HANSER

Leseprobe

zu

„Software-Engineering – kompakt“

von Anja Metzner

Print-ISBN: 978-3-446-45949-6
E-Book-ISBN: 978-3-446-46365-3

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-45949-6
sowie im Buchhandel

© Carl Hanser Verlag, München

Vorwort

– Guide to the Labyrinth of Software-Engineering –

Viele Bücher über Software-Engineering sind als vollständige Wissensbibliotheken des Fachgebiets konzipiert. Viele Fachbegriffe, Methoden und Vorgehensweisen sind lediglich in ausführlicher Form in meist umfangreichen, dicken Büchern zu finden. Deshalb ist es schwierig, an kurze und bündige Informationen in gut strukturierter Art und Weise zu gelangen.

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 Märtin 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 Strukturdiagramme 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* -[IEE18]

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.

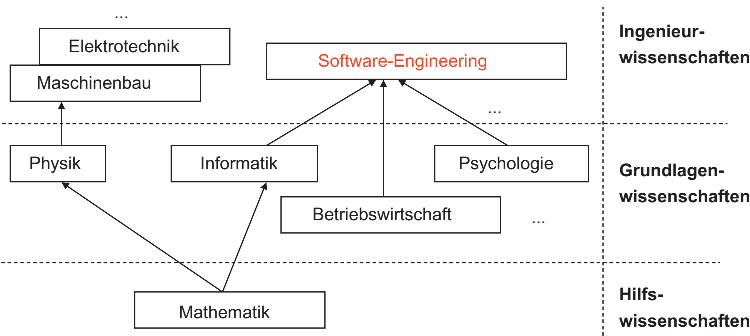

Abbildung 1.1 Gebietseinordnung nach Broy und Rombach [BR02, pp. 438–451]

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]	main() {...}
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 durchgeführten Projekts (oder Herstellungsprozesses). Ein Software-Produkt ist ein Produkt, das aus Software besteht.	Open Office

Begriff	Erklärung	Abbildung
CASE	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: ▪ UML ▪ Transformation in verschiedene Programmiersprachen ▪ Datenstrukturen ▪ Reverse Engineering ▪ Generierung von Prototypen. Aber auch Projektmanagement, Prozessmanagement und viele weitere Entwicklungswerkzeuge sind oft zusätzlich mit im Paket.	
UML [OMG19b]	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, Micrsosoft, . . .)." eine breite Unterstützung in der Industrie. (aus [RQSG12, S. 4])	UNIFIED MODELING LANGUAGE

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).

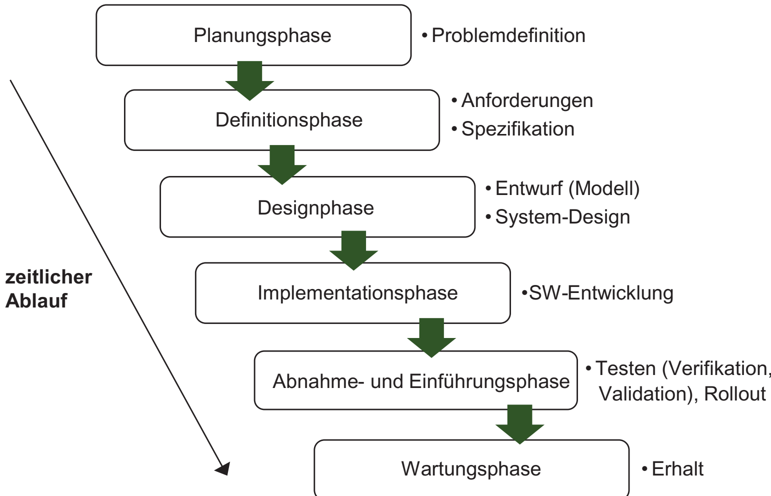

Abbildung 1.2 Software-Lebenszyklus

- **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 Stillegung 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 Projekterfahrungen 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 Teilpassagen 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

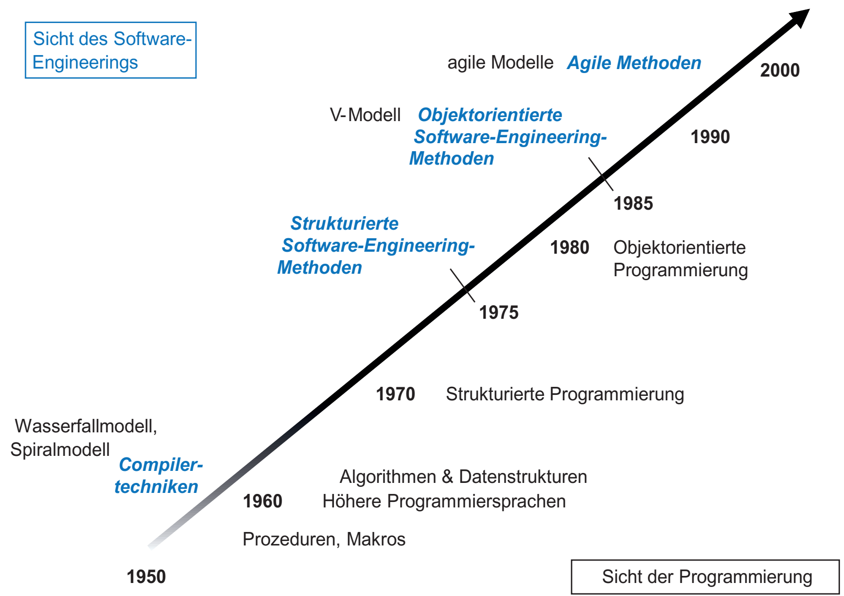

Abbildung 1.3 Software-Engineering-Zeitlinie

(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 Spiralmodel (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“. Erstmalig erkannte man hier, dass die Kosten für die Software die Kosten für die Hardware übersteigen und das 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-

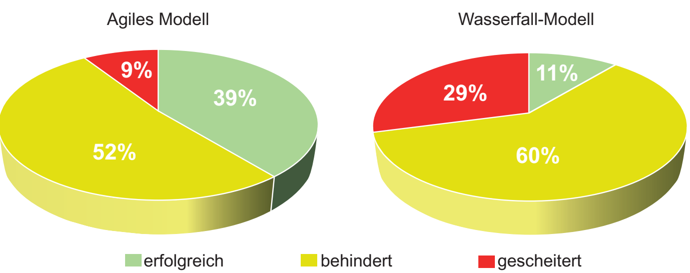

Abbildung 1.4 Standish Group CHAOS-Studie 2011–2015 (>10.000 Software-Projekte jeder Größe)

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**.

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 attributiert 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
Struktogamm 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