensystem erfolgreich gestartet werden konnte.“

Kritischer Pfad

Bei parallel ablaufenden Aufgaben ergeben sich aus der längsten Dauer Pufferzeiten für die kürzeren Aufgaben sowie der „kritische Pfad“, auf dem eine Verzögerung einer Tätigkeit eine Verspätung des gesamten Projektes zur Folge hat. Der kritische Pfad kann mithilfe der Graphentheorie berechnet werden.

Meilensteine

Meilensteine sind normale Vorgänge, jedoch mit 0 Stunden Dauer. Sie eignen sich als eine Art der Markierung geeigneter Stellen im Netzplan (meist an „engen“ Stellen), zu denen eine Überprüfung der Einhaltung von Terminen und Ergebnissen stattfinden kann.

Vorgangsknotennetzplan zeichnen

Um den Vorgangsknotennetzplan zu zeichnen, werden die Berechnungsformeln 2.1 eingesetzt.

Formel – Berechnungsformeln Projektmanagement

$$FA = \max(FE_{\text{der Vorgänger}})$$

$$FE = FA + \text{Dauer}$$

$$SE = \min(SA_{\text{der Nachfolger}}) \quad (2.1)$$

$$SA = SE - \text{Dauer}$$

$$\text{Puffer} = SA - FA = SE - FE$$

Folgende Schritte werden nun durchgeführt:

1. Aufstellen einer Tabelle (siehe Beispieldtabelle in Abbildung 2.16) mit laufender Nummer, Tätigkeit, Vorgang, Abhängigkeiten zu anderen Vorgängen, geschätzte Dauer (mit Variationsbreite), benötigte Ressource(n), Ausführende(r) und Platz für die Angaben FA, SA, FE, SE.
2. Die Zerlegung von Aufgaben erfolgt bis zu einer Einheit, die von einer „Rolle“ (Person) ausgeführt werden kann.
3. Aufzeichnen der Aufgaben aus der Tabelle als Netzplan (Beispiel siehe Abbildung 2.18).
4. Bestimmung des kritischen Pfads: jeweils längste Dauer von parallel verlaufenden Aktionen markieren. Entlang des kritischen Pfades sind die Pufferzeiten immer null.
5. Addition der Dauer entlang des kritischen Pfades zur Bestimmung der Dauer des gesamten Projektes.
6. Bei vorgegebenem Endtermin müssen Aktivitäten auf dem kritischen Pfad geeignet verkürzt werden. (Achtung: Parallele Pfade können dabei zum kritischen Pfad werden.)

2.2.2.2 Ressourcenplan

Die Verteilung von Tätigkeiten an Mitarbeiter (Ressourcen) ergibt sich nicht direkt aus dem Netzplan (siehe Beispiel in Abbildung 2.20). Je nach Anzahl und Auslastung von Ressourcen müssen die Aufgaben aus dem Netzplan verteilt werden. Nach Erstellung des Netzplans kann pro Ressource eine Liste mit Belegungszeiten erstellt werden; hierdurch werden die Tätigkeiten konkret an Mitarbeiter zugeteilt. Der tatsächliche Endtermin, die Auslastung und Kosten für die Ressourcen hängen von einem gekonnten Ressourcenplan ab.

	Name	Aufwand	29 Jul 19								
			M	D	F	S	S	M	D	M	F
1 Mitarbeiter 1	31 Stunden	8h 4h 0h 0h 0h						2,667h	8h 5,333h 0h 0h		
	Spezifikation schreiben	12 Stunden	8h	4h							
	Spezifikation schreiben	0 Stunden			0h						
	Prozessabläufe definieren	16 Stunden						2,667h	8h 5,333h		
	Prozessabläufe definieren	3 Stunden									
2 Mitarbeiter 2	Projektende	0 Stunden									
	101,667 Stu...	8h 4h 0h 0h 0h						6,667h	4h 4h 4h 4h		
	Spezifikation schreiben	12 Stunden	8h	4h							
	Systemarchitektur aufstellen	20 Stunden							4h	4h 4h 4h	4h
	Komponenten und Module	2,667 Stu...						2,667h			
	Programmierung Modul 5	64 Stunden									
	Test durchführen	64 Stunden									

Abbildung 2.20 Beispiel eines Ressourcenplans (mit [Bru19] erstellt)

2.2.2.3 Kalkulation

Durch die Zuordnung von Ressourcen zu den einzelnen Vorgängen, können nun durch den festgelegten Stundensatz eines jeden Mitarbeiters die Kosten pro Vorgang geschätzt bzw. ermittelt werden. Eine Beispielrechnung ist in Tabelle 2.2 zu finden. Durch Bildung der Summe der Kosten aller Vorgänge können die Gesamtkosten geschätzt werden.

Tabelle 2.2 Beispielhafte Kalkulation eines Vorgangs

Bezeichnung	Wert
Anforderungsnr.	433
Auftragsnr.	23
Funktion	UserLogin()
Aufwand	2 Personen
Zeit	2 Personentage
Stundensatz	80 EUR
Kosten	$8h * 2 \text{ Personen} * 2 \text{ Personentage} * 80 \text{ EUR} = 2560 \text{ EUR}$

2.2.2.4 Pufferzeiten, Ressourcenauslastung und Schätzung der Dauer von Tätigkeiten

Durch Vorwärts- und Rückwärtsrechnung im Netzplan ergeben sich Pufferzeiten. Die gesamte Planung hängt jedoch von einer möglichst realistischen Schätzung der Tätigkeitsdauer ab.

- **Pufferzeiten**

Durch Bestimmung der frühesten Anfangstermine vom Start des Netzplanes und der spätesten Endtermine vom Ende des Netzplanes her ergeben sich bei Tätigkeiten, die nicht auf dem kritischen Pfad liegen, Pufferzeiten. Bei hintereinander liegenden Tätigkeiten sind die Pufferzeiten nicht mehr voneinander unabhängig! (d.h es existieren freie Pufferzeiten – unabhängige Pufferzeiten – freie Rückwärts-pufferzeiten).

- **Auslastung**

Bei vorhandenen Pufferzeiten sind die Ressourcen nicht voll ausgelastet und können/sollten in anderen Tätigkeiten eingesetzt werden. Die Einsatztabelle für eine Ressource wird sich danach aber selten ohne Leerzeiten planen lassen, eine Optimierung erfordert ggf. Modifikationen im Netzplan.

- **Unsicherheit der Daten**

Die Dauer von Tätigkeiten kann durch eine optimistische und eine pessimistische Schätzung angegeben werden (siehe Tabelle 2.2), evtl. kann ein gewichteter Mittelwert mit Schwankungsbreite verwendet werden. Außerhalb des kritischen

Pfades kann man die Unsicherheit meist mit der Pufferzeit ausgleichen (auch mit dem Ressourcenplan). In welcher Form sich die Schwankungsbreiten entlang des kritischen Pfades addieren, ist eine Strategieentscheidung der Planung (gegenseitiges Ausgleichen oder Addition der Beträge).

■ 2.3 Zusammenfassung

In diesem Kapitel wird der Einsatz von Vorgehensmodellen allgemein (siehe Kapitel 2.1) und schließlich auch für unterschiedliche Vorgehensmodellarten diskutiert. Dabei werden die Ausprägungen Wasserfallmodell, verbessertes Wasserfallmodell, V-Modell, Spiralmodell, und agiles Modell erklärt. Die Wahl eines passenden Vorgehensmodells ist dabei projektabhängig zu entscheiden. Es spielen dabei Faktoren der Projektgröße, der Komplexität und der Vorgehensart (z. B. Prototypisches Vorgehen oder häufige Änderungswünsche) eine Rolle. Der Einsatz eines Vorgehensmodells dient dabei als Leitfaden und bietet eine gemeinsame und verbindliche Sicht der logischen und zeitlichen Struktur eines Projektes.

In Kapitel 2.2 geht es um Methoden des klassischen Projektmanagements. Hauptelemente sind Terminmanagement, Ressourcenmanagement und Kostenmanagement. Die Erstellung von Ablauf- und Netzplänen wird dargestellt.

■ 2.4 Aufgabensammlung

Aufgabensammlung der phasenübergreifenden Verfahren des Software-Engineerings.

1. Was beinhaltet ein Vorgehensmodell?
2. Welche Form der Software-Erstellung hat sich durchgesetzt und ist Erfolg versprechend?
3. Welche Hauptfaktoren eines Softwareprojektes werden beim Projektmanagement betrachtet?
4. Was liegt auf dem kritischen Pfad eines Vorgangsknotennetzes? Welche Aussage ist richtig?
 - A) Vorgangspfeile, Ereignisknoten, Meilensteine, Pufferzeiten
 - B) Vorgangspfeile, Ereignisknoten, Meilensteine

- C) Vorgangsknoten, Meilensteine, Vorgänger-Nachfolger-Beziehungen, Pufferzeiten
- D) Vorgangsknoten, Meilensteine, Vorgänger-Nachfolger-Beziehungen
5. Ein Vorgang hat folgende Werte: FA=0T, FE=5T, SA=3T, SE=8T. Welche Dauer und welche Gesamtpufferzeit hat der Vorgang?
6. Erstellen Sie einen Vorgangsknoten-Netzplan für die Aufgabentabelle in Abbildung 2.21. Zeichnen Sie den kritischen Pfad ein.

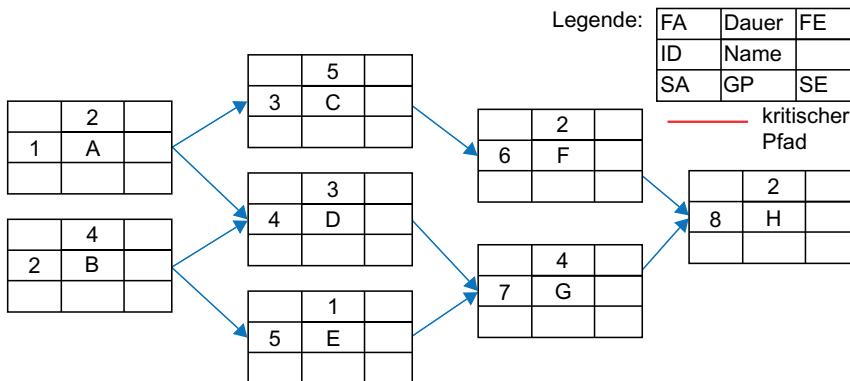


Abbildung 2.21 Übung Projektmanagement

3

Planungsphase

Die erste Phase im Software-Lebenszyklus ist die Planungsphase. In diesem Kapitel wird zunächst eine Übersicht über die Planungsphaseninhalte gegeben (siehe Kapitel 3.1). Ein wichtiges Ergebnisdokument dieser Phase ist das Lastenheft, welches ab Kapitel 3.2 beschrieben wird. Auch Methoden zur Aufwandsschätzung (siehe Kapitel 3.3) werden andiskutiert. Zuletzt folgt ein Einblick in das Risikomanagement von IT-Projekten (siehe Kapitel 3.4), was hilfreich ist, um etwaige Bedrohungen für ein Projekt zu erkennen, abzumildern oder zu verhindern.

■ 3.1 Übersicht Planungsphase

Grundsätzliche Vorgehensweise

In der Planungsphase eines IT-Projekts sind grundsätzlich vier Prozessschritte durchzuführen, nämlich (1) Ist-Aufnahme, (2) Ist-Analyse, (3) Diskussion der Verbesserungsvorschläge und (4) Erstellung des Soll-Konzepts. Grundsätzlich finden nach jedem Prozessschritt Treffen (engl. Meetings) und Reviews (siehe Kapitel 6.2.1) statt, in denen ein Konsens bezüglich der Ergebnisse gesucht wird.

1. Ist-Aufnahme

Dieser Prozessschritt enthält die schriftliche Beschreibung des momentanen Systems und dessen Zustand oder die Beschreibung des Problems, das mit neuer Software gelöst werden soll. Bedingungen, Umgebungen und zur Verfügung stehende Mittel werden definiert.

2. Ist-Analyse

Die im Vorschritt beschriebenen Artefakte werden nun analysiert und eine Schwachstellenanalyse findet statt.

Relevante Fragen sind:

- Wie ist der Zustand des momentanen Systems?
- Kann das System verbessert werden?
- Wo sollte das System verbessert werden?
- Wo sind Änderungen notwendig?

3. Verbesserungsvorschläge

Eine Sammlung von Lösungsalternativen entsteht in diesem Prozessschritt. Es werden grobe Beschreibungen erarbeitet, wie die Veränderungen aussehen sollen. Außerdem wird beschrieben, wie die Veränderungen herbeigeführt werden können.

4. Soll-Konzept

Aus den Lösungsalternativen wird nun das beste Konzept ausgewählt. Ein formales Modell vom momentanen Zustands des IT-Systems und ein formales Modell der geplanten Alternative wird schließlich in die sogenannte **Spezifikation (d. h. Lastenheft)** (siehe Kapitel 3.2) aufgenommen.

Aufgaben in der Planungsphase

Eine Reihe von Aufgaben sind in der Planungsphase zu erledigen. Häufig vorzufinden sind die folgenden:

1. Voruntersuchung
Ist-Aufnahme und Ist-Analyse, Verbesserungsvorschläge, Soll-Konzept, Festlegung der wichtigsten Anforderungen
2. Machbarkeitsstudie
Untersuchung der Realisierbarkeit, Prüfung auf Abdeckung Standard-Software, Verfügbarkeitsprüfung von Ressourcen, Risikoanalyse
3. Produktplanung
Auswählen des Produkts, Trendstudien, Marktanalysen, Integration von Forschungsergebnissen, Kundenanfragen und -analysen
4. Projektplanung
Projektplan: Aufwandsschätzung mit Kosten-, Ressourcen- und Terminplan.

Schriftliche Ergebnisse

Aus den zuvor genannten Aufgaben gehen folgende schriftliche Ergebnisse hervor:

1. Voruntersuchung: Lastenheft (siehe Kapitel 3.2)
2. Machbarkeitsstudien: Studien
3. Produktplanung: Studien, Risikoanalyse (siehe Kapitel 3.4)
4. Projektplanung: Kosten-, Termin- u. Ressourcenplan (siehe Kapitel 2.2).

Beteiligte

Neben dem Auftraggeber (Kunde) nehmen in diesem Prozess der Projektleiter, eventuell Anwendungsspezialisten (oft „Poweruser“ des Kunden oder so genannte Multiplikatoren, d. h. Benutzer, die Wissen an andere Benutzer weitergeben) und natürlich die Software-Engineering-Spezialisten (d. h. die fachlichen Entwickler) teil.

■ 3.2 Lastenheft

Ein wichtiges Ergebnisdokument der Planungsphase ist das Lastenheft, auch **Spezifikation** genannt. Dieses Ergebnisdokument wird später in der Definitionsphase zum Pflichtenheft verfeinert (siehe Kapitel 4.2). Leider wird begrifflich auch hier oft von der „Spezifikation“ gesprochen, so dass diese beiden Dokumente jedoch vom Team trotzdem gut unterschieden werden sollten.

Was sind also die Unterschiede zwischen diesen beiden Dokumenten?

- **Lastenheft:** Auftraggebersicht
- **Pflichtenheft:** Auftragnehmersicht
Vertragsgegenstand ist das (verfeinerte) Pflichtenheft.

Es gibt zwei gängige Vorgehensweisen, wie die Erstellung eines Lastenheftes stattfinden kann:

- **Outside-in:** Zuerst Systemumgebung modellieren, dann Systeminterna modellieren
- **Inside-out:** Zuerst Systeminterna modellieren, dann Systemumgebung modellieren.

Aufgabe

Mit dem Lastenheft werden alle fachlichen Basisanforderungen zusammengefasst. Es entsteht somit eine Produktbeschreibung, die aus Sicht des Auftraggebers geschrieben wird. Verfasser ist normalerweise der Kunde. Im Lastenheft wird beschrieben „**WAS**“ für eine Software entwickelt werden soll, jedoch nicht „**WIE**“ die Lösung dazu aussieht.

Der Inhalt des Lastenhefts wird auch benötigt, damit der Softwareproduzent dem Kunden ein Angebot unterbreiten kann.

Das Schreiben von Lastenheften wird meist durch die Verwendung von Dokumentvorlagen erleichtert. Die Basisanforderungen werden in natürlicher Sprache verfasst und oft bereits durch Diagramme und Graphiken ergänzt. Alle Anforderungen sollten

schon zu diesem Zeitpunkt durchnummertiert werden, um frühzeitig eine einfache Referenzierungsmöglichkeit zu erhalten.

Beteiligte (engl.: Stakeholder)

An der Lastenhefterstellung sind beteiligt:

- **Auftraggeber:** Verfasser des Lastenheftes ist der Auftraggeber, d. h. in der Regel der Kunde, ein Vertreter des Kunden oder auch die auftraggebende Abteilung.
- **Projektleiter:** Der Projektleiter des Softwareproduzenten nimmt in einem „Kick-off-Meeting“ das Lastenheft als (möglichen) Auftrag entgegen.
- **Anwendungsspezialist:** Hier handelt es sich um Domänen-Experten für die Aufgaben, welche im Lastenheft beschrieben sind. Oft sind Anwendungsspezialisten Angestellte des Kunden und an der Lastenhefterstellung beteiligt. Auch Vielbenutzer von Altsystemen werden oft als Anwendungsspezialisten zu Projekten hinzugezogen.
- **Software-Engineering-Spezialist:** Der fachliche Entwickler gehört zum Team des Softwareproduzenten und ist dafür zuständig, das Lastenheft entgegenzunehmen und später daraus ein Pflichtenheft zu generieren.

■ 3.3 Aufwandsschätzung

Der Projektleiter muss zur Erstellung eines Angebots eine Aufwandsschätzung vornehmen. In diesem Kapitel werden hierfür einige Grundlagen vermittelt.

Tabelle 3.1 Beispiel einer Projektaufgabe mit Aufwandsschätzung

Bezeichnung	Wert
Anforderungsnr.	433
Auftragsnr.	23
Funktion	UserLogin()
Aufwand	2 Personen
Zeit	2 Personentage
Stundensatz	80 EUR
Kosten	$8h * 2 \text{ Personen} * 2 \text{ Personentage} * 80 \text{ EUR} = 2560 \text{ EUR}$

Grundsätzlich wird während der Projektplanung das Projekt bereits in Aufgaben unterteilt. Jede Aufgabe wird hinsichtlich ihres Aufwands, ihrer Zeit und ihrer Kosten geschätzt (d. h. Ist- und Sollwerte). Regelmäßige Soll-Ist-Analysen sind wichtig. Die Aufgabenaufteilung wird in der Regel so vorgenommen, dass jede Aufgabe überschaubar von einer (oder wenigen Personen) in einem kleinen Zeitraum (meist 0–2 Tage) erledigt werden kann (siehe Beispiel Tabelle 3.1). Durch Summierung aller Aufwände (d. h. Abhängigkeiten der Tätigkeiten voneinander und Pufferzeiten sind mit zu berücksichtigen) entstehen schließlich Schätzungen.

Die **Gesamtkosten** eines Projekts bestehen jedoch nicht nur aus Personalkosten. Weitere zu berücksichtigende Faktoren sind hauptsächlich:

- Hardware-, Software-Kosten, Wartung
- Reisekosten, Schulungskosten
- Personalkosten
- Software-Entwicklerkosten
- Raumkosten (Heizung, Beleuchtung, Miete)
- Supplementäres Personal: techn. Personal, Sekretärinnen, ...
- Kommunikationskosten (Netze, Email ...)
- Kosten für zentrale Einrichtungen:
 - Druckereidienste
 - Sozialversicherungen (SV, KV, Mitarbeiterzuwendungen, Renten ...).

Projektleiter müssen daher diese Faktoren bei ihrer Kalkulation berücksichtigen und in die Rechnung mit einbeziehen.

Produktivität

Eine interessante Fragestellung des Software-Engineerings ist, wie die Produktivität bei Software-Projekten eigentlich ermittelt werden kann. Da die Produktivität überall in der Fertigungsindustrie eine große Rolle spielt und einfach berechnet werden kann, wurde zunächst untersucht, ob die dort verwendeten Formeln auch bei IT-Projekten eingesetzt werden können.

Formel

Analogie zum Fertigungssystem?

$$\text{Produktivität} = \frac{\text{produzierte Stückzahl}}{\text{Anzahl für die Produktion erforderliche Personenstunden}} \quad (3.1)$$

Feststellbar ist jedoch, dass die Produktivität in Software-Systemen **anderen Regeln** unterliegt. Dies liegt generell schon einmal an den Eigenschaften von Software (siehe Kapitel 1.2.2), wie beispielsweise der Immaterialität und der dadurch nicht möglichen Stückzahlmessung. Zudem hat Software ebenfalls Performance- bzw. Effektivitätsunterschiede, die nicht mit obiger Formel berechnet werden können.

Dennoch sind einige unterschiedliche Problemlösungen möglich. Dadurch wird eine Aufwandsschätzung erreicht, welche natürlich unterschiedliche Fehlertoleranzgrenzen hinnimmt und aus diesem Grund immer respektvoll von Projektexperten beurteilt und eingeschätzt werden sollten.

Maße für Software-Produktivität

Aufwandsschätzverfahren können grundsätzlich eingeteilt werden in größenspezifische Maße und funktionsspezifische Maße. Erstere werden aufgrund von Programmängen, Anzahl von Operatoren und Operanden und ähnlichen Software-Maßen ermittelt. Zweitere sind abhängig von der Zählung verschiedener Programmierelemente. Die wohl bekanntesten Vertreter sind folgende:

- **Größenspezifische Maße: LOC** (engl. „Lines-of-Code“ siehe auch Seite 50), Anzahl gelieferter Objektcodeanweisungen, Anzahl Seiten der Dokumentation
- **Funktionsspezifische Maße: Function-Points** (siehe Seite 51), **Object-Points** (siehe [Bal09]).

Einflussfaktoren auf die Produktivität

Eine Schwierigkeit bei der Aufwandsschätzung ist, dass es einige größere Einflussfaktoren auf die Produktivität in Software-Projekten gibt. Die bekanntesten sind:

- Erfahrungen im Anwendungsgebiet
- Kenntnisse ermöglichen effektive Software-Entwicklung
- Prozessqualität
- verwendetes Vorgangsmodell
- Projektgröße (Beispiel: große Projekte erfordern mehr Kommunikation)
- technische Unterstützung
- Verwendung guter Hilfswerkzeuge
- Arbeitsumgebung
- persönliches Wohlbefinden der Teammitglieder.

Probleme mit LOC

Das Größenmaß „Lines-of-Code“ (Abk. LOC) ist ein Maß zur groben Einschätzung eines Projektes. Jedoch ist es für die Ermittlung präziser Schätzwerte nicht geeignet.

Probleme bei der Messung des LOC sind, dass beispielsweise keine Standards zur Zählung der Quellcodezeilen vorhanden sind. Die Quellcodelänge ist jedoch **programmiersprachenabhängig**. Der Quellcode der Programmiersprache C kann beispielsweise meist sehr viel kompakter (und oft auch unlesbarer) geschrieben werden als ein Quellcode der Programmiersprache Ada (Programmiersprache geeignet für sicherheitsrelevanten IT-Projekte). Grundsätzlich kann jedoch nicht entschieden werden, ob eine ausführliche Programmierung besser als eine komprimierte Programmierung ist, da dies sehr stark vom Anwendungsfall abhängt. Weiterhin sind auch **Datendeklarationen** für die LOC Messung nicht vereinheitlicht und werden somit leider unterschiedlich gezählt. Zuletzt seien auch noch die **Leerzeilen** eines Quellcodes genannt. Auch diese werden nicht vereinheitlicht mit in den LOC-Wert eingerechnet, so dass eine große Ungenauigkeit und wenig Vergleichbarkeit bei der Aufwandsschätzung mit LOC entsteht.

Ein wichtiges Kriterium ist auch die Aussagekraft einer LOC-basierten Aufwandschätzung. **Guter** oder **schlechter Quellcode** sind hier **nicht unterscheidbar**. Daher sind Produktivitätsvergleiche zwischen Systemen nur mit Vorbehalten durchführbar.

Function-Points (Albrecht, 1979 [Alb79], Albrecht und Gaffney, 1983 [AG83])

Bei der Function-Points-Methode handelt es sich um ein programmiersprachen-unabhängiges Instrument zur Aufwandsschätzung von IT-Projekten. Es dient zum Vergleich der Produktivität zwischen Software-Systemen. Die Methode eignet sich für ein- und ausgabeorientierte Datenverarbeitungssysteme. Ein FP (d. h. *Function Point*) wird dabei beispielsweise pro Personenmonat gezählt.

Methode

Gesamtanzahl an **Function-Points** = Messung der Anzahl von:

- externen Ein- und Ausgaben
- Benutzerinteraktionen
- externen Schnittstellen
- vom System verwendeten Dateien.

Basis sind die Produktanforderungen. Schritte zur Ermittlung der Function-Points (siehe Abbildung 3.1):

- 1. Schritt: Kategorisierung der Anforderungen in Eingabedaten, Abfragen, Ausgabedaten, Datenbestände, Referenzdaten
- 2. Schritt: Klassifizierung jeder Anforderung in einfach-mittel-schwierig

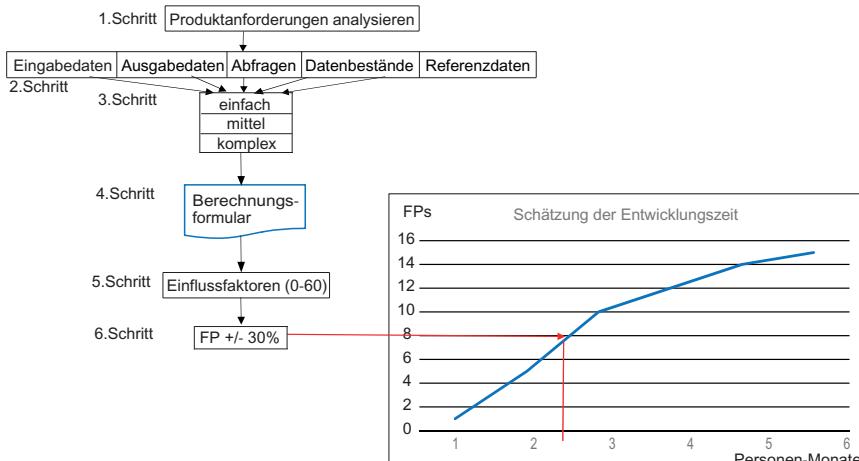


Abbildung 3.1 Function-Points-Methode aus [Bal09]

- 3. Schritt: Eintragung der einzelnen Anzahl in das Berechnungsformular; Ermittlung unbewerteter Function-Points
- 4. Schritt: Bewertung der FPs mit projektrelevanten Einflussfaktoren (0–60) wie beispielsweise „Erfahrung des Teams“ oder „Kenntnisgrad der Programmiersprache“
- 5. Schritt: Berechnung der bewerteten Function-Points
- 6. Schritt: Ablesen des geschätzten Aufwandes aus einem Diagramm, indem die Function-Points (y-Achse) in Bezug gesetzt werden zu dem steigenden Aufwand (x-Achse in Personen-Monaten) von Projekten.

Das zum Vergleich herangezogene, zugrunde liegende FP-Diagramm wird dabei aufgebaut aus der tatsächlichen Dauer früherer Projekte. Es basiert daher auf Erfahrungswerten.

Zusammenfassung der Probleme bei Maßen mit Volumen/Zeit

Zusammenfassend ergeben sich mit großen- und funktionsspezifischen Maßen folgende Probleme:

- nicht-funktionale Eigenschaften der Software werden nicht berücksichtigt (Beispiele: Zuverlässigkeit, Wartungseigenschaften, Verbesserung von Quellcode, etc.)
- Wiederverwendung wird nicht berücksichtigt
- sie setzen voraus: **mehr = besser**.

Manager sollten daher gut geschult werden und vorsichtigen Umgang mit diesen Maßen pflegen.

Übersicht einiger alternativer Schätzverfahren

Neben den bisher besprochenen **algorithmischen Aufwandsschätzmodellen** gibt es noch einige weitere Verfahren, die sich in der Praxis bewährt haben. Diese werden hier gelistet.

- **Algorithmisches Aufwandsmodell**

LOC mit Schätzung der Projektkosten, Function-Points, Object-Points,
COCOMO (engl. Constructive-Cost-Model)(siehe [Som18])

- **Expertenbeurteilung**

Mehrere Schätzungen werden gemittelt oder die Einigung auf eine Schätzung erzielt

- **Analogieschätzung**

Kosten werden analog zu abgeschlossenem ähnlichen Projekt geschätzt

- **Parkinsons-Gesetz**

Nach dem Motto: „Arbeit dehnt sich immer so aus, dass sie genau die Zeit braucht, die man für sie erübrigen kann.“

Kosten = verfügbare Ressourcen und Lieferfrist

- **Price-to-win**

Hier entsprechen die Projektkosten dem Kunden-Budget.

Allgemeine Formel für Softwarekosten (Kitchenham 1990)

Formel

$$\text{Aufwand} = A * \text{Größe}^B * M \quad (3.2)$$

- **A:** konstanter Faktor für unternehmenseigene Praktiken und Software-Art
- **Größe:** Schätzung der Größe LOC, Function Points oder Object Points
- **B:** Aufwand für übergroße Projekte und liegt zwischen 1,0..1,5
- **M:** Dieser Multiplikator setzt sich aus einer Kombination von Prozess-, Produkt- und Entwicklungsmerkmalen zusammen.

Grobe Schätzungen sind auch mit der Formel nach Kitchenham berechenbar. Jedoch unterliegen die einsetzbaren Werte oft jahrelanger Erfahrung von Projektleitern und werden durch unternehmensintern geführte Tabellen mit Werten bestückt.

Unsicherheit der Schätzung

Grundsätzlich sollte sich jeder, der sich mit Aufwandsschätzung beschäftigt, der Unsicherheit der Schätzwerte bewusst sein. Zu Anfang eines Projektes wird daher beispielsweise vielfach zu hoch geschätzt (siehe Abbildung 3.2). Schreitet das Projekt im Zeitverlauf fort, so werden die Schätzungen in der Regel genauer. Am Ende des Projektes sind die genauen Kosten schließlich bekannt.

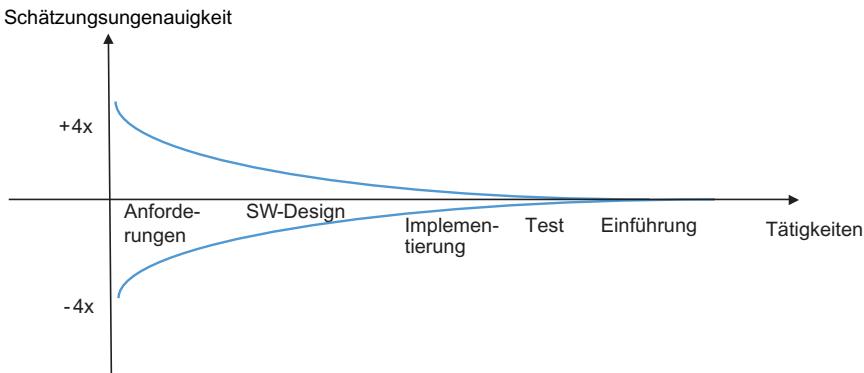


Abbildung 3.2 Ungenauigkeit der Schätzung in IT-Projekten [Som18, aus Boehm et al. 1995]

Das Teufelsviereck (Sneed, 1983 [Sne05])

In Abbildung 3.3 sind vier Einflussfaktoren auf Projekte zu sehen. Es handelt sich um

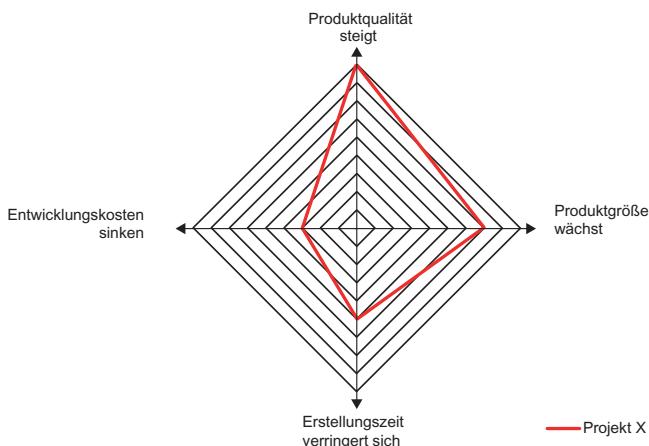


Abbildung 3.3 Teufelsviereck nach Sneed [Sne05]

die Faktoren Produktqualität, Produktgröße, Entwicklungskosten und die Erstellungszeit eines Software-Projektes. Sneed fand heraus, dass falls einer der Faktoren innerhalb eines Projektes sich ändert, mindestens noch ein weiterer Faktor sich ebenfalls ändert. Genauer gesagt bleibt nach Sneed der Umfang des sogenannten „Teufelvierecks“ immer gleich. Ist man sich dessen als Projektleiter bewusst, kann man natürlich auch entsprechend Projekte besser beherrschen.

■ 3.4 Risikomanagement

In diesem Abschnitt wird ein kurzer Einblick in das Risikomanagement gegeben, mit dem in der Planungsphase bei entsprechend großen Projekten begonnen wird. Risikomanagement beschäftigt sich mit den Risiken, die eine Softwareerstellung aus dem vorgesehenen Plan laufen lassen.

Risikokategorien

Risiken werden grundsätzlich in Kategorien eingeteilt, um sie besser zu verstehen und die Auswirkungen einfacher einschätzen zu können.

1. **Projektrisiken:** Zeitplan, Ressourcen

Personalveränderungen, Managementveränderungen, Nichtverfügbarkeit von Hardware/Software, Änderung von Anforderungen, Verzögerungen, Unterschätzung des Umfangs

2. **Produktrisiken:** Qualität, Leistung

Änderung von Anforderungen, Verzögerungen, Unterschätzung des Umfangs, schlechte CASE-Tools, unzureichende Hilfsmittel

3. **Wirtschaftliche Risiken:** Auswirkung auf Unternehmen

Technologieveränderungen, Konkurrenz.

Ziele des Risikomanagement sind die Risikoerkennung, die Vermeidung und die optimale Behandlung von eingetretenen Risiken.

Auch im Risikomanagement gibt es wiederum eine Art Vorgehensmodell, bei dem ein erprobter Prozessablauf vorgeschlagen wird (siehe Abbildung 3.4). Prozessschritte sind in der Abbildung hellblau hinterlegt, während Prozessergebnisse dunkelblau erscheinen. Zunächst muss eine Auflistung aller möglichen Risiken (natürlich vor dem Eintreten!) vorgenommen werden. Den zugehörigen Prozessschritt nennt man Risikoerkennung. Ergebnis ist eine Liste potentieller Risiken.

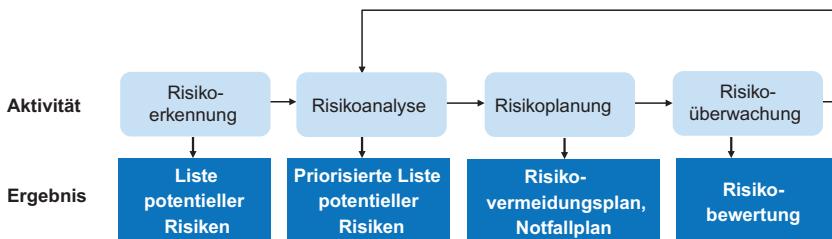


Abbildung 3.4 Prozessablauf beim Risikomanagement nach [Som18]

Zweiter Schritt ist die Risikoanalyse, aus der als Ergebnis eine priorisierte Liste potentieller Risiken hervorgeht (d. h. Risiken werden nach der Wahrscheinlichkeit des Auftretens sortiert). Anschließend findet die Risikoplanung statt. Dabei werden Vermeidungspläne und Notfallpläne für die einzelnen Risiken erdacht und aufgestellt. Während der Projektlaufzeit findet schließlich eine Risikoüberwachung statt, bei der kontinuierlich auch eine Risikobewertung vorgenommen wird. Treten neue Risiken auf, so kann jederzeit wieder in die Risikoanalyse eingetreten werden.

Einzelne Risiken sollten dabei so genau wie möglich auf die Einzelprojekte zugeschnitten sein, da „Allround-Risiken“ wie beispielsweise „Zeitplan in Gefahr“ sehr häufig in Projekten auftreten. Wenn jedoch projektspezifisch eine Zeitplanverschiebung wegen eines bestimmten Mitarbeiterausfalls droht, ist dies ein konkretes Risiko, das auch entsprechend durch Notfallpläne behandelt werden kann.

■ 3.5 Zusammenfassung

In der Planungsphase wird das Software-Projekt aufgesetzt. Ein wichtiges Ergebnis-dokument der Planungsphase ist das Lastenheft, das immer aus Kundensicht die Aufgabenbeschreibung enthält. Es dient dem Projektteam des Softwareproduzenten als Eingabe für die Erstellung des Pflichtenheftes.

Neben dem im letzten Kapitel beschriebenen Projektmanagement findet in der Planungsphase häufig auch noch eine Aufwandsschätzung statt, um Kundenangebote erstellen zu können. Methoden wie LOC, „Function-Points“, „Object-Points“, COCOMO nennen sich algorithmische Aufwandschätzung. Daneben gibt es noch die Expertenbeurteilung, Analogieschätzung, Parkinsons Gesetz und die Methode „Price-to-win“. Ein Projektleiter sollte sich jedoch darüber im Klaren sein, dass Schätzmethoden weit entfernt von den realen Kosten des Projektes sein können, sich jedoch

gegen Projektende diesen annähern. Einflussfaktoren werden auch durch das Teufelsviereck von Sneed verdeutlicht.

Zuletzt wird noch ein Blick auf das Risikomanagement von IT-Projekten geworfen. Der zugehörige Prozessablauf wird beschrieben: Nach einer Risikoanalyse findet eine Risikoplanung mit der zugehörigen Erstellung von Vermeidungs- und Notfallplänen statt.

■ 3.6 Aufgabensammlung

In diesem Abschnitt sind Fragen und Aufgaben bezüglich der Planungsphase zu finden.

1. Was sind die Aufgaben und Ergebnisse der Planungsphase?
2. Lasten- und Pfichtenheft: Welche Aussage stimmt?
 - A) Das Lastenheft ist aus der Sicht des Auftraggebers geschrieben.
 - B) Das Lastenheft ist aus der Sicht des Auftragnehmers geschrieben.
 - C) Das Pfichtenheft ist aus der Sicht des Auftraggebers geschrieben.
 - D) Das Pfichtenheft ist nicht aus der Sicht des Auftragnehmers geschrieben.
3. Welche Schätzverfahren gibt es für den Aufwand einer Software?
4. Eine Anforderung hat folgenden geschätzten Aufwand:
Ein Entwickler benötigt 5 Stunden, der Software-Architekt benötigt 12 Stunden und der Projektmanager 2 Stunden zur Erledigung der Anforderung.
Wie hoch sind die geschätzten Kosten der Anforderung, wenn für alle Beteiligten ein Stundensatz von 90 Euro angenommen werden kann?
5. Was sind Ziele des Risikomanagements bei der Softwareerstellung?
6. Welche Dokumente benötigt man beim Risikomanagement üblicherweise?
 - A) Spezifikation
 - B) Lastenheft
 - C) Notfallplan
 - D) Netzplan

4

Definitionsphase

4.1 Überblick Definitionsphase

Grundsätzliche Vorgehensweise

In der Definitionsphase eines Softwareprojekts werden die (Software-Produkt-) Anforderungen erstellt. Die Anforderungen beinhalten nach Fertigstellung die **fachliche Lösung**. Da man sie neben der schriftlichen, tabellarischen Form üblicherweise mit UML-Anwendungsfall-Diagrammen grafisch darstellt (siehe Abbildung 4.1, Notation siehe Kapitel 4.3.6.1), spricht man hier von der **Modellierung der fachlichen Lösung**. Alternativ werden Anforderungen heute in agilen Methoden in strukturierter Textform (siehe Kapitel 2.1.5) dokumentiert. Anforderungsmanagement bedeutet, dass man die oft zahlreichen Anforderungen eines IT-Projektes geordnet handhabt.

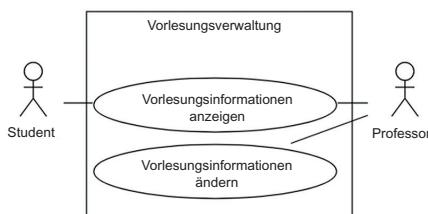


Abbildung 4.1 Anforderungen modelliert in einem UML-Use-Case-Diagramm



Definition

Was ist eine **Anforderung** in einem Software-Projekt?

- Eine Anforderung spezifiziert die qualitativen und quantitativen Eigenschaften eines Produkts aus Sicht des Auftraggebers.
- Eine Anforderung muss eindeutig und testbar sein.
- Anforderungen sind die Grundlage für die Systemarchitektur.

Aufgabe

Die Aufgabe in der Definitionsphase ist demnach das **Requirements-Engineering** (auf Deutsch: **Systemanalyse, Anforderungsanalyse**). Dabei werden Anforderungen ermittelt, analysiert, beschrieben und als fachliche Lösung modelliert. Optional können Anforderungen animiert, simuliert oder ausgeführt werden. Letztlich werden Anforderungen vom Projektteam verabschiedet (d. h. deren Implementierung wird beschlossen).

Woher kommen die Anforderungen?

Der Software-Engineering-Spezialist ist zuständig dafür, dass Anforderungen an die Software gesammelt werden. Dafür muss er kreativ eigene Ideen für die Funktionalität und die Gestaltung der Anwendung finden. Grundsätzlich können Anforderungen auch vom Auftraggeber erfragt werden (bzw. im Lastenheft enthalten sein). Aber auch die Fachabteilungen sollten zu Rate gezogen und befragt werden. Oft helfen auch Benutzer oder Repräsentanten der Endbenutzer, um Anforderungen zu klären. Manchmal ist auch die Marketingabteilung beteiligt, die Vorschläge unterbreitet, um sich von Konkurrenzprodukten zu unterscheiden.

Requirements-Engineering-Prozess

Prozessbeobachtungen, Anwenderwissen, jegliche Dokumentation (wie beispielsweise das Lastenheft) und alle Arten von Vorschriften, welche die Software betreffen, können gesammelt werden und als Eingabe für den ersten Prozessschritt im Requirements-Engineering-Prozess dienen (siehe Abbildung 4.2). Der Prozessvorschlag sieht vor, Anforderungen nacheinander zu ermitteln, zu analysieren, zu dokumentieren und schließlich zu prüfen. Wichtig zu erkennen ist, dass die einzelnen Prozessschritte nach einem Erstdurchlauf jeweils wiederholt werden können. Wichtigstes Ergebnisdokument ist schließlich das **Pflichtenheft** (bzw. die agile Anforderungsbeschreibung), das im nachfolgenden Unterkapitel genauer beschrieben wird.

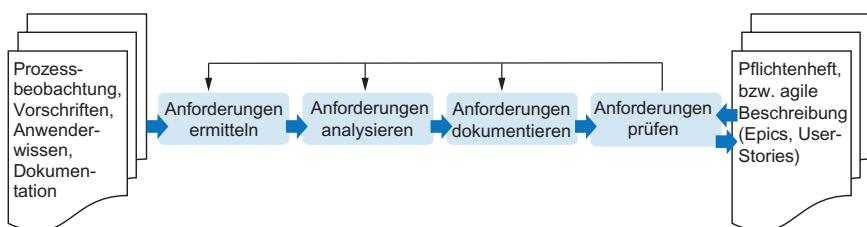


Abbildung 4.2 Der Requirements-Engineering-Prozess

Beteiligte

Prozessbeteiligte sind **Auftraggeber** (liefert Lastenheft) und **Projektleiter** (managt und verwaltet das Projekt). Wichtigste Rolle ist jedoch die des **Software-Engineering-Spezialisten**, der die fachlichen Anforderungen ermittelt und niederschreibt. Nach Balzert werden auch Systemanalytiker, Software-Ergonomen und technische Redakteure am Prozess beteiligt ([Bal09, S.99]). Häufig hängt die Beteiligung solcher Rollen jedoch auch von der Art der zu entwickelnden Software ab.

■ 4.2 Pflichtenheft

Wichtigstes schriftliches Ergebnis der Definitionsphase ist das Pflichtenheft (engl.: Specification, deutsches Synonym: Spezifikation). Es enthält alle fachlichen Anforderungen, die das Software-Produkt **aus Sicht des Auftraggebers** enthalten muss. Wichtig zu erkennen ist, dass es eine Beschreibung des Inhaltes der zu erstellenden Anwendung enthält (d. h. „**WAS**“) und nicht die technische Umsetzungsbeschreibung (d. h. „**WIE**“, siehe Kapitel 5 – Designphase). In vielen Projekten ist das **Pflichtenheft** auch heute noch **Vertragsgegenstand**, das den Lieferumfang eines Softwareprojektes definiert. Wird jedoch beispielsweise mit agilen Methoden Software entwickelt (siehe Kapitel 4.2), so werden aus dem Pflichtenheft die „**User-Stories**“ (oder textuelle Beschreibungen wie die sogenannten „**Epics**“), die jedoch nicht als juristischer Vertragsgegenstand genutzt werden, sondern nur die inhaltliche, fachliche Beschreibung der zu entwickelnden Anwendung enthalten.

Unterschiede Lastenheft – Pflichtenheft

Grundsätzlich erhalten Pflichtenhefte oft den gleichen Aufbau und die gleiche Gliederung wie Lastenhefte. Jedoch gibt es bei den beiden Dokumenten einige signifikante Unterschiede. Das Lastenheft ist eine Problemspezifikation aus Sicht des Kunden oder der beauftragenden Abteilung. Es wird dem Software-Produzenten übergeben, damit dieser seinen Auftrag besser verstehen kann. Das Pflichtenheft ist jedoch eine Anforderungsspezifikation aus Sicht des Software-Produzenten, um zu beschreiben, welches Produkt er dem Kunden übergeben wird. Aus diesem Grund können sich beide Dokumente von Inhalt und beschriebener Funktionalität unterscheiden.

- Lastenheft:
 - Problem spezifikation
 - Sicht des Kunden
 - geschrieben für den Software-Produzent
- Pflichtenheft:
 - Anforderungsspezifikation
 - Sicht des Software-Produzenten
 - geschrieben für den Kunden.

Agiles Modell

Moderne Pflichtenhefte, die der Vorgehensweise eines **agilen** Modells entsprechen, sind als **Product-Backlog** mithilfe von **User-Stories** (und anderen textuellen Beschreibungen wie **Epics**) verfasst. Sie werden meist in elektronischen „**Task-Boards**“ gespeichert. (Bei agilen Modellen wird hier jedoch der Begriff des Pflichtenhefts nicht mehr verwendet, beziehungsweise durch Begriffe wie „User-Stories“ und „Product-Backlog“ ersetzt). Da selbst-organisierend und interdisziplinär gearbeitet wird, hat das gesamte Scrum-Team Zugriff auf das Product-Backlog.

Benutzer-Pflichtenheft

Folgende Rollen arbeiten mit dem Pflichtenheft:

- Software-Engineering-Spezialist / Anwendungsspezialist
 - legt die Anforderungen fest
 - überprüft, ob Anforderungen den Erwartungen entsprechen
 - nimmt Anforderungsänderungen vor
- Projektmanager
 - erstellen des Angebots
 - Planung des Software-Entwicklungsprozesses
- Kunde
 - im Fall von agilem Vorgehen überprüft und/oder erstellt der Kunde User-Stories
 - der Kunde priorisiert die User-Stories
- Software-Entwickler
 - aufgrund Anforderungen verstehen, was für ein System entwickelt werden soll
 - nimmt aufgrund der Vorgaben Aufwandsschätzung einzelner Anforderungen vor
- Testpersonal
 - aufgrund Anforderungen verstehen, was für ein System entwickelt werden soll
 - Benutzung der Anforderungen für Validierungstests
- Wartungspersonal
 - Systemintegration, Wartung, Pflege.

■ 4.3 Requirements-Engineering

Im Requirements-Engineering (auf Deutsch: Systemanalyse, Anforderungsanalyse) werden Anforderungen an eine Software ermittelt, analysiert, beschrieben und als fachliche Lösung modelliert. Daher ist es zum Verständnis nützlich, sich mit Anforderungen, deren Arten und Inhalten auseinanderzusetzen.

4.3.1 Anforderungen

Grundsätzlich beinhaltet eine Anforderung den fachlichen Inhalt, der erforderlich ist, um eine qualitativ hochwertige Anwendung zu erstellen. Zusätzlich können Anforderungen jedoch auch noch Angaben zur Art des Systems, über verwendete Normen und Standards, über die Kritikalität des Systems, den gewünschten Systemumfang, die Komplexität der Systemabläufe, die Beobachtbarkeit der Arbeitsschritte, die Art der Anforderungen, vorhandene Erfahrung im Fachgebiet und Angaben zum DetAILlierungsgrad der Anforderungen einbeziehen. Da selbst in kleinen Projekten oft schon eine große Anzahl von Anforderungen auftreten, ist eine Nummerierung derselben obligatorisch. Um eine große Anzahl an Anforderungen besser überschauen zu können, werden Anforderungen oft in sogenannte Anforderungsarten eingeteilt. Neben „MUSS“, „KANN“ und „SOLL“-Anforderungen (siehe beispielsweise MoSCoW-Methode in [BS02]) gibt es noch einige inhaltliche Kategorien, die hier zu unterscheiden sind und welche im Folgenden anhand von Beispielen beschrieben werden.

4.3.2 Anforderungsarten

Anforderungen werden üblicherweise zur besseren Übersicht in die Kategorien funktional, nicht-funktional, Problembereich-, Benutzer- und Systemanforderungen eingeteilt.

4.3.2.1 Funktionale Anforderung

Bei funktionalen Anforderungen handelt es sich um Aspekte, welche die Funktionalität der zu erstellenden Software betreffen. In der Regel entstehen daraus die Funktionen des IT-Systems oder Dienste, die das System leisten soll. Auch Aktionen, die das System selbstständig ausführen soll, können damit beschrieben werden. Letztlich können auch allgemeine funktionale Vereinbarungen und Einschränkungen beschrieben werden.

Beispiel

- „Senden Fehlermeldung XYZ zum Nachbarsystem, wenn Ereignis ABC eingetreten ist“
- „Lesen Datei XYZ in den Hauptspeicher ein“.

4.3.2.2 Nicht-funktionale Anforderung

Grundsätzlich gilt natürlich, alle Anforderungen, welche nicht direkt als Programm codiert werden (d.h. in Methoden oder Funktionen umgesetzt werden), sind nicht-funktionale Anforderungen. Dazu gehören technische Anforderungen, Anforderungen an die zu verarbeitenden Informationen, Qualitätsanforderungen und viele mehr. In Tabelle 4.1 werden einige der möglichen Beispiele aufgelistet.

Tabelle 4.1 Liste möglicher nicht-funktionaler Anforderungen nach [Som18]

Produktanforderungen	Unternehmens-anforderungen	Externe Anforderungen
<ul style="list-style-type: none"> ▪ Benutzbarkeitsanforderungen ▪ Effizienzanforderungen (z. B. Leistung/Performance, Speicherplatz, Geschwindigkeit) ▪ Zuverlässigkeitssicherungsanforderungen ▪ Portierbarkeitsanforderungen 	<ul style="list-style-type: none"> ▪ Lieferanforderungen ▪ Umsetzungsanforderungen ▪ Vorgehensanforderungen 	<ul style="list-style-type: none"> ▪ Kompatibilitätsanforderungen ▪ Ethische Anforderungen ▪ Rechtliche Anforderungen (z. B. Datenschutz, Sicherheit, Normen)

Beispiel

- „Pflichtfelder für den Benutzer werden mit einem Stern markiert“
- „Der Algorithmus XYZ muss in 2 Sekunden ausführbar sein“.

4.3.2.3 Problembereichsanforderung

Anforderungen, die sich aus dem Anwendungsbereich ergeben, nennt man Problembereichsanforderung oder auch **Domänenanforderung**. Sie enthalten spezielle Bedürfnisse von Systembenutzern, die sich aus der Domäne ergeben. Die Domäne ist dabei die Umgebung, in der die neue Software eingesetzt werden soll. Beispielsweise ist die Domäne einer Museumsführer-Software das Museum. Sollen etwa Exponate auch bestimmten Räumen zugeordnet werden können, so ist die zugehörige Anforderung „Exponat Raum zuordnen“ eine Problembereichsanforderung.

Beispiel

- „Es sollte eine Standardbenutzerschnittstelle zu allen beteiligten Datenbanken geben, die auf ORM (engl. Object Relational Mapper) basiert“ (Domäne Datenbanken).

4.3.2.4 Benutzeranforderung

Eine Benutzeranforderung beschreibt oft das externe Verhalten des Systems, wie beispielsweise Ein- und Ausgaben. Dabei sollen die Anforderungen verständlich für Systembenutzer geschrieben werden. Oft handelt es sich bei Benutzeranforderungen auch um sogenannte „Komfortfunktionen“ für den Benutzer, die ihm das Leben mit der neuen Software zusätzlich erleichtern sollen.

Beispiel

- „Die aktuelle Ansicht ist zu drucken, wenn die Schaltfläche *Drucken* vom Benutzer gewählt wird“.

4.3.2.5 Systemanforderung

Bei Systemanforderungen handelt es sich meist um Anforderungen, die als genauere Beschreibung der Benutzeranforderungen dienen. Sie beschreiben, wie die Benutzeranforderungen umgesetzt werden sollen (nicht zu verwechseln damit, wie sie implementiert werden sollen!).

Beispiel

- „Das Datenaustauschformat für die zu übertragende Datei der Benutzeranforderung A soll XML sein“
- „Das System soll eine Client-Server Architektur besitzen“.

4.3.3 Qualitätsmerkmale für Anforderungen

Anforderungen sollten definierten Qualitätskriterien entsprechen, die vom Projektteam festgelegt werden. Ein Vorschlag möglicher, sinnvoller Kriterien ist im Buch von Rupp et.al [RSG14, S.21] wie folgt beschrieben. Die festgelegten Qualitätskriterien sollten alle jedoch möglichst **messbar**, **nachprüfbar** und **verfolgbar** sein.

- **Vollständigkeit:** Unvollständige Anforderungen führen zu einem unvollständigen Softwareprodukt.
- **Korrektheit:** Die Richtigkeit von Anforderungen ist Voraussetzung dafür, dass das richtige Produkt entwickelt werden kann.

- **Klassifizierbarkeit:** Eine gute Anforderung lässt sich in eine einzige Kategorie (siehe Kapitel 4.3.2 – Anforderungsarten) einteilen.
- **Konsistenz:** Eine gute Anforderung ist konsistent in sich selbst.
- **Prüfbar:** Eine gute Anforderung ist nachprüfbar durch geeignete Testmethoden.
- **Eindeutigkeit:** Eine gute Anforderung ist eindeutig und enthält niemals mehrdeutige Begriffe oder Textpassagen.
- **Verständlichkeit:** Ein qualitativ hochwertige Anforderung ist in verständlicher natürlicher Sprache formuliert.
- **Aktualität:** Eine Anforderung, die nicht aktuell ist, sollte aus dem aktuellen Anforderungskatalog herausgenommen werden, da sie nicht umgesetzt werden muss.
- **Realisierbarkeit:** Eine Anforderung muss als Software realisiert werden können.
- **Notwendigkeit:** Eine Anforderung, die nicht notwendig ist, sollte (evtl. für zukünftige Versionen) ausgelagert werden.
- **Verfolgbarkeit:** Eine Anforderung muss verfolgbar sein.
- **Bewertbarkeit:** Eine gute Anforderung ist bewertbar. Das Projektteam ist gefordert, hier geeignete Bewertungskriterien zu finden.

4.3.4 Beschreibung von Anforderungen

Grundsätzlich können Anforderungen an ein Software-System in natürlicher Sprache, in strukturierter Sprache oder in grafischer Notation formuliert werden.

4.3.4.1 Natürliche Sprache

Standard ist, dass Anforderungen in natürlicher Sprache verfasst werden. Ein Beispiel für eine natürliche sprachliche Anforderung ist:

„Es muss das aktuell angezeigte Rechnungsformular ausgedruckt werden, wenn die Schaltfläche *Drucken* vom Benutzer gewählt wird.“

Häufig werden hierfür Formularvorlagen verwendet, um die Lesbarkeit über viele Projekte hinweg zu vereinfachen.

Methoden

- Standardformulare
- Vorlagen.

Beispiele

- Word/Excel-Vorlagen.

Probleme

- Vorlagen sind oft zu wenig strukturiert
- Mangel an Genauigkeit
- Verwirrung bei Anforderungen
- Verschmelzung von Anforderungen.

Zur Strukturierung und Verbesserung der Wortwahl bei Anforderungen können Sprachschablonen wie in Abbildung 4.3 verwendet werden.

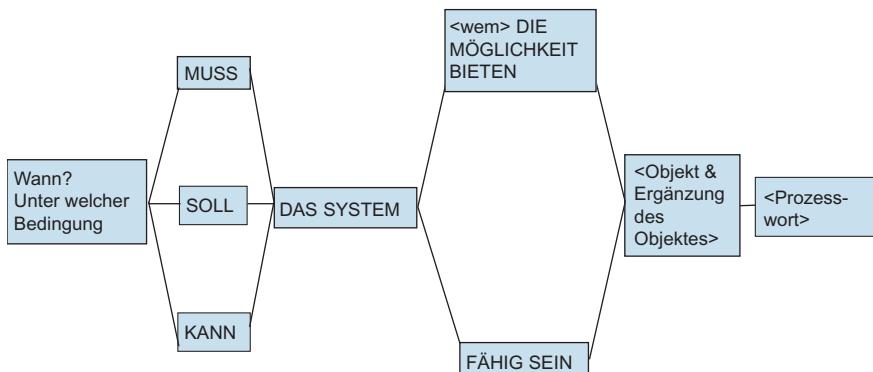


Abbildung 4.3 Beispiel einer Sprachschablone von Rupp [RSG14]

Beispiel (mit angewandter Sprachschablone):

„Das System ‚Kfz-Vermietung‘ muss dem Administrator die Möglichkeit bieten, die aktuelle Fehlermeldung auf einem der momentan verfügbaren Drucker zu drucken.“

User-Stories:

Bei der agilen Vorgehensweise wird ebenfalls mit natürlicher Sprache und mithilfe von Sprachschablonen gearbeitet (siehe Beispiel in Abbildung 4.4).

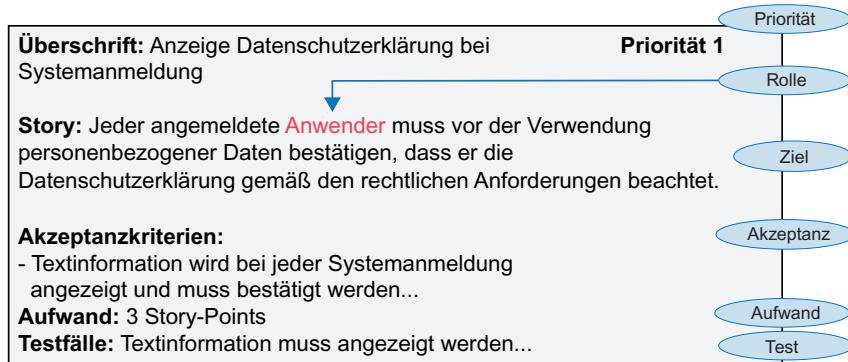


Abbildung 4.4 Beispiel einer User-Story

4.3.4.2 Anforderungen in strukturierter Sprache

Strukturierte Sprachen sind alle Sprachen, die (teil-)formalisiert sind. Sie können meist einfach von Programmierern verstanden werden, sind jedoch auch oft unverständlich oder schwer lesbar für den Kunden.

Methode

- ähnlich wie Programmiersprache
- Definition eines „Befehlssatzes“.

Beispiele

- PDLs (engl. Program Description Languages)
- Eigenkreationen
- Pseudocode.

Anwendungsbeispiel:

In Listing 4.1 ist ein Ausschnitt aus einer PDL-Sprache zu sehen.

Listing 4.1 Beispiel einer PDL

```

if ((Datei laden) == ok)
then {Datei lesen}
else {print (Fehlermeldung 'Datei nicht ladbar!')}
    
```

Mögliche Probleme

- Befehlssatz nicht ausreichend oder nicht genau genug
- nur für Software-Entwickler verständlich
- Anforderungen können für einen Software-Entwurf gehalten werden, was zu vermeiden ist.

4.3.4.3 Anforderungen in grafischer Notation

Die grafische Notation von Anforderungen ist mittlerweile Standard und ergänzt die Anforderungsbeschreibung in natürlicher Sprache durch mehr Übersicht. Üblicherweise werden hierfür die **UML-Use-Case-Diagramme** eingesetzt.

Methode

- grafische Sprache
- ergänzt durch Kommentare und Anwendungsfallbeschreibungen.

Beispiele

- **SA – Strukturierte Analyse** (engl.: Structured Analysis; Methode für prozedurale Programmierung, siehe [Mar79])
- **UML: Use-Case-Diagramm** (Methode für objektorientierte Programmierung).

Ein ausführlicheres Praxis-Beispiel ist in Kapitel 4.3.6 (Anforderungsanalyse) in Form eines Use-Case-Diagrammes zu finden.

Probleme

- manchmal sagen Worte mehr als Bilder
- Kommentare werden oft zu spärlich eingesetzt
- nur für Software-Engineering-Spezialisten und Software-Entwickler verständlich.

Probleme bei der Anforderungsanalyse

Nur durch eine vollständige und gelungene Anforderungsanalyse wird der Weg für die Entwicklung eines erfolgreichen Softwareprojekts bereitet. Jedoch ergeben sich in der Praxis oft vielfältige Schwierigkeiten, welche die erfolgreiche Softwareentwicklung letztlich behindern. Einige der typischen Probleme sind hier aufgezählt:

- unklare Zielvorstellungen
- hohe Komplexität
- Sprachbarrieren
- veränderliche Anforderungen
- schlechte Qualität der Anforderungen
- Beschreibung unnötiger Merkmale
- ungenaue Planung und Verfolgung des Projekts.

Alle Projektteilnehmer sind wichtig und angehalten, diesen Problemen mit geeigneten Maßnahmen entgegenzutreten.

4.3.5 Erhebung von Anforderungen

Die Tätigkeit des „Ausdenkens“ von Anforderungen einer Anwendung gehört zu den kreativsten Arbeiten im Erstellungsprozess. Man nennt sie *Erhebung* von Anforderungen. Die Projektteilnehmer, welche hier gute Ideen entwickeln sollen, sind hauptsächlich die Software-Engineering-Spezialisten.

Die Erhebung von Anforderungen ist oft gar nicht so leicht. Sie kann jedoch erlernt und geübt werden. Dazu gibt es einige häufig verwendete Ermittlungstechniken, die nun zusammengefasst werden und in [RSG14] ausführlicher beschrieben sind.

Einige mögliche Ermittlungstechniken

- Kreativitätstechniken
 - **Brainstorming**
Ereignisse sammeln
 - **Methode 6-3-5**
6 Teilnehmer entwickeln je 3 Ideen und geben diese auf je einem Kärtchen dem jeweiligen Nachbarn, der diese dann kommentiert. Karten weitergeben, bis jeder Teilnehmer jede Karte besessen hat (5-mal)
- Beobachtungstechniken
 - **Feldbeobachtung**
Systemanalytiker beobachtet die Arbeitsabläufe von Benutzern während ihrer täglichen Arbeit
 - **Apprenticing** („in die Lehre gehen“)
Systemanalytiker lernt die Arbeitsabläufe der Benutzer unter deren Anleitung

- Befragungstechniken
 - **Fragebogen**
Multiple-Choice-Fragen, offene Fragen an geeignete Benutzer und Entscheider
 - **Interview**
Systemanalytiker stellt mündliche Fragen
 - **On-Site Customer**
Ein versierter Vertreter des Kunden begleitet die Entwicklung beim Software-Produzenten vor Ort
- Vergangenheitsorientierte Techniken
 - **Systemarchäologie**
Analoge existierende Systeme und die dazugehörigen Dokumente werden beobachtet bzw. gelesen
 - **Reuse**
Anforderungen eines ähnlichen Systems werden wiederverwendet und abgeändert.

Tabelle 4.2 Beispiele zur Wahl einer Ermittlungstechnik aus [RSG14]

Ermittlungstechnik	Geeignet wenn ein Wissensträger...
Apprenticing	<ul style="list-style-type: none"> ▪ ... schlechte kommunikative Fähigkeiten besitzt, ▪ ... implizites Wissen besitzt, ▪ ... geringes Abstraktionsvermögen hat ▪ ... bei problematischer Gruppendynamik
Fragebogen	<ul style="list-style-type: none"> ▪ ... geringe Motivation besitzt, ▪ ... geringes Abstraktionsvermögen hat, ▪ ... bei divergierenden Stakeholder-Meinungen
On-Site Customer	<ul style="list-style-type: none"> ▪ ... implizites Wissen besitzt, ▪ ... bei problematischer Gruppendynamik

In Tabelle 4.2 finden sich Beispiele zur Anwendbarkeit von Ermittlungstechniken. Der Tabelle kann entnommen werden, welche Ermittlungstechnik bei der Kommunikation mit Wissensträgern zur aktuellen Situation passt. Eine umfangreichere Tabelle ist in [RSG14] publiziert.

4.3.6 Anforderungsanalyse – Notation im Überblick

Die schriftliche Anforderungsanalyse kann mithilfe der UML grafisch unterstützt werden. Zur Steigerung der Übersichtlichkeit der natürlich sprachlichen Anforderun-

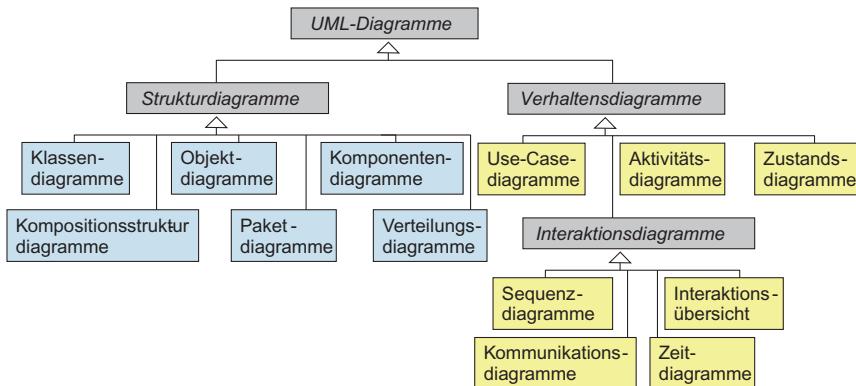


Abbildung 4.5 UML Diagrammarten – Einordnung Use-Case-Diagramme

gen werden sogenannte **Use-Case-Diagramme** (auf Deutsch: **Anwendungsfalldiagramme**) ergänzend verwendet. Die Use-Case-Diagramme gehören zu den Verhaltensdiagrammen, da mit ihnen die grundlegenden Funktionalitäten und hauptsächlich die Benutzerinteraktionen übersichtlich dargestellt werden (s. Abb. 4.5).

4.3.6.1 Anwendungsfalldiagramm (engl. Use-Case-Diagramm)

Use-Case-Diagramme geben eine erste Übersicht, welche Funktionen ein Software- system grundsätzlich bietet und wer diese ausführen kann. Sie bestehen hauptsächlich aus Systemrahmen, Akteuren, Assoziationen und Anwendungsfällen (engl.: Use- Case). Die Darstellung dieser Notationselemente und deren Bedeutung ist in Tabelle 4.3 zu finden.

Tabelle 4.3 Wichtige Notationselemente des Use-Case-Diagramms

Notation	Element	Bemerkung
Systemname	Systemrahmen	Systemgrenzen – Sie beschreiben, welche Komponenten und Akteure systemintern und welche systemextern beteiligt sind. Ein Systemrahmen ist mit einem Systemnamen zu beschriften.

Notation	Element	Bemerkung
 Akteur	Akteur	Akteure können beteiligte Personen oder auch Fremdsysteme sein. Bei Personen wird hier die Bezeichnung der Rolle zur Beschriftung verwendet. (Fremdsysteme können auch als Würfelsymbol dargestellt werden.)
—	Assoziation	Verbindungslien zwischen Akteur und Use-Case; Sie geben Antwort auf die Frage "Wer tut oder darf was?"
	Anwendungsfall (engl.: Use-Case)	Funktionen und Funktionalität, welche das zu erstellende System leisten soll. Ein Use-Case wird (zur Vollständigkeit) mindestens mit einem Nomen (d. h. wer oder was tut etwas?) und einem Verb (d. h. was geschieht/ passiert?) beschrieben.

Wichtig zu wissen ist auch, dass eine **Verfeinerung des Softwaresystems** durch weitere Use-Case-Diagramme und durch andere Diagrammarten aus der UML (z. B. Aktivitätsdiagramme) üblich ist. Ein Beispiel dafür ist auch in Kapitel 5.3.2.2 (Aktivitätsdiagramme) in Abbildung 5.18 und Abbildung 5.19 zu finden.



Merke

Im Anwendungsfalldiagramm erhält ein zu entwickelndes Softwaresystem einen **Systemrahmen**. Das System muss benannt und der Name im Systemrahmen oben links vermerkt werden.

Akteure werden durch Rollenangaben benannt (z. B. Mitarbeiter, Benutzer, Administrator). Sie werden durch **Assoziationen** (das sind die Verbindungslien) mit einem als ovalen Kreis dargestellten Use-Case verbunden.

Ein **Use-Case** wird mindestens mit einem **Nomen** und einem **Verb** beschrieben (z. B. „Partner suchen“) und üblicherweise in **Präsenz-Gegenwartszeitform** geschrieben. Er stellt die Funktionalität dar, die ein Benutzer mit dem System ausüben kann.

Beispiel

In Abbildung 4.6 ist ein Sportpartnerschaftsvermittlungssystem dargestellt. Es soll einem Benutzer die Möglichkeit bieten, interessierte Sportpartner zu finden und mit ihm zusammenzubringen.

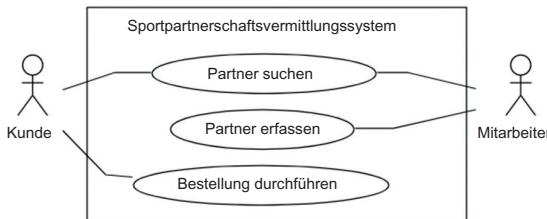


Abbildung 4.6 Beispiel eines UML-Use-Case-Diagramms

Tabelle 4.4 Tabellarische Beschreibung des Use-Case „Partner suchen“

Use-Case	„Partner suchen“
Use-Case-ID	1
primärer Akteur	Kunde
sekundärer Akteur	Mitarbeiter
Kurzbeschreibung	Ein Kunde stellt eine Suchanfrage an das Partnerschaftsvermittlungssystem, um passende Partner zu finden. Die Suche soll nach Alter, Geschlecht und Gr\u00f6\u00dfe des Partners einschr\u00e4nkbbar sein.
Vorbedingung	Kunde muss eingeloggt sein.
Ablauf der Ereignisse	<ol style="list-style-type: none"> 1. Kunde gibt Attribute f\u00fcr die Suchanfrage ein. 2. Kunde best\u00e4tigt mit Schaltfl\u00e4che „OK“. 3. Serverroundtrip da Client-Server-System: Server sucht nach Partnern mit gegebenen Suchkriterien in der Datenbank. 4. Ausgabe der Liste mit m\u00f6glichen Partnern.
Nachbedingungen	
alternativer Ablauf/Ausnahmen	Falls keine Treffer vorhanden: Ausgabe von Fehlermeldung „Keine passenden Partner vorhanden!“.
nicht-funktionale Anforderungen	Der Kunde muss \u00e4lter als 18 Jahre alt sein. Dies muss vor der Ausgabe der Ergebnisliste gepr\u00fctft werden.
Annahmen	Kunde kann sich nur erfolgreich einloggen, wenn er \u00e4lter als 18 Jahre ist.
Quelle	Tim Meier
Autor	Anja Metzner
Datum	01.10.2019

Das Diagramm enthält einen **primären** Akteur, den Benutzer, und einen **sekundären** Akteur, den Mitarbeiter. Beide Akteure können die Funktion „*Partner suchen*“ auslösen. Der Mitarbeiter kann zusätzlich „*Partner erfassen*“. Ein Kunde kann auch die Funktion „*Bestellung durchführen*“ aufrufen, was ihm die Adressdaten eines möglichen Sportpartners liefern soll. (Anmerkung: Man könnte die Funktion natürlich auch genauer „Adressdatenbestellung“ o. ä. nennen, jedoch soll hier die Analogie zwischen der Bestellung von normalen Waren(-Produkten) und Daten als Ware hervorgehoben werden.)

Zwei Bemerkungen zu diesem Beispiel sind wichtig. Erstens, um das Beispiel gut verstehen zu können ist eine weitere **textuelle Beschreibung** notwendig, die in **tabellarischer Form** für das gegebene Beispiel gleich in Tabelle 4.4 präsentiert wird. Solche tabellarischen Beschreibungen sind üblich. Die gezeigte Tabelle kann daher als Vorlage für beliebige andere Use-Cases verwendet werden.

Zweitens, die Use-Cases mit den dahinter liegenden Funktionalitäten können in **beliebiger Reihenfolge** aufgerufen werden. Grundsätzlich gibt es in UML-Use-Case-Diagrammen nicht die Möglichkeit, eine zeitliche Ablauffolge der Aufrufe zu definieren. In der Praxis sieht man jedoch häufig die Verwendung von Use-Case-**Nummerierung** in Verbindung mit einem UML-Kommentarfeld, das schließlich auf eine gewünschte Aufrufreihenfolge aufmerksam macht.

Auch um programmierbare Arbeitsportionen zu erhalten, können durch Nummerierung „große“ Use-Cases in „kleinere“ Use-Cases zerlegt werden. Im Beispiel von Abbildung 4.7 wird durch Nummerierung der Use-Case „1. Partner ändern“ in die beiden Use-Cases „1.1 Adresse ändern“ und „1.2 E-Mail ändern“ zerlegt.

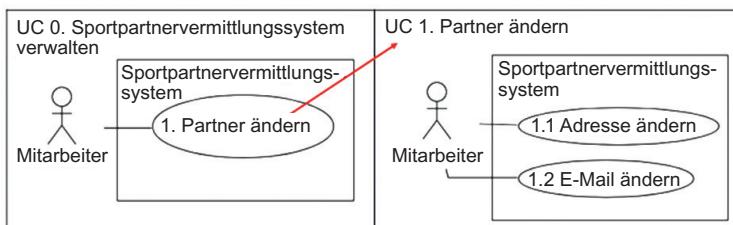


Abbildung 4.7 Mögliche Nutzung einer Nummerierung für Use-Cases

Neben den einfachen Assoziationsbeziehungen zwischen Akteur und Use-Case gibt es auch noch drei Beziehungsarten, die zwischen zwei Use-Cases stehen dürfen. Es handelt sich um die **Include-Beziehung**, die **Extend-Beziehung** und um den **spezialisierten Anwendungsfall**.

Include-Beziehung

Im Use-Case-Diagramm wird eine **Include-Beziehung** in zwei Fällen verwendet. Erstens, wenn ein großer Use-Case in kleinere Use-Cases gesplittet werden soll (siehe Abbildung 4.8 a). Dies kann bei der Nivellierung des Arbeitsaufwandes eines Use-Cases helfen. Und zweitens, wenn gemeinsame Teile des Verhaltens aus zwei oder mehreren Use-Cases ausgelagert werden sollen. Dies dient zur Reduzierung von redundantem Programmcode (siehe Abbildung 4.8 b).

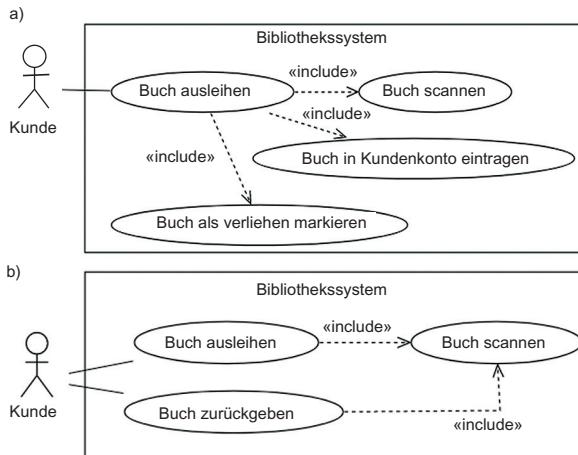


Abbildung 4.8 Beispiel zur Verwendung der Include-Beziehung des UML-Use-Case-Diagramms

Im Beispiel von Abbildung 4.8 a) wird der Use-Case „*Buch ausleihen*“ in drei kleinere Anwendungsfälle zerlegt. Er besteht also aus den Aktionen „*Buch scannen*“, „*Buch in Kundenkonto eintragen*“ und „*Buch als verliehen markieren*“.

Im Fall von Abbildung 4.8 b) wird die gemeinsame Aktivität „*Buch scannen*“ aus den beiden anderen Anwendungsfällen ausgelagert. Redundanzen sind damit beseitigt.

Extend-Beziehung

Eine **Extend-Beziehung** wird eingesetzt, wenn Anwendungsfall A um Anwendungsfall B erweitert wird (siehe Abbildung 4.9). (Zu beachten gilt, dass die Pfeilrichtung bei einer Include-Beziehung genau andersherum ist.)

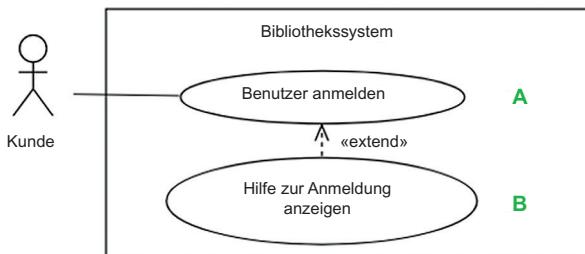


Abbildung 4.9 Extend-Beziehung des UML-Use-Case-Diagramms

Extend-Beziehung mit **Erweiterungspunkt** (engl.: Extension-Point): Anwendungsfall A wird unter der angegebenen **Bedingung** um Anwendungsfall B erweitert (s. Abb. 4.10).

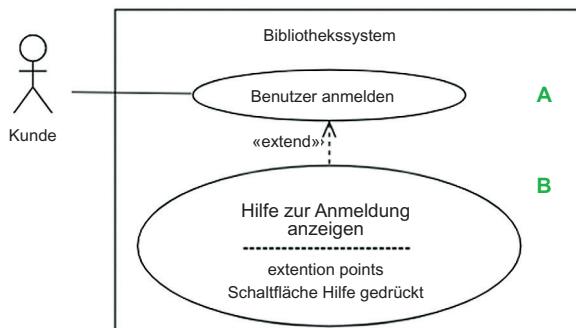


Abbildung 4.10 Extend-Beziehung mit Erweiterungspunkt und Bedingung

Häufige Stichwörter für eine Extend-Beziehung in Beschreibung sind unter anderen folgende: **optional**, **Option**, **auswählbar**, **erweitert**, **Erweiterung**, **Ausnahme**, usw.

Spezialisierter Anwendungsfall

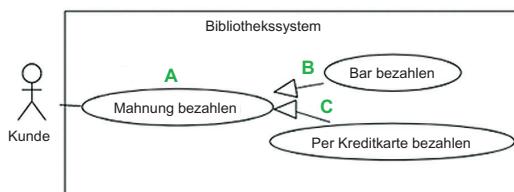


Abbildung 4.11 Spezialisierter Anwendungsfall

Anwendungsfall A wird durch die Anwendungsfälle B und C **spezialisiert** (siehe Abbildung 4.11). Anwendungsfall B und C sind daher **spezialisierte Anwendungsfälle**.

4.3.6.2 Diskussion von Use-Case-Diagrammen

Der Einsatz und Nutzen von Use-Case-Diagrammen wird auch kontrovers diskutiert. Die am häufigsten genannten Punkte sind hier zusammengetragen.

Vorteile:

- Möglichkeit zur Abbildung komplexer Systeme
- leicht erlernbar
- funktionale Anforderungen aus Sicht des Anwenders gut darstellbar
- Übersichtlichkeit durch Top-Down-Vorgehen
- durch Beschreibungen für natürliche Sprache geeignet
- gut geeignet für objektorientierte Software-Entwicklung.

Nachteile

- zeitliche Reihenfolge der Ereignisse nicht festgelegt
- schwer lesbar für Nicht-Informatiker
- nicht-funktionale Anforderungen sind grafisch nicht darstellbar.

4.3.6.3 Anwendungsfälle im Zusammenhang mit dem Software-Lebenszyklus

Wie stehen nun die Use-Cases, über den Software-Lebenszyklus gesehen, im Zusammenhang mit anderen Diagrammarten der UML?

Abbildung 4.12 bietet eine grafische Übersicht, in der gezeigt wird, dass Anwendungsfälle hauptsächlich in der Analysephase (d. h. Definitionsphase) eingesetzt werden. Zusätzlich können in der Analysephase auch noch dynamische Verhaltensdiagramme genutzt werden, um Prozessabläufe und Programmabläufe genauer zu beschreiben.

In der Software-Designphase entsteht schließlich mithilfe der Anwendungsfälle eine zugehörige Softwarearchitektur. Sie besteht üblicherweise bei objektorientierten Projekten aus einem statischen Klassendiagramm, das besonders gut die Struktur der zu erstellenden Software beschreibt. Die Software-Architektur wird ergänzt um die dynamische Beschreibung der Abläufe im System. Dies geschieht mithilfe von Verhaltensdiagrammen wie typischerweise den Aktivitätsdiagrammen, Sequenzdiagrammen und Zustandsdiagrammen.

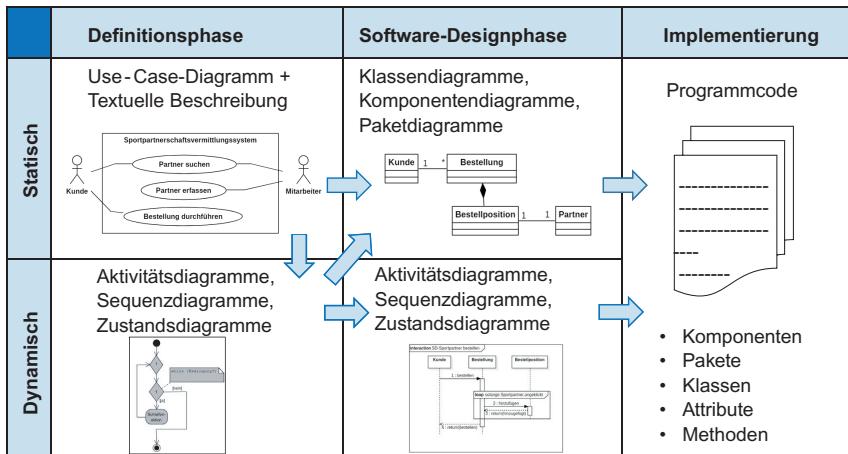


Abbildung 4.12 Zusammenhang der UML-Diagramme im Software-Lebenszyklus nach [Sch03]

Steht die Softwarearchitektur, kann als nächstes mit der Implementation begonnen werden.

Die Inhalte der Use-Cases bestimmen daher, welche Inhalte im Klassendiagramm zu modellieren sind. Ein übersichtliches Beispiel dafür ist in Abbildung 4.13 zu sehen.

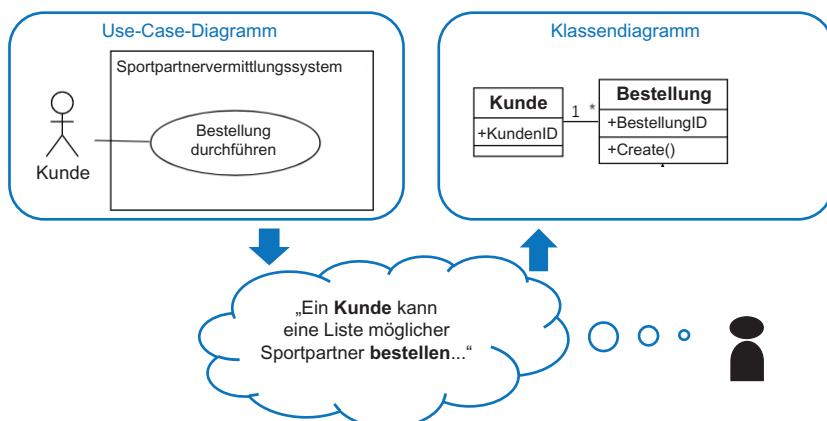


Abbildung 4.13 Zusammenhang eines Use-Cases mit dem Klassendiagramm nach [Sch03]

Darin wird beschrieben, wie aus der Anforderung und dem Use-Case, aus deren Substantiven „*Kunde*“ und „*Bestellung*“, jeweils eine zugehörige Klasse modelliert wird. Zur eindeutigen Identifikation erhalten beide Klassen jeweils ein Attribut, die im Beispiel „*KundenID*“ und „*BestellungID*“ genannt werden. Damit eine Bestellung aufgenommen werden kann, erhält die Klasse „*Bestellung*“ noch eine Methode „*Create()*“ dazu. (In der Abbildung fehlen hier natürlich noch die Angaben von Datentypen, Initialwerten und Parametern.)

Da in den überwiegenden Fällen eine Anforderungsanalyse in Unternehmen und Organisationen angewendet wird, sollte der Zusammenhang zwischen der eher Wirtschafts-Informatik-orientierten Business-Analyse mit den zugehörigen Geschäftsprozessmodellen und der eher Informatik-orientierten Systemanalyse verstanden werden. In Abbildung 4.14 ist dieser Zusammenhang übersichtlich dargestellt. Eine Business-Analyse geht der Systemanalyse üblicherweise voraus. Mit der Business-Analyse werden die Unternehmensprozesse analysiert und dokumentiert. Mit der Systemanalyse wird nun ein Vorschlag begonnen, wie man die Unternehmensprozesse durch IT verbessern bzw. unterstützen kann. Mit der Systemanalyse ist hier auch die Definitionsphase des Software-Lebenszykluses gemeint. Anschließend laufen die weiteren Lebenszyklusphasen wie schon beschrieben ab.

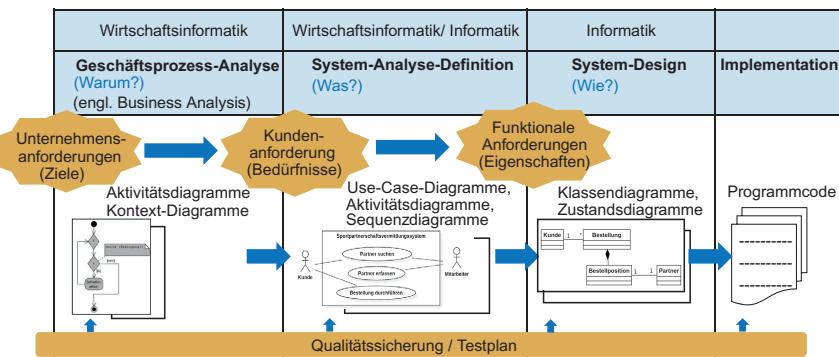


Abbildung 4.14 Zusammenhang von Business-Analyse und Systemanalyse nach [Sch03]

4.3.7 Validation von Anforderungen

Wird Requirements-Engineering ernsthaft betrieben, werden Anforderungen auch getestet und geprüft wie jedes andere Software-Artefakt auch. Denn es ist statistisch nachweisbar, dass eine Fehlererkennung zu einem möglichst frühen Zeitpunkt kostengünstiger ist als die Erkennung zum späteren Zeitpunkt.

4.3.7.1 Ziele guter Anforderungen

Die Ziele guter Anforderungen beziehen sich typischerweise auf deren Syntax, deren Semantik und deren Wiederverwendbarkeit. Hier werden nun die Qualitätskriterien dieser drei Bereiche aufgelistet. Die Interpretation, was die Qualitätskriterien im konkreten Projektfall für die Anforderungsanalyse bedeuten, obliegt jedoch dem Projektteam und sollte konkret formuliert und dokumentiert werden.

- Syntaktisch:
 - Verfolgbarkeit, Redundanzfreiheit, gute Struktur, angemessener Umfang, Eindeutigkeit, Notwendigkeit, rechtliche Verbindlichkeit
- Semantisch (inhaltlich fachliche Aspekte):
 - Fachliche Richtigkeit, Korrektheit, Gültigkeit, Vollständigkeit
- Weiterverwendung:
 - Realisierbarkeit, Testbarkeit, Bewertbarkeit, Verständlichkeit.

4.3.7.2 Prüfung von Anforderungen

Eine effiziente Prüfung von Anforderungen erspart Kosten für Fehlerbeseitigung und Änderungsmaßnahmen.

Ziele

Bei der Prüfung von Anforderung ist das Hauptziel, die im Team festgelegten Qualitätskriterien für die Anforderung zu erfüllen. Fehler und Schwächen von Anforderungen sollen aufgedeckt werden. Fehlende Informationen (d. h. Lücken) sollen entdeckt, Widersprüche in Anforderungen aufgehoben, Redundanzen sollen erkannt und fehlerhaftes Systemverhalten soll identifiziert werden.

Vorgehen

Der Zeitpunkt für die Prüfung ist optimal nach der Definition von Anforderungen, jedoch liegt er natürlich mindestens vor der Implementation von Software. Es müssen nacheinander Prüfer identifiziert, die Prüftechnik festgelegt und schließlich die Prüfung durchgeführt werden.

Potentielle Prüfer

In Tabelle 4.5 sind Beispiele einer sinnvollen Einteilung zu finden, welche potentiellen Prüfer sich für unterschiedliche Prüfungsaspekte eignen.

Tabelle 4.5 Beispiele potentieller Prüfer für Anforderungen nach [RSG14]

Prüfungsaspekt	Geeigneter Prüfer
Fachliche Richtigkeit	<ul style="list-style-type: none"> ▪ Anwender ▪ Software-Engineering-Spezialist ▪ Projektleiter
Vollständigkeit	<ul style="list-style-type: none"> ▪ Software-Engineering-Spezialist ▪ projekexterne Qualitätssicherung ▪ Anwender
Realisierbarkeit	<ul style="list-style-type: none"> ▪ Software-Entwickler

Prüftechniken

Zur Prüfung von Anforderungen sei hier eine Liste möglicher Prüftechniken vorgeschlagen, aus der sich Projektteams passende Verfahren aussuchen können.

- Stellungnahmen von Stakeholdern
- Prototyp/Simulationsmodell: Verhalten des Prototypen testen
- Analysemodell, z. B. Mengengerüste für Daten aufstellen
- Reviews, Inspektion, Walkthrough: gemeinsames Verständnis aller Beteiligten suchen (siehe Kapitel 6.2.1 statische Testverfahren)
Manchmal: Kickoff-, Follow-up- und Exit-Meetings
- sprachliche Schablonen prüfen
- Abnahmekriterien aufstellen und prüfen: Entscheidungstabellen aufstellen.

4.3.8 Anforderungsmanagement

Anforderungsmanagement im Sinne des Managements wird typischerweise von einem Projektleiter o. ä. durchgeführt, der die Leitung und Übersicht darüber behält, welche Anforderungen aufgenommen, bearbeitet und geprüft werden sollen.

Jedoch fällt auch die Aufgabe der Verwaltung von Anforderungen in diesen Bereich. Diese Aufgabe wird meist von den Software-Engineering-Spezialisten durchgeführt. Um eine bessere Übersicht zu behalten, werden Anforderungen daher oft nach folgenden Merkmalen gruppiert:

- nach Anforderungsart (siehe Kapitel 4.3.2)
- nach Merkmalen (Beispiele: Zuverlässigkeit, Benutzbarkeit, Effizienz, etc.)

- nach Detailebenen des Systems
- nach Priorität
- nach Kritikalität
- nach rechtlicher Verbindlichkeit: Muss-, Kann-, Soll- Verbindlichkeit.

Requirements-Engineering-Werkzeuge

Die Verwaltung von Anforderungen in großer Anzahl ist nicht einfach und wird daher üblicherweise werkzeug-basiert durchgeführt. Die meisten UML-Werkzeuge haben mittlerweile auch einen Bereich, in dem die Anforderungsdokumentationen gleich mit gespeichert werden können. Es gibt jedoch auch einige Werkzeuge, mit denen pures Anforderungsmanagement betrieben werden kann. Beispiele hierfür sind IBM Rational DOORS-Next Generation [[IBM19a](#)] oder Caliber von Micro Focus (ehemals Borland [[Foc19](#)]).

■ 4.4 Zusammenfassung

Die Definitionsphase dient der Erhebung von Anforderungen an ein Softwaresystem. In diesem Kapitel wird zunächst ein Überblick über die Inhalte dieser Phase geschaffen (siehe Kapitel 4.1). Beispielsweise entsteht ein sogenanntes **Pflichtenheft** (engl.: Specification). Es wird auch im Deutschen häufig als **Spezifikation** bezeichnet. Inhalt des Pflichtenhefts sind alle Arten von Anforderungen an das zu entwickelnde Softwaresystem (siehe Kapitel 4.2). Zusätzlich werden Motivation und Ziele der erwarteten Software ergänzt. Das Inhaltsverzeichnis kann dabei dem eines Lastenheftes entsprechen. Unterschiede zwischen dem Lastenheft und dem Pflichtenheft sind jedoch in der Sichtweise beider Dokumente zu finden. Während das Lastenheft aus Sicht des Kunden als eine Art Wunschübersicht geschrieben wird, findet sich im Pflichtenheft die Sichtweise des Software-Herstellers, welcher darin eine **Leistungsbeschreibung** dokumentiert. Verfasser des Pflichtenhefts ist hauptsächlich der **fachliche Software-Engineering-Spezialist**.

Moderne Pflichtenhefte, die der Vorgehensweise eines agilen Modells entsprechen, sind als sogenannte User-Stories und Epics verfasst. Den Begriff „Pflichtenheft“ gibt es jedoch in der agilen Vorgehensweise nicht mehr. Verfasser (bzw. Verantwortlicher) der User-Stories und Epics ist hauptsächlich der **Product-Owner**.

Der englische Begriff für Anforderungsanalyse ist „**Requirements-Engineering**“, der auch im deutschen Sprachgebrauch verwendet wird. In Kapitel 4.3 werden alle Begriffe und Arbeitsschritte des Requirements-Engineering erklärt. Begonnen wird mit der Klärung des Begriffs „**Anforderung**“ (siehe Kapitel 4.3.1). Es handelt sich dabei um sprachlich möglichst unkomplizierte und einfach formulierte Sätze, welche Funktionalitäten in einem Softwaresystem enthalten sein sollen. Um eine gute Übersicht und Handhabbarkeit zu erzielen, werden Anforderungen in die Anforderungsarten (siehe Kapitel 4.3.2) funktional, nicht-funktional, Benutzeranforderungen, Problembereichsanforderungen und Systemanforderungen eingeteilt. Zu jeder Art werden Beispiele gegeben. Manchmal findet man hier auch nur eine Einteilung in Muss-, Kann-, und Soll-Anforderungen.

Anschließend werden in Kapitel 4.3.3 Qualitätsmerkmale guter Anforderungen diskutiert. Die Beschreibung der Anforderungsinhalte kann in natürlicher Sprache, in formalisierter Sprache oder auch in grafischer Notation erfolgen (siehe Kapitel 4.3.4). Zu empfehlen ist hier die Beschreibung in natürlicher Sprache unter zu Hilfenahme von Sprachschablonen. Die Beschreibungen sollten durch grafische UML-Use-Case-Diagramme zur Übersicht ergänzt werden, da diese oftmals in der Praxis die Diskussion mit dem Kunden erleichtern. Eine der interessantesten Aufgaben im Requirements-Engineering ist die Erhebung von Anforderungen. In Kapitel 4.3.5 werden daher einige Techniken dafür vorgestellt. Es handelt sich dabei beispielsweise um Kreativitätstechniken. Die sinnvolle grafische Ergänzung kann durch die Anwendung von Notationen für Anforderungsmodellierung geschaffen werden. Kapitel 4.3.6 beschreibt zwei Notationen. Die etwas ältere strukturierte Analyse für funktionale Programmierung und UML-Use-Case-Diagramme für objektorientierte Programmierung. Bei Letzterem sind wichtige Notationselemente die Akteure, die Use-Cases und die Assoziationen dazwischen. In Kapitel 4.3.7 wird schließlich die Prüfung der Anforderungen erklärt. Prüftechniken können hier beispielsweise Expertenmeinungen oder Reviews sein. Zuletzt folgen in Kapitel 4.3.8 noch Gedanken zum erfolgreichen Management und der Verwaltung von Anforderungen.

■ 4.5 Aufgabensammlung

Dies ist der Aufgabenbereich für Fragen zur Definitionsphase.

1. Was ist Requirements-Engineering?
2. Welche Eigenschaften haben gute Anforderungen?
3. Was ist eine Systemanforderung?
 - A) Anforderungen, die als genauere Beschreibung der Benutzeranforderungen dienen.
 - B) Anforderungen, die das externe Verhalten des Systems betreffen (Ein-, Ausgaben).
 - C) Anforderungen, die sich aus dem Anwendungsbereich ergeben.
 - D) Anforderungen, die speziellen Bedürfnissen von Systembenutzern genügen.
4. Wie findet man Anforderungen?
5. Welche Beziehungsart in einem Use-Case-Diagramm würden Sie wählen, wenn der Use-Case „Rechnung drucken“ um eine Fehlermeldung ergänzt werden soll?
 - A) Include-Beziehung, weil ein großer Use-Case in kleinere Use-Cases zerlegt werden soll
 - B) Include-Beziehung, weil ein Use-Case einen anderen Use-Case aufrufen soll
 - C) Extend-Beziehung ohne Erweiterungspunkt
 - D) Extend-Beziehung mit Erweiterungspunkt
6. Wie testet man Anforderungen?

5

Software-Design-Phase

Die Software-Design-Phase wird häufig auch Entwurfsphase genannt. Sie sollte jedoch nicht mit der Implementationsphase verwechselt werden. Es geht hier nicht um die Programmierung der Software, sondern um die Modellierung einer so genannten Softwarearchitektur. Im Zentrum steht hier die Frage, „**WIE**“ die Software fachlich und technisch erstellt werden soll.

Eine Übersicht über die Software-Design-Phase wird in Kapitel 5.1 gegeben. Zur Veranschaulichung von Softwarearchitekturen wird mit grafischen Notationen, wie beispielsweise mit den **UML-Diagrammen**, gearbeitet (siehe Kapitel 5.3). Beschrieben werden **Strukturdigramme**, wie die Klassendiagramme und Komponentendiagramme, sowie **Verhaltensdiagramme**, wie Struktogramme, Aktivitätsdiagramme, Sequenzdiagramme und Zustandsdiagramme. Einige der **typischen Softwarearchitekturen** werden anschließend in Kapitel 5.4 vorgestellt und diskutiert. Strategien und Methoden, wie man zu eigenen Softwarearchitekturen kommt, behandelt Kapitel 5.5. Zuletzt folgen in Kapitel 5.6 einige Ansätze für die Wiederverwendung von Software. Beispielsweise leistet die Verwendung vorher erprobter Software-Artefakte meist einen Beitrag zur Qualitätssicherung. Initial sind Wiederverwendungsmaßnahmen allerdings eher mit Mehraufwand verbunden. Einmal etabliert führen Sie jedoch in der Praxis häufig schneller und kosten-effizienter zum Ziel.

■ 5.1 Überblick

Die zwei Sichtweisen auf Software-Design

Zwei Aspekte sind zu unterscheiden, wenn von Software-Design gesprochen wird. Diese Aspekte werden auch **Prozesssicht** und **Ergebnissicht** genannt.



Definition

Software-Design ist ...

- „... der Prozess, die Softwarearchitektur, -Komponenten, -Schnittstellen und andere Merkmale eines Systems oder einer Komponente zu definieren“
- „... das Ergebnis dieses Prozesses“.

[IEEE Std 610.12-1990 (R2002), Standard Glossary of Software Engineering Terminology, 1990]

Was ist dementsprechend nun der Unterschied zwischen Prozesssicht und Ergebnissicht?

1. Die Prozesssicht

Software-Design ist die **Lebenszyklus-Aktivität** im Software-Engineering, in der die Anforderungen analysiert werden, um die interne Struktur der Software zu produzieren.

2. Die Ergebnissicht

Software-Design ist die **Beschreibung** der Softwarearchitektur, d. h. wie Software zerlegt wird, in Komponenten organisiert wird und wie die Schnittstellen zwischen den Komponenten sind.

Aufgaben

Die Software-Design-Phase ist eine sehr wichtige Phase bei der Herstellung qualitativ hochwertiger Software und unverzichtbar, wenn Software komplex wird. Es wird dabei die Definition der Vorgehensweise vorgenommen (gemeint ist die Erstellung eines Architektenplans für Software). Unverzichtbar ist dabei eine **Modellbildung** für die Software. Dabei produzieren die Software-Engineering-Spezialisten eine Vielzahl an unterschiedlichen Modellen. **Die Modelle** bilden einen **Plan**, wie die **Lösung implementiert** werden kann. Die Modelle werden analysiert und evaluiert, ob sie die Anforderungen (erstellt in der Definitionsphase und dokumentiert in der Spezifikation) erfüllen. Alternative Lösungen und deren Zielkonflikte werden diskutiert. Die resultierende Modellierung dient als **Startpunkt für die Implementierung** und als **Vorlage für die Testphase**.

Aktivitäten

Zuerst wird die sogenannte Softwarearchitektur erstellt. Als Fachbegriff wird diese auch **Makroarchitektur** genannt. Anschließend wird das Detailkonzept, die sogenannte **Mikroarchitektur** festgehalten.

- **Top-Level-Design/Makroarchitektur:**

Eine Beschreibung der Software auf hohem Abstraktionsniveau und deren Organisation. Es findet die Identifikation der Software-Komponenten statt.

- **Detailkonzept/Mikroarchitektur:**

Es handelt sich um die detaillierte Beschreibung jeder Einzelkomponente. Damit wird die Implementierung ermöglicht.

Ergebnis

Das Ergebnis dieser Phase ist das **Pflichtenheft** (engl.: Specification) und ein eventuell erweitertes **Entwurfsdokument** (Ein Entwurfsdokument kann ein Pflichtenheft um genauere Implementierungsdetails erweitern.). Das Pflichtenheft wird dabei in der Definitionsphase (durch Füllen mit Anforderungen) begonnen und in der Designphase (durch Ergänzung der Softwarearchitektur) vervollständigt. Der Inhalt des Pflichtenheftes ist eine Vielzahl von Modellen für jede Komponente der Software. Fragen, die in diesem Dokument beantwortet werden, sind: **WIE** soll die Software entwickelt werden?! Beispielsweise werden die Komponenten festgelegt und auch die Kommunikation zwischen den Komponenten beschrieben. Außerdem ist das Pflichtenheft im klassischen Vorgehen die Vertragsgrundlage mit dem Kunden.

Bei der Verwendung eines **agilen** Vorgehensmodells wird der Begriff „Pflichtenheft“ nicht mehr verwendet. Hier werden, beispielsweise bei SCRUM, Anforderungen als „**User-Stories**“ definiert und im sogenannten „**Product-Backlog**“ gesammelt. Die User-Stories werden priorisiert und rechtzeitig den Teilnehmern des SCRUM-Teams zugeordnet (vgl. [SS18]).

Beteiligte

Es arbeiten Software-Engineering-Spezialisten, Systemanalytiker und Software-Entwickler zusammen, um die Modelle möglichst realitätsnah zur Lösung der Aufgaben zu gestalten.

Abschließender Hinweis:

Allgemeine, jedoch sehr ausführlichere Informationen zum Thema Software-Design können beispielsweise auch in den umfangreichen Lehrbüchern von Sommerville [Som18] und Balzert [Bal11] gefunden werden.

■ 5.2 Ein durchgängiges Beispiel

Als ein kleines Beispiel für den Lebenszyklusprozess von der Analyse bis zur fertigen Softwarearchitektur, sei hier die Entwicklung eines webbasierten Shops skizziert

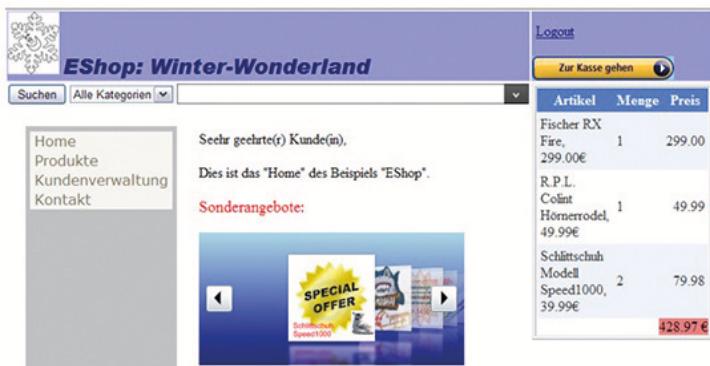


Abbildung 5.1 Beispiel: Webbasiertes Shop für Wintersportwaren mit Namen „E-Shop“

(Name: E-Shop). Mit dem Shop sollen Wintersportwaren verkauft werden. Es soll ein Menü, eine Suche, eine Anzeige von aktuellen Angeboten, einen Warenkorb und eine Bezahlfunktion geben. Das Endergebnis könnte aussehen wie in Abbildung 5.1.

Da es sich um ein webbasiertes Projekt handelt, ist die zu Grunde liegende Makroarchitektur bereits als typische „Client-Server“-Architektur vorgegeben (siehe Abbildung 5.2). Dies bedeutet grob beschrieben, dass ein so genannter einfacher „Client“ komplexe Dienste von einem potenzen „Server“ aus einem angeschlossenen Netzwerk benutzt. Eine übliche Three-tier-Architektur (d. h. dreigeteilt) gibt hier Sinn, da eine Datenbank mit den Wintersportprodukten hinterlegt wird. Ein Vorteil der Web-Anwendungen ist, dass auf Client-Seite nur eine simple Browser-Software installiert sein muss (Beispiel: Mozilla Firefox [Coo19]). Marktübliche Browser verstehen außerdem so genannte Clientskript-Sprachen. Üblicherweise handelt es sich dabei heutzutage um JavaScript.

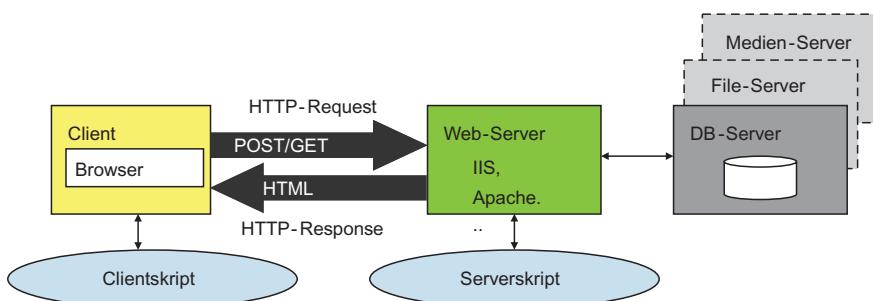


Abbildung 5.2 Die gewählte „Client-Server“-Architektur des Anwendungsbeispiels E-Shop

tage um die Skriptsprache „JavaScript“. Dadurch können statische Webseiten, die auf HTML (Abk. engl. Hyper Text Markup Language) [W3C19] basieren, dynamisiert werden. Komplexere Aufgaben, wie beispielsweise der weltweite Zugriff auf eine Datenbank, werden als Dienstanfrage an den Server versendet. Das derzeit relevante Protokoll zur Übertragung der Anfrage nennt sich HTTP (Abk. engl. Hyper Text Transport Protocol) [W3C19], mit dessen Befehlen „get“ und „post“ die Anfragen ausgelöst werden können. Ein Beispiel für einen Webserver ist der Apache-Webserver [Apa19a]. Auf Webservern laufen wiederum Serverskriptsprachen wie beispielsweise ASP.net Core [Mic19a], PHP [TPG19], RubyOnRails [RCT19] oder Python [Pyt19]. Auf dem dritten Tier läuft typischerweise in unserem Beispiel ein Datenbankserver mit Datenbankssoftware (z. B. Oracle [Ora19], SQL Server [Mic19b] oder MySQL [Mic19c]). Aus technischer Sicht wird man sich im Projekt also eine geeignete webbasierte Systemumgebung heraussuchen und einsetzen.

Nun werden die konkreten Anforderungen des Shops in der **Definitionsphase**, wie in Tabelle 5.1 zu sehen, festgelegt und mit der Sprachschablone (siehe Abbildung 4.3 aus Kapitel 4 – Definitionsphase) zu eindeutigen, testbaren Sätzen ausformuliert.

Tabelle 5.1 Requirements-Engineering in der Definitionsphase am Beispiel des E-Shops

Akteur	Anforderung
Kunde	<ul style="list-style-type: none"> ■ 24 Stunden täglich muss das System dem Kunden die Möglichkeit bieten, Produkte (die zu Kategorien gehören) anzusehen und zu suchen. ■ Das System muss dem Kunden die Möglichkeit bieten, einen Warenkorb zu füllen und anzusehen. ■ Das System muss dem Kunden die Möglichkeit bieten, zur Kassenansicht zu gehen und zu bezahlen. ■ Das System soll dem Kunden die Möglichkeiten bieten, ein Kontaktformular auszufüllen und abzusenden. ■ ...
Mitarbeiter	<ul style="list-style-type: none"> ■ Das System muss dem Mitarbeiter die Möglichkeit bieten, die Kunden zu verwalten. ■ ...

In der **Designphase** kann nun auf diesen Grundlagen die Mikroarchitektur des Shops erstellt werden (siehe Abbildung 5.3). Aufgrund der Anforderungen entstehen nun, je nach Bedarf, UML-Diagramme, wie beispielsweise Klassenmodelle und Aktivitätsdiagramme.

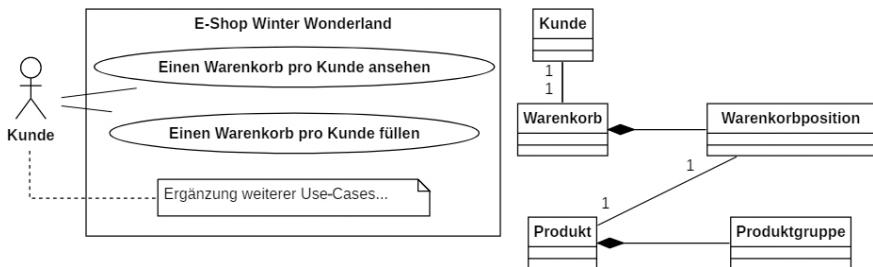


Abbildung 5.3 Anwendungsfälle des E-Shops mit Klassendiagramm als frühes Begriffsmodell

Erstellung eines Präsentationsmodells

Bei webbasierten Systemen wird häufig auch das Layout (auf Deutsch: Aussehen) der Webseiten modelliert, um Benutzer zu einem Kauf im fertigen Shop zu motivieren (siehe Abbildung 5.4). (Fachliche Stichworte, unter denen weitere Informationen hierzu gefunden werden können, sind **Software-Ergonomie**, **Usability-Engineering** und **User-Experience**). Dafür finden sogenannte „Wireframe“-, „Mock-up“-, oder „Layout“-Werkzeuge, wie beispielsweise „Balsamiq“ [BS19] Verwendung.

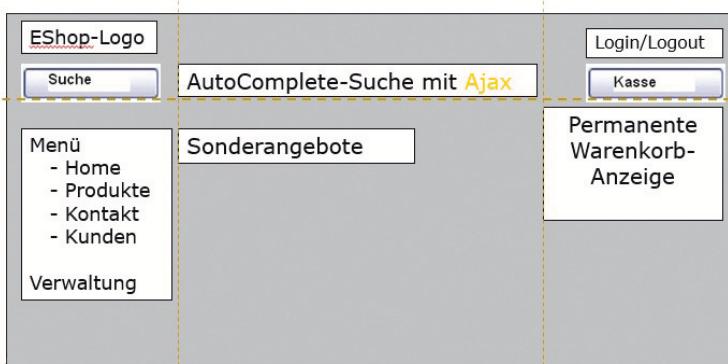


Abbildung 5.4 Präsentationsmodell des Anwendungsbeispiels E-Shop

Erstellung eines Navigationsmodells

Eine Eigenart der webbasierten Software ist die Aufteilung der Anwendung in Webseiten. Dabei werden aktuell die Darstellung der Anwendung (im Browser), dessen Layout und die Steuerung der Anwendung getrennt (siehe auch „Model-View-Controller“-Pattern in Kapitel 5.4.3 – Entwurfsmuster). Die Darstellung erfolgt auf

Webseiten im HTML-Format (siehe [W3C19]). Zumeist werden diese statischen Seiten durch die Verwendung von JavaScript-Funktionen und Bibliotheken dynamisiert. Ein ansprechendes Layout erreicht man durch den Einsatz von CSS (engl.: Cascading Stylesheets siehe [W3C19]). Reicht die Dynamisierung der Seiten durch Einsatz von JavaScript nicht aus oder will man aus Gründen der Skalierung die Anwendung optimieren, werden Funktionalitäten durch Serverskript ergänzt.

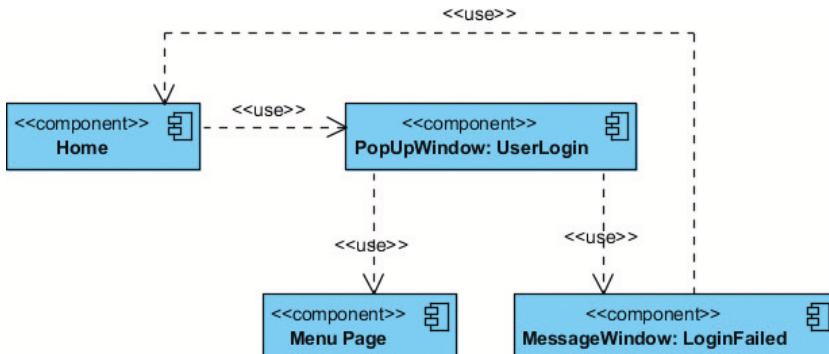


Abbildung 5.5 Beispiel eines Navigationsdiagramms für das Anmelden bei der Webanwendung „E-Shop“, dargestellt als UML-Komponentendiagramm

Diese Art der Schichtenbildung und Modularisierung hat zur Folge, dass Webprojekte eine sogenannte **Navigationsstruktur** abbilden. Diese wird durch **Navigationsmodelle** (siehe [DLWZ03]) dargestellt. Als Hilfsmittel aus der UML sind dafür unter anderem Komponentendiagramme einsetzbar. Ein Beispiel ist in Abbildung 5.5 zu sehen. Sogar bei modernen Single-Page-Webanwendungen (siehe beispielsweise Angular-Anwendungen [FM19]), die aus einer einzigen Webseite gebildet werden, ist durch den Einsatz einer Vielzahl von Bibliotheken und untergeordneter Webseitenbereiche, ein UML-Komponentendiagramm hilfreich und bietet eine Übersicht (siehe Kapitel 5.3.1.2 – Komponentendiagramm).

Auf Grundlage all dieser Modelle könnte nun schließlich mit der Programmierung der Webanwendung begonnen werden. An diesem praxisnahen Beispiel wurde demonstriert, wie die Systemarchitektur einer Software schrittweise entsteht. Dabei hängt die entstehende Systemarchitektur häufig auch von der verwendeten Technologie und der Systemart ab. Manchmal werden dabei auch Modelle benötigt und gebaut, die nicht mit der UML-Notation umgesetzt werden (können). Dies ist durchaus manchmal notwendig und daher gängige Praxis. Sogar Handzeichnungen sind oft eine hilfreiche Unterstützung zur Gewinnung der Übersicht.

5.3 Notationen

Notationen dienen dazu, **Software-Design-Artefakte** zu repräsentieren. Sie helfen dabei, beim Software-Design, eine einheitliche Art von „Sprache“ zu bilden und können daher von allen Fachleuten interpretiert werden. Häufig wird Software-Design durch grafische Darstellungen repräsentiert, jedoch sind auch andere Darstellungsformen manchmal sinnvoll.

Einsatz

Manche Notationen eignen sich mehr zur Darstellung der **Makroarchitektur**, andere eher zur Darstellung der **Mikroarchitektur** (siehe Kapitel 5.4 – Softwarearchitekturen). Manche Notationen können in der Software-**Definitionsphase** (siehe Kapitel 4 – Definitionsphase) und/oder in der **Designphase** eingesetzt werden.

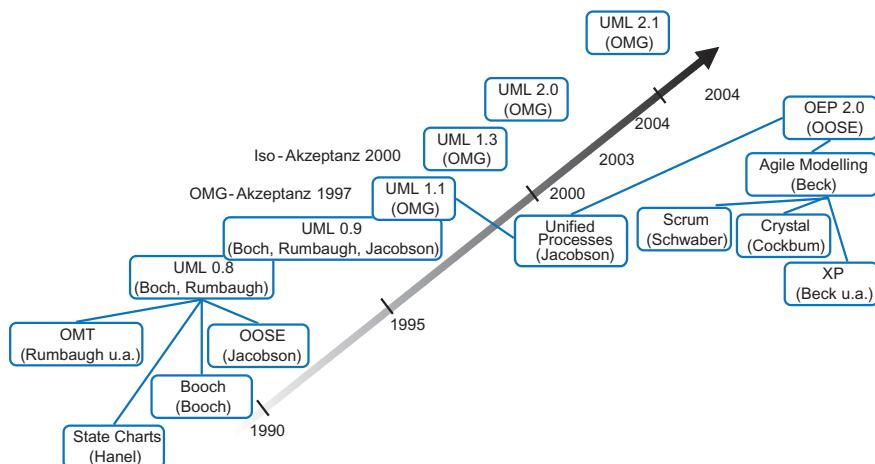


Abbildung 5.6 Historie der UML nach [ÖS14]

In Abbildung 5.6 ist die historische Entwicklung der **UML** zu sehen. Zunächst wurde jedoch lange nach geeigneten Methoden zur Darstellung von Softwarearchitekturen gesucht und viele Alternativvorschläge entwickelt. Mittlerweile ist die UML jedoch in der Branche als Standard etabliert. UML ist die englische Abkürzung für „**Unified Modeling Language**“. Es handelt sich dabei um die aktuell eingesetzte Notation für objektorientierte Software-Modellierung und -Entwicklung.

Überblick UML (engl.: Unified Modeling Language) – (Version 2.5.1, von 12/2017)

In diesem Buch wird eine kompakte, kleine Übersicht der wichtigsten Notationselemente der UML gegeben. Diese Übersicht ist daher wohlüberlegt nicht vollständig. Ein Leser, der eine vollständige Beschreibung studieren möchte, kann die Beschreibung des Standards bei der „Object Management Group“ [OMG19a] finden, die Beschreibung von OOSE verwenden [Oll18] oder in einschlägigen UML-Büchern lesen, welche die Notation detailliert beschreiben. Empfehlen kann man beispielsweise besonders die „UML Kurzreferenz“ ([ÖS14]) und das umfangreichere Buch „UML glasklar“ ([RQSG12]). Von den Autoren der UML, Rumbaugh, Jacobson und Booch, gibt es auch ein aktualisiertes Buch zur Beschreibung der UML, nämlich „The Unified Modeling Language, Reference Manual“ ([RJB10]), welches lesenswert ist.

Diagrammarten

Die Beschreibung der UML wurde grundsätzlich mithilfe von Baumstrukturen durch Graphentheorie entwickelt.

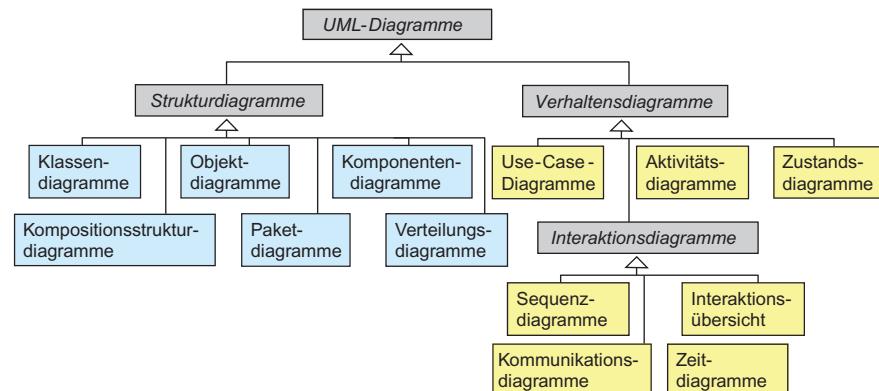


Abbildung 5.7 UML-Diagrammarten

In der UML werden hauptsächlich **Strukturdiagramme** und **Verhaltensdiagramme** unterschieden (siehe Abbildung 5.7). Mit Strukturdiagrammen können statische Strukturen der Software beschrieben werden (siehe Kapitel 5.3.1 – Strukturdiagramme). Mit Strukturen einer Software sind typischerweise die darin enthaltenen Komponenten, Module und Klassen gemeint.

Mit Verhaltensdiagrammen kann dagegen das Verhalten einer Software beschrieben werden (siehe Kapitel 5.3.2 – Verhaltensdiagramme). Prozessabläufe und schrittweise nacheinander ausgeführte Aktivitäten, sind dafür typische Beispiele.

Die wichtigsten Diagrammarten werden im Folgenden beschrieben und anhand von Beispielen erklärt.

Hinweis:

In diesem Schriftwerk wird davon ausgegangen, dass der Leser die objektorientierte Programmierung und deren Begriffe versteht und anwenden kann. Daher werden die dafür notwendigen Begriffe nicht weiter erklärt und damit vorausgesetzt. Wer sich mit objektorientierten Programmiersprachen detaillierter auseinandersetzen möchte, wird beispielsweise im „Taschenbuch Programmiersprachen“ von Henning und Vogelsang [HV07] fündig.

5.3.1 Strukturdiagramme

Mit Strukturdiagrammen werden die statischen Strukturen einer Software beschrieben. Gemeint ist hier die Organisation einer Software in Subsysteme, Module und Komponenten. In objektorientierten Programmen wird meist die Organisation in Klassen modelliert. Je nach Systemart kann es aber auch Sinn machen, noch andere statische Modelle zu erstellen. Beispielsweise wird in web-basierten Programmen neben dem Klassenmodell meist auch die Navigationsstruktur der Webseiten modelliert (siehe UML-Komponentendiagramm aus Abbildung 5.5).

5.3.1.1 UML-Klassendiagramm

Ein Klassendiagramm ist eine Übersicht über die Klassen eines objektorientiert geschriebenen Programms. Es wird verwendet, um eine Menge von Klassen (und Objekte) und deren Beziehungen zueinander darzustellen.

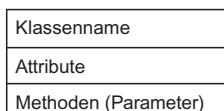


Abbildung 5.8 Grundsätzlicher Aufbau einer Klasse in der grafischen Darstellung der UML

Eine Klasse wird dargestellt als Rechteck und unterteilt in die drei Bereiche **Klassenname**, **Attribute** und **Methoden** (siehe Abbildung 5.8).

In Abbildung 5.9 ist das Beispiel der Klasse „Acrobat“ zu sehen. Klassen und Methoden können Stereotypen wie „*abstract*“ zugeordnet werden. Jedes Attribut erhält

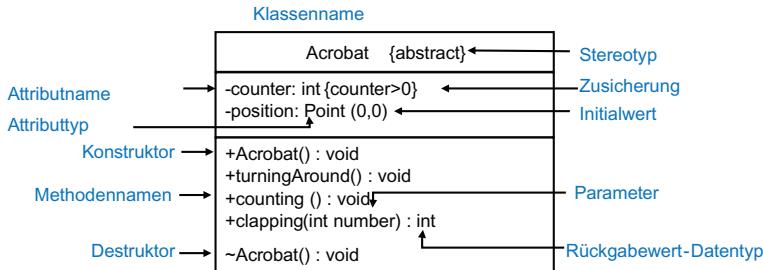


Abbildung 5.9 Beispiel der Klasse „Acrobat“

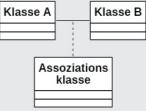
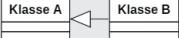
einen Attributtyp. Außerdem kann ein Attribut einen Initialwert und Zusicherungen erhalten. Methoden können Rückgabewerte und Parameter erhalten. Je nach objektorientierter Programmiersprache können Klassen zum Aufbau Konstruktoren und zum Zerstören Destruktoren besitzen.

Sogenannte **Sichtbarkeiten** regeln die Zugriffsrechte auf die Klasse, deren Attribute und ihrer Methoden. Üblicherweise gibt es minimal die Ausprägungen „`+ :public`“, „`- :private`“ und oft auch „`# :protected`“.

Eine grundlegende Eigenschaft von objektorientierter Software ist, dass die Klassen miteinander in **Beziehung** stehen. Dies ermöglicht die **Kommunikation** zwischen den Objekten und regelt die **Zugriffsmöglichkeiten** aufeinander. Die wichtigsten Beziehungselemente zwischen Klassen sind in Tabelle 5.2 zu sehen.

Tabelle 5.2 Wichtige Beziehungselemente des UML-Klassendiagramms

Notation	Element	Beschreibung
	Assoziation	Eine Assoziation beschreibt eine Beziehung zwischen zwei oder mehr sogenannten Klassifizierern, im häufigsten Fall eine Verbindung zwischen genau zwei Klassen. Dies beinhaltet, dass die beiden Klassen miteinander kommunizieren können.
	gerichtete Assoziation	Die Navigationsrichtung einer Beziehung kann mit Pfeil an der Assoziation angegeben werden. Der Pfeil drückt die Zugriffsrichtung der Objekte aus.

Notation	Element	Beschreibung
	qualifizierte Assoziation	Eine qualifizierte Assoziation unterteilt die Menge der Objekte auf einer Seite der Assoziation in Partitionen. Qualifizierer haben (wie Attribute) einen Typ, der auch eine Klasse sein kann.
	Attributierte Assoziation	Ist eine Klasse von dem Vorhandensein einer Assoziation zwischen zwei Klassen abhängig, so kann dies durch eine Assoziationsklasse ausgedrückt werden. Die Assoziationsklasse beschreibt Eigenschaften, die keiner der an der Assoziation beteiligten Klassen sinnvoll zugeordnet werden können. Die Assoziationsklasse wird mit einer gestrichelten Linie mit der Assoziation, von der sie abhängt, verbunden. Hat die Assoziation einen Namen, dann muss die Assoziationsklasse denselben Namen erhalten.
	Vererbung	Vererbung wird auch Generalisierung oder Spezialisierung genannt. Vererbungsbeziehungen werden mit einem Pfeil dargestellt. Die Pfeilspitze zeigt auf die Oberklasse. Die Oberklasse vererbts ihre Eigenschaften an die Unterklassen.
	Realisierung	Eine Klasse realisiert die Schnittstelle, die in einer anderen Klasse modelliert ist.

Notation	Element	Beschreibung
	Abhängigkeit	Eine Abhängigkeit ist die schwächste Art einer Beziehung. Eine Änderung in einer unabhängigen Klasse kann eine Änderung in der abhängigen Klasse erfordern. Häufig wird dies als eine „braucht“-Beziehung bezeichnet.
	Aggregation / Komposition	Beide Beziehungsarten sind Darstellung einer Ganze-Teile-Beziehung. Bei einer Komposition sind die Teile des Ganzen existenzabhängig vom Ganzen, d. h. wenn ein Objekt vom Typ „Ganzen“ gelöscht wird, werden auch die zugehörigen Teile gelöscht. Bei der Aggregation sind die Teile nicht existenzabhängig vom Ganzen.
	Mehrgliedrige Assoziation	Bei drei oder mehrstelligen Assoziationen entfällt die Leserichtung. Assoziationen können als eigene Assoziationsklasse ausgebildet und mit Attributen versehen werden. Findet beispielsweise häufig Verwendung, wenn die zu den Klassen zugehörigen Datenbanktabellen normalisiert werden sollen.

Beispiel

Ein konkretes Anwendungsbeispiel des Klassendiagramms ist der Ausschnitt aus der fortgeführten Modellierung des Systems zur Sportpartner-Vermittlung, das in Abbildung 5.10 zu finden ist. Die Klassen *Partner* und *Bestellung* enthalten die vier wichtigsten Methoden: Anlegen mit Laden (engl. create), Eintragen (engl. insert), Ändern (engl. update) und Zerstören (engl. destroy). Außerdem ist die Zeichnung unterteilt in ein **fachliches Konzept**, das im Regelfall Software-Engineering-Spezialisten entwerfen und ein **technisches Konzept**, das häufig die Entwickler ergänzen, um beispielsweise programmatische Vereinfachungen und sprachabhängige Details zu beschreiben oder um eine Beschleunigung von Prozessen auf technischer Ebene zu erzielen.

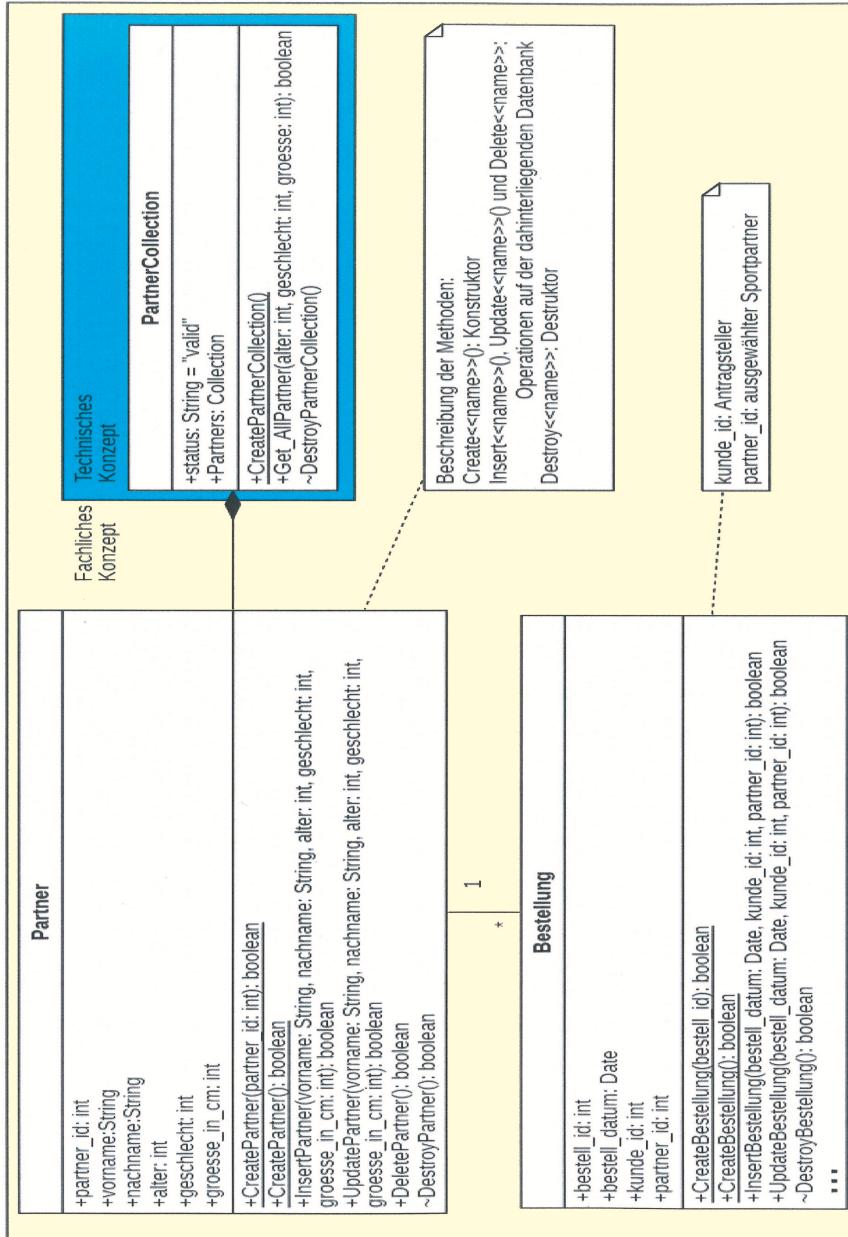


Abbildung 5.10 Beispiel: Ausschnitt aus dem Klassendiagramm der Sportpartner-Vermittlung

In unserem Beispiel wurde eine Liste (engl. collection) ergänzt, um technisch gesuchten schnelleren (und später evtl. sortierten) Zugriff auf die Partner zu erhalten. Würde die Liste der Partner (d. h. das Objekt *PartnerCollection*) nicht mehr gebraucht, stehen durch die Kompositionenbeziehung auch alle darin enthaltenen Partner nicht mehr zur Verfügung. Außerdem sei angemerkt, dass die Klasse *Bestellung* den Antragsteller durch das Attribut „*kunde_id*“ abbildet und der ausgewählte Sportpartner durch das Attribut „*partner_id*“ repräsentiert wird. Dadurch kann ein Antragsteller in unserem Modell gleichzeitig nur einen einzigen Sportpartners „anfordern“.

In dieser Art und Weise sind praktisch viele kleine, wichtige Details in solchen (Softwarearchitektur-) Modellen enthalten. Sie sind daher sehr sinnvoll und werden laufend von den Beteiligten konsultiert, diskutiert und verändert.

Für den Use-Case „*Partner suchen*“ kann, passend zum hier gezeigten Klassendiagramm, in Abbildung 5.19 ein Aktivitätsdiagramm und in Abbildung 5.22 ein Sequenzdiagramm betrachtet werden.

5.3.1.2 UML-Komponentendiagramm

Komponentendiagramme finden Verwendung, um eine Menge von Komponenten und Subkomponenten eines Software-Artefakts darzustellen. Es kann sich dabei auch um physikalische und ersetzbare Teile eines Systems handeln oder eben um

Tabelle 5.3 Einige wichtige Elemente des UML-Komponentendiagramms

Notation	Element	Beschreibung
<p>bereitgestellte Schnittstelle genutzte Schnittstelle</p>	Komponente	Komponenten haben einen eindeutigen Namen. Sie können Schnittstellen (eng. Interfaces) anbieten oder bereitstellen.
<p>Komponente A Port1 Port2 Schnittstelle X Schnittstelle Y Schnittstelle Z</p>	Port	Komponenten können über Ports oder direkt angesprochen werden.
<p>«manifest» Komponente A Datei.xml Komponente B Komponente C Komponente D Klasse 1 (from Komponente A)</p>	Schachtelung	Komponenten können geschachtelt werden und bei Bedarf intern und extern kommunizieren.

Softwareteile. Dargestellt werden auch Schnittstellen, die angeboten werden, beziehungsweise die Schnittstellen, die genutzt werden (siehe Tabelle 5.3).

Ein typisches Beispiel der Nutzung von Schnittstellen ist eine Sender- und Empfänger-Komponente wie in Abbildung 5.11 zu sehen. Wobei der Empfänger Nachrichten empfängt, die der Sender anbietet.

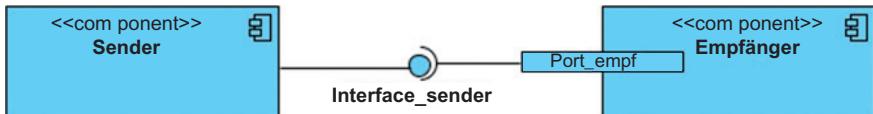


Abbildung 5.11 Beispiel: Sender- und Empfänger-Komponente

Ein weiteres Beispiel ist das Komponentendiagramm des Museumsführers aus Abbildung 5.12. Dieses Diagramm ist klar in Schichten eingeteilt. Die Einteilung der Schichten erfolgt hier in Komponenten für Daten (engl. model), Steuerung (engl. controller) und Anzeige (engl. view). Diese Einteilung wird auch MVC-Entwurfsmuster (Abk.: Model-View-Controller) genannt und ist damit ein Beispiel für gutes Software-Design, das vielfach als qualitativ hochwertige Architektur geprüft und in der Fachwelt akzeptiert ist. Die Entwurfsmuster (engl.: Design-Pattern) werden in Kapitel 5.4.3 diskutiert.

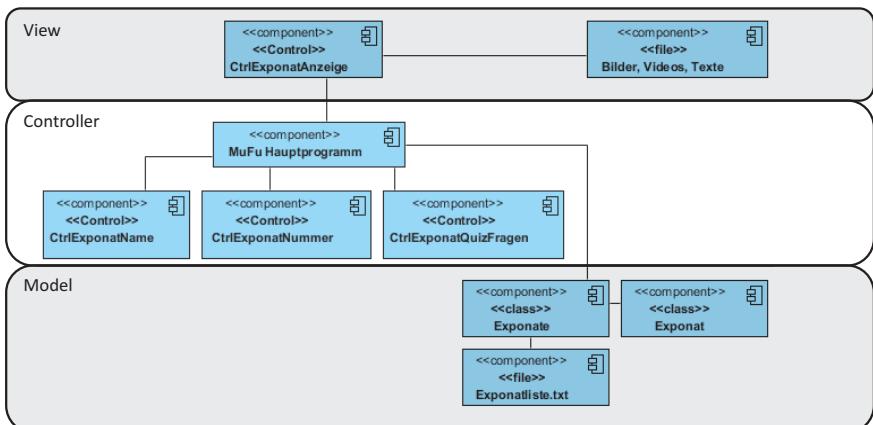


Abbildung 5.12 Beispiel: Komponenten der Software eines Museumsführers

Dabei gehören zur Datenschicht die Komponenten „class Exponate“, das ist die Listenansicht, „class Exponat“ zur Verwaltung des Einzelelements und die Datei „Exponatliste.txt“, welche in diesem Fall alle Exponate als Textform enthält.

Zur Steuerung gehören die Komponenten „*Hauptprogramm*“, „*CtrlExponatName*“ zur Suche mit dem Exponatsnamen, „*CtrlExponatNummer*“ zur Suche nach der Nummer und die Anzeige in Quizform „*CtrlExponatQuizFragen*“.

Schließlich gehören zur Anzeigeschicht die Komponenten „*CtrlComponentAnzeige*“ und „*files (Bilder, Videos und Texte)*“. Vorteil dieser Schichtenbildung ist auch, dass Projektbeteiligte einfach und schnell Software-Artefakte auffinden können.

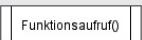
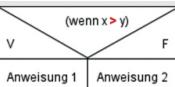
5.3.2 Verhaltensdiagramme

Mit Verhaltensdiagrammen kann grundsätzlich das Verhalten einer Software beschrieben werden. Damit sind Prozessabläufe oder Programmabläufe gemeint. Hier liegt der Fokus darauf, die Reihenfolge der schrittweisen Vorgänge zu beschreiben.

5.3.2.1 Struktogramme

Struktogramme, auch Nassi-Shneiderman-Diagramm genannt, wurden zur Modellierung von Software-Verhalten in der Vergangenheit genutzt. Sie sind nicht Teil der UML, jedoch nach **DIN 66261** genormt. Struktogramme sind gleichwohl gut geeignet als **Lernobjekte**, wie gute Softwarearchitektur modelliert werden kann, und finden

Tabelle 5.4 Einige wichtige Elemente des Struktogramms

Symbol	Element	Beschreibung
	Anweisung	Anweisungen werden in Programmen sequentiell nacheinander ausgeführt. In auf C basierenden Sprachen folgt nach jeder Anweisung beispielsweise ein Abschlusszeichen wie das Semikolon „;“.
	Funktionsaufruf	Da Struktogramme aus der Zeit des funktionalen Programmierens stammen, sind Funktionsaufrufe mit Parameterangaben möglich. (Beispiel: <i>summe(int x, int y);</i>)
	Fallunterscheidung	In Pseudocode ist die typische Schreibweise: <code>(if x > y){Anweisung1;} else {Anweisung2;}</code> .

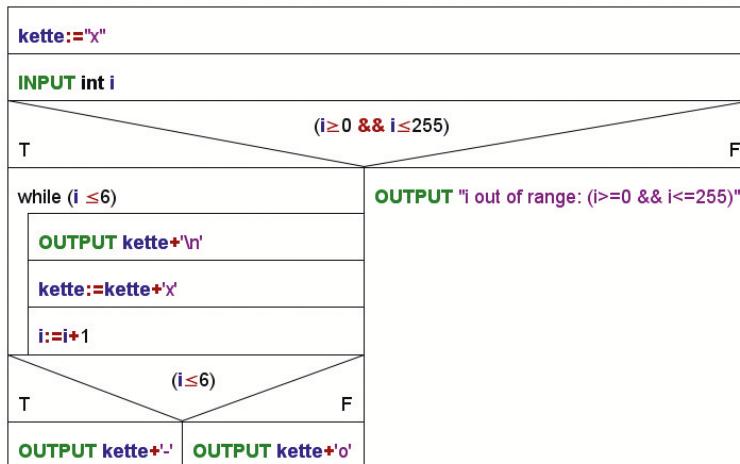
Symbol	Element	Beschreibung
	Mehrfachauswahl	In C-Derivaten handelt es sich hier um die Switch-case-Anweisung. (Beispiel: <code>switch (int i) case 1: Anweisung1; case 2: Anweisung2; else Anweisung 3;</code>)
	Abweisende Schleife	Die entsprechende Struktur in Programmen ist die While-Schleife. Merkmal ist, dass zuerst die Schleifenbedingung geprüft wird, bevor der Schleifenkörper ausgeführt wird. Der Schleifenkörper kann daher auch nicht (d. h. null mal) ausgeführt werden. (Beispiel: <code>while (x>y) {Anweisung1;} else {Anweisung2;}</code>)
	Annehmende Schleife	Die entsprechende Struktur in Programmen ist die Do-while-Schleife. Merkmal ist, dass zuerst der Schleifenkörper mindestens einmal ausgeführt wird, bevor die Schleifenbedingung geprüft wird. (Beispiel: <code>do {Anweisung1;} while (x>y)</code>)

daher hier kurz Erwähnung. Im Kapitel 5.3.2.2 über Aktivitätsdiagramme werden die Struktogramme zur Veranschaulichung des Begriffs „Wohlgeformtheit“ daher wieder verwendet. Die wichtigsten Notationselemente sind in Tabelle 5.4 zu finden.

Die zeitliche Reihenfolge der Vorgänge wird im Struktogramm von oben nach unten modelliert (siehe Abbildung 5.13). Es können hauptsächlich Sequenzen, Verzweigungen und Schleifen modelliert werden. Das Struktogramm eignet sich daher für eher programmmahe Prozessabläufe und eventuell weniger gut für abstraktere Darstellungen der Vorgänge (was jedoch mit UML-Aktivitätsdiagrammen sehr wohl möglich ist).

In dem gegebenen Beispiel von Abbildung 5.13 ist mit der Eingabe $i=0$ folgende Ausgabe zu erwarten:

```
X
XX
XXX
XXXX
XXXXX
XXXXXX
XXXXXXXX
XXXXXXXXXo
```

FunPainting()**Abbildung 5.13** Beispiel: Struktogramm der Funktion FunPainting()

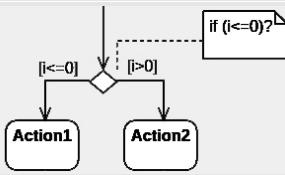
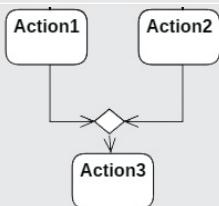
Heutzutage werden Programmabläufe oder auch abstraktere Vorgangsdarstellungen mit den UML-Aktivitätsdiagrammen visualisiert, die im nachfolgenden Abschnitt diskutiert werden.

5.3.2.2 UML-Aktivitätsdiagramme

Aktivitätsdiagramme finden Verwendung zur **Darstellung des Kontrollflusses** einer Software. Auch **abstraktere Prozessabläufe** sind damit visualisierbar. Ein Aktivitätsdiagramm zeigt das schrittweise Abarbeiten eines Vorgangs. Einige wichtige Notationselemente sind in Tabelle 5.5 wiedergegeben. Wie im Fall von Struktogrammen, können hier Sequenzen von Ablaufschritten, Verzweigungen und Wiederholungen modelliert werden. Zusätzlich können aber auch noch beliebige abstrakte Prozessabläufe, mit beispielsweise vielen Ein-, Ausgängen oder Abbrüchen, beschrieben werden. Das Aktivitätsdiagramm eignet sich somit auf allen Ebenen der Konzepterstellung zur Visualisierung von beliebigen Abläufen.

Eine komplette Darstellung aller möglichen Notationselemente des Aktivitätsdiagramms kann in [RQSG12] oder auch bei [Ollie18] gefunden werden. Für den Anfang reichen die hier gezeigten Elemente jedoch zunächst aus.

Tabelle 5.5 Einige wichtige Elemente des UML-Aktivitätsdiagramms

Symbol	Element	Beschreibung
●	Start	Startpunkt im Aktivitätsdiagramm.
○	Ende	Endpunkt im Aktivitätsdiagramm.
Action1	Aktion	Aktion, Aktivität oder Prozessschritt. (In der UML werden die Begriffe Aktion (i. d. R. Einzelanweisung) und Aktivität (d. h. kann weitere Aktionen enthalten) voneinander unterschieden).
→	Kontrollfluss	Die Pfeile von Symbol zu Symbol stellen den Kontrollfluss oder den Prozessablauf dar.
	Fallunterscheidung	In Pseudocode ist die typische Schreibweise: <code>(if x>y){Anweisung1;} else {Anweisung2;}</code> . Kennzeichen ist, dass ein Pfeil hineinführt und mehrere Pfeile hinausführen. Im Aktivitätsdiagramm wird daher aus dem Raute-Symbol auch die Mehr-fachauswahl generiert, wenn aus der Raute mehr als zwei Pfeile hinaus gehen. An jedem ausgehenden Pfeil muss ein sogenannter Guard (d. h. die Bedingungsoption, z. B. [$i < 0$]) stehen, damit der Programmablauf eindeutig entschieden werden kann. Mit der Raute können auch Schleifen erzeugt werden, wie noch gezeigt wird.
	Zusammenführung	Die Zusammenführung (engl. Merge) fungiert wie die schließende Klammer nach Fallunterscheidungen oder zum Abschluss von Schleifen. Mehrere Pfeile führen hinein. Nur ein Pfeil führt hinaus.

In Abbildung 5.14 ist ein Beispiel eines gültigen Aktivitätsdiagramm zu finden. Es handelt sich um das gleiche Modell und die gleiche Funktion *FunPainting()* wie das Struktogramm aus Abbildung 5.13. Gezeigt werden hier zwei ineinander geschachtelte Verzweigungen und eine Schleife.

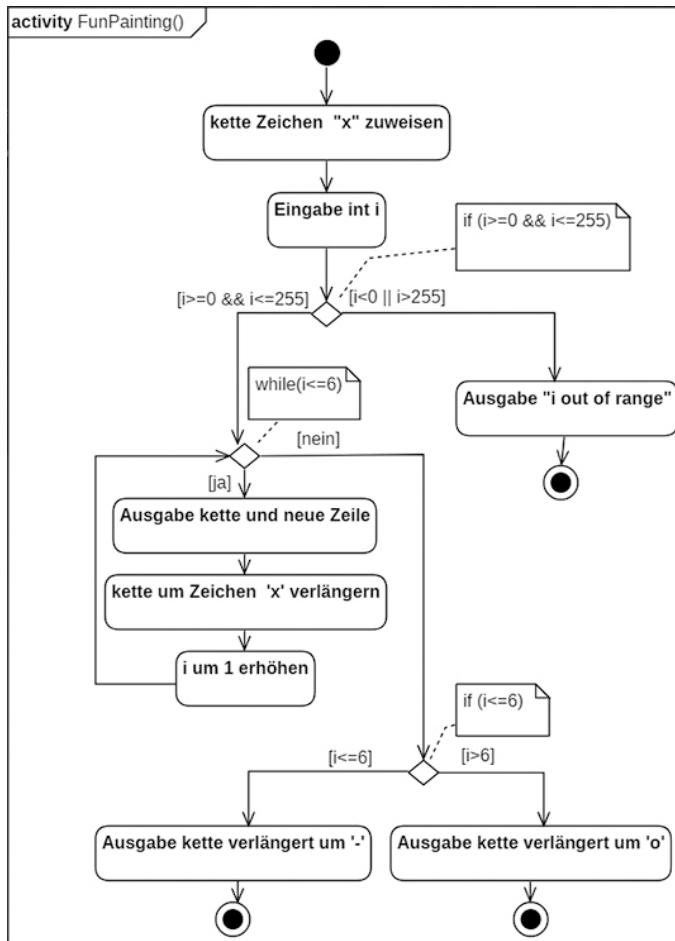


Abbildung 5.14 Beispiel eines gültigen Aktivitätsdiagramms – Funktion *FunPainting()*

Gelesen wird das Aktivitätsdiagramm normalerweise von oben nach unten und von links nach rechts. Jedoch ist die Leserichtung nicht vorgegeben und daher kann davon abgewichen werden.

Viel interessanter an dem Beispiel ist jedoch, dass zwar für die Funktion *FunPainting()* die gleiche Funktionalität wie im Struktogramm visualisiert wird, jedoch die Übersichtlichkeit über das Aktivitätsdiagramm leichter verloren geht (besonders natürlich bei komplexeren und größeren Modellen). Dies liegt bei genauerer Betrachtung an der legitimen Verwendung von drei Ende-Symbolen, der gleichzeitigen Verwendung der While-Schleifen-Raute als Verzweigungssymbol und Zusammenführungssymbol und der Nicht-Schließung der Klammer für die zweite Fallunterscheidung. Um also gut lesbare und programmierbare Aktivitätsdiagramme modellieren zu können, ist die Anwendung der sogenannten „Wohlgeformtheit“ nötig.

Aktivitätsdiagramm wohlgeformt

Obwohl im Aktivitätsdiagramm abstrakte Abläufe modellierbar sind, sollte sich ein guter Software-Engineering-Spezialist darum bemühen sogenannte **wohlgeformte Aktivitätsdiagramme** zu erstellen (vgl. [HV04]).

In wohlgeformten Aktivitätsdiagrammen werden hauptsächlich alle Verzweigungen durch eine Zusammenführung (englisch: Merge) wieder geschlossen (siehe Beispiel in Abbildung 5.15). Genauso wie bei Schleifen ein Zusammenführungssymbol eingefügt werden muss, um am Ende wieder zu einem einzigen Pfad im Ablauf zu kommen. (Schließlich handelt es sich hier ja nicht um zwei parallel ablaufende Threats eines Prozesses, welche wiederum mit ganz anderer Symbolik dargestellt werden müssten.)

Vergleicht man nun das Struktogramm der Funktion *FunPainting()* aus Abbildung 5.13 mit dem Aktivitätsdiagramm der Funktion *FunPainting2()* in Abbildung 5.15, sieht man sehr gut die genaue Entsprechung der beiden Modelle. (Somit ist klar, wofür die Darstellung der Struktogramme in diesem Buch genutzt wurde.)

Wohlgeformt mit vollständiger Beschriftung

Nicht direkt zur Wohlgeformtheit gehört die **Nummerierung** der Rauten eines Aktivitätsdiagramms (siehe Abbildung 5.15) und die Ergänzung der Fallunterscheidungs- und Schleifen-Bedingungen in den **Notizfeldern** der Rauten. Jedoch sei hier angemerkt, dass die vollständige Beschriftung die Lesbarkeit eines Aktivitätsdiagramms ungemein erhöht und daher unbedingt verwendet werden sollte!

Ein Software-Engineering-Spezialist kann mit Hilfe des Konstrukts der Wohlgeformtheit nun selbst entscheiden, ob sein Modell wohlgeformt oder eher abstrakt zu zeichnen ist!

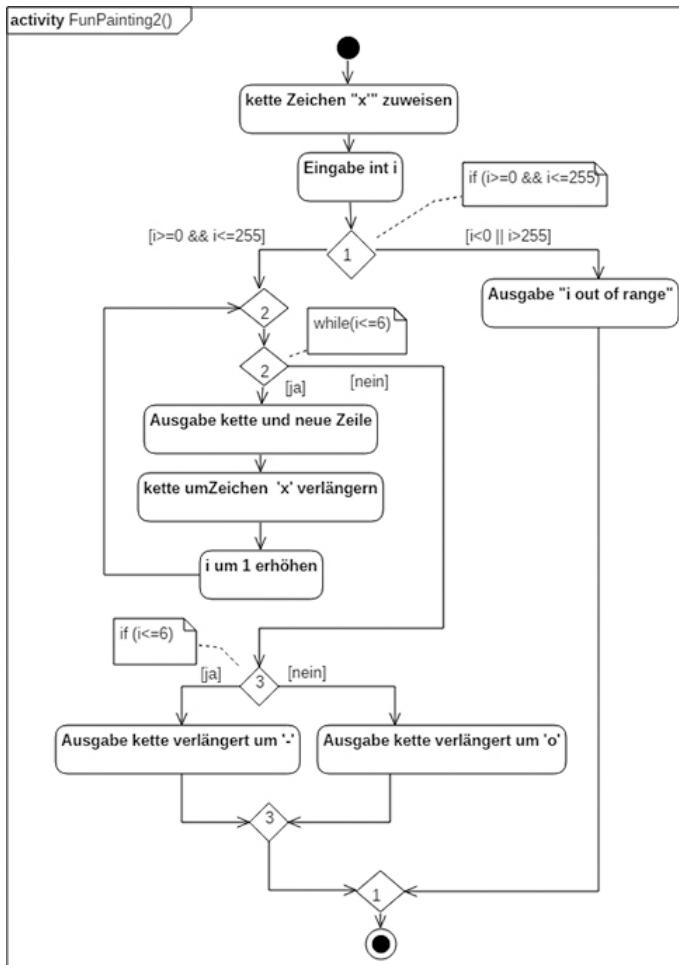


Abbildung 5.15 Funktion `FunPainting2()` – ein wohlgeformtes Aktivitätsdiagramm

Zusammenfassung der Wohlgeformtheit

Computerprogramme bestehen hauptsächlich aus den Kontrollstrukturelementen, die in Abbildung 5.16 zusammengefasst sind. Dort wird deutlich, wie diese Kontrollstrukturelemente im Struktogramm und mit der UML als Aktivitätsdiagramme modelliert werden können.

Fachbegriff	Struktogramm	Aktivitätsdiagramm
Modulare Sub-Struktur		
Sequenz		
Fallunterscheidung		
Fachbegriff	Struktogramm	Aktivitätsdiagramm
Schleife		
Parallelisierung	- kein Symbol -	
Prozesseinheit		

Abbildung 5.16 Aktivitätsdiagramm – Kontrollstrukturelemente nach [HG08]

In Abbildung 5.16 sind auch die Symbole für **Parallelisierung** und **Serialisierung** zu finden, die nur von Fachleuten verwendet werden sollten, wenn ein **Prozess** in mehrere sogenannte **Threats** aufgeteilt wird. In der Notation der UML spricht man hier auch davon, dass ein sogenanntes Token (imaginäre Darstellung durch ein Bällchen) am Parallelisierungssymbol (in der Zeichnung oben) geteilt wird in zwei Token. Je ein Token „fließt“ schließlich durch seinen eigenen Kontrollflusspfad bis zum nächsten Serialisierungssymbol (in der Zeichnung unten). Am Serialisierungssymbol wird auf alle weiteren Token bis zu deren Ankunft gewartet. Erst dann wird serialisiert und alle angekommenen Token werden wiederum zu einem einzigen Token verschmolzen. Von da ab läuft neuerlich nur noch ein einziger Kontrollfluss weiter, nämlich

der Prozess (siehe auch [RQSG12]). Diese Konstrukte dürfen **nicht** mit der einfachen Fallunterscheidung verwechselt werden (was Einsteigern zuweilen passiert)!

Schleifenknoten und Entscheidungsknoten

Wem die Schleifenbildung mit einfachen Mitteln zu kompliziert erscheint oder auch wer nur kleine, kurze Schleifen modellieren möchte, der kann die in Abbildung 5.17 angegebenen alternativen UML-Notationselemente verwenden. Schleifen werden als sogenannte Schleifenknoten und Fallunterscheidungen als Entscheidungsknoten bezeichnet. Das Vorgehen ist jedoch erst zu empfehlen, wenn die Schleifenbildung mit einfachen Mitteln perfekt beherrscht wird.

Fachbegriff	Aktivitätsdiagramm
Alternative Schleifendarstellung mit UML: Schleifenknoten	<pre> graph TD subgraph "while Schleifenbedingung" direction TB A1[do Schleifenrumpf] --> B1[while Schleifenbedingung] B1 --> C1[do Schleifenrumpf] C1 --> D1[while Schleifenbedingung] end subgraph "do Schleifenrumpf" direction TB A2[do Schleifenrumpf] --> B2[do Schleifenrumpf] B2 --> C2[while Schleifenbedingung] C2 --> D2[do Schleifenrumpf] end subgraph "for Bereich mit initialen Aufgaben" direction TB A3[do Schleifenrumpf] --> B3[for Bereich mit initialen Aufgaben] B3 --> C3[do Schleifenrumpf] end </pre>
Alternative Fallunterscheidung mit UML: Entscheidungsknoten	<pre> graph TD A4[if Bedingung] --> B4[then auszufhrende Aktionen] A4 --> C4[else auszufhrende Aktionen] </pre>

Abbildung 5.17 Aktivitätsdiagramm – Schleifenknoten und Entscheidungsknoten nach [RQSG12]

Verbindung mit den Anwendungsfällen (Use-Cases) aus der Definitionsphase

Aktivitätsdiagramme werden häufig dazu verwendet, um Anforderungen aus der Definitionsphase im Detail zu spezifizieren (auch als **Feinkonzept** bezeichnet).

Ein Beispiel, wie Use-Case-Diagramme mit Aktivitätsdiagrammen verfeinert werden können, ist ein Anwendungsfall aus Abbildung 4.6 (und dessen tabellarische Beschreibung aus Tabelle 4.4), welches das schon vorgestellte Sportpartnerschaftsvermittlungssystem repräsentiert (siehe relevanter Ausschnitt in Abbildung 5.18).



Abbildung 5.18 Use-Case-Diagrammausschnitt des Beispiels „Partner suchen“

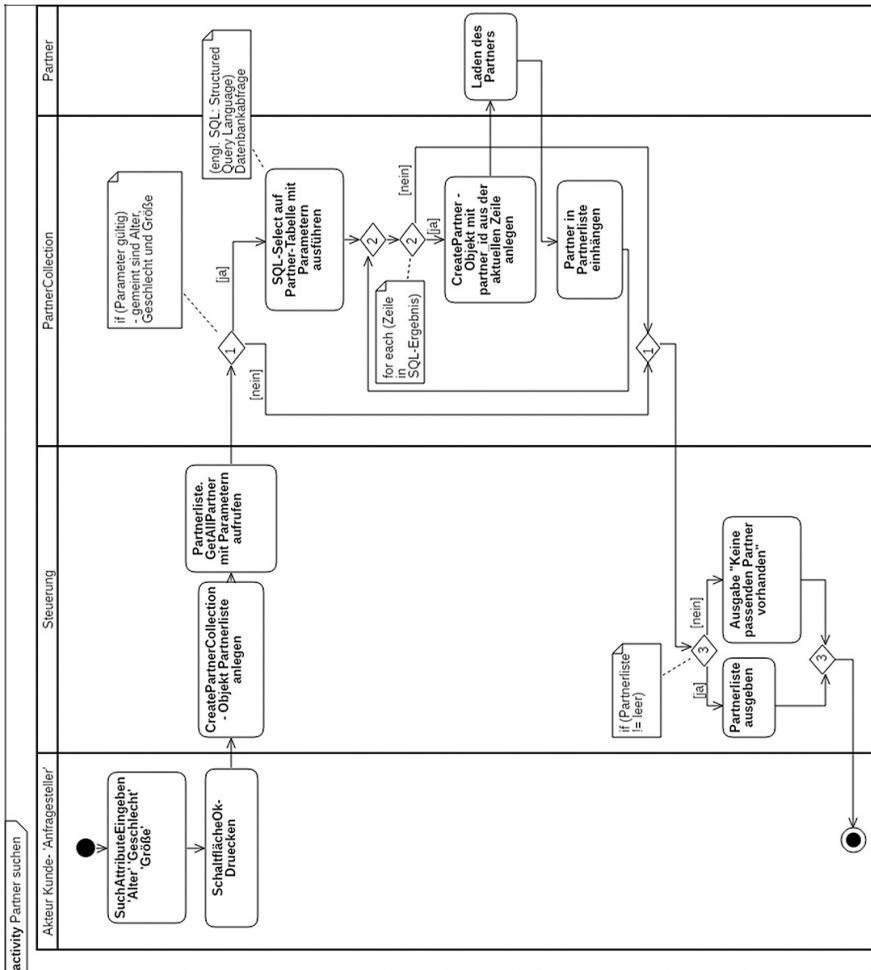


Abbildung 5.19 Aktivitätsdiagramm Beispiel – Aktivität „Partner suchen“

(Der zugehörige Ausschnitt des Klassendiagramms kann schließlich Abbildung 5.10 entnommen werden.)

Der Use-Case „*Partner suchen*“ wird nun im Aktivitätsdiagramm Abbildung 5.19 verfeinert dargestellt. Zu sehen ist dabei auch das Notationselement der so genannten „**Swimlanes**“ (auf Deutsch: Schwimmbahnen), mithilfe deren man Verantwortlichkeiten bzw. Verantwortungsbereiche für alle enthaltenen Aktivitäten definieren kann. Swimlanes können sowohl vertikal als auch horizontal eingesetzt werden. Im Aktivitätsdiagramm des Beispiels ist nun viel genauer der Prozessablauf abzulesen, der durch den zugehörigen Use-Case nur kurz beschrieben wird.

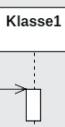
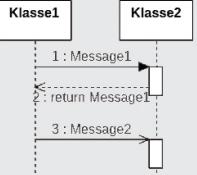
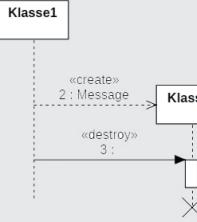
Inhaltlich ist bei diesem Beispiel im Aktivitätsdiagramm zu lesen, dass ein Kunde (d. h. der Antragsteller) auf einer Benutzeroberfläche Attribute, wie Alter, Geschlecht und Größe, eingibt. Er bestätigt anschließend seinen Vorgang mit der Schaltfläche „Ok“. Die Steuerung der Sportpartnerverwaltung legt darauf ein Objekt vom Typ *PartnerCollection* mit dem Namen *Partnerliste* an. Die Steuerung ruft danach die Methode *GetAllPartner()* des Objekts *Partnerliste* auf. Falls die an die Methode übergebenen Parameter (Alter, Geschlecht und Größe) nicht gültig sind, bleibt die *Partnerliste* leer. Sind diese jedoch gültig, dann wird an die Datenbank der Befehl zur Suche mit diesen Parametern abgesetzt. Werden Datensätze gefunden, so wird pro Datensatz (d. h. pro Zeile), mithilfe einer For-each-Schleife, ein *Partner-Objekt* angelegt, mit den gefundenen Zeilenwerten geladen und schließlich in die *Partnerliste* eingehängt. Die Steuerung gibt nun, mithilfe einer Verzweigung, entweder bei gefüllter Partnerliste dieselbe nun aus, oder erzeugt die Ausgabe „Keine passenden Partner vorhanden“. Zuletzt wird der Vorgang noch beendet und dem Kunden die Möglichkeit gegeben, über die Benutzeroberfläche, weiterzuarbeiten.

5.3.2.3 UML-Sequenzdiagramm

Sequenzdiagramme werden hauptsächlich zur Darstellung von **Interaktionen zwischen Objekten** und Objektgruppen eingesetzt. Die Betonung liegt auf der **zeitlichen Reihenfolge von Nachrichten**, die zwischen den Objekten versendet werden. Es geht also hauptsächlich um **Kommunikation**. Eine Übersicht über die wichtigsten Notationselemente ist in Tabelle 5.6 zu finden.

Tabelle 5.6 Einige wichtige Notationselemente des UML-Sequenzdiagramms

Symbol	Element	Beschreibung
	Rahmen	Rahmen eines Sequenzdiagramms, in dem der darzustellende Ablauf modelliert wird. Rahmen erhalten einen Namen.

Symbol	Element	Beschreibung
	Klasse, Objekt	Ein Sequenzdiagramm enthält Klassen oder Objekte. An jedem dieser Elemente hängt eine eigene gestrichelte Lebenslinie . Ob eine Klasse aktiv ist (d. h. Prozessorzeit erhält), kann mit einem Balken auf der Lebenslinie modelliert werden. Jede Klasse kann Nachrichten empfangen oder versenden. Nachrichten werden mit Pfeilen dargestellt. Der zeitliche Ablauf im Sequenzdiagramm wird immer von oben nach unten dargestellt.
	Nachrichten	Pfeile repräsentieren einfache Nachrichten, Events oder Signale. In der Regel steht als Beschriftung an einem Pfeil ein Methodenaufruf (z. B. <i>Create-Partner()</i>). Der Pfeiltyp gibt die Nachrichtenart an. <i>Message1</i> ist eine synchrone Nachricht , die eine Rückantwort erwartet. <i>Message2</i> ist eine asynchrone Nachricht und erwartet daher keine Antwort. Nachrichten werden immer zwischen zwei Lebenslinien versendet (Ausnahme: Ankommende/Abgehende Nachricht aus/zu anderem Diagramm.)
	Erzeugung, Zerstörung	Klassen und Objekte können durch Erzeugungsnachrichten erzeugt und durch Angabe des Kreuzes auf der Lebenslinie zerstört werden.

Symbol	Element	Beschreibung
<pre> sequenceDiagram participant K1 as Klasse1 participant K2 as Klasse2 K1->>K1: opt if(i<=0) activate K1 K1-->>K1: 4 : Message3 activate K1 K1-->>K1: 5 : return Message3 deactivate K1 activate K1 K1-->>K1: i>0 activate K1 K1-->>K1: 6 : Message4 activate K1 K1-->>K1: 7 : return Message 4 deactivate K1 </pre>	Kontrollstrukturelemente	Genauso wie beim Aktivitätsdiagramm können auch beim Sequenzdiagramm Kontrollstrukturelemente eingesetzt werden. Die wichtigsten sind die Fallunterscheidung mit <code><opt></code> (Abk. Option), Mehrfachauswahl mit <code><alt></code> (Abk. Alternative) und die Schleife mit <code><loop></code> . Es gibt noch einige weitere wie z. B. den Programmaufruf mit <code><ref></code> (Abk. Referenz). Dabei müssen jeweils die notwendigen Bedingungen und Optionen im jeweiligen Zweig angegeben werden (im Beispiel bei der Fallunterscheidung <code>[i>0]</code>). Wichtige Regel dabei ist, dass Kontrollstrukturelemente immer über mindestens zwei Lebenslinien gespannt werden .

Eine komplette Referenz der Möglichkeiten kann wiederum in Kurzform bei [ÖS14] oder ausführlich in [RQSG12] gefunden werden.

Fachbegriff	Symbol
Synchrone Nachricht	→
Antwortnachricht	→ - - - -
Asynchrone Nachricht	→

Abbildung 5.20 Sequenzdiagramm – Nachrichtenelemente

Die Zusammenfassung der Notation, über die wichtigsten synchronen und asynchronen Nachrichtenelemente, ist in Abbildung 5.20 zu finden.

Zeitliche Abläufe lassen sich im Sequenzdiagramm durch Nachrichten, die quer verlaufen, darstellen, wie in Abbildung 5.21 gezeigt.

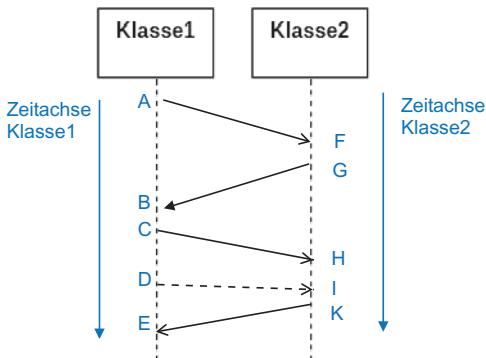


Abbildung 5.21 Sequenzdiagramm – Kommunikation

Dabei können sich theoretisch die Einheiten der Zeitachsen auch unterscheiden. Eine beliebte Frage ist daher auch, in welcher Reihenfolge die Nachrichteneignisse eintreffen. Im Beispiel werden gleiche Einheiten der Zeitachsen angenommen. Daher wäre die korrekte Antwort: A-F-G-B-C-H-D-I-K-E

Beispiel

Um nun ein komplexeres Beispiel zu zeigen, wird wieder auf den Use-Case „Partner Suchen“ des Sportpartnervermittlungssystems zurückgegriffen (siehe Abbildung 5.22).

Ein wirklich wichtiger Aspekt ist dabei folgender. Vergleicht man nun dieses Sequenzdiagramm mit dem zugehörigen Aktivitätsdiagramm aus Abbildung 5.19, so ist es wichtig, die komplette Entsprechung der beiden Diagramme zu erkennen.



Merk Aktivitätsdiagramme und Sequenzdiagramme können die gleichen Prozess- und Programmabläufe darstellen.

Der Fokus im **Aktivitätsdiagramm** liegt jedoch bei der Darstellung der **Ablaufschritte**, während der Schwerpunkt im **Sequenzdiagramm** klar die **Kommunikation zwischen Objekten** ist.

Ein Modellierer kann also einfach wählen, welche Aspekte er in einem Architekturteil akzentuiert möchte.

Inhaltlich ist im Sequenzdiagramm aus Abbildung 5.22 also der gleiche Vorgang, wie im Aktivitätsdiagramm (siehe Abbildung 5.19) abgebildet. Daher ist die zugehörige inhaltliche Beschreibung im Kapitel 5.3.2.2 Aktivitätsdiagramme zu finden. Jedoch werden im Sequenzdiagramm noch zusätzlich die Nachrichten gezeigt, welche

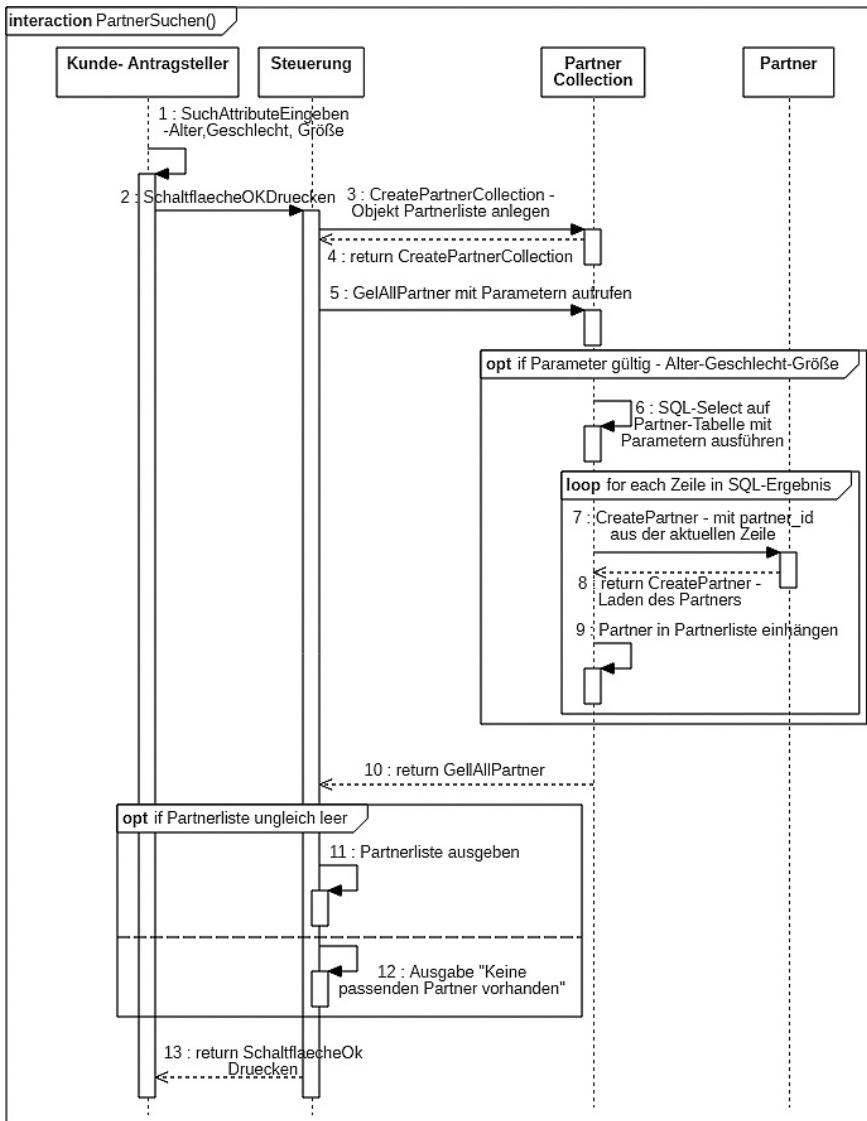


Abbildung 5.22 Sequenzdiagramm Beispiel – „Partner suchen“

im Programmablauf gesendet und empfangen werden. Beispielsweise wird von der Steuerung das Objekt *Partnerliste* vom Typ *PartnerCollection* angelegt. Die Steuerung erhält mit einem Return-Befehl die Rückantwort dazu.

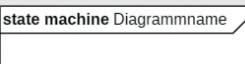
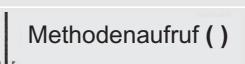
(Anmerkung: Streng genommen müsste nun der Rückgabewert (laut Klassendiagramm aus Abbildung 5.10 ein *Boolean*, also „true“ oder „false“) abgefangen und damit die Anlage des Objekts auf Korrektheit überprüft werden. Solche Details wurden hier im Modell zur Komplexitätsreduktion und zur Bewahrung der Übersichtlichkeit vernachlässigt. Zur Fehlervermeidung in realen Implementierungen darf darauf aber natürlich nicht verzichtet werden.)

Außerdem wird die korrekte Schachtelung von Kontrollstrukturelementen im Beispiel demonstriert. Die Verzweigung, zur Abfrage der Gültigkeit aktueller (Such-) Parameter, umgibt die For-each-Schleife, mit der die einzelnen Datensätze (d. h. gespeicherter Partner aus der Datenbank) geladen werden.

5.3.2.4 UML-Zustandsdiagramm

Das Zustandsdiagramm wird zur Darstellung des **Kontrollflusses** von **Zustand zu Zustand** in einem **Zustandsautomaten** verwendet. Eine Übersicht der wichtigsten Notationselemente ist in Tabelle 5.7 zu finden. Damit sind erste, einfache Zustandsdiagramme modellierbar.

Tabelle 5.7 Einige wichtige Notationselemente des UML-Zustandsdiagramm

Symbol	Element	Beschreibung
	Rahmen	Rahmen eines Zustandsdiagramms, in dem die darzustellenden Zustände modelliert werden. Rahmen erhalten einen Namen.
	Start	Startsymbol des Zustandsdiagramms.
	Ende	Endesymbol des Zustandsdiagramms.
	Zustand	Jeder Zustand wird benannt (z. B. „wartend“). In der UML-Gemeinde gilt die Regel, dass Zustände in der Zeitform Präteritum (1.Vergangenheitsform) geschrieben werden.
	Transition	Der Pfeil stellt eine Transition dar. Das ist ein Zustandsübergang von einem Zustand zum nächsten Zustand. Als Transition ist in der Regel ein Methodenauftrag zu modellieren (z. B. „Buch_oeffnen()“). In der UML-Gemeinde gilt die Regel, dass Methodenaufträge in der Zeitform Präsens (Gegenwartsform) geschrieben werden.

Zustandsdiagramme werden **immer** genau für **ein** zu modellierendes **Objekt** erstellt. Für dieses Objekt wird dargestellt, welche **Zustände** dieses annehmen kann und welche **Übergänge** von Zustand zu Zustand möglich sind. (Ein Beispiel eines solchen Objektes ist ein „*Buch*“.)

Beispiel

In Abbildung 5.23 ist ein Zustandsdiagramm für das Objekt „*Buch*“ zu sehen.

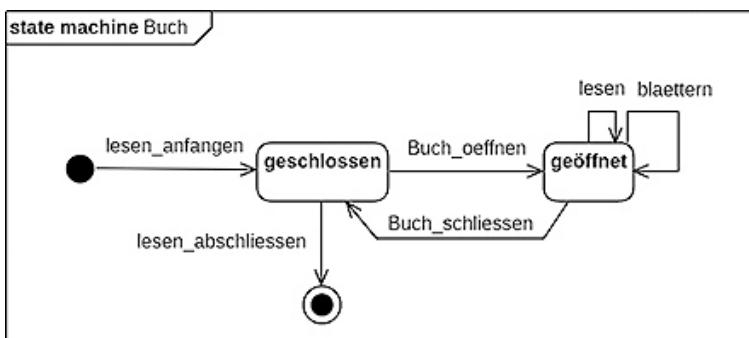


Abbildung 5.23 Beispiel: Zustandsdiagramm für das Objekt „*Buch*“

Als erstes hat das Buch den Zustand *geschlossen*. Durch Aufruf der Methode *Buch_oeffnen()* wechselt das Buch in den Zustand *geoeffnet* über. Der Aufruf der Methode *blaettern()* bewirkt, dass das Buch wieder im gleichen Zustand *geoeffnet* landet. Das gleiche geschieht beim Aufruf der Methode *lesen()*. Wird jedoch die Methode *Buch_schliessen()* aufgerufen, so wechselt das Buch wieder in den Zustand *geschlossen*. Der Aufruf der Methode *lesen_abschliessen()* bewirkt zuletzt, dass der Endezustand des Diagramms erreicht wird.

Nicht jede Software ist zustands-basiert, d. h. nicht jede Software hat Zustände. Aus diesem Grund wird das Zustandsdiagramm erwartungsgemäß auch nur in **zustands-basierter Software** eingesetzt.

5.4 Softwarearchitekturen

Zuerst ist wiederum die Begriffsklärung einer Softwarearchitektur wichtig.



Definition

Eine **Softwarearchitektur** ist eine Beschreibung der Subsysteme und Komponenten eines Software-Systems und der Beziehungen dazwischen.



Merke

Eine **Softwarearchitektur** beinhaltet die

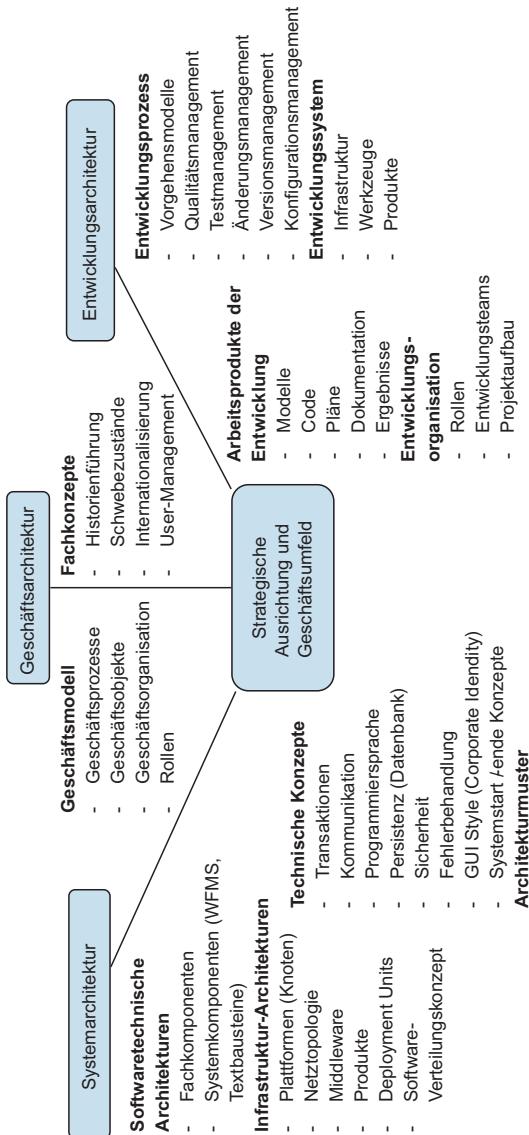
- **Makroarchitektur:** Darunter versteht man eine sogenannte High-Level-Architektur einer Software,
z. B. Web-Systeme basieren auf verteilten Architekturen
- **Mikroarchitektur:** Darunter wird die Softwarestruktur eines Software-Artefaktes verstanden,
z. B. Programmstruktur, Low-Level-Modelle und die sogenannten „**Design-Pattern**“ (siehe Kapitel 5.4.3).

Softwarearchitekturen gibt es in vielen unterschiedlichen Arten. Diese entstehen, da unterschiedliche Aspekte eines IT-Projektes beschrieben und visualisiert werden müssen. Eine mögliche Einteilung und Übersicht ist in Abbildung 5.24 gegeben.

Nach Foegen und Battenfeld lassen sich Architekturen in die Bereiche **Geschäftsarchitektur**, **Systemarchitektur** und **Entwicklungsarchitektur** einteilen (vgl. [FB01]). Mit großer Wahrscheinlichkeit gibt es auch noch viele weitere, hier nicht genannte Architekturen in dieser Übersicht zu ergänzen. In der Regel sind viele der darunter eingeordneten Architekturen für ein Software-Projekt notwendig. Dadurch wird nun klar, dass die Verwaltung der vielfältigen Diagramme für ein Projekt aufwendig ist und Strategie und Sorgfalt erfordert.

Musterarchitekturen

Ein empfehlenswertes Vorgehen ist möglichst bekannte Muster einzusetzen. Man spricht hier von **Musterarchitekturen**, **Architekturmustern** und **Entwurfsmustern**. Ein Beispiel für unterschiedliche Architekturbilder der gleichen Software ist in Abbildung 5.25 zu sehen. Oberste Schicht ist hier die Musterarchitektur SOA (Abk.: Service-orientierte Architektur). Es handelt sich dabei um den Ansatz, dass eine Anwendung bei Bedarf so genannte Services aus dem Netz aufruft. Dabei wird bei einer Anfrage aus einer Liste möglicher Service ein passender bzw. der günstigste ausgewählt und schließlich angewendet. Die mittlere Schicht ist das Architekturmuster „Client-Server“ (siehe Kapitel 5.2), die typische Web-Architektur. Das bedeutet, dass die gezeigte SOA auf einer Client-Server-Architektur basiert, bzw. dass sich aus der Client-Server-Architektur eine SOA bauen lässt. Die unterste Schicht in der Abbildung wird durch Klassen (siehe Kapitel 5.3.1.1 Klassendiagramm) und Entwurfsmuster (siehe Kapitel 5.4.3 Entwurfsmuster) wie beispielsweise ein Singleton (englisch: Einzelstück) beschrieben. Die Client-Server-Architektur basiert also wiederum auf Klassen und Entwurfsmustern. Ein solcher schichtenweiser Aufbau schafft Struktur, Übersicht und hat Wiedererkennungswert.


Abbildung 5.24 Architekturbereiche nach [FB01])

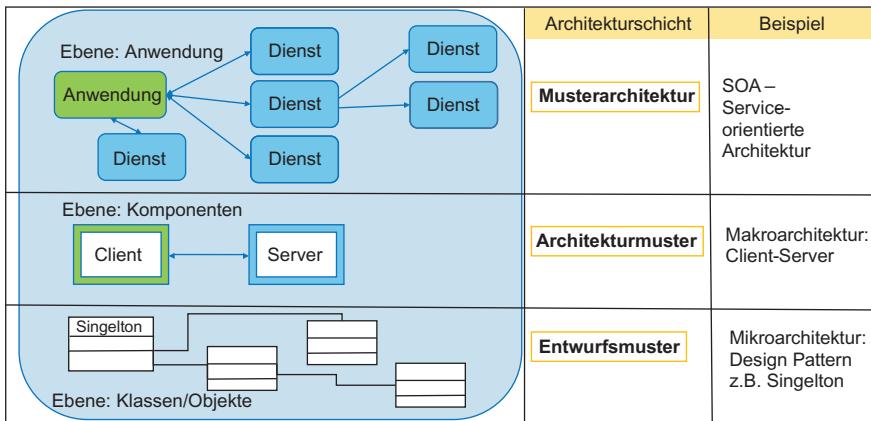


Abbildung 5.25 Beispiel unterschiedlicher Architektschichten nach [DLWZ03]

Welche Dienste und Service die SOA also schließlich anbietet, wird letztlich auf die Klassen und Entwurfsmustern auf der untersten Ebene heruntergebrochen und modelliert. So erhält man übersichtliche Softwarearchitekturen. Durch die Verwendung der bekannten Muster können die beteiligten Fachleute die Softwarearchitektur schnell, sicher und einfach verstehen.

5.4.1 Subsysteme und Komponenten

Am Beispiel in Abbildung 5.26 kann nachvollzogen werden, dass die Zerlegung von großen, komplexen Systemen in kleinere Subsysteme und Komponenten sinnvoll ist. Die Modellierung von Subsystemen und Komponenten in UML kann etwa durch Verschachtelung im **Komponentendiagramm** (siehe Kapitel 5.3.1.2 Komponentendiagramm) oder Zerlegung in Pakete mit dem UML-Paketdiagramm (nachlesbar in [RQSG12]) erreicht werden.

Gibt es nun einen Unterschied zwischen Subsystemen und Komponenten?

In großen oder komplexen Systemen macht die Unterscheidung in Subsysteme und Komponenten Sinn. Sie ist jedoch nicht in jedem Modellierungsfall zwingend erforderlich. Ein Vorschlag zur sinnvollen Unterscheidung kann beispielsweise in Abbildung 5.27 gefunden werden.

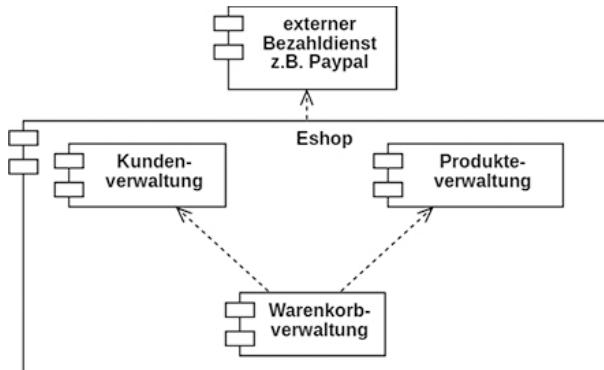


Abbildung 5.26 Beispiel: Mögliche Subsysteme und Komponenten des E-Shop-Beispiels

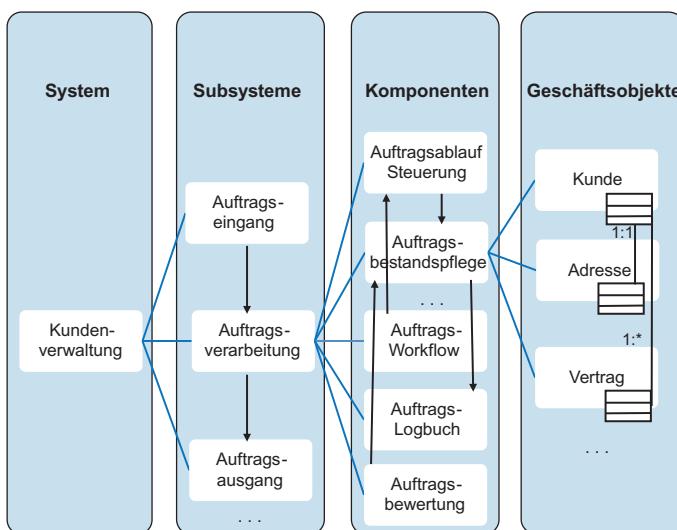


Abbildung 5.27 Der Unterschied von Subsystemen und Komponenten nach [FB01], einsetzbar im E-Shop

Die Kommunikation zwischen Subsystemen oder auch zwischen Komponenten kann dabei durch Sequenzdiagramme dargestellt werden (siehe Abbildung 5.28).

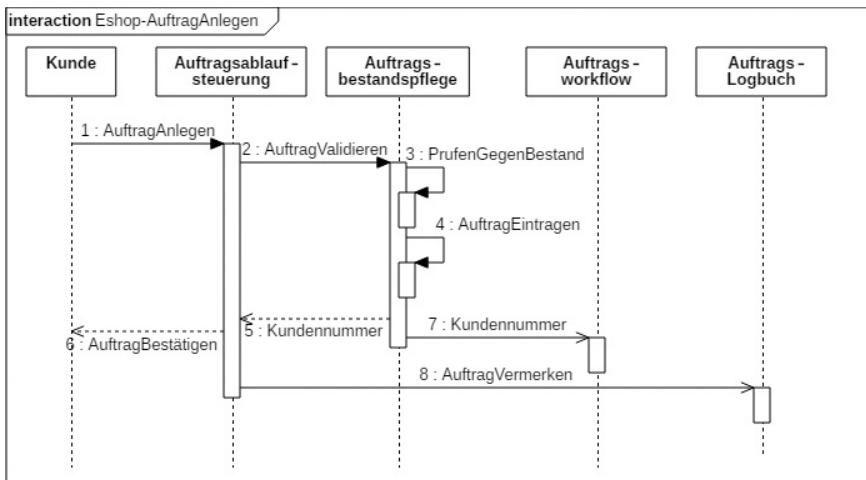


Abbildung 5.28 Modellierung der Kommunikation in Subsystemen und Komponenten nach [FB01], einsetzbar im E-Shop Beispiel

5.4.2 Makroarchitekturen

Es gibt einige grundsätzliche und nützliche Architekturvorschläge für Makroarchitekturen, die hier zur Ideenfindung aufgelistet werden. Es ist generell sinnvoll, viele solche Vorschläge einzusetzen, da diese in sehr vielen Projekten bereits erprobt wurden und sich dadurch bewährt haben. Es macht daher Sinn, diese zu studieren und zu kennen, um für konkrete Projekte eine geeignete Auswahl treffen zu können.

5.4.2.1 Allgemeine Architekturen

Einige der bekanntesten allgemeinen Softwarearchitekturen, die unbedingt in IT-Projekten eingesetzt werden sollten, sind Schichten-, Pipe-, Filter- und Blackboard-Architekturen.

- **Schichten**

Beispiele: ISO-OSI 7-Schichtenmodell, Entwurfsmuster Model-View-Controller (siehe auch Entwurfsmuster in Kapitel 5.4.3 und siehe Abbildung 5.29)

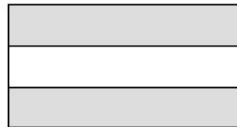


Abbildung 5.29 Makroarchitektur – Schichten

- **Pipes:** FIFO- (engl. First-In-First-Out), LIFO- (engl. Last-In-First-Out) und Stack-Konstrukte
Beispiel: Message Queues, allgemeine Warteschlangen (siehe Abbildung 5.30)
- **Filter**
Beispiel: Spamfilter, Filter durch die Verwendung von Registerkarten zur Verkleinerung großer Menüstrukturen
- **Blackboards**
Architekturmödell: Expertengruppe, Zusammenarbeit, Lösungen in hierarchisch organisierter Form.



Abbildung 5.30 Makroarchitektur – Warteschlangen

5.4.2.2 Verteilte Architekturen

Es gibt viele unterschiedliche verteilte Architekturen (siehe [AST03]). Zwei Beispiele aus diesem Bereich sind die sogenannte Three-tier-Architektur und die Request-Broker-Architektur.

- **Three-tier-Architekturen** (siehe Abbildung 5.31)
Diese Architekturen bestehen typischerweise aus drei Teilen. Ein typischer Vertreter ist ein web-basiertes System, das aus den drei Teilen Client, Server und Datenbank besteht (siehe auch durchgängiges Beispiel in Kapitel 5.2). Eine typische Three-tier-Architektur ist eine **Client-Server-Architektur**.

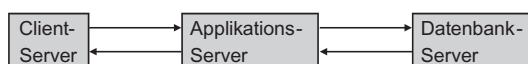


Abbildung 5.31 Typische Three-tier-Architektur

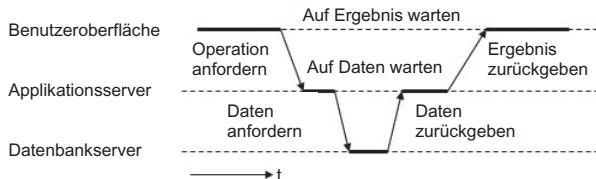


Abbildung 5.32 Typische Client-Server-Architektur

In Abbildung 5.32 ist die typische Kommunikation eines Client-Server-Systems abgebildet. Über eine Benutzeroberfläche wird eine Operation angefordert. Der Applikationsserver verarbeitet die Anfrage und setzt eine Abfrage auf die Datenbank ab. Das Ergebnis wird, über den gleichen Kommunikationsweg, zurück zur Benutzeroberfläche gegeben.

■ Broker-Architektur

Ein anderes Beispiel einer verteilten Architektur ist die sogenannte Broker-Architektur.

Beispiel: Request-Broker (Vermittler-Komponenten)

In Abbildung 5.33 wird gezeigt, wie die Kommunikation über einen steuernden Request-Broker geleitet wird. Der Request-Broker entscheidet darüber, welcher Server derzeit zur Beantwortung einer Anfrage zur Verfügung steht und leitet entsprechend die Anfragen zwischen Client und Server weiter.

Einige Vorteile der Architektur sind, dass die Serverseite skalierbar wird (z. B. Möglichkeit der Mengenskalierung von ganzen Serverfarmen). Die Clientseite kann dagegen einfach und wenig komplex ausgelegt sein. Die Proxy-Komponenten können in der Regel standardisiert werden und dienen rein der Kopplung über Netzwerke (d. h. Ortsunabhängigkeit wird hiermit erreicht).

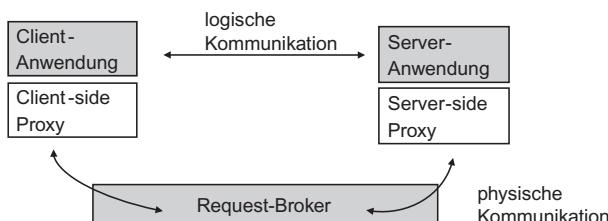


Abbildung 5.33 Typische Request-Broker-Architektur

Interaktive Systeme

Bei interaktiven Systemen handelt es sich um Systeme, in denen durch Kommunikation zwischen Objekten der Ablauf entschieden wird. Eine typische Architektur ist hier die „Modell-View-Controller“-Architektur.

- **Modell-View-Controller-Architektur**

Der Aufbau dieser Architektur ist dreigeteilt. Im sogenannten **Modell** befinden sich die Daten. Im **View** wird die Ausgabe der Daten auf Benutzeroberflächen geregelt. Die Steuerung aller Aufgaben übernimmt dabei der **Controller** (siehe Abbildung 5.34).

- **Presentation-Abstraction-Control-Architektur**

Dies ist im Wesentlichen die gleiche Architektur wie die MVC-Architektur. Die Aufteilung wird hier typischerweise in Präsentationslogik, Businesss-Logik und Datenpersistenzschicht bzw. Datenschicht vorgenommen (siehe auch Abbildung 5.34).

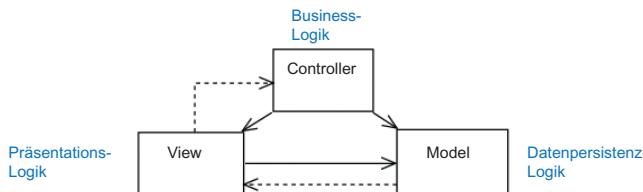


Abbildung 5.34 Modell-View-Controller-Architektur

5.4.2.3 Adaptive Systeme

Bei adaptiven Systemen handelt es sich um Architekturvorschläge, bei denen sich das Programm zur Laufzeit entscheidet, welche Bibliotheken letztlich verwendet werden. Deshalb nennt man diese Software **anpassungsfähig**. Typische Vertreter sind Mikrokernelsysteme oder Reflektion.

- **Mikrokernell**

Ein Mikrokernell ist ein Betriebssystemkern. Nur grundlegende Funktionen werden erfüllt, wie beispielsweise Speicherverwaltung, Prozessverwaltung, Synchronisation und Kommunikation. Alle weiteren Funktionen werden als eigene Prozesse (Server) implementiert. Diese kommunizieren mit nachfragenden Programmen (Clients).

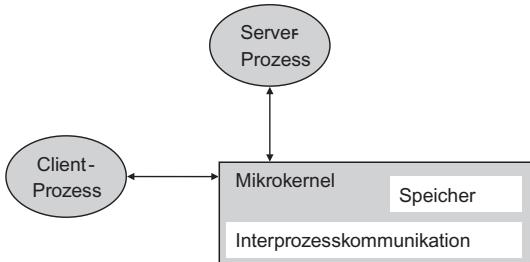


Abbildung 5.35 Mikrokern-Architektur

Manche Funktionen sind auch als Programmbibliothek implementiert. Diese werden von nachfragenden Programmen schließlich eingebunden.

- **Reflektion**

Zur Laufzeit (per Programm) wird nachgesehen, welche Funktionen eine Bibliotheksfunktion (engl. Assembly-Methode) anbietet.

5.4.2.4 Andere Architekturen

Einige weitere Architekturen seien hier noch ohne weitere Erklärungen aufgezählt. Die Liste hier könnte natürlich noch beliebig weitergeführt werden und zeigt die vielfältigen Möglichkeiten, die für Software-Architekten zur Verfügung stehen. Daher sollte ein lebenslanges Bestreben dieser Berufsgruppe die Studie unterschiedlichster Architekturen sein.

- Batch-Prozesse
- Interpreter
- Prozess-Kontrolle
- regelbasierte Systeme
- wissensbasierte Systeme
- künstlich-intelligente Systeme.

5.4.3 Mikroarchitekturen

Auf der Ebene der Mikroarchitekturen lohnt sich das Studium der vielfältig in der Literatur zu findenden **Entwurfsmuster** (engl.: **Design Pattern**). Ein erster gelungener Versuch der strukturierten Beschreibung und Kategorisierung gelang den Autoren des Buches „Entwurfsmuster“ der sogenannten *Gang of Four* [GHJV94].

Was sind Entwurfsmuster?



Definition

Entwurfsmuster beschreiben elementare Lösungsvorschläge zu Software-Problemen, die immer wieder in unseren Programmierumgebungen auftreten. Die Lösungsvorschläge sind dabei so aufbereitet, dass sie in vielen unterschiedlichen analogen Bereichen einsetzbar sind. Diese flexible Einsatzmöglichkeit macht gute Entwurfsmuster so wertvoll. Es handelt sich damit um „Best-Practice“-Vorschläge für Software-Design.

Das Studium möglichst vieler Entwurfsmuster ist daher Pflicht für jeden, der qualitativ hochwertige Software modellieren und schreiben möchte.

Ausgehend von dem sehr gut strukturierten Erstlingswerk der „Gang of Four“ übernehmen auch heutzutage neuere Entwurfsmustervorschläge die Aufteilung in deren Bestandteile und Kategorien, die deshalb nun kurz beschrieben wird.

Die fünf wichtigsten Bestandteile eines Entwurfsmusters

- **Name**
Name des Entwurfsmusters eindeutig und beschreibend. Somit wird ein höherer Abstraktionslevel erreicht
- **Problem**
Beschreibung des zu lösenden Problems und seines Kontexts; manchmal Aufzählung möglicher Designfehler
- **Lösung**
Elemente, die das Design ausmachen und deren Beziehungen; Verantwortlichkeiten und Kollaboration
- **Konsequenzen**
Zeit- und Platzabstimmungen; eventuell sprachabhängige und implementationsabhängige Bedenken.

Kategorien der Entwurfsmuster

Die Einordnung und die Sortierung der Entwurfsmuster erleichtern die Auswahl für den konkreten Einsatzfall. Es gibt **Erzeugungsmuster**, **Strukturmuster** und **Verhaltensmuster**. Wird beispielsweise ein objekt-basiertes Verhaltensmuster gesucht, so kann durch Tabellen, wie die in Abbildung 5.36, schnell ein passendes Entwurfsmuster gefunden werden. Diesem konkreten Beispiel kann entnommen werden, dass **objekt-basiert** bedeutet, dass zur Laufzeit der Software ein konkretes Objekt verwendet wird und **Verhaltensmuster** bedeutet, dass für das Objekt ein bestimmtes Verhalten modelliert werden soll.

		Aufgabe		
		Erzeugungs-muster	Strukturmuster	Verhaltensmuster
Gültigkeits- bereich	Klassen- basiert	Fabrikmethode	Adapter (klassenbasiert)	Interpreter Schablonenmethode
	Objekt- basiert	Abstrakte Fabrik Erbauer Prototyp Singelton (Einzelstück)	Adapter (objekt- basiert) Brücke Dekorierer Fassade Fliegengewicht Kompositum Proxy	Befehl Beobachter Besucher Iterator Memento Strategie Vermittler Zustand Zuständigkeitskette

Abbildung 5.36 Kategorien der originalen Design-Pattern aus [GHJV94]

Beispiel: Entwurfsmuster Beobachter (engl. Observer)

Der Beobachter wird zur Weitergabe von (Status-)Änderungen an einem Objekt, an von diesem Objekt abhängige Strukturen verwendet (siehe Abbildung 5.37).

Dabei wird ein aktueller Status des Subjekts an die Beobachter weitergegeben, ohne dass die Beobachter ständig nach einer Änderung des Status fragen müssen. An diesem Beispiel kann man gutes Software-Design studieren, da beispielsweise an horizontale und vertikale Teilung gedacht wurde. Dabei dient die **horizontale Teilung** in Subjekt und konkretes-Subjekt zur möglichen Integration mehrerer Subjektarten, während die **vertikale Teilung** in Subjekt und Beobachter die Unabhängigkeit zwischen den beiden Teilnehmern ermöglicht.

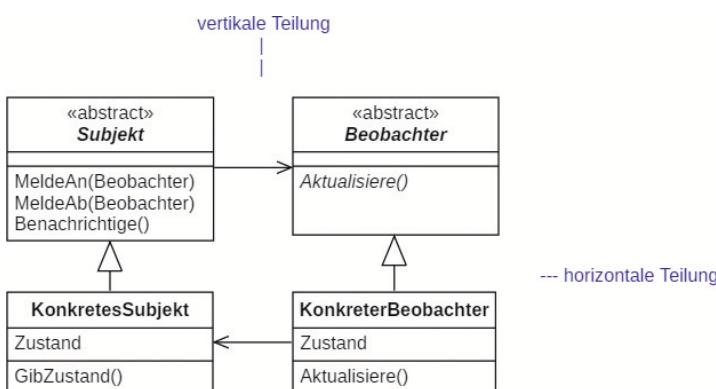


Abbildung 5.37 Entwurfsmuster: Beobachter

■ 5.5 Strategien und Methoden

In diesem Abschnitt werden unterschiedliche Vorgehensweisen zur Erstellung von Software-Design aufgeführt. Je nach Art der Software kann eine passende Methode für ein Projekt ausgesucht werden.

▪ **Funktionsorientiertes Design**

Hierbei handelt es sich um eine klassische Methode, die Anwendung findet in eher **prozeduraler** Software (und daher nicht den objektorientierten Weg beschreibt).

Erstellungsschritte

– **Dekomposition**

Es findet eine Zerlegung der Aufgabe in Subsysteme, Komponenten und Module statt.

– Identifizieren der wichtigsten **Funktionen** zuerst

Durch die Aufgabenzerlegung lassen sich schließlich die benötigten Funktionen identifizieren.

Vorgehen

Verwendung findet hier häufig ein **Top-down-Vorgehen** zur Identifizierung der Architektur. Die Zerlegung in kleinere Portionen geschieht meist durch Bearbeiten und Verfeinern.

Methoden

In der Vergangenheit fanden Diagrammtypen wie die **Strukturierte Analyse** von Tom de Marco [Mar79] und die Vorläufer der Aktivitätsdiagramme, die **Strukturcharts**, Verwendung. Heute kann natürlich hierfür die UML mit Use-Cases und Aktivitätsdiagrammen verwendet werden.

▪ **Objektorientiertes Design**

Bei dieser Strategie der Gewinnung einer Softwarearchitektur steht die **Wiederverwendung von Programmcode** im Vordergrund. Dies geschieht, indem zu Anfang einfach die Objekte identifiziert werden. Eine einfache Regel der Interpretation von Spezifikationen dabei ist:

- **Nomen** sind die Objekte und werden oft zu Klassen
- **Verben** sind die Methoden der Klassen
- **Adjektive** sind die Attribute der Klassen.

Dabei sei angemerkt, dass es für diese Regel natürlich auch Ausnahmen gibt. Nicht alle genannten Nomen einer Spezifikation werden schließlich als Klassen modelliert, sondern beispielsweise nur jene, die weiter zur Bearbeitung benötigt werden.

Vorgehensmöglichkeiten

- Komponentenbasiertes Vorgehen
Komponenten, Vererbung und Polymorphismus identifizieren
- Datenorientiertes Vorgehen
Datenabstraktion und Datenmodellierung verwenden
- Zuständigkeitsorientiertes Vorgehen
Zuständigkeiten und Rollen identifizieren.

Erstellungsschritte

- Meta-Informationen sammeln, gewinnen und anwenden durch z. B. **Reflektion**
- Klassen identifizieren
- die Programmstruktur ermitteln durch das Aufstellen von Klassendiagrammen
- Verhaltensmodellierung durch Aktivitäts- und / oder Sequenzdiagramme.

Methoden

Verwendung von allen UML-Diagrammtypen

Datenstruktur-orientiertes Design

Zuerst wird bei dieser Strategie ein Datenbankmodell erstellt und anschließend die Programmstruktur herausgearbeitet.

Methode

Es werden Datenbankmodelle, sogenannte **Entity-Relationship-Diagramme** [Che76] und Ein-Ausgabe-orientierte Vorgangsdiagramme, sogenannte HIPO-Diagramme [Cor74] erstellt:

- der Software-Engineering-Spezialist beginnt zuerst, Inputs und Outputs zu beschreiben
- die Kontrollstruktur des Codes wird aufgrund der entstandenen Datenstrukturen entworfen
- häufig werden Heuristiken eingesetzt, beispielsweise wenn ein Versatz zwischen Eingabe- und Ausgabedaten vorkommt.

▪ Komponenten-orientiertes Design

Über die Aufstellung der Komponenten gelangt man zur System-Architektur.

Methode

Bei dieser Methode werden zuerst die UML-**Komponentendiagramme** erstellt. Sinnvoll ist dabei auch die Schachtelung von Komponenten und damit die Zerlegung in Sub-Systeme und Komponenten (siehe Kapitel 5.4.1):

- eine Komponente ist eine unabhängige Einheit eines Software-Artefakts
- Schnittstellen werden zwischen den Komponenten definiert
- Abhängigkeiten zwischen den Komponenten werden abgeschätzt
- wichtig ist, die Unabhängigkeit der Komponenten zu fördern, falls möglich
- schrittweises Vorgehen ist das Anbieten, Entwickeln und Integrieren von Komponenten
- dabei ist die Wiederverwendung von Komponenten zu stärken, wo möglich.

▪ Andere Methoden

Es gibt eine Vielzahl weiterer Methoden für die Erstellung von Software-Design. Einige wichtige seien hier genannt. Der interessierte Leser sei hier auf die Literatur verwiesen. Die angegebene Liste legt natürlich keinen Wert auf Vollständigkeit:

- formale Methoden [KKB18]
- Transformationsmethoden
- Transaktionsmethoden [WV01].

■ 5.6 Software-Wiederverwendung

Der englische Begriff für Wiederverwendung ist „**reuse**“ und wird häufig auch im deutschen Sprachgebrauch in diesem Kontext so verwendet.

Heutzutage können gewinnbringende Projekte oft nur durch Nutzung wiederverwendbarer Software kostengünstig, zeit- und ressourcensparend erstellt werden. Dabei spielt die Kenntnis über und die geschickte Nutzung von vorhandenen Software-Bibliotheken eine entscheidende Rolle. Zwei in der Praxis übliche Techniken werden stellvertretend in diesem Kapitel nun vorgestellt.



Merke

Die **Wiederverwendung** von Software-Artefakten spielt beim Software-Engineering eine entscheidende Rolle zur Ressourcenschonung und zur Zeit- und Kostenersparnis.

▪ Produktlinien

Die Erfindung der Produktlinien [GB04] stammt ursprünglich von einer Firma in England, die Motoren für Traktoren herstellte. Um nicht für jede neue Traktor-Variante einen komplett neuen Motor herzustellen, entschied sich die Firma für die Aufstellung von sogenannten Produktlinien. Damit konnte erreicht werden, dass die Motoren grundsätzlich gleich aufgebaut waren und sich nur durch spezielle Details unterschieden (z. B. höhere Drehzahl und dadurch mehr Leistung etc.). Ein weiteres bekanntes Beispiel, diesmal aus dem Hardware- und Softwarebereich, sind die Serien für Smartphones, die sich nur durch Abkürzungen im Namen unterscheiden (z. B. „Samsung S8“ und „Samsung S8 Plus“). Dieses erfolgreiche Konzept wird mittlerweile häufig auch bei Software eingesetzt.

Vorgehensweise (siehe Abbildung 5.38)

- Identifizieren von gemeinsamen Eigenschaften von Teilnehmern und Gründung einer Produktfamilie.
- Grundlagen-Entwickler erstellen die Basis-Architektur und tauschen sich mit Produkt-Entwicklern aus. Manager steuern dabei die Ziele der Basis-Architektur.
- Produkt-Entwickler entwickeln die spezialisierten Architekturen für die Software-Varianten. Sie geben Feedback an Grundlagen-Entwickler über gewünschte Veränderungen in der Basis-Architektur weiter. Manager steuern, welche Produktlinien als Software-Varianten erstellt werden sollen.
- Manager steuern die Ziele der Basis-Architektur und bestimmen die Software-Produktlinien, die erstellt werden sollen.

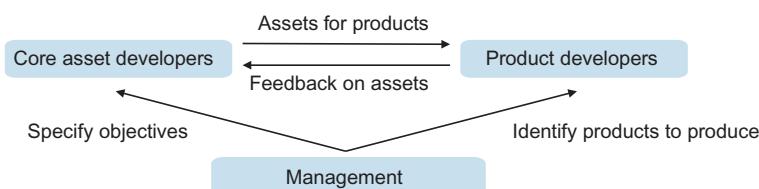


Abbildung 5.38 Produktlinien für Software [GB04]

▪ Frameworks

Kaum ein modernes Software-Projekt kommt heutzutage ohne Software-Bibliotheken aus. Der Fachbegriff hierfür ist englisch „Frameworks“, welcher mittlerweile auch im Deutschen gebraucht wird.

Es handelt sich hierbei um ein teilweise fertiges Software-System, das vom Software-Entwickler erweitert werden kann. Da Objektorientierung das derzeit vorherrschende Programmierparadigma ist, sind die modernen Frameworks häufig auch objektorientiert. Oft werden dabei auch spezielle Plug-ins instanziert.

Beispiel

Ein weitläufig, aktuell genutztes Framework ist das Microsoft .NET Framework [Mic18, 2018]:

- Common Language Infrastructure (CLI)
- Common Language Runtime (CLR) ist die Virtual Machine von .NET
- Client-PC benötigt zur Ausführung das installierte .NET Framework
- veröffentlichter Code unter Microsoft-Reference-License; Es ist jedoch nicht erlaubt, den Quellcode zu modifizieren.

Dabei sind Frameworks häufig in folgender hierarchischer Form in Unternehmen zu finden:

- .Framework
- ..Frameworkveränderungen für die Firma
- ... Projekt-Bibliothek.

Am Beispiel von .NET könnte eine Projektstruktur dann konkret so aussehen:

- .dotNet
- ..dotNet.„Firmenname“
- ... „Projektname“.

■ 5.7 Zusammenfassung

Der Begriff „Software-Design“ wird zweideutig verwendet. Aus Prozesssicht ist Software-Design die Lebenszyklus-Aktivität, in der Anforderungen an eine zu erstellende Software analysiert werden, um die interne Struktur einer Software produzieren zu können. Aus Ergebnissicht ist Software-Design die Beschreibung der Softwarearchitektur. Aufgaben, Aktivitäten, Ergebnisse und Beteiligte werden beschrieben.

Anhand eines durchgängigen Beispiels wird gezeigt, wie ein Software-Projekt von der Idee bis zur Realisierung entsteht.

Anschließend wird ein Überblick über die wichtigsten zugehörigen Notationen gegeben. Dabei erfolgt hauptsächlich eine Einführung in die UML (engl. Unified Modeling Language). Die Einteilung wird in Strukturdigramme und Verhaltensdiagramme vorgenommen. Erklärte Strukturdigramme sind Klassendiagramme und Komponentendiagramme. Wichtige Verhaltensdiagramme sind Struktogramme, Aktivitätsdiagramme, Sequenzdiagramme und Zustandsdiagramme.

Als Nächstes folgt die Diskussion einiger wichtigen Softwarearchitekturen. Dabei wird eine Einteilung in Makro- und Mikroarchitekturen vorgenommen. Zunächst wird die Zerlegung in Subsysteme und Komponenten diskutiert. Danach folgt eine Übersicht über wichtige Makroarchitekturen, wie beispielsweise Schichtenarchitekturen, verteilte System und dergleichen. Darauf folgt eine Übersicht über wichtige Mikroarchitekturen. Hier werden hauptsächlich Entwurfsmuster am Beispiel des „Beobachters“ behandelt.

Strategien und Methoden zur Erstellung von Software-Design sind funktionsorientiertes Design, objektorientiertes Design, datenstruktur-orientiertes Design und komponentenorientiertes Design.

Entscheidend für eine qualitativ hochwertige und wirtschaftlich rentable Software ist die Wiederverwendung von bereits bestehenden und somit getesteten Software-Artefakten. Zwei praxisrelevante Verfahren sind die Verwendung von sogenannten Produktlinien und die Verwendung von „Frameworks“. Beide Verfahren finden in der Praxis häufig Verwendung.

■ 5.8 Aufgabensammlung

An dieser Stelle findet sich eine Zusammenstellung von Fragen zur Software-Design-Phase.

1. Welche Diagrammtypen gibt es grundsätzlich, um Software-Design darzustellen?
2. Was ist der Unterschied zwischen Makroarchitektur und Mikroarchitektur?
3. Nennen Sie Beispiele, wann Sie Pipes oder Filter in einer Architektur vorschlagen würden.
4. Welche Aussage trifft auf die folgende Klasse zu?

Kunde
+name:String
+kontoNummer:int
+wohnort:String
+bestellen(artikel:int)
+bezahlen(betrag:float)
+kontoLoeschen()

- A) Die Klasse hat 6 Attribute.
- B) Es gibt eine Methode, die keine Parameter bekommt.
- C) Es gibt eine Methode, die als Parameter einen String-Wert bekommt.
- D) Es gibt ein Attribut vom Datentyp „float“.
5. Welche Aussage trifft auf eine einfache Assoziation in einem Klassendiagramm zu?
 - A) Eine einfache Assoziation ist gerichtet.
 - B) Eine einfache Assoziation beschreibt die Multiplizität zwischen zwei Klassen.
 - C) Eine einfache Assoziation beschreibt die Beziehung zwischen zwei Klassen.
 - D) Eine einfache Assoziation beschreibt eine Ganze-Teile-Beziehung zwischen zwei Klassen.
6. Wann verwendet man einen asynchronen Operationsaufruf im Sequenzdiagramm?
7. Wo findet man im Zustandsdiagramm Operationsaufrufe?
8. Programmieren Sie in einer objektorientierten Programmiersprache Ihrer Wahl das Beobachter-Entwurfsmuster nach.

6

Testphase – Verifikation und Validation



Merke

Ziel der Testphase ist es, **qualitativ hochwertige** Software unter betriebswirtschaftlich akzeptablen Rahmenbedingungen zu erhalten. Dafür sind **Softwaretests** (und Hardwaretests) erforderlich.

In diesem Kapitel wird daher über das Testen von Software eine kleine Übersicht gegeben. In Kapitel 6.1 werden die Grundlagen zum Thema Software-Tests eingeführt. Einige wichtige Software-Testverfahren findet man in Kapitel 6.2. Dazu gehören statische, dynamische und auch diversifizierende Testverfahren.

■ 6.1 Grundlagen

Grundsätzlich möchte man Software mit hoher Qualität herstellen. Aber wie definiert sich Software mit hoher Qualität? Und welche Messungen können vorgenommen werden, um Qualität zu beschreiben?

Diese Fragen sind schwierig und beschäftigen bis heute die Fachwelt. Aus diesem Grund wurde eine Norm erschaffen (siehe Abbildung 6.1).



Abbildung 6.1 ISO Norm 9126, Deutsches Institut für Normung e. V. (DIN), 1994,
vgl.[De94]

Nach der **Norm ISO 9126 (DIN 66272)** werden folgende Kriterien für die Qualität einer Software definiert:

- Funktionalität
 - Korrektheit, Angemessenheit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit
- Zuverlässigkeit
 - Reife, Fehlertoleranz, Wiederherstellbarkeit
- Benutzbarkeit
 - Verständlichkeit, Bedienbarkeit, Erlernbarkeit, Robustheit
- Effizienz
 - Wirtschaftlichkeit, Zeitverhalten, Verbrauchsverhalten
- Wartungsfreundlichkeit
 - Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit
- Übertragbarkeit
 - Anpassbarkeit, Installierbarkeit, Konformität, Austauschbarkeit.

Überraschend mag erscheinen, dass in dieser Norm nur Wortterme definiert sind. Es wird dagegen nicht definiert, wie die einzelnen Ziele erreicht werden können und es wird auch nicht beschrieben, wie eine Messung der Parameter vorgenommen werden kann. Entsprechende Software-Maße existieren jedoch teilweise bzw. wo immer möglich. Ein typisches Maß für die Zuverlässigkeit einer Software ist beispielsweise das Maß „**MTBF**“ (engl. Mean Time Before Failure), das mit Formel 6.1 berechnet wird.

Formel – MTBF (engl. Mean Time Before Failure)

$$\text{MTBF} = \frac{\text{gesamte Betriebszeit}}{\text{Anzahl der Ausfälle}} \quad (6.1)$$

Was ist Testen?

Beim Testen handelt es sich also um **Aktivitäten** im Software-Lebenszyklus eines Projekts.

Definition

Testen ist eine Aktivität, die ausgeführt wird, um die Software-Qualität zu evaluieren und zu verbessern, indem Defekte und Probleme identifiziert werden. Geprüft wird immer gegen **erwartetes Verhalten**, das in der **Anforderungsspezifikation** enthalten ist.

Warum ist Testen wichtig?

Testen ist wichtig, um komplexe (und oft enorm große) Software mit hoher Qualität herstellen zu können. Der grobe Ablauf eines Tests folgt im Wesentlichen dem Prozess aus Abbildung 6.2.

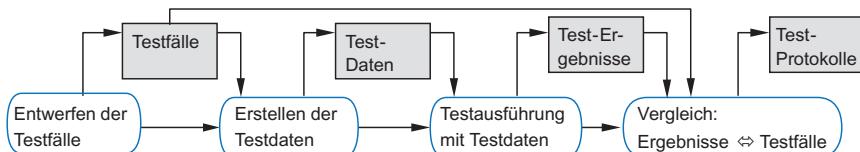


Abbildung 6.2 Testablauf aus Sommerville [Som18]

Die eckigen Kästchen sind entstehende Dokumente, während die runden Kästchen für Aktivitäten stehen. Zunächst werden Testfälle erzeugt und dokumentiert. Testfälle können in der Regel nicht ohne Testdaten durchgeführt werden, so dass der nächste Schritt die Erzeugung und Dokumentation der Testdaten ist. Beide Aktivitäten können in der Regel schon recht früh im Software-Lebenszyklus gestartet werden, nämlich schon in der Definitionsphase, wenn die Use-Cases und User-Stories entstehen. Gleichzeitig kann also bereits mit der Sammlung von Testfällen und Testdaten begonnen werden.

Erst wenn Testfälle und Testdaten bekannt sind, kann der Test mit diesen auch durchgeführt werden. Es ist wichtig, dabei das Testergebnis zu dokumentieren. In der Regel finden die Tests nach der Implementation des zugehörigen Software-Artefakts statt, also spätestens in bzw. nach der Implementationsphase.

Zuletzt werden die Testfälle mit den Testergebnissen verglichen. Das Ergebnis wird schließlich protokolliert und den Testern bzw. dem Software-Entwickler bei notwendigen Korrekturen übergeben.

Dokumentation und Protokollierung dienen neben der Steuerung des Korrekturprozesses auch der Feststellung von Software-Maßen wie beispielsweise der Fehlerquote eines Projekts.

Begriffe der Testphase

Es gibt in der Testphase zwei wichtige Begriffe, die unterschieden werden.



Merke

▪ **Verifikation**

Bei der Verifikation einer Software wird die Frage betrachtet, ob das entstandene Software-Artefakt auch seiner Anforderungsspezifikation entspricht,

(d. h. Pflichtenheft s. Kap. 4.2, bzw. Use-Cases s. Kap. 4.3.6.1 oder Items im Backlog bei agilem Vorgehensmodell s. Kap. 4.2). Da die Spezifikation maßgebend für die Abnahme einer Software ist, wird spätestens in diesem Prozessabschnitt klar, wie wichtig klare und gute Spezifikationen sind.

▪ **Validation**

Wird eine Software validiert, so untersucht man die Eignung eines Software-Artefakts bezogen auf seinen Einsatzzweck.

Im Fokus steht hier die Frage, ob die Software in der Realität auch qualitativ hochwertig die Aufgaben erfüllt, zu deren Zweck sie geschrieben wurde.

Aufgaben

Die Aufgaben, die in der Testphase zu bewältigen sind, sind je nach verwendetem Testverfahren hauptsächlich die nun folgenden. Bei einigen Testverfahren, wie zum Beispiel bei statischen oder auch automatisierten Tests, können manche der angegebenen Tätigkeiten aber auch wegfallen (z. B. werden Testfälle bei statischen Tests nicht benötigt und bei automatisierten Tests wird, bis auf das Implementieren der Tests, zur eigentlichen Testausführung kein Testpersonal benötigt).

▪ **Test planen**

Bevor ein Software-Artefakt getestet werden kann, ist festzulegen, was (genaue Definition des Umfangs und der Genauigkeit) und wie (d. h. mit welchen Methoden) getestet werden soll.

▪ **Test organisieren**

Testpersonal (und automatisierte Tests) müssen organisiert werden. Die Zeitplanung muss erstellt werden. Außerdem muss eventuell eine Testumgebung zur Verfügung gestellt werden. Testfälle und Testdaten müssen entworfen und bereitgestellt werden. Es ist wichtig, dass bei jedem Testfall ein zu erwartendes Ergebnis dokumentiert wird.

▪ **Test durchführen**

Das Testpersonal führt hier die bereitgestellten Testfälle zusammen mit den Testdaten aus. Wichtig ist, dass die Testergebnisse dokumentiert werden, damit die Tests bei Bedarf nachvollzogen und wiederholt werden können. Der Vergleich zwischen dem Testergebnis und dem zu erwartendem Ergebnis der Testfälle findet statt und wird idealerweise dokumentiert.

▪ **Letzte Kontrollinstanz vor Freigabe**

Das Ergebnisprotokoll des Vergleichs dient als Entscheidungsgrundlage für das weitere Vorgehen. Ist die Software qualitativ hochwertig genug, wird das Artefakt freigegeben. Andernfalls kann eine Nachbesserung notwendig werden.

Beteiligte

Hauptsächlich sind in der Testphase das Prüfpersonal (bewährt haben sich primär unabhängige Organisationen), die Software-Entwickler und die Software-Engineering-Spezialisten und -Architekten beteiligt.

■ 6.2 Software-Testverfahren

Es gibt mehrere Möglichkeiten, Testverfahren zu klassifizieren. Vorschläge in der Literatur unterscheiden die Testverfahren beispielsweise nach Prüftechnik, Einsatzzeitpunkt (z. B. Modultest, Integrationstest, Systemtest etc.) oder Systemart (Web-Test, Hardware-Test, Sicherheits-Test etc.) und vielem mehr. In diesem Buch wird daher exemplarisch eine brauchbare, verbreitete Klassifikation nach der Prüftechnik verwendet.



Merke

Unterschieden werden grundsätzlich sogenannte **statische** und **dynamische** Testverfahren. Zusätzlich sind noch die **diversifizierenden** Verfahren zu nennen, bei denen immer zwei Versionen einer Software gegeneinander getestet werden.

Auf alle drei Arten wird nun anhand jeweils eines Beispielverfahrens eingegangen. Es existieren jedoch weitaus mehr Verfahren. Eine geeignete Übersicht kann beispielsweise in [Lig09] gefunden werden.

6.2.1 Statische Testverfahren

Die statischen Testverfahren können nach [Lig09] in **verifizierende** und **analysierende** Verfahren eingeteilt werden. Verifizierende Testverfahren können wiederum **formale** Verfahren wie beispielsweise das „Zusicherungsverfahren“ sein oder **symbolische** Verfahren wie **algebraische Techniken** oder **automatenbasierte Techniken**.

Beispiele für analysierende Verfahren sind „Stilanalyse“, „Datenflussanomalieanalyse“ oder „Inspektions- und Review-Technik“. Reviews werden, als Beispiel statischer Verfahren und aufgrund deren Verbreitungsgrades in der Wirtschaft, nun im Anschluss erläutert.

Die wichtigsten Merkmale von statischen Testverfahren sind folgende. Es findet **keine Ausführung** der zu prüfenden Software statt. Testfälle werden dabei nicht gewählt. Vielmehr wird der **Programmcode analysiert**, wobei prinzipiell die statische Analyse ohne Computerunterstützung ausgeführt werden kann. Eine vollständige Aussage über die Korrektheit oder Zuverlässigkeit eines Software-Artefakts kann dabei jedoch nicht erzeugt werden.

Inspektions- und Reviewtechnik (analysierende Technik)

Reviews können in verschiedenen Varianten abgehalten werden. Die Wahl besteht zwischen einfachen Reviews in **Kommentartechnik**, in denen Software-Artefakte nur grob, schnell und unkompliziert untersucht werden. Der Vorteil liegt hier darin, dass Vorbereitung, Durchführung und Nachbereitung in kürzester Zeit stattfinden können.

Eine weitere Form sind sogenannte **informale Reviews**. Diese finden gründlich und zeitaufwendig statt, sind jedoch günstig durchzuführen, da nur wenige Formalitäten und Ergebnisdokumente gefordert werden.

Die aufwendigste Art dieses Verfahrens sind **formale Reviews**, welche sehr gründlich, zeitaufwendig und damit relativ teuer durchführbar sind. Obwohl Reviews und sogenannte Inspektionen sehr ähnlich sind, werden in der Literatur manchmal diese Begriffe unterschieden.

Reviews sind Testverfahren, die nach dem Prinzip der **externen Qualitätskontrolle** arbeiten. Wichtige Eigenschaften sind, dass es festgelegte Eingangs- und Ausgangskriterien und definierte Inspektionsphasen mit Zielvorgaben für Qualitätssicherung gibt. Teilnehmer übernehmen dabei bestimmte **Rollen** (z. B. Moderator, Reviewer

oder Projektleiter). Dabei werden **Inspektionsdaten** gesammelt und analysiert, einschließlich einer eventuellen **Rückkopplung** auf den Inspektionsprozess. Als Ergebnis entsteht ein Protokoll über erkannte Fehler oder Schwachstellen im Software-Artefakt. Ein Review durchläuft die in Abbildung 6.3 dargestellten Inspektionsphasen.

In der Planung geschieht die organisatorische Vorbereitung des Reviews. In der Überblicksveranstaltung werden die Informationen über das zu testende Software-Artefakt und die Rollen verteilt. In der Vorbereitungsphase bereitet sich jeder Teilnehmer getrennt voneinander, durch Studium der Informationen und des Software-Artefaktes, auf die eigentliche Inspektionssitzung vor. In Sitzungstechnik erfolgt danach die hauptsächliche Inspektionssitzung, in der die gefundenen Probleme und Fehler besprochen und dokumentiert werden. Während der Nacharbeitsphase werden gefundene Fehler der Software korrigiert. In der Follow-Up Sitzung werden die Fehlerkorrekturen überprüft und die endgültigen Inspektionsberichte angefertigt.

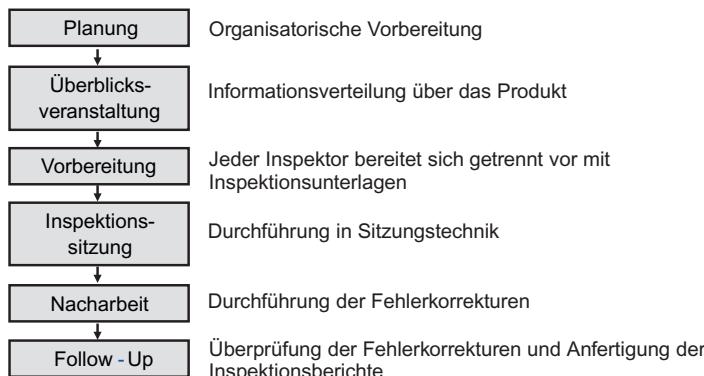


Abbildung 6.3 Insektionsphasen nach [Lig09]

Reviews werden hauptsächlich für semantische Prüfungen eingesetzt. Sie werden oft als Meilenstein am Ende jeder Software-Entwicklungsphase eingesetzt. Häufig werden Reviews in den frühen Phasen des Software-Lebenszyklus sehr genau durchgeführt.

Vorteilhaft an Reviews ist deren Effektivität in Bezug auf die Anzahl früh gefundener Fehler. Werden Fehler frühzeitig entdeckt, können in der Regel Kosten reduziert werden. Aussage über das Laufzeitverhalten des Software-Artefaktes können dagegen mit diesem Testverfahren nicht getroffen werden.

6.2.2 Dynamische Testverfahren

Bei dynamischen Testverfahren wird die übersetzte, ausführbare Software mit **konkreten Eingabewerten** versehen, ausgeführt und gegen **erwartetes Verhalten** begutachtet. Oft kann in der realen Betriebsumgebung getestet werden. Dynamische Testtechniken sind **Stichprobenverfahren**, bei denen normalerweise keine hundertprozentige **Testabdeckung** angestrebt wird. Sie können die Korrektheit der getesteten Software daher in der Regel nicht beweisen.

Ziel dieser Verfahren ist es, **Testfälle** zu erzeugen, die **repräsentativ, fehlersensitiv, redundanzarm** und **ökonomisch** sind.

Dynamische Testverfahren können nach [Lig09] in **strukturorientierte (White-Box-)** und **funktionsorientierte (Black-Box-)** Verfahren eingeteilt werden. Bei Ersteren sind außerdem **kontrollflussorientierte** und **datenflussorientierte Tests** unterscheidbar. Ein Beispiel eines kontrollflussorientierten Verfahrens ist ein sogenannter „Zweigüberdeckungstest“, der in Kapitel 6.2.2.1 beschrieben wird.

Ein Beispiel eines wichtigen funktionsorientierten Verfahrens ist ein sogenannter „Unit-Test“ (siehe Kapitel 6.2.2.2).

6.2.2.1 White-Box-Techniken

White-Box-Techniken (siehe Abbildung 6.4) sind, wie schon erwähnt, strukturorientierte Verfahren, deren Testvollständigkeit anhand der Abdeckung der Strukturelemente des Programmcodes gemessen wird. Die Korrektheit des Software-Artefakts wird mittels der Spezifikation beurteilt. Bei kontrollflussorientierten White-Box-Verfahren wird der Programmablauf, während bei datenflussorientierten Verfahren die Eingaben, die zugehörigen Ausgaben und der Fluss, den Daten im Programmablauf nehmen, getestet werden.

Ablauf

Zuerst liest und interpretiert ein Tester die Software-Spezifikation (siehe Abbildung 6.4). Dadurch gewinnt er eine Vorstellung der Spezifikation. Nun wendet er Testfälle und Testdaten auf ein Software-Artefakt (z. B. eine Funktion/Methode) an. Dabei hat er Einsicht in den Quellcode und kann nachvollziehen, welche Anweisungen des Quellcodes dabei durchlaufen werden und welche nicht. Anhand der Ausgaben des Programmes kann ein Tester schließlich beurteilen, ob diese zur Spezifikation passen (d. h. ob die Testkriterien erfüllt sind oder nicht). Das Ergebnis wird wiederum protokolliert.

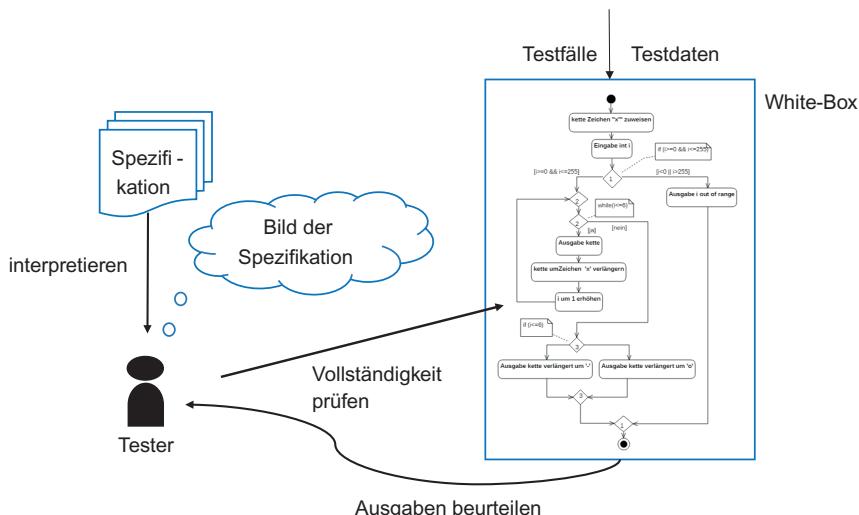


Abbildung 6.4 White-Box-Test nach [Lig09]

Ein Beispiel eines typischen White-Box-Tests ist der Zweigüberdeckungstest, der nun beschrieben wird.

Zweigüberdeckungstest (strukturorientierte, kontrollflussorientierte Technik)

Das Ziel des Zweigüberdeckungstests ist die Ausführung aller Zweige des Software-Artefakts. Der Zweigüberdeckungstest subsumiert den sogenannten *Anweisungsüberdeckungstest* (Test, bei dem das Ziel ist, alle Anweisungen zu durchlaufen). Die Aufgabe dabei ist, geeignete Testfälle mit definierten Eingabedaten zu finden, so dass alle Zweige des Prüflings (d. h. des zu prüfenden Software-Artefaktes) abgedeckt werden.

Es gibt für diesen Test das Testmaß $c_{\text{primitive}}$ (siehe Formel 6.2), das die **Test-Überdeckungsrate** abbildet. Maximalwert für $c_{\text{primitive}}$ ist 1 (d. h. 100 % Überdeckungsrate). Je größer der Wert, desto vollständiger ist das Software-Artefakt mit diesem Test abgedeckt.

Formel

$$c_{\text{primitive}} = \frac{\text{Anzahl ausgeführter primitiver Zweige}}{\text{Anzahl primitive Zweige}} \quad (6.2)$$

▪ **Beispiel**

In Abbildung 6.5 ist neben dem Programmcode das wohlgeformte Aktivitätsdiagramm der Methode *FunPainting2()* (siehe Kap.5.3.2.2) zu sehen.

```

1 void FunPainting2() {
2
3     String kette = "x";
4     int i;
5     cin >> "Eingabe i:"; i;
6     if ( i>=0 && i<=255 ) //if 1
7         while (i<=6) //while 2
8     {
9         cout << kette;
10        kette = kette + 'x';
11        i=j+1;
12    } //end while 2
13    if (i<=6) { //if 3
14        cout << kette + '.';
15    }
16    else {
17        cout << kette + 'o';
18    } //end if 3
19 }
20 else {
21     cout >> "i out of range (0255)"
22 } //end if 1
23 } //end function

```

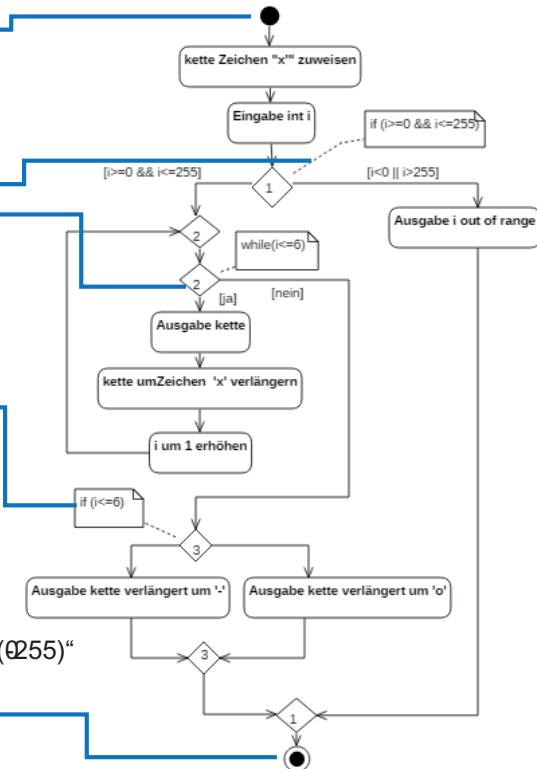


Abbildung 6.5 Zweigüberdeckungstest der Methode *FunPainting2()* nach [Lig09]

Nun müssen geeignete Testfälle und Testdaten gefunden werden, damit alle Zweige der Methode mindestens einmal durchlaufen werden. In Tabelle 6.1 sind die Eingaben (d. h. Testdaten) angegeben und der damit durchlaufene Pfad aus Abbildung 6.5 zu sehen.

Damit im Beispiel eine Überdeckungsrate von 100 % erreicht wird, müssen insgesamt 4 Testfälle aufgestellt werden. So wäre die Rechnung: $c_{\text{primitive}} = 4/4 = 1$. Da jedoch im Beispiel Fallnr.2 und Fallnr.3 mit den gleichen Testdaten erfolgen würde, kann in diesem speziellen Fall der zweite Testfall entfallen. Eine Reduktion der Testfälle ist stets anzustreben, um die Tests effizient und möglichst wirtschaftlich durchzuführen.

Tabelle 6.1 Zweigüberdeckungstest Beispiel – Testfälle und Testdaten

Fallnr.	Eingabe von i	Durchlaufener Pfad (Zeilennummer)
1	256	(1,2,3,4,5,6,20,21,22,23)
2	3	wie Fallnr.3, d.h. Testfall kann eingespart werden!
3	3	(1,2,3,4,5,6,{7,8,9,10,11,12},13,14,15,18,19,22,23)
4	8	(1,2,3,4,5,6,7,12,13,16,17,18,19,22,23)

- **Einsatz**

Der Zweigüberdeckungstest ist weit verbreitet und oft Standard in Modultestwerkzeugen.

- **Vorteile**

Ein Aufspüren von nicht ausführbaren Programmzweigen ist mit diesem Testverfahren möglich.

- **Nachteile**

Das Testverfahren ist ungeeignet für den Test von zusammengesetzten Entscheidungen und für den Test von Schleifen. Zu berücksichtigen ist außerdem der nichtlineare Zusammenhang zwischen Überdeckungsrate und Testfallanzahl. Die geeignete Auswahl an Testfällen gestaltet sich oft schwierig.

6.2.2.2 Black-Box-Techniken

Bei den Black-Box-Tests (s. Abb. 6.6) handelt es sich um **funktionsorientierte Verfahren**. Der Programmcode selbst wird dabei nicht betrachtet und bildet somit die „Black-Box“, in die man nicht hineinsehen kann. Es wird daher die **Spezifikation** in **Testfälle** umgesetzt.

Zuerst liest und interpretiert ein Tester die Software-Spezifikation. Dadurch gewinnt er eine Vorstellung der Spezifikation. Nun wendet er Testfälle und Testdaten auf ein Software-Artefakt (z. B. eine Funktion/Methode) an. Dabei hat er keine Einsicht in den Quellcode. Anhand der Ausgaben des Programmes kann ein Tester schließlich beurteilen, ob diese zur Spezifikation passen (d. h. ob die Testkriterien erfüllt sind oder nicht). Das Ergebnis wird auch bei diesem Verfahren wiederum protokolliert.

Die Testvollständigkeit wird anhand der **Abdeckung der Eingabewerte** und sogenannter **Äquivalenzklassen** (z. B. Aufteilung des möglichen Zahlenraumes eines Integer-Eingabewertes in alle positiven ganzen Zahlen, Null und alle negativen Zahlen, zwischen $[MININT; MAXINT]$) ermittelt. Die Korrektheit wird anhand der Spezifikation beurteilt. Da der Quellcode der Software nicht eingesehen wird, kann die Vollständigkeit der Abdeckung der Programmstruktur nicht garantiert werden.

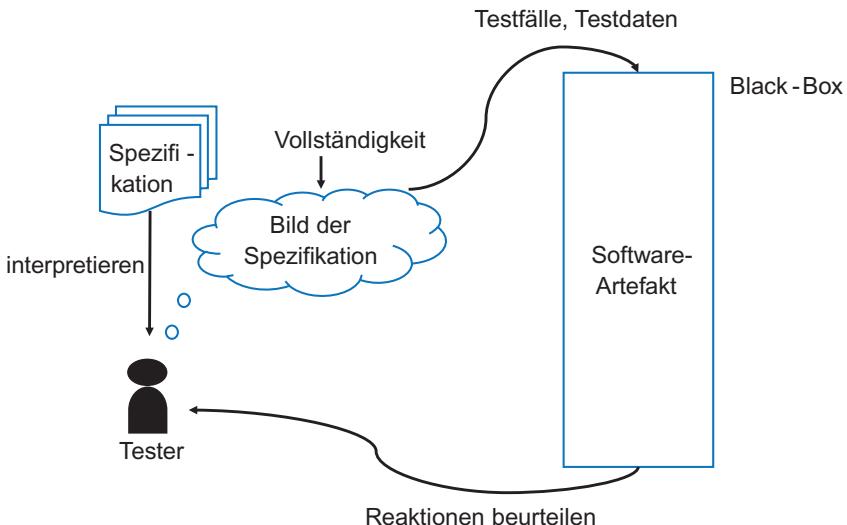


Abbildung 6.6 Black-Box-Test aus [Lig09]

Ein Beispiel eines Black-Box-Tests ist der **Unit-Test**, der nun beschrieben wird.

Unit-Test (funktionsorientierte Technik)

Ein Unit-Test ist ein Modultest, bei dem die Funktionalität einzelner Module oder Methoden getestet wird.

Unit-Tests können automatisiert werden; d. h. sie werden einmal (i. d. R. vom Entwickler) implementiert und stetig mithilfe von Testwerkzeugen (ohne menschliches Zutun) wiederholt. Zu jeder **Klasse** des zu prüfenden Software-Artefakts wird eine entsprechende **Testklasse** entwickelt. Es findet dabei die Prüfung eines sehr kleinen und autarken Teils der Software statt, z. B. der Test genau einer **Methode**. Ein Testfall ist in diesem Fall daher eine Methode, die mit **Testdaten** konfrontiert wird. Die laut Spezifikation zu erwartenden Ausgabewerte werden verglichen mit den Testergebnissen. Stimmt der erwartete Ausgabewert mit dem gelieferten Ergebnis der Funktion oder Methode überein, so gilt der Test als bestanden. Häufig findet hier die **Betrachtung von Grenzfällen** (z. B. sehr große/kleine Werte) und **besonderer Werte** (beispielsweise Null-Zeiger oder Objekte in speziellen Zuständen) statt.

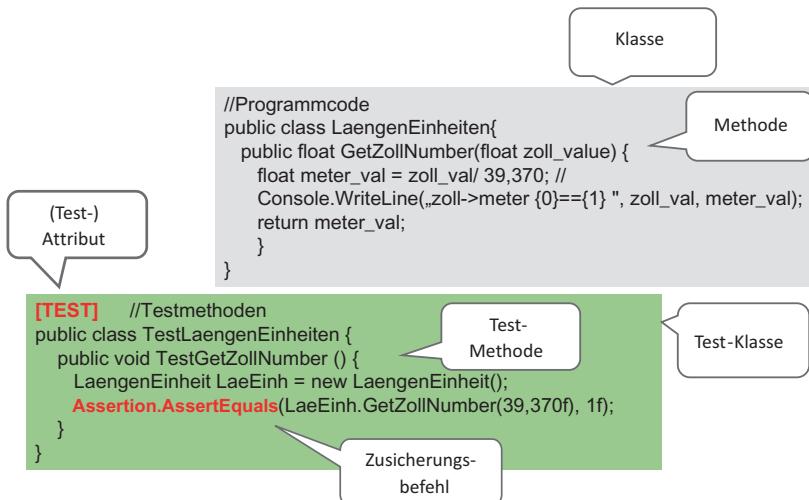


Abbildung 6.7 Beispiel Unit-Test in C#

▪ Beispiel

In Abbildung 6.7 ist ein typischer Unit-Test zu sehen, der aus seinem Prüfling (d. h. einer Klasse) und seiner zugehörigen Testklasse besteht. Durch sogenannte Zusicherungsbefehle können zu erwartende Ergebnisse mit Testergebnissen verglichen werden. Der Zusicherungsbefehl wird im Falle eines falschen Ergebnisses eine Meldung ausgeben. Im Beispiel handelt es sich um eine Methode, die einen Wert mit der Einheit Zoll in die Einheit Meter umrechnet.

▪ Einsatz

Der Unit-Test wird meist im Modultest eingesetzt. Am gebräuchlichsten ist seine Anwendung in objektorientierten Systemen.

▪ Vorteile

Es findet eine automatisierte stetige Wiederholung eines minimalen Modultests statt. Entwickler werden daher von zeitaufwendiger Testwiederholung befreit. Die Software-Entwickler werden angehalten, Methoden möglichst einfach zu halten und Objektstrukturen sauber zu entwerfen, um einfache Tests implementieren zu können. Schnell fallen unglücklich gewählte Schnittstellen auf. Komplexer Code wird erkannt und kann **refaktorisiert** (d. h. vereinfacht) werden. Bei Änderungen oder Erweiterungen bietet der Unit-Test somit auch eine schnelle Prüfung, ob bestehende Software noch fehlerfrei läuft. Dieses Verfahren zeigt daher den Weg auf zur sogenannten **testgetriebenen Entwicklung**.

- **Nachteile**

Es gibt keine wirklichen Nachteile für den Unit-Test. Die Prüfung von Benutzeroberflächen gelingt jedoch je nach Testtool weniger gut. Bibliotheksfunktionen sind damit nicht testbar. Auch Nebenläufigkeit ist mit diesem Verfahren schwierig zu testen.

6.2.3 Diversifizierende Tests

Bei einem diversifizierenden Test (siehe Abbildung 6.8) findet der **Vergleich der Testergebnisse** mit einer sogenannten **diversitären Software** statt; d. h. artgleiche, aber dennoch unterschiedliche Software, wie etwa ein Vergleich zur Vorversion. Es entfällt hier die aufwendige Bewertung der Korrektheit der Testergebnisse gegen die Spezifikation. Häufig ist eine Automatisierung durch Werkzeuge möglich. Ein Beispielverfahren ist ein **Regressionstest**.

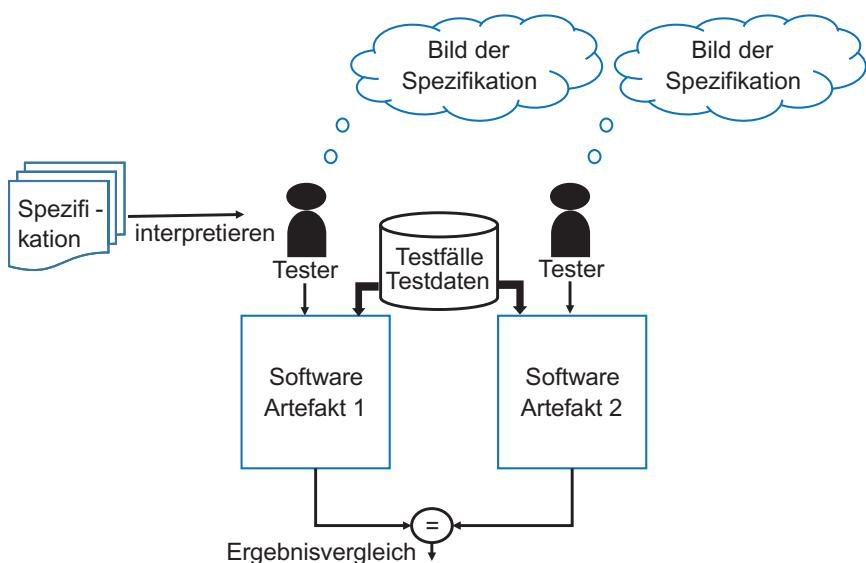


Abbildung 6.8 Schema eines diversifizierenden Testverfahrens aus [Lig09]

Regressionstest

Ein Regressionstest gehört zu den diversifizierenden Testverfahren (s. Abb. 6.9). Regressionstests bieten die Möglichkeit eines Nachweises, dass Modifikationen am Software-Artefakt keine unerwünschten Auswirkungen auf dessen Funktionalität besitzen. Es findet dabei die mehrmalige Ausführung einer Teilmenge der Testfälle statt. Die Ergebnisse werden mit der Spezifikation und meist mit der Vorgänger-Version der Software verglichen. Ein Werkzeug zur Ausführung eines automatisierten Tests ist hier sinnvoll.

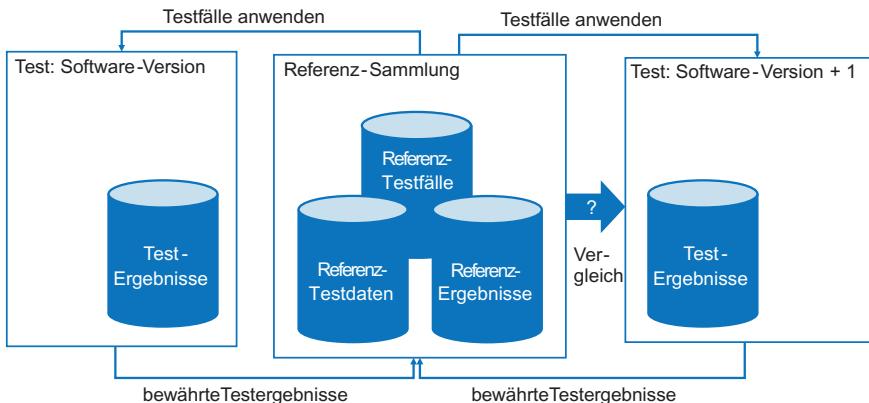


Abbildung 6.9 Schemabild eines Regressionstests nach [Lig09]

- **Einsatz**

Der Einsatz eines Regressionstests findet häufig in allen drei Testebenen statt, d. h. Modultest, Integrationstest und Systemtest.

- **Vorteile**

Der Vorteil eines Regressionstests ist, dass sich leicht Veränderungen zur Vorversion einer Software erkennen lassen. Daher ist der Test sehr praxisrelevant. Werkzeuge, die Regressionstests beinhalten, bieten oft auch Möglichkeiten zur **Lasterzeugung für Leistungs- und Stresstests**.

- **Nachteile**

Fehler, die in der Vorversion eines Software-Artefakts schon vorhanden waren, werden nicht sicher erkannt (Idealerweise sollten solche Fehler aber natürlich schon beim Test der Vorversion abgefangen werden).

■ 6.3 Zusammenfassung

In diesem Kapitel wird die Testphase des Software-Lebenszyklus erklärt. Ziel der Software-Entwicklung ist es ja, Software mit Qualität herzustellen. In der Norm ISO 9126 (DIN 66272) wird definiert, welche Eigenschaften qualitativ hochwertige Software besitzen soll. Die Messung der Eigenschaften ist häufig schwierig. Dennoch existieren einige etablierte Software-Maße. Um Software-Qualität überprüfen zu können, muss Software getestet werden. Unterschieden und beschrieben werden die Begriffe Verifikation und Validation einer Software.

Eine mögliche Einteilung von Software-Testverfahren ist deren Kategorisierung in statische, dynamische und diversifizierende Verfahren. Anhand jeweils eines Beispielverfahrens werden die Kategorien erklärt. Ein Beispiel für einen statischen Test sind Reviews. Dynamische Testverfahren werden in White-Box-Test und Black-Box-Test eingeteilt. Ein Beispiel eines White-Box-Testverfahrens ist der Zweigüberdeckungstest. Ein bekanntes Black-Box-Testverfahren ist ein sogenannter Unit-Test. Ein gebräuchliches diversifizierendes Testverfahren ist ein Regressionstest. Alle genannten Testverfahren werden kurz beschrieben und anhand von Beispielen erklärt.

■ 6.4 Aufgabensammlung

Mit den folgenden Aufgaben und Fragen können Sie ihr Wissen über die Testphase überprüfen.

1. Welche Merkmale enthält Qualitätssoftware? Nennen Sie jeweils ein passendes Testverfahren dazu.
2. Wie unterscheiden sich statische Testverfahren von dynamischen Testverfahren?
3. Welche Merkmale werden dem dynamischen Test zugeordnet?
(Wählen Sie eine oder mehrere Antworten:)
 - A. Kontrolliertes Ausführen der Software mit festgelegten Eingabedaten.
 - B. Die Software wird nicht ausgeführt.
 - C. Bestimmen von Testfällen.
 - D. Dynamische Tests dienen dem Prüfen von Quelltext.
4. Was ist ein Unit-Test?

5. Bei einem Zweigüberdeckungstest wird das Testmaß c-primitive gemessen. Welches Ergebnis wäre aus Qualitätsgründen das beste?
A)c-primitive = 5/6
B)c-primitive = 1/6
C)c-primitive = 4/6
D)c-primitive = 3/6
6. Welche Aussagen sind richtig?
A) Bei diversifizierenden Tests werden verschiedene Versionen einer Software gegeneinander getestet.
B) Tests planen, organisieren und durchführen sind Aufgaben der Abnahme- und Einführungsphase.
C) Bei testgetriebener Entwicklung wird zuerst der Softwarecode geschrieben und danach die Tests.
D) MTBF = Anzahl der Ausfälle / gesamte Betriebszeit.

7

Wartungsphase

Die letzte Phase des Software-Lebenszyklus ist die Wartung. Alle Aktivitäten in der Wartung zielen darauf ab, die nun eingeführte Software in Betrieb zu halten. Fehlermeldungen werden gesammelt und gegebenenfalls korrigiert. Meist wird Unterstützung für die Anwender geboten. Wird für eine Software die Wartung eingestellt, so wird sie nach einiger Zeit, wegen Mangel an Unterstützung bzw. der Nichtbeseitigung von Fehlern, außer Betrieb gehen.

In diesem Kapitel werden Grundlagen der Wartungsphase zusammengefasst (siehe Kapitel 7.1). Der Wartungsprozess wird in Kapitel 7.2 erklärt. Verschiedene Begriffe und Wartungstechniken werden in Kapitel 7.3 diskutiert.

■ 7.1 Grundlagen

Wartung ist eine Phase im Lebenszyklus eines Software-Produkts. Änderungsanfragen werden dokumentiert und verfolgt. Der Einfluss von geplanten Software-Änderungen wird bestimmt. Programmcode und auch die Spezifikation wird modifiziert, getestet und gezielt freigegeben. Das Training und täglicher Support für Anwender wird in dieser Phase geleistet.

Ziel in der Wartungsphase

In der Wartungsphase wird bereits existierende Software modifiziert. Dabei soll jedoch die Integrität des Software-Produkts erhalten bleiben (d. h. dass Änderungen an der Software nicht zu einer neuen Versionsnummer derselben führen sollen).



Definition

Was ist Wartung?

- Es handelt sich um Maßnahmen und Dienstleistungen zum Erhalt der Verwendbarkeit und Betriebssicherheit einer Software.
- Die Phase im Lebenszyklus eines Software-Produkts wird ebenfalls Wartung genannt.
- Alle Aktivitäten, um Software kosteneffektiv am Leben zu erhalten und dafür Support zu leisten, nennt man Wartung.

„Maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.“

(Standard IEEE 1219 [[IEE98](#)])

Beteiligte

Beteiligt am Wartungsprozess sind vornehmlich folgende Personen und Rollen:

- **Software-Entwickler:**

Die Aufgabe ist die Wartung des Programmcodes und der Programmcode-Dokumentation. Dazu gehören Sachverhalte wie Fehlersuche, Fehlerbehebung, Leistungskorrekturen und vieles mehr.

- **Software-Engineering-Spezialist:**

Anpassungen und Änderungen am Konzept und der Modellierung der Software sind Hauptaufgaben. Oft wird hier auch die Anpassung der Projektdokumentation und Handbücher angesiedelt.

- **Support-Mitarbeiter:**

Projektextern oder -intern angesiedelte Sammelstelle für Anwenderfragen und Kundenwünsche. Die Support-Mitarbeiter werden oft durch sogenannte „Help-Desk“-Software (auch „Ticketing-Systeme“ oder „Issue-Tracking-Systeme“) unterstützt. Ein Beispiel derartiger Software ist Bugzilla [[MF19](#)].

- **1-Level-Support:**

Aufnahme und Klärung von Anwenderfragen. Wird oft von Mitarbeitern durchgeführt, die zwar in der Anwendung geschult werden, jedoch meist keinen Einblick in das Innenleben der Software haben.

- **2-Level-Support:**

Wird meist durch Anwendungsexperten durchgeführt. Diese haben in der Regel Einblick sowohl in die Anwendung als auch in die Entwicklung selbst. Sie nehmen den Software-Entwicklern die leicht zu lösenden Problemfälle ab.

- **3-Level-Support:**

Dahinter stecken oft schon die Software-Entwickler selbst. Alle Probleme, die durch 1-Level- und 2-Level-Support nicht lösbar waren, landen zur Behandlung hier.

- **Projektmanager:**

Planung, Leitung, Steuerung und Überwachung der Wartungstätigkeiten und Mitarbeiter.

Änderungsanforderungsprozess

In der Wartungsphase ändern sich oft einige der Anforderungen an die Software. Dies geschieht einerseits, weil sich durch die Nutzung der Software der reale Arbeitsprozess der Anwender ändert und dadurch neue Begehrlichkeiten bezüglich des Leistungsumfangs der Software entstehen. Andererseits fallen durch den Einsatz der Software nun auch deren Fehler und Schwächen auf, so dass der Wunsch nach Korrektur von Anforderungen entsteht. Damit die Menge und Art der Änderungswünsche im Software-Projekt gut beherrschbar bleiben, werden häufig Änderungsanforderungsprozesse wie in Abbildung 7.1 eingeführt.

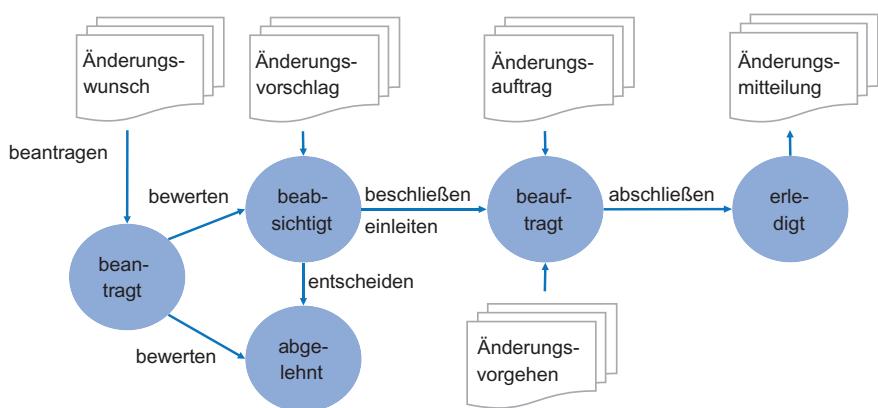


Abbildung 7.1 Änderungsanforderungsprozess nach [Bal09, Band 1]

Zu Beginn des Prozesses wird eine Änderungsanforderung schriftlich formuliert. Deinen Status wird auf „beantragt“ gesetzt. Diese wird dann begutachtet und falls sie für relevant genug erachtet wird, ändert sich deren Status auf „beabsichtigt“. Andernfalls wird sie als „abgelehnt“ markiert. Wird die Änderung beabsichtigt, wird die Anforderung, falls genügend Ressourcen, Zeit und finanzielle Mittel bereit stehen, beschlossen und eingeleitet. Der Status ändert sich damit auf „beauftragt“. Wurde die Änderungsanforderung schließlich realisiert, ändert sich der Status der zugehörigen

Dokumente auf „erledigt“. In Tabelle 7.1 ist als Beispiel ein typisches Änderungsformular zu finden.

Tabelle 7.1 Änderungsformular nach [Som18]

Änderungsantrag	Nummer: 114, Projekt: Sportpartnervermittlung
Priorität: Hoch	Antragsstatus: beauftragt
Kurzbeschreibung:	Alle Pflichtfelder der Formulare müssen mit Sternchen markiert werden. Pflichtfelder sind jeweils die Attribute <i>Name</i> , <i>Vorname</i> und <i>Email</i> .
Antragsteller: Anja Metzner	Antragsdatum: 1.10.2019
Betroffene Komponenten:	Benutzerregistrierung, Sportpartneranfrage-Formular
Damit verbundene Komponenten:	
Beurteilung der Änderung	Relativ einfach zu implementieren. Erfordert die Änderung des Attributes <i>required</i> pro betroffenem Eingabefeld. An den damit verbundenen Komponenten ist keine Änderung erforderlich.
Geschätzter Aufwand:	0,5 Personentage
Analyse: Tim Schuster	Analysedatum: 3.10.2019
Datum der Vorlage beim Änderungskontrollgremium: 2.10.2019	Datum der Entscheidung: 04.10.2019 Entscheidung: Implementierung in Version 2.1
Implementierer: Team (Verantwortlicher: Tim Schuster)	Änderungsdatum:
Voragedatum bei der QS:	Entscheidung der QS:
Voragedatum Konfigurationsmanagement:	
Kommentare:	

Der Prozessablauf zum Umgang mit dem Änderungsantrag ist der gleiche wie bei einer neuen Anforderung auch. Nach der Formulierung des Änderungsantrags schätzt ein Software-Entwickler den Aufwand. Sobald die Änderungsanforderung beauftragt ist, wird die Entwicklung eingeplant und schließlich realisiert, getestet und eingeführt.

Aufgaben des Wartungspersonals

Die Hauptaufgabe des Wartungspersonals ist das Beibehalten der Kontrolle über die gebräuchlichen Funktionen der Software. Außerdem muss die Kontrolle über Soft-

ware-Modifikationen erhalten werden. Existierende Funktionen sollen dabei falls möglich verbessert werden. Das Wartungspersonal sollte Vorsorge treffen, dass die Leistungsdaten der Software nicht unakzeptabel werden.

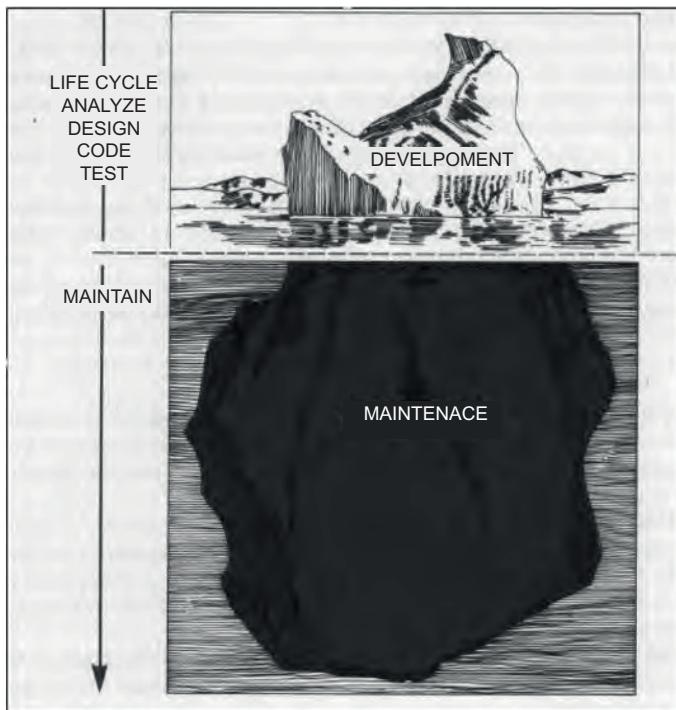


Abbildung 7.2 Der Wartungseisberg (aus [MM83], S.7)

Warum wird Wartung überhaupt benötigt?

Wartung stellt sicher, dass die Software immerzu den Anwenderwünschen des Kunden entspricht. Jeder, der sich mit Wartung beschäftigt, sollte wissen:



Merke

Software ist evolutionär, d. h. Anforderungen können sich über die Zeit ändern.

Software altert, d. h. wird Software im Laufe der Zeit nicht mehr verändert, so verliert sie den Zuspruch der Anwender und deren Verwendung läuft schließlich aus. Die Software geht dann zwangsläufig außer Betrieb.

Aufwand für die Wartung

Oft unterschätzen Projektverantwortliche den tatsächlichen Aufwand, der für die Wartungsphase anfällt. Der Abbildung mit dem Eisberg (s. Abb. 7.2) kann entnommen werden, dass ein Großteil des gesamten Projektaufwands alleine auf die Wartungsphase entfällt. Es handelt sich dabei sogar in der Realität um circa **80 % des Gesamtaufwandes**. Dem entgegen stehen oft die finanziellen Interessen und die Grenzen der zur Verfügung stehenden Ressourcen der Verantwortlichen, welche die Aufwände der Wartungsphase gerne so gut wie möglich reduzieren möchten.

Kategorien der Wartung (ISO/IEC 14764)

Wartung wird nach ISO-Norm in folgende Kategorien eingeteilt:

- **Korrigierende Wartung**
Reaktive Modifikation eines Software-Produkts nach Auslieferung, um beobachtete Probleme zu korrigieren.
- **Adaptive Wartung**
Modifikationen eines Software-Produkts nach Auslieferung, um Software in einer veränderten oder veränderlichen Umgebung am Leben zu erhalten.
- **Perfektive Wartung**
Modifikation eines Software-Produkts nach Auslieferung, um Leistungsfähigkeit oder Wartbarkeit zu verbessern.
- **Präventive Wartung**
Modifikation eines Software-Produkts nach Auslieferung, um latente Fehler zu entdecken und zu korrigieren, bevor diese Fehler zu effektiven Fehlern heranwachsen.

■ 7.2 Wartungsprozess

Wie bei der Ersterstellung einer Software, so wird auch bei ihrer Wartung ein iterativer, sich wiederholender Prozess stattfinden. Ein entsprechendes Prozessmodell ist in Abbildung 7.3 zu sehen. Änderungswünsche werden dabei kategorisiert, priorisiert und sukzessive eingearbeitet. Dabei wird die Software erweitert, verbessert und modernisiert. Wer genau hinsieht, kann erkennen, dass es sich bei dem Prozessmodell im Wesentlichen erneut um die Wiederholung des Software-Lebenszyklus handelt.

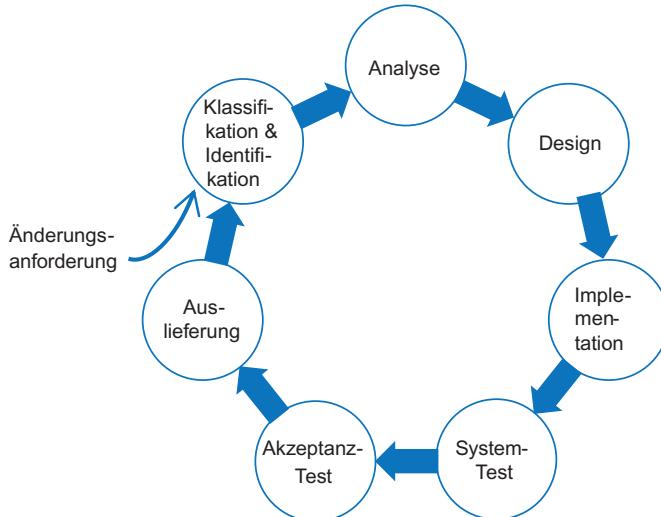


Abbildung 7.3 Wartungsprozessmodell nach [IEE98]

Wartungsspezifische Aktivitäten

Einige wichtige Fachbegriffe für die Aktivitäten in der Wartung sind die nun Folgenden.

- **Transition**

Sequenz von Aktivitäten, die kontrolliert und koordiniert zur Übergabe zwischen Software-Entwickler und Wartungspersonal stattfindet, falls es sich dabei um unterschiedliche Personengruppen handelt.

- **Änderungsanforderungsmanagement**

Änderungsanforderungen werden von Benutzern und Kunden gestellt und vom Wartungspersonal dokumentiert. Wie oben beschrieben, können Änderungsanforderungen vom Projektteam entweder akzeptiert oder abgelehnt werden. Änderungsanforderungen mit größerem Umfang können zur Begutachtung auch zu Entwicklern weitergeleitet werden.

- **Betreibung von Help-Desks und Software-Support**

Heutzutage meist online, wird ein Help-Desk betrieben, der den Anwender-Support leistet. Der Help-Desk ist in den meisten Fällen der Trigger für die Bewertung, Priorisierung und Ausführung von Änderungsanforderungen.

Der Anwender-Support bietet Hilfe und Rat für Anwender bezüglich Fragen rund um die Software. Außerdem bietet er Hilfe zur Erlangung von Sonderinformationen an (z. B. Ad-hoc-Reports).

- **Impact-Analyse**

Eine Impact-Analyse ist eine Untersuchung und deren Dokumentation, in der eine Begutachtung der Software bezüglich deren Einfluss auf den zu verbesserten Prozess stattfindet.

- **Erstellung und Einhaltung von Service-Level-Agreements**

Das Wartungspersonal stellt sogenannte SLAs (engl. Service-Level-Agreements) auf. Es handelt sich dabei um Vertragspunkte im Wartungsvertrag (z. B. wann und wie intensiv Wartung angeboten wird). Ein Beispiel ist ein Vertragspunkt einer Webseite, die 24 Stunden am Tag online ist und für die tagsüber 12 Stunden Support angeboten werden. Enthalten sind typischerweise auch die Verantwortlichkeitsbereiche des Wartungspersonals.

■ 7.3 Wartungstechniken

Während der Wartung sind Verfahren wie das **Re-Engineering** und das **Reverse-Engineering** gängig. Deshalb werden im Folgenden diese Begriffe erklärt.

7.3.1 Re-Engineering

Beim Re-Engineering findet keine Neuentwicklung wie in Abbildung 7.4 statt. Es handelt sich dabei um die Identifikation der Komponenten und deren Beziehungen einer Software. Es geht dabei um die (Teil-) Neuimplementierung von sogenannten Altsystemen (auch engl. legacy systems). Der grobe Prozessablauf ist in Abbildung 7.5 zu finden.

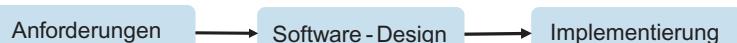


Abbildung 7.4 Software-Engineering

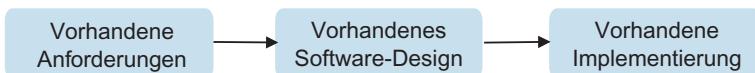


Abbildung 7.5 Re-Engineering

Typische **Ergebnisse** sind die Beschreibungen des Software-Systems, die Beschreibung deren Architektur oder auch die Transformation einer Repräsentation der Software in eine andere. Manchmal finden auch Untersuchungen und Änderungen (d. h.

Reverse-Engineering) der Software statt. Es kann auch eine Nachqualifizierung vorhandener Software erfolgen.

Häufige Aufgaben sind Plattformanpassungen, Änderung der Programmiersprache, oder die Einführung neuer Standards und Paradigmen.

Der Prozessablauf nach Sommerville ist in Abbildung 7.6 zu finden.

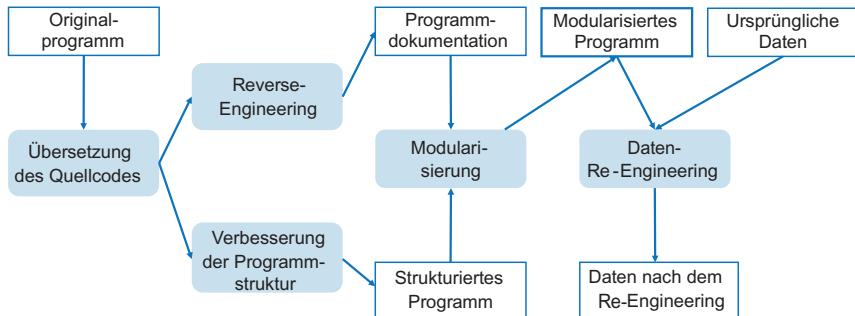


Abbildung 7.6 Re-Engineering [Som18, Sommerville, S.633]

Zusammengefasst lassen sich folgende Vor- und Nachteile des Re-Engineerings finden:

- **Vorteile**
 - Verbesserung der Wartungeigenschaften eines Software-Systems
 - Alternative zur kosteneffektiven Weiterentwicklung eines Altsystems
 - ökonomische Vorteile durch Einsatz bereits vorliegender, bewährter Produkte
 - Zuverlässigkeitsssteigerung der Software möglich
 - geringeres Risiko als bei Neuentwicklung
 - geringere Kosten im Vergleich mit Neuentwicklung.
- **Nachteile**
 - Neuentwicklung beinhaltet bessere, neuere Spezifikationen
 - Strukturierung und Modularisierung ist bei Neuentwicklungen i. A. verbessert
 - Neuentwicklung verwendet in der Regel moderne Techniken.

7.3.2 Reverse-Engineering

Reverse-Engineering (siehe Abbildung 7.7) darf nicht mit Re-Engineering verwechselt werden. Es handelt sich hierbei um völlig unterschiedliche Verfahren. Die Ziele des Reverse-Engineerings sind, Programmcode, Spezifikation oder Daten wiederzugewinnen oder zu aktualisieren. Altsysteme sollen dabei verstanden werden, um Veränderungen einzubauen zu können. Eine Wartungsvereinfachung ist dadurch möglich.

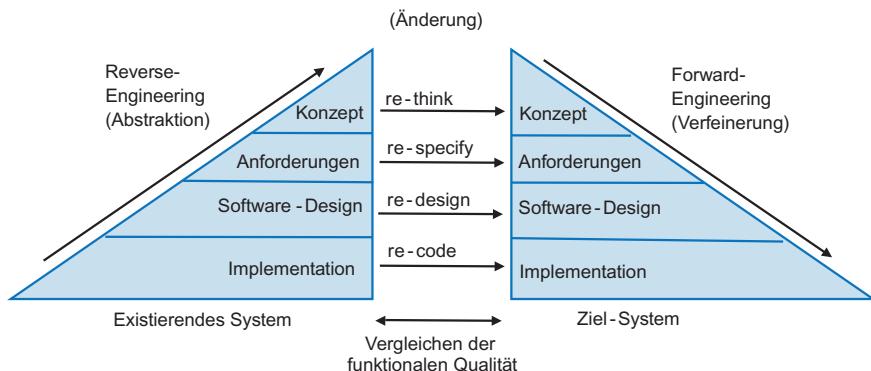


Abbildung 7.7 Reverse-Engineering [Byr92, pp. 226-235]

Der Prozessablauf nach Sommerville ist in Abbildung 7.8 zu sehen.

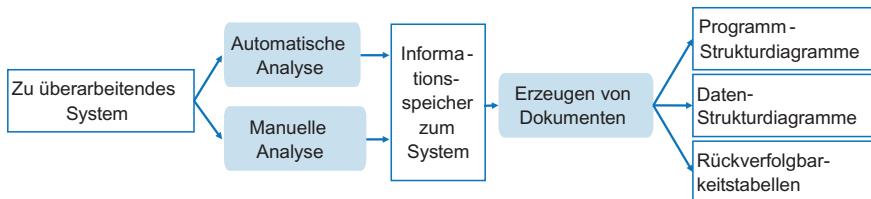


Abbildung 7.8 Re-Engineering [Som18]

Es gibt beim Reverse-Engineering auch unterstützende Werkzeuge. Ein Beispiel für objektorientiertem Code und dessen Spezifikationen ist etwa das CASE-Tool *erwin* [erw19], das beispielsweise Hilfestellung zur Konstruktion und Rekonstruktion der Datenmodellierung bietet.



Merk Re-Engineering ist **nicht gleich** Reverse-Engineering.
Auf Reverse-Engineering kann jedoch Re-Engineering folgen.

Beispiel:

Listing 7.1 zeigt einen älteren Ausschnitt eines Programmcodes, der modernisiert werden soll. Der verbesserte und lesbare Codeausschnitt ist in Listing 7.2 zu finden.

Listing 7.1 Unstrukturierter Programmcode

```

START: get(Switch)           /*Zeichen
       einlesen*/
       if Switch=off goto OFF /*Vergleich,
          Sprung*/
       if Switch=on goto ON
       goto CNTRLD
OFF:   status:='quit' /*String speichern*/
       save (status)
       goto START
ON:    status:='apply'
       save (status)
       goto START
CNTRLD: save (status) /*theoretisch status=NULL
                      */
END:   nop

```

Deutlich erkennbar ist, dass, neben der Nutzung der in der Programmierung verdeckten Sprünge, der Programmcode schwer lesbar ist.

Listing 7.2 Reverse-Engineering am Beispiel: strukturierte Modernisierung des Programmcodes aus Listing 7.1

```

loop
  status:='apply'; /*Initialisierung
                     status*/
  get (Switch);    /*Zeichen einlesen*/
  case Switch of   /*Mehrfachauswahl*/
    when ON => status:='quit';
    end when;
    when OFF=> status:='apply';
    end when;
  end case;
  save (status); /*status speichern*/
end loop;

```

Die Struktur und die Lesbarkeit wird in Listing 7.2 deutlich verbessert.

Anhand dieses kleinen, aber extremen Beispiels ist leicht zu erkennen, dass die Wartung nicht immer einfach durchzuführen ist. Je besser dokumentiert und modelliert eine Software ist, desto einfacher kann auch dazu Wartung stattfinden.

- Vielleicht inspirieren Sie ja die gezeigten Beispiele – wie mich – stets weiter an den eigenen Software-Engineering-Kenntnissen zu arbeiten! –*

■ 7.4 Zusammenfassung

In der Wartungsphase finden Maßnahmen und Dienstleistungen zum Erhalt der Verwendbarkeit und Betriebssicherheit einer Software statt. Zunächst werden die Grundlagen der wichtigsten Aktivitäten erklärt, um Software kosteneffektiv am Leben zu erhalten. Dazu gehören der Support und die Einarbeitung von Änderungsanfragen. Dafür wird ein Änderungsanforderungsprozess eingeführt und erklärt, dass der Zeitaufwand für Wartung im Schnitt etwa 80 % der Softwarelebenszeit beträgt und damit nicht zu unterschätzen ist. Nach ISO/IEC 14764 werden die unterschiedlichen Wartungsarten eingeführt.

Danach wird der wiederum iterative Wartungsprozess näher betrachtet. Für Änderungswünsche wird ein weiterer Entwicklungszyklus der Software fällig. Wartungsspezifische Aktivitäten wie Transition, Änderungsanforderungsmanagement und Support werden näher erläutert.

Eingesetzt werden hierfür verschiedene Wartungstechniken, die erklärt werden. Forward-Engineering, Re-Engineering und Reverse-Engineering sind wichtige Begrifflichkeiten, die zu unterscheiden sind.

■ 7.5 Aufgabensammlung

Zur letzten Phase im Lebenszyklus, der Wartungsphase, sind schließlich die nun folgenden Aufgaben zu lösen.

1. In welche Kategorien kann Wartung eingeteilt werden und wie unterscheiden sich diese Kategorien?
2. Warum bezeichnet man Software als evolutionär?
3. Was ist der Unterschied zwischen Re-Engineering und Reverse-Engineering?

4. In der agilen Softwareentwicklung gibt es Refactoring. Es handelt sich dabei um die Vereinfachung von kompliziertem Programmcode. Um welche Wartungstechnik handelt es sich dabei am wahrscheinlichsten?
- Engineering
 - Software-Engineering
 - Re-Engineering
 - Reverse-Engineering
5. In Listing 7.3 ist ein Ausschnitt eines web-basierten Systems gegeben. Es handelt sich dabei um die lauffähige, aber verbesserungswürdige Datei *Quadratzahl.html*, bei der eine positive, ganze Zahl über ein Formularfeld eingegeben und deren Quadratzahl ausgegeben wird. Nennen Sie drei Aspekte, die, aus Wartungssicht, an der Datei verbessert werden sollten.

Listing 7.3 Inhalt der Datei: *Quadratzahl.html*

```

1 <html>
2   <head>
3     <title>Quadratzahl-Berechnung</title>
4     <script type="text/javascript">
5       // Funktion quadratzahl(): Pruefung, ob in feld_x eine positive, ganze
6       // Zahl eingegeben wurde und Berechnung ihrer Quadratzahl.
7       function quadratzahl() {
8         //Abfrage ob Formularfeld feld_x leer ist
9         if ((isNaN (document.Formular.feld_x.value) || 
.           document.Formular.feld_x.value ==' ')) {
10           //Falls feld_x leer ist, Fehlermeldung anzeigen
11           label_x.innerHTML='Bitte eine positive, ganze Zahl eingeben';
12         else{ //Quadratzahl ausgeben
13           label_x.innerHTML = document.Formular.feld_x.value * 
.             document.Formular.feld_x.value; }
14         }; //Ende-Funktion
15       </script>
16     </head>
17     <body>
18       <form name="Formular">
19         <p><label id="label_qb">Zahl:</label></p>
20         <input type="text" id="feld_x"/>
21         <input type="button" id="btncalc" value="ok"
.           onclick="javascript:quadratzahl();"/>
22         <label id="label_x" style="color:red"></label>
23       </form>
24     </body>
25 </html>
```


A

Lösungen zur Aufgabensammlung

Hier finden Sie Antworten zu den Fragen und Aufgaben aus den einzelnen Kapiteln dieses Buches. **Weiteres Material** zum Üben und Vertiefen finden Sie auf meiner Homepage unter <https://www.hs-augsburg.de/Informatik/Anja-Metzner.html>

Unter anderem finden Sie dort beispielsweise auch noch mehr kurzweilige Kreuzworträtsel, einige Spiele wie UML-Memory und immer wieder neue gesammelte Übungsaufgaben aus den Vorlesungen zum Thema.

Lösungen zu Kapitel 1 – Einführung:

1. Was ist Software-Engineering und warum ist die Erstellung von qualitativ hochwertiger Software so schwer?
 - Software-Engineering ist eine Ingenieurstätigkeit zur Erstellung qualitativ hochwertiger Software.
 - Software ist immateriell und deren Modellbildung ist daher anspruchsvoll.
2. Wie heißen die Software-Lebenszyklus-Phasen?
 - Die Software-Lebenszyklus-Phasen sind Planungsphase, Definitionsphase, Software-Designphase, Implementationsphase, Abnahme-/Einführungsphase und Wartungsphase.
3. Nennen Sie einige wichtige und nützliche Verfahren, die dazu beitragen, Software mit Qualität zu produzieren
 - Beispiele sind Vorgehensmodelle, Requirements-Engineering, UML, Softwarearchitekturen, Testverfahren, Managementmethoden, Wartung und Weiterentwicklungsmethoden.

4. Kreuzworträtsel-Lösung

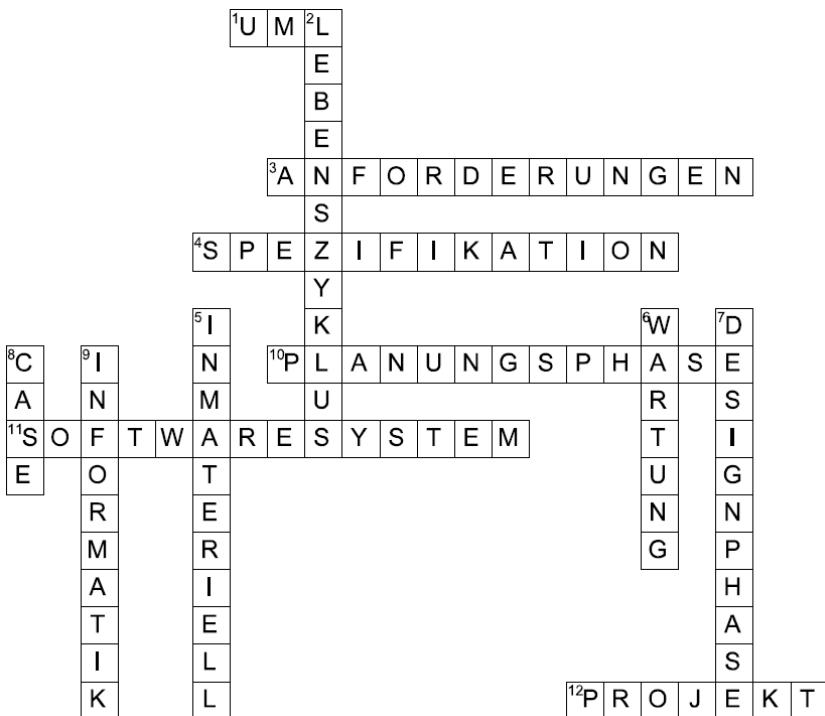


Abbildung A.1 Kreuzworträtsellösung Kapitel 1 – Aufgabe 4

■ **Horizontal:**

1. Grafische Modellierungssprache zur Spezifikation, Konstruktion und Dokumentation von Software
3. Was ist der Hauptgegenstand im Requirements-Engineering?
4. Wichtiges Ergebnis der Definitionsphase
10. Einheit im Softwarelebenszyklus
11. Ein System, dessen Subsysteme und Komponenten aus Software bestehen
12. Form, in der Softwareerstellung stattfindet.

■ **Vertikal:**

2. Was durchläuft jede Software?
5. Haupteigenschaft von Software
6. Einheit im Softwarelebenszyklus
7. Einheit im Softwarelebenszyklus
8. Abkürzung für Computerunterstützung im Software-Engineering
9. Grundlagenwissenschaft für Software-Engineering.

Lösungen zu Kapitel 2 – Phasenübergreifende Verfahren:

- Was beinhaltet ein Vorgehensmodell?
 - Ein Vorgehensmodell beinhaltet Aktivitäten, die den Software-Lebenszyklus darstellen. Das Vorgehensmodell gibt die Reihenfolge der Aktivitäten vor. Die Aktivitäten sind zu Phasen zusammengefasst.
- Welche Form der Software-Erstellung hat sich durchgesetzt und ist Erfolg versprechend?
 - Die passende Form ist ein Projekt. Es ist eingeteilt in verschiedene Phasen und untergliedert in Einzelaufgaben. Die Teilnehmer erhalten jeweils eine „Rolle“ (d. h. Beschreibung des Aufgabenbereiches einer Person).
- Welche Hauptfaktoren eines Softwareprojektes werden beim Projektmanagement betrachtet?
 - Hauptsächlich werden Zeit, Kosten und Ressourcen betrachtet.
- Was liegt auf dem kritischen Pfad eines Vorgangsknotennetzes? Welche Aussage ist richtig?
 - Die richtige Antwort ist D) – Vorgangsknoten, Meilensteine, Vorgänger-Nachfolger-Beziehungen
- Ein Vorgang hat folgende Werte: FA=0T, FE=5T, SA=3T, SE=8T. Welche Dauer und welche Gesamtpufferzeit hat der Vorgang?
 - Dauer=5T, Gesamtpufferzeit=3T
- Erstellen Sie einen Vorgangsknoten-Netzplan für die gegebene Aufgabentabelle. Zeichnen Sie den kritischen Pfad ein.

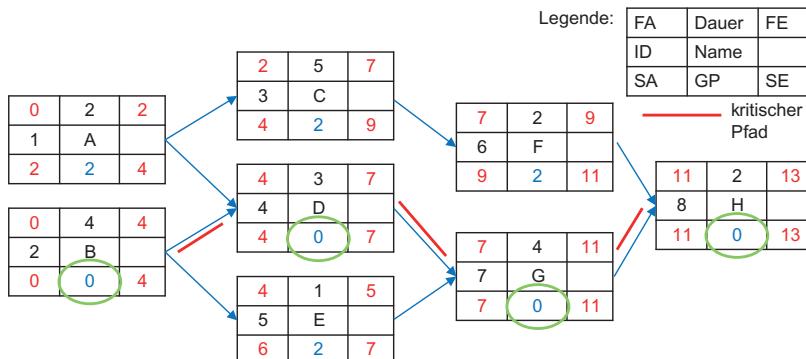


Abbildung A.2 Übung Projektmanagement

Lösungen zu Kapitel 3 – Planungsphase:

1. Was sind die Aufgaben und Ergebnisse der Planungsphase?
 - Die Aufgaben sind Voruntersuchung, Machbarkeitsstudie, Produktplanung, Projektplanung.
 - Die Ergebnisse sind Lastenheft, Studien, Risikoanalyse, Kosten-, Termin- und Resourcenplan.
2. Lasten- und Pflichtenheft: Welche Aussage stimmt?
 - Aussage A) stimmt: Das Lastenheft ist aus der Sicht des Auftraggebers geschrieben.
3. Welche Schätzverfahren gibt es für den Aufwand einer Software?
 - Es gibt beispielsweise das algorithmisches Aufwandsmodell, Expertenbeurteilung, Analogieschätzung, „Parkinsons-Gesetz“ und „Price-to-win“.
4. Eine Anforderung hat folgenden geschätzten Aufwand:
Ein Entwickler benötigt 5 Stunden, der Software-Architekt benötigt 12 Stunden und der Projektmanager 2 Stunden zur Erledigung der Anforderung.
Wie hoch sind die geschätzten Kosten der Anforderung, wenn für alle Beteiligten ein Stundensatz von 90 Euro angenommen werden kann?
 - $5h * 90\text{€} + 12h * 90\text{€} + 2h * 90\text{€} = 1710 \text{ Euro}$
5. Was sind Ziele des Risikomanagements bei der Softwareerstellung?
 - Ziele des Risikomanagements sind, Projekt-, Produkt- und Wirtschaftliche Risiken frühzeitig zu erkennen, zu dokumentieren und zu vermindern.
6. Welche Dokumente benötigt man beim Risikomanagement üblicherweise?
 - Antwort C) ist richtig: Notfallplan

Lösungen zu Kapitel 4 – Definitionsphase:

1. Was ist „Requirements-Engineering“?
 - Requirements-Engineering umfasst die Anforderungsanalyse und das Anforderungsmanagement mit ingeniermäßigem Vorgehen.
2. Welche Eigenschaften haben gute Anforderungen?
 - Eigenschaften guter Anforderungen sind: vollständig, korrekt, klassifizierbar, konsistent, prüfbar, eindeutig, verständlich, aktuell, realisierbar, „hat-Notwendigkeit“, verfolgbar, bewertbar.
3. Was ist eine Systemanforderung?
 - Antwort A) ist richtig: Anforderungen, die als genauere Beschreibung der Benutzeranforderungen dienen.
4. Wie findet man Anforderungen?
 - Man findet Anforderungen beispielsweise durch Kreativität, Beobachten (Vergangenheit oder aktuelles Vorgehen), Befragen und die Nutzung unterstützender Techniken.
5. Welche Beziehungsart in einem Use-Case-Diagramm würden Sie wählen, wenn der Use-Case „Rechnung drucken“ um eine Fehlermeldung ergänzt werden soll?
 - Die richtige Antwort ist D) – Extend-Beziehung mit Erweiterungspunkt.
6. Wie testet man Anforderungen?
 - Der Zeitpunkt ist nach der Definition der Anforderungen, aber minimal vor deren Implementation. Wichtige Schritte sind Prüfer identifizieren, Prüftechnik festlegen und Prüfung durchführen.

Lösungen zu Kapitel 5 – Designphase:

1. Welche Diagrammtypen gibt es grundsätzlich, um Software-Design darzustellen?
 - Grundsätzlich gibt es Strukturdiagramme und Verhaltensdiagramme. Strukturdiagramme beschäftigen sich mit der statischen Struktur eines Programmes (z. B. Aufbau und Art). Verhaltensdiagramme visualisieren den schrittweisen Ablauf in Programmen oder Prozessen.
2. Was ist der Unterschied zwischen Makroarchitektur und Mikroarchitektur?
 - Eine Makroarchitektur ist die Beschreibung der Software auf hohem Abstraktionsniveau und deren Organisation.
 - Bei der Mikroarchitektur handelt es sich um die Darstellung der Softwarestruktur und des Programmverhaltens.
3. Nennen Sie Beispiele, wann Sie Pipes oder Filter in einer Architektur vorschlagen würden.
 - Pipes helfen immer dann, wenn Daten durch mehrere Komponenten zu „schleusen“ sind. Auch zum Nachrichtenaustausch zwischen mehreren Komponenten können sie geeignet sein.
 - Filter sind angesagt, wenn Daten nach bestimmten Merkmalen zu „filtern“ bzw. auszusuchen sind.
4. Welche Aussage trifft auf die folgende Klasse zu?

Kunde
+name:String
+kontoNummer:int
+wohnort:String
+bestellen(artikel:int)
+bezahlen(betrag:float)
+kontoLoeschen()

Abbildung A.3 Übung Klasse – Kapitel5 – Aufgabe 4

- Richtig ist Antwort B) – Es gibt eine Methode, die keine Parameter bekommt.
5. Welche Aussage trifft auf eine einfache Assoziation in einem Klassendiagramm zu?
 - Richtig ist Antwort C) – Eine einfache Assoziation beschreibt die Beziehung zwischen zwei Klassen.
6. Wann verwendet man einen asynchronen Operationsaufruf im Sequenzdiagramm?
 - Man verwendet einen asynchronen Operationsaufruf, wenn von der Operation keine Rückantwort erwartet wird
7. Wo findet man im Zustandsdiagramm Operationsaufrufe?
 - Die Operationsaufrufe stehen auf den Transitionen
8. Programmieren Sie in einer objektorientierten Programmiersprache Ihrer Wahl das Beobachter-Entwurfsmuster nach.
In Listing A.1 ist eine konkrete Implementierung des Beobachter-Entwurfsmusters in der Programmiersprache C# zu finden. Es wurde hier auf (die so wichtige) Kommentierung des Codes verzichtet, um die pure Klassendiagrammstruktur nicht zu unterbrechen bzw. das Beispiel so kurz wie möglich zu halten.

Listing A.1 Minimale C#-Implementierung des Beobachter-Entwurfsmuster

```

class Programm
{
    static void Main(string[] args)
    {
        KonkretesSubject s = new
            KonkretesSubject();
        KonkreterBeobachter b1 = new
            KonkreterBeobachter();
        KonkreterBeobachter b2 = new
            KonkreterBeobachter();

        b1.Aktualisiere(s);
        b2.Aktualisiere(s);
        s.SetzeZustand(1);
        s.SetzeZustand(2);
        Console.ReadKey();
    }
}

public abstract class Subject
{
    public abstract void Benachrichtige()
    ;
    public abstract void MeldeAb(
        Beobachter b);
    public abstract void MeldeAn(
        Beobachter b);

}

public class KonkretesSubject:Subject
{
    private System.Collections.SortedList
        beobachterList = new System.
            Collections.SortedList();
    private int subjektZustand = 0;
    private int bCnt = 0;

    public override void Benachrichtige()
    {
}

```

```

        foreach (Beobachter b in
            beobachterList.GetValueList()
        )
    {
        b.Aktualisiere();
    }
}

public override void MeldeAb(
    Beobachter b)
{
    beobachterList.Remove(b);
    if (bCnt > 0) { bCnt--; }
}

public override void MeldeAn(
    Beobachter b)
{
    bCnt++;
    beobachterList.Add(bCnt, b);
}

public int GibZustand()
{
    return subjektZustand;
}

public void SetzeZustand(int neu)
{
    subjektZustand = neu;
    Benachrichtige();
}

public abstract class Beobachter
{
    public abstract void Aktualisiere();
    public abstract void Aktualisiere(
        KonkretesSubject s);
}

class KonkreterBeobachter:Beobachter
{
    protected KonkretesSubject Ksubjekt =
        null;
}

```

```

        private int beobachterZustand;

        public override void Aktualisiere(
            KonkretesSubject s) {
            if (this.Ksubjekt != null)
                this.Ksubjekt.MeldeAb(this);

            this.Ksubjekt = s;

            if (this.Ksubjekt != null)
                this.Ksubjekt.MeldeAn(this);
        }

        public override void Aktualisiere() {
            beobachterZustand=this.Ksubjekt.
                GibZustand();
            Console.WriteLine("Zustand: " +
                beobachterZustand);
        }
    }
}

```

Lösungen zu Kapitel 6 – Verifikation und Validation:

1. Welche Merkmale enthält Qualitätssoftware? Nennen Sie jeweils ein passendes Testverfahren dazu.
 - Funktionalität – Unit-Test
 - Zuverlässigkeit – „Mean Time Before Failure“
 - Benutzbarkeit – Regressionstest
 - Effizienz – O-Notation, Stresstest
 - Wartbarkeit – Regressionstest
 - Übertragbarkeit – Black-Box-Test.
2. Wie unterscheiden sich statische Testverfahren von dynamischen Testverfahren?
 - Statische Testverfahren: Keine Ausführung der zu prüfenden Software. Der Test findet eher durch verschiedene Arten der Begutachtung statt.
 - Dynamische Testverfahren: Übersetzte und ausführbare Software wird mit konkreten Eingabewerten versehen und ausgeführt. Anhand der Spezifikation wird entschieden, ob ein Test erfolgreich war.
3. Welche Merkmale werden dem dynamischen Test zugeordnet?
(Wählen Sie eine oder mehrere Antworten:)
 - Richtig ist Antwort A: Kontrolliertes Ausführen der Software mit festgelegten Eingabedaten.
 - Richtig ist Antwort C: Bestimmen von Testfällen.

4. Was ist ein Unit-Test?
 - Zu jeder Klasse wird eine entsprechende Testklasse implementiert.
 - Es erfolgt eine Prüfung eines sehr kleinen und autarken Programmteils, z. B. eine Methode oder Funktion.
 - Ein Testfall ist eine Methode oder eine Funktion, die mit Testdaten konfrontiert wird.
 - Häufig werden Grenzfälle betrachtet (z. B. Minus Unendlich, Null, Plus Unendlich).
5. Bei einem Zweigüberdeckungstest wird das Testmaß c-primitive gemessen. Welches Ergebnis wäre aus Qualitätsgründen das beste?
 - Antwort A) ist richtig: $c\text{-primitive} = 5/6$ (da 0,8333 den höchsten der gegebenen Werte darstellt).
6. Welche Aussagen sind richtig?
 - Richtig ist Antwort A): Bei diversifizierenden Tests werden verschiedene Versionen einer Software gegeneinander getestet.
 - Richtig ist Antwort B): Tests planen, organisieren und durchführen sind Aufgaben der Abnahme- und Einführungsphase.

Lösungen zu Kapitel 7 – Wartungsphase:

1. In welche Kategorien kann Wartung eingeteilt werden und wie unterscheiden sich diese Kategorien?
 - Korrigierende Wartung (Fehlerbehebung)
 - Adaptive Wartung (Plattformänderung)
 - Perfektive Wartung (Systemerweiterung)
 - Präventive Wartung (Zuverlässigkeitsssteigerung)
2. Warum bezeichnet man Software als evolutionär?
 - IT-Systeme müssen sich im Laufe ihres Lebenszyklus verändern, da Anwender bei nicht-funktionierender Software sonst andere Produkte nutzen oder da sich beispielsweise die Anforderungen der Benutzer ändern.
3. Was ist der Unterschied zwischen Re-Engineering und Reverse-Engineering?
 - Re-Engineering ist die Neuimplementierung eines vorhandenen Softwaresystems in unveränderter Weise.
 - Bei Reverse-Engineering werden Softwarecode, Spezifikation oder Daten wiedergewonnen, um Änderungen einzubauen.
4. In der agilen Softwareentwicklung gibt es Refactoring. Es handelt sich dabei um die Vereinfachung von kompliziertem Programmcode. Um welche Wartungstechnik handelt es sich dabei am wahrscheinlichsten?
 - C) Re-Engineering
(Begründung: Es handelt sich um eine Teil-Neuimplementierung eines Programmcode-Artefakts (z. B. Überarbeitung einer Methode, Funktion oder Klasse.))

5. In Listing 7.3 ist ein Ausschnitt eines web-basierten Systems gegeben. Es handelt sich dabei um die lauffähige, aber verbesserungswürdige Datei *Quadratzahl.html*, bei der eine positive, ganze Zahl über ein Formularfeld eingegeben und deren Quadratzahl ausgegeben wird. Nennen Sie drei Aspekte, die aus Wartungssicht an der Datei verbessert werden sollten.

Verbesserungsvorschläge:

- Sicherheitsaspekt – mangelnde Eingabevalidierung:
Alle Anfragen des Clients sind generell nicht vertrauenswürdig. Daher müssen alle Daten, die der Server vom Client entgegennimmt, auf Manipulation und bösartigen Input unbedingt überprüft werden. Ein Beispiel eines solchen Angriffs ist Cross-Site-Scripting (siehe [Apa19b])
- Architekturaspekt:
Zur besseren Übersicht und Trennung der einzelnen Seitenaspekte wäre zumindest eine einfache MVC-Architektur (siehe Kapitel 5.34) angemessen. Dabei wäre wenigstens eine erste einfache Trennung nach Model (CSS-Anteile, wie Schriftfarben, auslagern in .css-Datei), View (HTML-Anteile) und Controller (JavaScript-Anteile auslagern in .js-Datei) sinnvoll.
- Inhaltlicher Aspekt:
Die Funktion *quadratzahl()* soll überprüfen, ob in *feld_x* eine positive, ganze Zahl eingegeben wurde. Dabei wird derzeit nur eine leere Eingabe abgefangen. Es sollten daher mindestens Abfragen ergänzt werden, welche, sauber je erlaubtem Eingabedatentyp (hier: positive, ganze Zahl), einen gültigen Zahlenwert evaluiert.

Feedback

Feedback zur Fragensammlung, Ergänzungen und Ideen für Erweiterungsmöglichkeiten sind jederzeit willkommen!

– Anja Metzner –
(anja.metzner@hs-augsburg.de)

Literatur

- [AG83] ALBRECHT, A. J. ; GAFFNEY, J. E.: Software Function, Lines of Code and Development Effort Prediction: a Software Science Validation. In: *IEEE Trans. on Software Engineering, SE-9(6)* (1983), S. pp. 639–47
- [Alb79] ALBRECHT, A. J.: Measuring Application Development Productivity. In: *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, Monterey, California, IBM Corporation* (1979), S. pp. 83–92
- [Apa19a] APACHE: *Apache Webserver*. Webseite, 2019. – Online erhältlich unter <https://httpd.apache.org/>; abgerufen am 25.07.2019.
- [Apa19b] APACHE: *Cross Site Scripting Info*. Webseite, 2019. – Online erhältlich unter <https://httpd.apache.org/info/css-security/>; abgerufen am 21.11.2019.
- [AST03] ANDREW S. TANENBAUM, Maarten van S.: *Verteilte Systeme: Grundlagen und Paradigmen*. 1. Aufl. Pearson Studium, 2003. – ISBN 978–3827370570
- [Bal08] BALZERT, Helmut: *Lehrbuch der Softwaretechnik: Softwaremanagement*. 2. Aufl. Spektrum Akademischer Verlag, 2008. – ISBN 978–3827411617
- [Bal09] BALZERT, Helmut: *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. 3. Aufl. Spektrum Akademischer Verlag, 2009. – ISBN 978–3827417053
- [Bal11] BALZERT, Heide ; BALZERT, Helmut (Hrsg.): *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML*. 2. 2. Aufl. Spektrum Akademischer Verlag, 2011. – ISBN 978–3827429032
- [Bec03] BECK, Kent: *Extreme Programming. Das Manifest*. 1. Aufl. Addison-Wesley, 2003. – ISBN 978–3827321398
- [BGM⁺01] BECK, Kent ; GRENNING, James ; MARTIN, Robert C. u. a.: *Manifesto for Agile Software Development*. Webseite, 2001. – Online erhältlich unter <https://agilemanifesto.org/>; abgerufen am 07.11.2019.
- [Boe88] BOEHM, Barry W.: A Spiral Model of Software Development and Enhancement. In: *IEEE Computer. Vol. 21, Ausg. 5* (1988), S. 61–72
- [BR02] BROY, Manfred ; ROMBACH, H.Dieter: Software Engineering Wurzeln, Stand und Perspektiven. In: *Informatik Spektrum* 25 (2002), Nr. 6, 438–451.
<http://dx.doi.org/10.1007/s002870200266>. – DOI 10.1007/s002870200266

- [Bru19] BRUNS, Jürgen: *Projekt Libre - Projektmanagement Software*. Webseite, 2019. – Open Source, Online erhältlich unter <http://www.projectlibre.de/>; abgerufen am 13.09.2019.
- [BS02] BITTNER, K. ; SPENCE, I.: *Use Case Modeling*. 1. Aufl. Addison Wesley Professional, 2002. – ISBN 978-0-201-70913-1
- [BS19] BALSAMIQ-STUDIOS: *Balsamiq*. Webseite, 2019. – Online erhältlich unter <https://balsamiq.com/wireframes/>; abgerufen am 02.03.2019.
- [Byr92] BYRNE, E. J.: *A Conceptual Foundation for Software Re-Engineering*. 1992. – ICSM
- [Che76] CHEN, Peter: The Entity-Relationship Model - Towards a Unified View of Data. In: *ACM Transactions on Database Systems* 1 (1976), S. 9–36
- [Coo19] COOPERATION, Mozilla: *Firefox*. Webseite, 2019. – Online erhältlich unter <https://www.firefox.com/>; abgerufen am 25.07.2019.
- [Cor74] CORPORATION, IBM: HIPO - A Design Aid and Documentation Technique. In: *Publication Number GC20-1851* (1974)
- [De94] DIN-EV: *Deutsche Institut für Normung e.V. (DIN)*. Webseite, 1994. – Online erhältlich unter <https://www.din.de>; abgerufen am 14.10.2019.
- [DLWZ03] DUMKE, Reiner ; LOTHER, Mathias ; WILLE, Cornelius ; ZBROG, Fritz: *Web Engineering*. 1. Aufl. Pearson Studium, 2003. – ISBN 978-3827370808
- [erw19] ERWIN, Inc.: *erwin Data Modeler*. Webseite, 2019. – Online erhältlich unter <https://erwin.com/products/erwin-data-modeler/>; abgerufen am 13.09.2019.
- [Eur19] EUROPE, Sparkx Systems C.: *Enterprise Arhitect*. Webseite, 2019. – Online erhältlich unter <https://www.sparxsystems.de/>; abgerufen am 02.03.2019.
- [FB01] FOEGEN, Malte ; BATTFENFELD, Jörg: Die Rolle der Architektur in der Anwendungsentwicklung. In: *Informatik Spektrum* 24 (2001), Nr. 5, 290–301. <http://dx.doi.org/10.1007/s002870100183>. – DOI 10.1007/s002870100183
- [FM19] FERDINAND MALCHER, Danny K. Johannes Hoppe H. Johannes Hoppe: *Angular: Grundlagen, fortgeschrittene Themen und Best Practices*. 2. Aufl. dpunkt.verlag, 2019. – ISBN 978-3864906466
- [Foc19] FOCUS, Micro: *Caliber*. Webseite, 2019. – (ursprünglich von Borland und nun Firma Micro Focus), Online erhältlich unter <https://www.microfocus.com/en-us/products/requirements-management-caliber/overview>; abgerufen am 02.03.2019.
- [GB04] GÜNTER BÖCKLE, et.al.: *Software-Produktlinien: Methoden, Einführung und Praxis*. 1. Aufl. dpunkt, 2004. – ISBN 978-3898642576
- [GHJV94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software-The "Gang of Four"*. 1. Aufl. Prentice Hall, 1994. – ISBN 978-0201633610
- [HG08] HOLL, Alfred ; GRÜNAUER, Karin: *Business Process Modeling*. Växjö University (*Master thesis*). Webseite, 2008. – Online erhältlich unter <https://www.in.th-nuernberg.de/professors/holl/Personal/HollHome.htm>; abgerufen am 20.07.2019.
- [HHMS09] HINDEL, Bernd ; HÖRMANN, Klaus ; MÜLLER, Markus ; SCHMIED, Jürgen: *Basiswissen Software-Projektmanagement: Aus- und Weiterbildung zum Certified Professional for Project Management nach iSQI-Standard*. 3. Aufl. dpunkt Verlag, 2009. – ISBN 978-3898645614

- [HS99] HOLL, Alfred ; SCHOLZ, Michael: *Objektorientierung und Poppers Drei-Welten-Modell als Theoriekerne der Wirtschaftsinformatik*, in: Schütte, Reinhard et al. (Hrsg.): *Wirtschaftsinformatik und Wissenschaftstheorie. Grundpositionen und Theoriekerne. Arbeitsbericht 4 des Instituts für Produktion und industrielles Informationsmanagement*. Webseite, 1999. – Online erhältlich unter <https://www.in.th-nuernberg.de/professors/holl/Personal/HollHome.htm>; abgerufen am 20.07.2019.
- [HV04] HOLL, Alfred ; VALENTIN, Gregor: *Structured Business Process Modeling, Congress Paper for Information Systems Research in Scandinavia (IRIS 27)*, Falkenberg, Schweden 2004. Webseite, 2004. – Online erhältlich unter <https://www.in.th-nuernberg.de/professors/holl/Personal/HollHome.htm>; abgerufen am 20.07.2019.
- [HV07] HENNING, Peter A. ; VOGELSANG, Holger: *Taschenbuch Programmiersprachen*. 2. Aufl. Carl Hanser Verlag, 2007. – ISBN 978-3446407442
- [IBM19a] IBM: *Rational Doors Next Generation*. Webseite, 2019. – Online erhältlich unter <https://www.ibm.com/products/software>; abgerufen am 02.03.2019.
- [IBM19b] IBM: *Rational Rhapsody Developer*. Webseite, 2019. – Online erhältlich unter <https://www.ibm.com/products/software>; abgerufen am 02.03.2019.
- [IEE98] IEEE.ORG: *Standard IEEE 1219-98*. PDF, 1998. – Online erhältlich unter <https://standards.ieee.org/standard/1219-1998.html>; abgerufen am 02.10.2019.
- [IEE18] IEEE.ORG: *Standard Glossary of Software Engineering*. PDF, 2018. – Online erhältlich unter <https://ieeexplore.ieee.org/servlet/opac?punumber=2238>; abgerufen am 02.03.2019.
- [Int19] INTERNATIONAL, Visual P.: *Visual Paradigm Tools*. Webseite, 2019. – Online erhältlich unter <https://www.visual-paradigm.com/>; abgerufen am 02.03.2019.
- [Ker08] KERZNER, Harald: *Projektmanagement - ein systemorientierter Ansatz zur Planung und Steuerung*. 2. Aufl. MITP, 2008. – ISBN 978-3826616662
- [KKB18] KASTENS, Uwe ; KLEINE BÜNING, Hans: *Modellierung: Grundlagen und formale Methoden*. 4. Aufl. Carl Hanser Verlag, 2018. – ISBN 978-3446454644
- [Lig09] LIGGESMEYER, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. 2. Aufl. Spektrum Akademischer Verlag, 2009. – ISBN 978-3827420565
- [LK18] LEOPOLD, Klaus ; KALTENECKER, Siegfried: *Kanban in der IT. Eine Kultur der kontinuierlichen Verbesserung schaffen*. 3. Aufl. Carl Hanser Verlag, 2018. – ISBN 978-3446453609
- [Mar79] MARCO, Tom de: *Structured Analysis and System Specification*. Prentice Hall, 1979. – ISBN 0-13-854380-1
- [Mar08] MARTIN, Robert C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. 1. Aufl. Prentice Hall, 2008. – ISBN 978-0132350884
- [MF19] MOZILLA-FOUNDATION: *Bugzilla*. Webseite, 2019. – Online erhältlich unter www.bugzilla.org; abgerufen am 14.10.2019.
- [MHS98] MELLIS, Werner ; HERZWURM, Georg ; STELZER, Dirk: *TQM der Softwareentwicklung*. 2. Aufl. Vieweg Verlagsgesellschaft, 1998. – ISBN 978-3528155315
- [Mic18] MICROSOFT: *dotNet Architektur*. Webseite, 2018. – Online erhältlich unter www.microsoft.com; abgerufen am 16.03.2019.

- [Mic19a] MICROSOFT: *ASP.net Core*. Webseite, 2019. – Online erhältlich unter <http://www.asp.net/core/>; abgerufen am 25.07.2019.
- [Mic19b] MICROSOFT: *SQL Server*. Webseite, 2019. – Online erhältlich unter <http://www.microsoft.com/de-de/sql-server/>; abgerufen am 25.07.2019.
- [Mic19c] MICROSYSTEMS, Sun: *MySQL Datenbank*. Webseite, 2019. – Online erhältlich unter <https://www.mysql.com/>; abgerufen am 25.07.2019.
- [MM83] MARTIN, James ; MCCLURE, Carma: *Software Maintenance: The problem and its Solutions*. Prentice Hall PTR, 1983. – ISBN 978-0138223618
- [OIIe18] OOSE-INNOVATIVE-INFORMATIK-EG: *UML-Notationsübersicht*. Webseite, 2018. – Online erhältlich unter <https://www.oose.de/wp-content/uploads/2012/05/UML-Notations%C3%BCbersicht-2.5.pdf>; abgerufen am 02.05.2019.
- [OMG19a] OBJECT-MANAGEMENT-GROUP: *Internationales Konsortium für Standards der objektorientierten Programmierung*. Webseite, 2019. – Online erhältlich unter <https://www.omg.org/>; abgerufen am 02.05.2019.
- [OMG19b] OBJECT MANAGEMENT GROUP, Inc.: *Standard Glossary of Software Engineering*. PDF, 2019. – Online erhältlich unter <http://www.uml.org>; abgerufen am 02.03.2019.
- [Ora19] ORACLE: *Oracle Datenbank*. Webseite, 2019. – Online erhältlich unter <http://www.oracle.com/>; abgerufen am 25.07.2019.
- [ÖS14] ÖSTERREICH, Bernd ; SCHEITHAUER, Axel: *Die UML-Kurzreferenz 2.5 für die Praxis: kurz, bündig, ballastfrei*. 6. Aufl. Oldenbourg Wissenschaftsverlag, 2014. – ISBN 978-3486749090
- [Pre14] PRESSMAN, Roger S.: *Software Engineering, A Practitioner's Approach*. 8. Aufl. McGraw-Hill Education Ltd, 2014. – ISBN 978-1259253157
- [Pyt19] PYTHON: *Python Software Foundation*. Webseite, 2019. – Online erhältlich unter <https://www.python.org/>; abgerufen am 25.07.2019.
- [RCT19] RAILS-CORE-TEAM: *Ruby on Rails*. Webseite, 2019. – Online erhältlich unter <http://rubyonrails.org/>; abgerufen am 25.07.2019.
- [RJB10] RUMBAUGH, James ; JACOBSON, Ivar ; BOOCHE, Grady: *The Unified Modeling Language, Reference Manual*. 2. Aufl. Pearson Education (US), 2010. – ISBN 978-0321718952
- [RQSG12] RUPP, Christine ; QUEINS, Stefan ; SOPHIST-GROUP: *UML 2 glasklar*. 4. Aufl. Carl Hanser Verlag, 2012. – ISBN 978-3-446-43057-0
- [RSG14] RUPP, Christine ; SOPHIST-GROUP: *Requirements-Engineering und –Management: aus der Praxis von klassisch bis agil*. 6. Aufl. Carl Hanser Verlag, 2014. – ISBN 978-3446438934
- [Sch03] SCHULZ, Joseph D.: Requirements based Unified Modeling Language (UML). In: *Borland Software Corporation, White Paper* (2003)
- [SG15] STANDISH-GROUP: *Standish Group CHAOS Studie 2011-2015*. Webseite, 2015. – Online erhältlich unter <http://www.standishgroup.com>; abgerufen am 20.01.2019.
- [Sne05] SNEED, Harry M.: *Software-Projektkalkulation: Praxiserprobte Methoden der Aufwandsschätzung für verschiedene Projektarten*. Carl Hanser Verlag, 2005. – ISBN 978-3446400054

- [Som18] SOMMERVILLE, Ian: *Software Engineering*. 10. Aufl. Pearson Studium, 2018. – ISBN 978–3868943443
- [SS18] SCHWABER, Ken ; SUTHERLAND, Jeff: *The Scrum Guide*. Webseite, 2018. – Online erhältlich unter <https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-German.pdf>; abgerufen am 29.01.2019.
- [TPG19] THE-PHP-GROUP: *PHP - Serverskriptsprache*. Webseite, 2019. – Online erhältlich unter <https://php.net/>; abgerufen am 25.07.2019.
- [W3C19] W3C: *World Wide Web Consortium*. Webseite, 2019. – Online erhältlich unter <https://www.w3.org>; abgerufen am 02.06.2019.
- [We18] WEIT-E.V.: *V-Modell XT*. Webseite, 2018. – Online erhältlich unter www.v-modell-xt.de; abgerufen am 15.06.2019.
- [Wie90] WIEKEN, John-Harry: *Software-Produktion*. 1990
- [WV01] WEIKUM, G. ; VOSSEN, G.: *Transactional Information Systems*. 1. Aufl. Morgan Kaufmann, 2001. – ISBN 978–1558605084
- [ZR06] ZÜLLIGHOVEN, H. ; RAASCH, J. ; RECHENBERG, Peter (Hrsg.) ; POMBERGER, Gustav (Hrsg.): *Informatik-Handbuch*. 4. Aufl. Carl Hanser Verlag, 2006. – ISBN 978–3446401853

Stichwortverzeichnis

- Abhängigkeits-Beziehung 99
Ablaufplan 36
Adaptive Systeme 127
Adaptive Wartung 162
Aggregation 99
Agiles Projektmanagement 27
Agiles Vorgehensmodell 26
Akteur 73
Akteur primär 74
Akteur sekundär 74
Aktivitätsdiagramm 105
Aktivitätsdiagramm wohlgeformt 108
algorithmische Aufwandsschätzmodelle 53
Analogie Fertigungssystem 49
Analogieschätzung 53
Änderungsanforderung 163
Anforderung 60, 64
Anforderungen – Benutzer 66
Anforderungen – Qualitätsmerkmale 66
Anforderungen – System 66
Anforderungen Domäne 65
Anforderungen funktional 64
Anforderungen nicht-funktional 65
Anforderungsanalyse 60
Anforderungsarten 64
Anforderungserhebung 71
Anforderungsmanagement 83
Anforderungsvalidation 82
Anwendungsfall 73
Anwendungsfall spezialisiert 78
Anwendungsfalldiagramm 73
Anwendungsspezialist 48
Apprenticing 71
Architektur Broker 126
Architektur Client-Server 125
Architektur Modell-View-Controller 127
Architektur Software 119
Architektur Three-tier 125
Architektur verteilte 125
Architekturbereiche 120
Architekturmuster 120
Architekturschichten 120
Assoziation 73, 97
Assoziation attribuiert 98
Assoziation gerichtet 97
Assoziation mehrgliedrig 99
Assoziation qualifiziert 98
Aufgaben 14
Aufwandsschätzmodelle algorithmisch 53
Aufwandsschätzung 48

Backlog 28
Baukastensystem 7
Befragungstechnik 72
Begriffseinführung 10
Benutzeranforderung 66
Beobachter-Entwurfsmuster 130
Beobachtungstechnik 71
Berechnungsformeln Projektmanagement 39
Blackboards 125
Black-Box-Test 149
Brainstorming 71
Broker-Architektur 126

- Client-Server-Architektur 125
 COCOMO 53
 Code-Konventionen 31
 Collective-Code-Ownership 31
 Constructive-Cost-Model 53
 Continuous-Integration 32
 CPM-Diagramm 37
 Critical-Path-Method 37

 Datenstruktur-orientiertes Design 132
 Definitionsphase 59
 Design-Pattern 128
 Designphase 87
 Detailkonzept 89
 diversifizierende Testverfahren 152
 Domänenanforderungen 65
 dynamische Testverfahren 146

 Eigenschaften von Software 7
 End-of-Life-Management 6
 Entity-Relationship-Diagramme 132
 Entwurfsmuster 120, 128
 Entwurfsmuster Beobachter 130
 Entwurfsmuster Observer 130
 Entwurfsphase 87
 ER-Diagramme 132
 Ereignisknotennetz 37
 ERM 132
 Ermittlungstechnik Anforderungen 71
 Erzeugungsmuster 130
 Expertenbeurteilung 53
 Extend-Beziehung 77
 eXtreme Programming 27

 Feldbeobachtung 71
 FIFO/LIFO Konstrukte 125
 Filter 125
 Folgeversion 6
 Fragebogen 72
 Frameworks 135
 Frühester Anfangstermin 38
 Frühester Endtermin 38
 Function-Points 50
 Funktionale Anforderungen 64
 funktionsorientiertes Design 131
 Funktionsspezifische Maße 50

 GANTT-Diagramm 36
 Gebietseinordnung 2
 Gesamtkosten 49
 Geschichte 8
 grafische Notation 70
 Größenspezifische Maße 50

 Help-Desks 163
 horizontale Teilung 130

 immaterielle Software 7
 Impact-Analyse 164
 Include-Beziehung 77
 Inside-out-Vorgehen 47
 Inspektion 144
 Interaktive Systeme 127
 Interviewtechnik 72
 Ist-Analyse 45
 Ist-Aufnahme 45

 Kanban 27
 Kick-Off-Meeting 48
 Klassendiagramm 96
 Komplexität Softwareentwicklung 6
 Komponenten 101, 122
 Komponentendiagramm 93, 101
 Komponenten-orientiertes Design 133
 Komposition 99
 Korrigierende Wartung 162
 Kreativitätstechnik 71
 Krise Software 10
 kritischer Pfad 39

 Lastenheft 47
 Lean-Development 27
 Leichtgewichtiges Vorgehensmodell 27
 Lines-of-Code 50
 LOC 50
 LOC Probleme 50

 Machbarkeitsstudie 46
 Makroarchitektur 89, 120, 124
 Marktanalysen 46
 Maße funktionsspezifisch 50
 Maße für Produktivität 50
 Maße größenspezifisch 50

- Meilenstein 39
 Methode 6-3-5 71
 Mikroarchitektur 89, 120, 128
 Modellbildung 12
 Modelle Bedeutung 13
 Modell-View-Controller-Architektur 127
 Musterarchitekturen 120
- Nassi-Shneiderman-Diagramm 103
 Netzplan 37
 Neuentwicklung 6
 Nicht-funktionale Anforderungen 65
 Notation 94
 Notation grafisch 70
- Object-Points 53
 objektorientiertes Design 131
 Observer-Pattern 130
 On-Site Customer 31, 72
 Outside-in Vorgehen 47
- Pair-Programming 31
 Parkinsons-Gesetz 53
 Perfektive Wartung 162
 Pert-Diagramm 37
 Pflichtenheft 62, 89
 phasenübergreifende Verfahren 17
 Pipes 125
 Planungsphase 45
 Port 101
 Präventive Wartung 162
 Price-to-win 53
 Problembereichsanforderungen 65
 Product-Backlog 28, 63
 Product-Owner 28
 Produktivität 49
 Produktivitätsmaße 50
 Produktlinien 134
 Produktplanung 46
 Program Description Language 69
 Programmierrichtlinien 31
 Projektmanagement agil 27
 Projektmanagement klassisch 32
 Projektmanager 34
 Projektplanung 33
 Pufferzeiten 41
- Qualität Software 139
 Qualitätsmerkmale Anforderungen 66
- Rapid-Code-Reviews 31
 Realisierungs-Beziehung 98
 Re-Engineering 164
 Refactoring 31
 Regressionstest 152
 Request-Broker 126
 Requirements-Engineering 64
 Requirements-Engineering-Prozess 61
 Ressourcenauslastung 41
 Ressourcenplan 40
 Reuse 72
 Reverse-Engineering 166
 Reviews 144
 Risikomanagement 55
- Schätzung Unsicherheit 54
 Schätzverfahren 53
 Schichtenarchitektur 124
 Schnittstelle 101
 Scrum 27
 Scrum-Master 28
 Sequenzdiagramm 113
 Service-Level-Agreements (SLA) 164
 Softwarearchitektur 119
 Software-Design-Phase 87
 Software-Design-Sichtweisen 87
 Software-Engineering 2
 Software-Engineering-Spezialist 14
 Software-Fabrik 17
 Softwarekosten Formel 53
 Software-Krise 10
 Software-Lebenszyklus 5
 Software-Support 163
 Softwaretest 139
 Soll-Konzept 46
 Spätester Anfangstermin 38
 Spätester Endtermin 38
 spezialisierter Anwendungsfall 78
 Spezifikation 47, 89
 Spiralmodell 24
 Sprachschablone 68
 Sprint 28
 Sprint-Backlog 28

- statische Testverfahren 144
- Stilllegung 6
- Story-Cards 31
- Story-Map 28
- Struktogramm 103
- Strukturdiagramme 96
- Strukturierte Analyse 70
- Strukturmuster 130
- Subsysteme 122
- Support 163
- Systemanalyse 60
- Systemanforderung 66
- Systemarchäologie 72
- Systemrahmen 73

- TaskBoard 63
- Teilung -horizontal/vertikal 130
- Test Black-Box 149
- Test Software 139
- Test White-Box 146
- Testablauf 141
- Testdriven-Development 32
- Testgetriebene-Entwicklung 32
- Testphase 139
- Testverfahren 143
- Testverfahren diversifizierend 152
- Testverfahren dynamisch 146
- Testverfahren statisch 144
- Teufelsviereck von Sneed 54
- Three-tier-Architektur 125
- Top-Level Design 89
- Transition Wartung 163
- Trendstudien 46

- UML-Diagrammarten 95
- UML-Geschichte 94
- Unit-Test 150
- Use-Case 73
- Use-Case-Diagramm 73

- Use-Case-Tabelle 76
- User-Story 27

- Validation 139, 142
- Verbessertes Wasserfallmodell 20
- Verbesserungsvorschläge 46
- Vererbung 98
- Vergangenheitsorientierte Technik 72
- Verhaltensdiagramme 103
- Verhaltensmuster 130
- Verifikation 139, 142
- Verknüpfungsvarianten
 - Projektmanagement 39
 - verteilte Architekturen 125
 - vertikale Teilung 130
 - V-Modell 22
 - Vorgangsknotennetz 37
 - Vorgangspfeilnetz 37
 - Vorgehensmodelle 17
 - Voruntersuchung 46

- Wartung 158
- Wartung adaptiv 162
- Wartung korrigierend 162
- Wartung perfektiv 162
- Wartung präventiv 162
- Wartungskategorien 162
- Wartungsphase 157
- Wartungsprozess 162
- Wartungstechniken 164
- Wasserfallmodell 19
- Wasserfallmodell verbessert 20
- White-Box-Test 146
- Wiederverwendung Software 133
- Wissensgebiete Projektmanagement 34

- Zeitmanagement 36
- Zustandsdiagramm 118
- Zweigüberdeckungstest 147

SOFTWARE-ENGINEERING //

- Kompakt, kurz und bündig
- Für alle grundständigen Studiengänge der Informatik oder Studiengänge mit Informatikanteil geeignet
- Die wichtigsten, anwendungsorientierten Grundkenntnisse und Praxistipps
- Aktuelles Nachschlagewerk für IT-Studium und -Beruf

Im Software-Engineering geht es um die Modellierung und Entwicklung komplexer, qualitativ hochwertiger Software und die für einen erfolgreich durchgeführten Realisierungsprozess geeigneten Methoden, Werkzeuge und Standards. In diesem kompakten Lehrbuch werden die wichtigsten Themen rund um Software-Engineering erklärt, zusammengefasst und mit kleinen Praxisbeispielen vertieft.

Von zentraler Bedeutung für das Software-Engineering ist der Software-Lebenszyklus. Gemeint ist damit der gesamte Prozess, der zur Erstellung und Erhaltung eines Softwaresystems führt. Sowohl in traditionellen als auch in agilen Softwareerstellungsprozessen läuft dieser Lebenszyklus ab. Bewährt hat sich in der Praxis die Einteilung in sogenannte Phasen, denen die Gliederung folgt.

Anfänger erhalten eine schnelle Orientierung und kompaktes, fundiertes Grundwissen. Fortgeschrittene Leser finden hier ein aktuelles, gut strukturiertes Nachschlagewerk.

Prof. Dr. Anja METZNER ist Professorin für Software-Engineering an der Hochschule für angewandte Wissenschaften Augsburg, Fakultät für Informatik, mit langjähriger Praxiserfahrung. Außer ihrem Fachgebiet sind webbasierte, mobile und datenbankbasierte Softwaresysteme ihre Passion.

HANSER

www.hanser-fachbuch.de/computer

€ 22,99 [D] | € 23,70 [A]

ISBN 978-3-446-45949-6



9 783446 459496