Software Engineering

10., aktualisierte Auflage

Ian Sommerville

Pearson

EXTRAS ONLINE

Software Engineering

Software Engineering

10., aktualisierte Auflage

Ian Sommerville

Pearson

Bibliografische Information der Deutschen Nationalbibliothek

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

10 9 8 7 6 5 4 3 2 1

22 21 20 19 18

ISBN 978-3-86894-344-3 (Buch)
ISBN 978-3-86326-835-0 (E-Book)

Lilienthalstraße 2, 85399 Hallbergmoos/Germany
Alle Rechte vorbehalten
www.pearson.de
A part of Pearson plc worldwide

Programmleitung: Birger Peil, bpeil@pearson.de
Übersetzung: Katharina Pieper, Berlin; Petra Alm, Saarbrücken
Fachlektorat und Korrektorat: Katharina Pieper, Berlin
Coverabbildung: ESA–Stephane Corvaja, 2016
Herstellung: Philipp Burkart, pburkart@pearson.de
Satz: Gerhard Alfes, mediaService, Siegen (www.mediaservice.tv)
Druck und Verarbeitung: DZS-Grafik d.d.o., Ljubljana

Printed in Slovenia

Inhaltsübersicht

Inhaltsverzeichnis

Vorwort

Der Fortschritt im Software-Engineering innerhalb der letzten 50 Jahre war erstaunlich. Unsere Gesellschaft könnte ohne große professionelle Softwaresysteme nicht funktionieren. Staatliche Einrichtungen und Infrastrukturen – Energie, Kommunikation und Transportwesen – stützen sich alle auf komplexe und größtenteils zuverlässige Computersysteme. Software hat es uns ermöglicht, den Weltraum zu erforschen und das World Wide Web zu erschaffen – das bedeutendste Informationssystem in der Menschheitsgeschichte. Smartphones und Tablets sind allgegenwärtig und eine ganz neue „App-Branche“, welche die Software für diese Geräte entwickelt, ist in den letzten paar Jahren aufgekommen.

Die Menschheit sieht sich nun gewaltigen Herausforderungen gegenüber: Klimawandel und extreme Wetterverhältnisse, Verknappung natürlicher Ressourcen und eine steigende Weltbevölkerung, die ernährt werden muss und Wohnraum benötigt, internationaler Terrorismus sowie die Notwendigkeit, ältere Menschen dabei zu unterstützen, ein befriedigendes und erfülltes Leben zu führen. Wir brauchen neue Technologien, die uns helfen, diese Herausforderungen anzugehen, und Software wird – ganz gewiss – eine zentrale Rolle in diesen Technologien spielen. Software-Engineering ist daher unglaublich wichtig für unsere Zukunft auf diesem Planeten. Wir müssen weiterhin Softwareentwickler ausbilden und das Fachgebiet weiter vorantreiben, um den steigenden Bedarf an Software decken zu können und um die zunehmend komplexen Softwaresysteme herzustellen, die wir zukünftig brauchen werden.

Natürlich gibt es immer noch Probleme mit Softwareprojekten. Systeme werden manchmal noch verspätet ausgeliefert und kosten mehr als erwartet. Wir erstellen zunehmend komplexe Systeme von Systemen und es sollte uns nicht überraschen, dass wir dabei Schwierigkeiten begegnen. Dennoch sollten wir nicht zulassen, dass diese Probleme die wahren Erfolge des Software-Engineerings und die eindrucksvollen Software-Engineering-Methoden und -Technologien verdecken, die bisher entwickelt wurden.

Dieses Lehrbuch gibt es nun – in mehreren Auflagen – seit über 30 Jahren und diese Auflage beruht auf den grundlegenden Prinzipien, die in der ersten Auflage etabliert wurden:

1. Ich schreibe über Software-Engineering, wie es in der Praxis angewandt wird, ohne eine missionarische Position bezüglich bestimmter Ansätze wie agiler Entwicklung oder formaler Methoden einzunehmen. In der Realität werden Techniken wie agile und planbasierte Entwicklung gemischt, und dies wird im Buch auch so abgebildet.

2. Ich schreibe darüber, was ich weiß und verstehe. Ich hatte viele Anregungen für zusätzliche Themen, die ausführlicher behandelt werden könnten, wie Open-Source-Entwicklung, der Einsatz von UML und Software-Engineering für mobile Geräte. Aber ich weiß einfach nicht genug über diese Bereiche. Meine eigene Arbeit beschäftigte sich mit Systemverlässlichkeit und Systems-Engineering, und dies spiegelt sich in meiner Auswahl der weiterführenden Themen für dieses Buch wider.

Ich glaube, die Schlüsselaspekte im modernen Software-Engineering sind: verwalten der Komplexität, integrieren von agilen mit anderen Methoden sowie gewährleisten, dass unsere Systeme sicher und resilient sind. Diese Themen waren die Triebfeder für die Veränderungen und Ergänzungen in dieser neuen Auflage meines Buchs.

Änderungen zur 9. Auflage

Die größeren Aktualisierungen und Ergänzungen in diesem Buch gegenüber der 9. Auflage sind kurz gefasst:

- Ich habe das Kapitel zum agilen Software-Engineering umfassend aktualisiert und neues Material zu Scrum eingebracht. Andere Kapitel wurden ebenfalls dahingehend aktualisiert, um den zunehmenden Einsatz von agilen Methoden des Software-Engineerings zu reflektieren.
- Ich habe neue Kapitel zum Thema Resilienz-Engineering, Systems-Engineering und Systeme von Systemen hinzugefügt.
- Ich habe drei Kapitel, in denen Zuverlässigkeit, Betriebssicherheit und Informationssicherheit behandelt wird, vollständig umgestellt.
- Ich habe neues Material zu REST-konformen Diensten im Kapitel über serviceorientiertes Software-Engineering hinzugenommen.
- Ich habe das Kapitel zum Konfigurationsmanagement mit neuem Material zu verteilten Versionskontrollsystemen überarbeitet und aktualisiert.
- Ich habe Kapitel zum aspektorientierten Software-Engineering und zur Prozessverbesserung von der Druckversion auf die Website verschoben.
- Neue zusätzliche Materialien wurden der Website hinzugefügt, darunter eine Reihe von Videos (in Englisch). In den Videos erkläre ich wichtige Punkte und verweise auf ähnliche YouTube-Videos.

Der Aufteilung in vier Teile wurde aus früheren Auflagen beibehalten, aber ich habe erhebliche Änderungen an den einzelnen Teilen des Buchs vorgenommen.

1 In *Teil 1*, der Einleitung ins Software-Engineering, habe ich *Kapitel 3* (agile Methoden) vollständig umgeschrieben und aktualisiert, um die zunehmende Verwendung von Scrum zu reflektieren. Eine neue Fallstudie zu einer digitalen Lernumgebung ist in *Kapitel 1* dazugekommen und wird in mehreren Kapiteln benutzt. Altsysteme werden ausführlicher in *Kapitel 9* behandelt. Kleinere Änderungen und Aktualisierungen wurden an allen anderen Kapiteln vorgenommen.

2 *Teil 2*, in dem es um verlässliche Systeme geht, wurde überarbeitet und neu strukturiert. Statt eines aktivitätsbasierten Ansatzes, bei dem Informationen zu Betriebssicherheit, Informationssicherheit und Zuverlässigkeit über mehrere Kapitel verteilt sind, habe ich die Inhalte so strukturiert, dass jedes Thema ein eigenständiges Kapitel bekommt. Dadurch ist es einfacher, ein einzelnes Thema wie Informationssicherheit als Teil einer allgemeineren Vorlesung zu behandeln. Zu Resilienz-Engineering habe ich ein vollständig neues Kapitel hinzugenommen, in dem Cybersicherheit, Resilienz eines Unternehmens und der Entwurf von resilienten Systemen behandelt wird.

3. In *Teil 3* sind neue Kapitel zum Systems-Engineering und Systeme von Systemen hinzugekommen, außerdem habe ich das Material zu serviceorientierten Systemen umfassend überarbeitet, um dem wachsenden Einsatz von REST-konformen Diensten gerecht zu werden. Das Kapitel zum aspektorientierten Software-Engineering wurde aus der Druckversion entfernt, bleibt aber als Webkapitel verfügbar.

4. In *Teil 4* habe ich das Material zum Konfigurationsmanagement aktualisiert, um die steigende Verwendung von Werkzeugen zur verteilten Versionskontrolle wie Git zu reflektieren. Das Kapitel zur Prozessverbesserung wurde aus der Druckversion entfernt, bleibt aber als Webkapitel verfügbar.

Eine wichtige Änderung bei den Zusatzmaterialien für dieses Buch ist die Ergänzung von Video-Links in allen Kapiteln. Ich habe mehr als 40 Videos zu einer Reihe von Themen erstellt, die auf meinem YouTube-Kanal verfügbar sind und auf die von der Webseite zum Buch aus zugegriffen werden kann. Zu den Themen, für die ich keine Videos gemacht habe, empfehle ich YouTube-Videos, die von Nutzen sein könnten.
Ich erkläre die Gründe für die vorgenommenen Änderungen in diesem kurzen Video:

http://software-engineering-book.com/videos/10th-edition-changes

Zielgruppe

Das Buch wendet sich in erster Linie an Studenten, die Einführungs- und Fortgeschrittenenkurse im Bereich Software- und Systems-Engineering belegen. Ich gehe davon aus, dass die Leser ein Basiswissen in Programmierung haben und grundlegende Datenstrukturen verstehen.
Professionelle Softwareentwickler finden in diesem Buch ein allgemeines Nachschlagewerk und eine Quelle, ihr Wissen zu bestimmten Themen auf den aktuellen Stand zu bringen, wie zum Beispiel zu Softwarewiederverwendung, Architekturentwurf, Verlässlichkeit und Informationssicherheit sowie Systems-Engineering.

Das Buch als Skript für eine Software-Engineering-Vorlesung

Das Buch ist so entworfen, dass es auf die drei folgenden unterschiedlichen Arten von Lehrveranstaltungen zum Thema Software-Engineering eingesetzt werden kann:

1. *Allgemeine Einführungen in das Software-Engineering:* Der erste Teil des Buches ist so gestaltet, dass eine einsemestrige Vorlesung zur Einführung in Software-Engineering unterstützt wird. In neun Kapiteln werden die grundlegenden Themen des Software-Engineerings behandelt. Falls Ihre Lehrveranstaltung eine praktische Komponente hat, können die Kapitel zum Management in Teil 4 dafür ausgewechselt werden.

2. *Einführende oder weiterführende Kurse zu bestimmten Themen des Software-Engineerings:* Auf der Basis der Kapitel in den Teilen 2 bis 4 können Sie eine Bandbreite von weiterführenden Kursen konzipieren, indem Sie die Kapitel in den Teilen 2 bis 4 verwenden. Zum Beispiel habe ich eine Lehrveranstaltung zum Systems-Engineering von kritischen Systemen gehalten, indem ich die Kapitel in Teil 2 sowie die Kapitel zum Systems-Engineering und zum Qualitätsmanagement verwendet habe. In einer Veranstaltung, die softwareintensives Systems-En-

gineering beinhaltete, habe ich Kapitel zu den Themen Systems-Engineering, Requirements-Engineering, Systeme von Systemen, verteiltes Software-Engineering, eingebettete Software, Projektmanagement und Projektplanung benutzt.

3 *Fortgeschrittenenveranstaltungen zu bestimmten Themen des Software-Engineerings:* In diesem Fall bilden die Kapitel aus dem Buch die Vorlesungsgrundlage. Diese müssen dann mithilfe weiterer Literatur ergänzt werden, um das Thema detaillierter zu erläutern. Zum Beispiel könnte eine Lehrveranstaltung zur Softwarewiederverwendung auf den Kapiteln 15 bis 18 beruhen.

Website zum Buch

Dieses Buch wurde als hybrider Druck-/Webtext ausgelegt, wobei die Kerninhalte in der Druckausgabe mit zusätzlichem Material im Web verknüpft ist. Einige Kapitel enthalten speziell dafür geschriebene „Webabschnitte“, die die Informationen in diesem Kapitel anreichern. Es gibt außerdem sechs „Webkapitel“ zu Themen, die ich nicht in der Druckversion des Buchs behandele. Alle Webinhalte sind in Englisch verfügbar.
Sie können eine umfangreiche Palette an zusätzlichem Material von der Buch-Website (*software-engineering-book.com*) herunterladen, unter anderem:

- PowerPoint-Präsentationen zu allen Kapiteln in diesem Buch;
- Videos, in denen ich eine Reihe an Software-Engineering-Themen behandele. Ich empfehle außerdem andere YouTube-Videos, die beim Lernen hilfreich sein können;
- ein Leitfaden für Dozenten mit Ratschlägen, wie das Buch in verschiedenen Lehrveranstaltungen eingesetzt werden kann;
- weitere Informationen zu den Fallstudien im Buch (Insulinpumpe, Patienteninformationssystem, Wildnis-Wetterstation, digitale Lernumgebung) sowie zu weiteren Fallstudien wie dem Versagen der Ariane-5-Trägerrakete;
- zusätzliche Fallstudien, die in Software-Engineering-Kursen verwendet werden können;
- sechs Webkapitel, die folgende Themen behandeln: Prozessverbesserung, formale Methoden, Interaktionsentwurf, Anwendungsarchitekturen, Dokumentation und aspektorientierte Entwicklung;
- Webabschnitte, die die Inhalte jedes Kapitels ergänzen. Die Weblinks zu den Webabschnitten finden Sie am Ende der gekennzeichneten Maus-Symbol-Kästen in jedem Kapitel;

Als Reaktion auf Leseranfragen habe ich eine vollständige Anforderungsspezifikation für eine der Fallstudien auf der Website zum Buch veröffentlicht. Es ist normalerweise schwierig für Studenten, auf solche Dokumente zugreifen zu können und dadurch ihre Struktur und Komplexität verstehen zu lernen. Um die Vertraulichkeit nicht zu gefährden, habe ich das Anforderungsdokument von einem realen System umgearbeitet, sodass es uneingeschränkt verwendet werden kann.

Kontakt

Website: *software-engineering-book.com*
E-Mail: *software.engineering.book@gmail.com*
Blog: *iansommerville.com/systems-software-and-technology*
YouTube: *youtube.com/user/SoftwareEngBook*
Facebook: *facebook.com/sommerville.software.engineering*
Twitter: *@SoftwareEngBook* oder *@iansommerville* (für allgemeinere Tweets)
Folgen Sie mir auf Twitter oder Facebook, um Aktualisierungen zu neuem Material zu erhalten und Kommentare zu Software- und Systems-Engineering zu lesen.

Danksagungen

Im Laufe der Jahre haben sich sehr viele Personen an der Weiterentwicklung dieses Buchs beteiligt und ich möchte all denen danken (Rezensenten, Studenten und sonstigen Lesern), die Kommentare zu den früheren Ausgaben abgegeben und konstruktive Änderungsvorschläge eingebracht haben.
Ganz besonders möchte ich meiner Familie, Anne, Ali und Jane, für ihre Liebe, Hilfe und Unterstützung danken, während ich an diesem Buch (und den Vorauflagen) gearbeitet habe.

Ian Sommerville

TEIL I

Einführung in Software-Engineering

Mein Ziel in diesem Teil des Buchs ist es, Ihnen eine allgemeine Einführung in das Software-Engineering zu geben. Die Kapitel in diesem Teil sind so gestaltet, dass sie als Vorlage einer einsemestrigen Einführungsvorlesung in Software-Engineering dienen. Ich führe in wichtige Konzepte wie Softwareprozesse und agile Methoden ein und beschreibe wesentliche Aktivitäten der Softwareentwicklung, von der Anforderungsspezifikation bis hin zur Softwareevolution.

Kapitel 1 ist eine allgemeine Einführung in das professionelle Software-Engineering und definiert einige Konzepte des Software-Engineerings. Außerdem finden Sie hier eine kurze Diskussion ethischer Probleme im Software-Engineering. Für Softwareentwickler ist es wichtig, über die weiteren Auswirkungen ihrer Arbeit nachzudenken. Dieses Kapitel führt auch in die vier Fallstudien ein, die ich im Buch benutze: ein System zur Verwaltung von Patientenakten, die aufgrund von psychiatrischen Problemen behandelt werden (Mentcare), ein Steuerungssystem für eine tragbare Insulinpumpe, ein eingebettes System für eine Wildnis-Wetterstation und eine digitale Lernplattform (iLearn).

Kapitel 2 und 3 behandeln Prozesse des Software-Engineerings und der agilen Entwicklung. In **Kapitel 2** führe ich generische Softwareprozessmodelle (Vorgehensmodelle) wie das Wasserfallmodell ein und bespreche die grundlegenden Aktivitäten, die Teil dieser Prozesse sind. **Kapitel 3** ergänzt dies um eine Diskussion der agilen Entwicklungsmethoden für Software-Engineering. Dieses Kapitel wurde gegenüber den vorigen Ausgaben umfassend verändert, wobei der Fokus jetzt auf agiler Entwicklung unter Verwendung von Scrum liegt. Außerdem werden agile Verfahrensweisen wie User-Storys für die Anforderungsdefinition sowie testgetriebene Entwicklung diskutiert.

Die restlichen Kapitel in diesem Teil sind erweiterte Beschreibungen von Softwareprozessaktivitäten, die in **Kapitel 2** präsentiert werden. **Kapitel 4** behandelt das äußerst wichtige Thema Requirements-Engineering, bei dem die Anforderungen an ein System und seine Funktionen festgelegt werden. **Kapitel 5** erläutert Systemmodellierung mithilfe von UML, wobei mein Fokus auf dem Einsatz von Anwendungsfalldiagrammen, Klassendiagrammen, Sequenzdiagrammen und Zustandsdiagrammen zur Modellierung eines Softwaresystems liegt. In **Kapitel 6** diskutiere ich die Bedeutung der Softwarearchitektur und den Einsatz von Architekturmustern beim Softwareentwurf.

Kapitel 7 stellt den objektorientierten Entwurf und die Verwendung von Entwurfsmustern vor. Ich behandle hier außerdem wichtige Implementierungsaspekte – Wiederverwendung, Konfigurationsverwaltung und Host-Ziel-Entwicklung – und diskutiere Open-Source-Entwicklung. **Kapitel 8** konzentriert sich auf Softwaretests, angefangen bei Modultests während der Systementwicklung bis hin zum Testen der Softwarereleases. Ich bespreche auch den Einsatz von testgetriebener Entwicklung – ein Ansatz, der zuerst in agilen Methoden benutzt wurde, aber vielseitig anwendbar ist. Zu guter Letzt präsentiert **Kapitel 9** einen Überblick über Aspekte der Softwareevolution. Ich behandle Evolutionsprozesse, Softwarewartung und die Verwaltung von Altsystemen.

Einführung

1

ÜBERBLICK

Einführung

Die Lernziele dieses Kapitels sind, Sie mit dem Software-Engineering vertraut zu machen und Ihnen ein Gerüst für das Verständnis des restlichen Buches zu geben. Wenn Sie dieses Kapitel gelesen haben, werden Sie

- verstehen, was Software-Engineering ist und warum es wichtig ist;
- verstehen, dass die Entwicklung von unterschiedlichen Typen von Softwaresystemen auch unterschiedliche Software-Engineering-Techniken erforderlich machen kann;
- die ethischen und beruflichen Aspekte verstehen, die für Softwareentwickler wichtig sind;
- vier Systeme verschiedener Typen kennengelernt haben, die als Beispiele im gesamten Buch benutzt werden.

Software-Engineering ist für das Funktionieren von Regierungen, Gesellschaften und nationaler und internationaler Organisationen sowie Institutionen von entscheidender Bedeutung.Wir können die moderne Welt ohne Software nicht aufrechterhalten. Staatliche Infrastrukturen und Versorgungseinrichtungen werden von computerbasierten Systemen gesteuert und die meisten elektrischen Produkte enthalten einen Computer und Steuerungssoftware. Industrielle Fertigungs- und Verteilungsprozesse sind – wie auch das Finanzsystem – vollständig computergestützt. Unterhaltung, einschließlich der Musikindustrie, Computerspiele, Film und Fernsehen, ist softwareintensiv. Mehr als 75 % der Weltbevölkerung haben ein softwaregesteuertes Mobiltelefon und fast alle dieser Geräte sind internetfähig (Stand 2016).

Softwaresysteme sind abstrakt und nicht greifbar. Sie werden weder durch Materialeigenschaften noch durch physikalische Gesetze oder durch Herstellungsprozesse beschränkt. Dies vereinfacht das Software-Engineering, da es keine natürlichen Grenzen des Potenzials von Software gibt. Jedoch können Softwaresysteme wegen der fehlenden physischen Einschränkungen schnell äußerst komplex, schwer durchschaubar und teuer bei Änderungen werden.

Es gibt viele unterschiedliche Arten von Softwaresystemen, von einfachen **eingebetteten Systemen** bis hin zu komplexen, weltweiten Informationssystemen. Es gibt keine universellen Notationen, Methoden oder Techniken des Software-Engineerings, weil unterschiedliche Softwaretypen unterschiedliche Ansätze erforderlich machen. Die Entwicklung eines Informationssystems für ein Unternehmen unterscheidet sich vollständig von der Entwicklung eines Steuergeräts für wissenschaftliche Instrumente. Und keines dieser Systeme hat viel gemeinsam mit einem grafikintensiven Computerspiel. Alle diese Anwendun-

gen benötigen Software-Engineering, doch sie brauchen nicht alle die gleichen Software-Engineering-Techniken.
Es gibt immer noch viele Berichte von misslungenen Softwareprojekten und „Softwarefehlern“. Software-Engineering wird als unangemessen für moderne Softwareentwicklung kritisiert. Meiner Meinung nach resultieren jedoch viele dieser sogenannten Softwarefehler aus zwei Faktoren:

1. *Steigende Systemkomplexität:* In dem Maße, in dem neue Techniken des Software-Engineerings uns helfen, größere und komplexere Systeme zu bauen, ändern sich auch die Anforderungen. Die Systeme müssen schneller gebaut und ausgeliefert werden; größere, noch komplexere Systeme werden benötigt; und die Systeme müssen ein Leistungsvermögen aufweisen, das zuvor für unmöglich gehalten wurde. Neue Software-Engineering-Techniken müssen erarbeitet werden, um diese neuen Herausforderungen, komplexere Software zu liefern, zu erfüllen.

2. *Versäumnis, Software-Engineering-Methoden zu verwenden:* Es ist recht einfach, Computerprogramme ohne die Anwendung von Software-Engineering-Methoden und -Techniken zu schreiben. Viele Unternehmen wurden im Zuge der Weiterentwicklung ihrer Produkte und Dienstleistungen in die Softwareentwicklung hineingetrieben. Sie benutzen die Methoden des Software-Engineerings nicht in ihrem Arbeitsalltag. Infolgedessen ist ihre Software häufig teurer und weniger zuverlässig als sie sein sollte. Das Fachgebiet Software-Engineering bedarf einer besseren Ausbildung und Schulungen, um diesem Problem zu begegnen.

Softwareentwickler können mit Recht stolz darauf sein, was sie erreicht haben. Natürlich haben wir noch Probleme bei der Entwicklung von komplexer Software, doch ohne Software-Engineering hätten wir das Weltall nicht erforscht und wir müssten ohne das Internet und ohne moderne Telekommunikation auskommen. Jede Art von Fortbewegung wäre gefährlicher und teurer. Die Herausforderungen für die Menschheit im 21. Jahrhundert sind der Klimawandel, schwindende Ressourcen, sich wandelnde Demografien und eine expandierende Weltbevölkerung. Wir werden auf Software-Enigeering angewiesen sein, um Systeme zu entwickeln, mit denen wir diese Probleme angehen können.

Geschichte des Software-Engineerings

Kurz bevor am 20. Juli 1969 die Entscheidung getroffen werden musste, ob die Mondfähre der Apollo-11-Mission die Landung wagen sollte, gab der Bordcomputer „Apollo Guidance Computer" (AGC) eine Fehlermeldung aus – der Rechner war überlastet. Es handelte sich um ein prioritätengesteuertes Multiprozesssystem, das vollständig in Assembler programmiert war. Innerhalb von Minuten konnte die Software aufgrund der vergebenen Prioritäten feststellen, dass kein Risiko für die Mission bestand – das „Go" für die Landung wurde ausgegeben und die „Eagle" landete sicher auf der Mondoberfläche. Margaret Hamilton, die Leiterin der Entwicklungsgruppe von AGC am MIT, kommentiert dies so: „The Apollo 11's Crew became the first humans to *walk* on the moon, and our software became the first software to *run* on the moon."[1]

Hamilton, zu der Zeit eine junge Mathematikerin, gehört zu den Pionieren des Software-Engineerings, und sie war diejenige, die den Begriff „Software-Engineering" popularisierte: Anfangs noch eher als scherzhafter Ausdruck benutzt und belächelt, wurde es zu ihrem Anliegen, den Begriff zu legitimieren. Zu dieser Zeit gab es noch keine Ausbildung für Softwareentwickler, Programmierer lernten durch praktische Erfahrungen und die Erstellung von Software wurde noch nicht als eigenständige Disziplin ernst genommen. Hamilton berichtet: „Es war ein denkwürdiger Tag, als einer der renommiertesten Hardware-Gurus allen Anwesenden eines Meetings erklärte, dass er mit mir einer Meinung sei, dass der Prozess der Softwareerstellung genau wie im Hardwarebereich als eine „Engineering"-Disziplin betrachtet werden sollte." Dies war der Startschuss, der zur Etablierung von Software-Engineering als eigenständigem Fachgebiet führen sollte.

Weitere historische Meilensteine finden Sie unter

http://software-engineering-book.com/web/history/

1.1 Professionelle Softwareentwicklung

Viele Menschen schreiben Programme. Geschäftsleute schreiben Tabellenkalkulationsprogramme, um sich die Arbeit zu erleichtern; Wissenschaftler und Ingenieure schreiben Programme, um ihre Versuchsdaten zu verarbeiten; Privatleute schreiben Programme für ihre eigenen Interessen und zum Vergnügen. Überwiegend ist die Softwareentwicklung jedoch eine professionelle Tätigkeit, bei der Software für Geschäftszwecke erstellt wird, zur Einbindung in andere Geräte oder als Softwareprodukte wie Informationssysteme und CAD-Systeme. Die Hauptunterschiede sind, dass professionelle Software außer vom Programmierer auch von anderen Menschen verwendet werden sollen und dass die Software normalerweise eher von Teams als von einzelnen Personen entwickelt wird. Diese Software wird während ihrer gesamten Lebensdauer gewartet und angepasst.

Software-Engineering soll eher die professionelle Softwareentwicklung als das individuelle Programmieren unterstützen. Es beinhaltet Techniken zur Programmspezifikation, für den Entwurf und zur Weiterentwicklung – nichts, was normalerweise für die private Softwareentwicklung von Belang ist. Um Ihnen zu helfen, einen umfassenden Einblick in das Software-Engineering zu bekommen, habe ich eine Liste von häufigen Fragen und Antworten in ▶ Abbildung 1.1 zusammengestellt.

1 Aus einer Rede von Hamilton auf der International Conference on Software Engineering (ICSE) 2018.

Frage	Antwort
Was ist Software?	Computerprogramme und zugehörige Dokumentation. Softwareprodukte können für einen bestimmten Kunden oder für den freien Markt entwickelt werden.
Was sind die Eigenschaften von guter Software?	Gute Software sollte dem Nutzer die geforderte Funktionalität und Performanz liefern und sollte wartbar, verlässlich und nützlich sein.
Was versteht man unter Software-Engineering?	Software-Engineering ist eine technische Disziplin, die sich mit allen Aspekten der Softwareherstellung beschäftigt, angefangen bei Konzeption über Betrieb bis hin zur Wartung.
Welches sind die grundlegenden Aktivitäten des Software-Engineerings?	Softwarespezifikation, Softwareentwicklung, Softwarevalidierung und Softwareweiterentwicklung.
Worin liegt der Unterschied zwischen Software-Engineering und Informatik?	Informatik konzentriert sich auf Theorie und Grundlagen; beim Software-Engineering geht es um die Entwicklung in der Praxis und die Herstellung nützlicher Software.
Worin liegt der Unterschied zwischen Software-Engineering und Systems-Engineering?	Systems-Engineering beschäftigt sich mit allen Aspekten computerbasierter Systementwicklung, darunter Hardware-, Software- und Verfahrensentwicklung. Software-Engineering ist ein Teil dieses Prozesses.
Worin liegen die größten Herausforderungen für das Software-Engineering?	Der erfolgreiche Umgang mit zunehmender Vielfalt, mit Forderungen nach verkürzten Lieferzeiten und mit der Entwicklung vertrauenswürdiger Software.
Was kostet das Software-Engineering?	Etwa 60 % der Kosten sind Entwicklungskosten, 40 % sind Kosten für Tests. Bei Auftragssoftware übersteigen die Kosten für die Weiterentwicklung oft die für die eigentliche Entwicklung.
Welches sind die besten Techniken und Methoden des Software-Engineerings?	Da alle Softwareprojekte professionell verwaltet und entwickelt werden müssen, sind unterschiedliche Techniken für verschiedene Systemarten angemessen. Zum Beispiel sollten Spiele immer mithilfe einer Folge von Prototypen entwickelt werden, wohingegen sicherheitskritische Steuerungssysteme eine vollständige und analysierbare Spezifikation zur Entwicklung erfordern. Es gibt keine Methoden und Techniken, die für alles gut sind.
Wie hat das Internet das Software-Engineering verändert?	Das Internet hat nicht nur zur Entwicklung von gewaltigen, hochgradig verteilten servicebasierten System zu entwickeln. Die Entwicklung von webbasierten Systemen geführt, es hat auch die Entstehung einer „App"-Industrie für mobile Geräte unterstützt, die die ökonomischen Aspekte von Software verändert hat.

Abbildung 1.1: Häufig gestellte Fragen zum Thema Software-Engineering.

Viele Menschen denken, dass „Software" nur ein anderes Wort für „Computerprogramm" ist. Wenn wir jedoch über Software im Zusammenhang mit Software-Engineering sprechen, dann sind damit nicht nur die Programme selbst gemeint, son-

dern auch die dazugehörige Dokumentation, Bibliotheken, Support-Webseiten und Konfigurationsdaten, die nötig sind, damit diese Programme einsetzbar sind. Ein professionell entwickeltes Softwaresystem ist oft mehr als ein einzelnes Programm. Ein System kann aus mehreren separaten Programmen und Konfigurationsdateien bestehen, die zur Einrichtung dieser Programme verwendet werden. Es kann die Systemdokumentation enthalten, die die Systemstruktur beschreibt, die Benutzerdokumentation, die die Anwendung des Programms erklärt, sowie Websites, von denen der Benutzer die neuesten Produktinformationen herunterladen kann.
Dies ist einer der wichtigsten Unterschiede zwischen professioneller und amateurhafter Softwareentwicklung. Wenn man ein Programm für sich selbst schreibt, dann wird es niemand anderes nutzen und man muss sich nicht die Mühe machen, Programmanleitungen zu schreiben, den Programmentwurf zu dokumentieren usw. Schreibt man dagegen Software, die andere Menschen benutzen und andere Programmierer ändern werden, dann muss man gewöhnlich zusätzliche Information sowie den Programmcode zur Verfügung stellen.
Softwareentwickler beschäftigen sich mit der Entwicklung von Softwareprodukten, d. h. mit Software, die an einen Kunden verkauft werden kann. Es gibt zwei grundlegende Arten von Softwareprodukten:

1. *Generische Produkte* sind eigenständige Systeme, die von einem Softwareentwicklungsunternehmen hergestellt wurden und auf dem freien Markt an jeden Kunden verkauft werden, der sie sich leisten kann. Beispiele für diese Art von Produkten sind Apps für mobile Geräte, Software für PCs, wie Datenbanken, Textverarbeitungsprogramme, Grafikpakete oder Projektverwaltungswerkzeuge. Zu dieser Softwareart gehören auch „vertikale“ Anwendungen, die für einen speziellen Markt entworfen wurden, wie Bibliotheksinformationssysteme, Abrechnungssysteme oder Systeme zur Verwaltung von zahnärztlichen Aufzeichnungen.

2. *Angepasste (oder bestellte) Softwaresysteme* sind Systeme, die im Auftrag eines bestimmten Kunden für diesen hergestellt werden. Ein Softwareanbieter entwickelt die Software speziell für diesen Kunden. Beispiele für solche Auftragssoftware sind Steuerungssysteme für elektronische Geräte, Systeme zur Unterstützung eines bestimmten Geschäftsprozesses und Flugsicherungssysteme.

Der entscheidende Unterschied zwischen diesen Softwaretypen ist, dass bei generischen Produkten das Unternehmen, das die Software entwickelt und implementiert hat, auch die Spezifikation der Software selbst bestimmt. Das heißt, falls Probleme in der Entwicklung auftreten, kann neu überlegt werden, was entwickelt werden soll. Bei Auftragsprodukten wird die Spezifikation von dem Unternehmen entwickelt und kontrolliert, das die Software kauft. Die Softwareentwickler müssen sich an diese Spezifikation halten.
Die Unterscheidung zwischen diesen Systemprodukttypen verwischt jedoch zunehmend. Es werden heutzutage immer mehr Systeme mit einem generischen Produkt als Grundlage gebaut, das dann an die Kundenbedürfnisse angepasst wird. **ERP-Systeme** (*Enterprise Resource Planning System*), wie die Systeme von **SAP** und Oracle, sind beste Beispiele für diesen Ansatz. Hier wird ein großes und komplexes System für ein Unternehmen angepasst, indem Informationen über Geschäftsregeln und -prozesse, erforderliche Berichte usw. integriert werden.

Wenn wir über die Qualität von professioneller Software sprechen, dann müssen wir berücksichtigen, dass die Software von anderen Menschen als ihren Entwicklern genutzt und verändert wird. Qualität betrifft deshalb nicht nur das, was die Software macht. Vielmehr muss sie das Verhalten der Software, während diese ausgeführt wird, sowie die Struktur und Organisation der Systemprogramme und der zugehörigen Dokumentationen beinhalten. Dies schlägt sich in den Qualitäts- oder nichtfunktionalen Merkmalen der Software nieder. Beispiele dieser Merkmale sind die Antwortzeit der Software auf die Anfrage eines Benutzers und die Verständlichkeit des Programmcodes.

Die konkrete Menge dieser Attribute, die man von einem Softwaresystem erwarten könnte, hängt offensichtlich von seiner Anwendung ab. Ein Flugsteuerungssystem muss sicher sein, ein interaktives Spiel muss reaktionsfähig sein, ein Fernsprechvermittlungssystem muss zuverlässig sein und so weiter. Daraus kann man eine Reihe allgemeiner Merkmale herleiten, die in ▶ Abbildung 1.2 aufgelistet sind und von denen ich denke, dass sie die wesentlichen Charakteristika eines professionellen Softwaresystems darstellen.

Produkteigenschaft	Beschreibung
Akzeptanz (*acceptability*)	Software muss von den Benutzern akzeptiert werden, für die sie entwickelt wurde. Das bedeutet, dass sie verständlich, nützlich und kompatibel mit anderen Systemen sein müssen, die diese Benutzer verwenden.
Verlässlichkeit (*dependability*)[1] und Informationssicherheit (*security*)	Die Softwareverlässlichkeit umfasst eine ganze Reihe von Merkmalen, darunter **Zuverlässigkeit** (*reliability*), **Informationssicherheit** (*security*) und **Betriebssicherheit** (*safety*).[2] Verlässliche Software sollte keinen körperlichen oder wirtschaftlichen Schaden verursachen, falls das System ausfällt. Software muss so sicher sein, dass böswillige Benutzer auf das System zuzugreifen oder es beschädigen können.
Effizienz (*efficiency*)	Software sollte nicht verschwenderisch mit Systemressourcen wie Speicher und Prozessorkapazität umgehen. Effizienz umfasst somit Reaktionszeit, Verarbeitungszeit, Ressourcennutzung usw.
Wartbarkeit (*maintainability*)	Software sollte so geschrieben werden, dass sie weiterentwickelt werden kann, um sich verändernden Kundenbedürfnissen Rechnung zu tragen. Das ist ein entscheidendes Merkmal, weil Softwareanpassungen eine unvermeidliche Anforderung einer sich verändernden Geschäftsumgebung sind.

Abbildung 1.2: Wesentliche Merkmale guter Software.

2 Anm. d. Übersetzers: Der englische Begriffe *dependability* umfasst ein weites Spektrum an Eigenschaften, wie Verfügbarkeit, Zuverlässigkeit, Stabilität, Informations- und Betriebssicherheit. Wir haben den Begriff in diesem Buch durchgehend mit „Verlässlichkeit“ übersetzt.

3 Anm. des Übersetzers: Die englischen Begriffe *security* und *safety* werden im Deutschen oft beide vereinfachend mit „Sicherheit“ übersetzt. Um den Unterschied zwischen den beiden Arten der Sicherheit hervorzuhaben, haben wir in diesem Buch durchgängig die Übersetzungen „Informationssicherheit“ und „Betriebssicherheit“ verwendet.

1.1.1 Software-Engineering

Software-Engineering ist eine technische Disziplin, die sich mit allen Aspekten der Softwareherstellung beschäftigt, von den frühen Phasen der Systemspezifikation bis hin zur **Wartung** des Systems, nachdem der Betrieb aufgenommen wurde.[4] In dieser Definition gibt es zwei Schlüsselbegriffe:

1. *Technische Disziplin:* Techniker bringen Dinge zum Laufen. Sie benutzen wenn möglich Theorien, Methoden und Werkzeuge. Sie wählen sie jedoch bewusst aus und versuchen immer, Lösungen für Probleme zu finden, auch wenn es für diese Probleme keine anwendbaren Theorien oder Methoden gibt. Techniker erkennen auch an, dass sie innerhalb organisatorischer und finanzieller Beschränkungen arbeiten müssen und ihre Lösungen innerhalb dieser Beschränkungen suchen müssen.

2. *Alle Aspekte der Softwareherstellung:* Software-Engineering beschäftigt sich nicht nur mit den technischen Prozessen der Softwareentwicklung, sondern umfasst auch Aktivitäten wie die Softwareprojektverwaltung und die Entwicklung von Werkzeugen, Methoden und Theorien, die die Softwareentwicklung unterstützen.

Im Ingenieurwesen geht es darum, Ergebnisse in der geforderten Qualität innerhalb des Zeitplans und des Budgets zu erhalten. Dazu müssen häufig Kompromisse eingegangen werden – Ingenieure dürfen dann keine Perfektionisten sein. Jemand, der Programme zur Eigenanwendung schreibt, kann sich hingegen so viel Zeit für die Programmentwicklung nehmen, wie er will.
Im Allgemeinen arbeiten Softwareentwickler vorwiegend systematisch und organisiert, weil dies oft die effektivste Art ist, qualitativ hochwertige Software herzustellen. Im Software-Engineering geht es jedoch hauptsächlich darum, die am besten geeignete Methode für die jeweiligen Umstände zu wählen. Ein kreativeres, weniger formelles Verfahren kann daher das richtige Vorgehen für einige Softwarearten sein. Ein flexiblerer Softwareprozess, der schnelle Änderungen zulässt, ist besonders für die Entwicklung von webbasierten Systemen und mobilen Apps angemessen, weil diese eine Mischung aus Software und Grafikdesign verlangen.
Software-Engineering ist aus zwei Gründen wichtig:

1. Individuen und die Gesellschaft sind immer mehr auf fortschrittliche Softwaresysteme angewiesen. Wir müssen zuverlässige und vertrauenswürdige Systeme wirtschaftlich und schnell herstellen.

2. Auf lange Sicht ist es gewöhnlich billiger, Methoden und Techniken des Software-Engineerings für professionelle Softwaresysteme zu verwenden, anstatt Programme so zu schreiben, als wären sie ein privates Programmierprojekt. Das Versäumnis, Methoden des Software-Engineerings einzusetzen, führt zu höheren Kosten beim Testen, bei der Qualitätssicherung und der langfristigen Wartung.

Der systematische Ansatz, der beim Software-Engineering eingesetzt wird, wird manchmal **Softwareprozess** genannt. Ein Softwareprozess ist eine Folge von Aktivitä-

4 Anm. d. Übersetzers: Der Begriff „Engineering“ geht somit um über die reine Softwareentwicklung hinaus, deshalb haben wir den Ausdruck unübersetzt (auch für andere Engineering-Arten) beibehalten, wenn wie in dieser Definition die gesamte Disziplin gemeint ist.

ten, die zur Herstellung eines Softwareprodukts führen. Alle Softwareprozesse haben vier grundlegende Aktivitäten gemeinsam haben:

1. Softwarespezifikation: Kunden und Entwickler definieren die zu produzierende Software und die Rahmenbedingungen für ihren Einsatz.
2. Softwareentwicklung: Die Software wird entworfen und programmiert.
3. Softwarevalidierung: Die Software wird überprüft um sicherzustellen, dass sie den Kundenanforderungen entspricht.
4. Softwareevolution: Die Software wird weiterentwickelt, damit sie die sich ändernden Bedürfnisse der Kunden und des Marktes widerspiegelt.

Unterschiedliche Systemtypen erfordern unterschiedliche Entwicklungsprozesse, worauf ich in *Kapitel 2* weiter eingehen werde. So muss z. B. in einem Flugzeug die Echtzeitsoftware vollständig spezifiziert werden, bevor die Entwicklung beginnt. In E-Commerce-Systemen werden die Spezifikation und das Programm in der Regel gemeinsam entwickelt. Folglich können diese allgemeinen Aktivitäten für unterschiedliche Softwaretypen auf unterschiedliche Arten organisiert und auf verschiedenen Detailebenen beschrieben werden – abhängig vom Softwaretyp, der entwickelt wird.
Software-Engineering ist sowohl mit der Informatik als auch mit dem **Systems-Engineering** verbunden:

1. In der Informatik geht es um die Theorien und Methoden, die Computern und Softwaresystemen zugrunde liegen, während sich Software-Engineering mit den praktischen Problemen der Softwareherstellung beschäftigt. Einige Kenntnisse auf dem Gebiet der Informatik sind für Softwareentwickler auf dieselbe Weise unverzichtbar, wie Grundlagenkenntnisse der Physik für Elektrotechniker unerlässlich sind. Die Theorie der Informatik ist jedenfalls insbesondere auf relativ kleine Programme anwendbar. Elegante Theorien der Informatik sind für große, komplexe Probleme, die nach einer Softwarelösung verlangen, selten relevant.
2. Systems-Engineering beschäftigt sich mit allen Aspekten der Entwicklung und Evolution von komplexen Systemen, in denen Software eine wesentliche Rolle spielt. In der Systementwicklung geht es somit um Hardwareentwicklung, Vorgehens- und Verfahrensentwurf, Verteilung des Systems im Einsatz sowie Software-Engineering. Systementwickler beteiligen sich an der Spezifikation des Systems, der Definition der Gesamtarchitektur und an der Integration der verschiedenen Einzelteile, um das fertige System zu schaffen. Ihr Interesse gilt weniger der Entwicklung der Systemkomponenten (Hardware, Software usw.).

Im nächsten Abschnitt stelle ich eine Vielzahl von Softwaretypen vor. Es gibt keine universellen Methoden oder Techniken des Software-Engineerings, die verwendet werden können. Doch vier allgemeine Probleme betreffen viele unterschiedliche Softwaretypen:

1. *Heterogenität:* Systeme müssen verstärkt als **verteilte Systeme** in Netzwerken arbeiten, die aus verschiedenen Computertypen und mobilen Geräten bestehen. Die Software muss nicht nur auf Vielzweckcomputern, sondern möglicherweise auch auf Mobiltelefonen und Tablets ausgeführt werden. Man muss häufig neue Software in ältere Systeme integrieren, die in verschiedenen Programmierspra-

chen geschrieben sind. Die Herausforderung hier besteht also darin, Techniken zu entwickeln, mit denen verlässliche Software hergestellt werden kann, die flexibel genug ist, um mit dieser Heterogenität umzugehen.

2 *Geschäftliche und soziale Veränderungen:* Geschäftsleben und Gesellschaft ändern sich unglaublich schnell, ebenso wie sich neu entstehende Wirtschaftszweige entwickeln und neue Technologien verfügbar werden. Unternehmen müssen in der Lage sein, ihre vorhandene Software zu ändern und schnell neue Software zu entwickeln. Viele traditionelle Software-Engineering-Techniken sind sehr zeitaufwendig und die Auslieferung neuer Systeme dauert länger als geplant. Diese Techniken müssen weiterentwickelt werden, um die Zeit zu reduzieren, bis die Software den Kunden einen Mehrwert bietet.

3 *Sicherheit und Vertrauen:* Da Software mit allen Aspekten unseres Lebens verflochten ist, müssen wir dieser Software vertrauen können. Dies gilt ganz besonders für entfernte Softwaresysteme, auf die über eine Webseite oder eine Webserviceschnittstelle zugegriffen wird. Wir müssen sicherstellen, dass böswillige Benutzer unsere Software nicht erfolgreichangreifen können und dass die Informationssicherheit gewahrt bleibt.

4 *Skalierung:* Software muss über ein breites Spektrum von Skalierungen entwickelt werden, von sehr kleinen eingebetteten System in portablen oder z. B. innerhalb von Kleidung tragbaren Geräten bis hin zu Cloud-basierten Systemen in Internetgröße, die einer globalen Gemeinschaft dienen.

Um diesen Herausforderungen begegnen zu können, werden wir neue Werkzeuge und Techniken brauchen. Außerdem wird es notwendig sein, auf innovative Art und Weise bestehende Methoden des Software-Engineerings zu kombinieren und anzuwenden.

1.1.2 Vielfalt des Software-Engineerings

Software-Engineering ist ein systematischer Ansatz der Softwareherstellung, der Kosten, Zeitplanung und Verlässlichkeitsprobleme sowie die Bedürfnisse von Softwarekunden und -herstellern berücksichtigt. Die im Einzelfall verwendeten Methoden, Werkzeuge und Techniken hängen von dem Unternehmen ab, das die Software entwickelt, von der Art der Software und von den Menschen, die am Entwicklungsprozess beteiligt sind. Es gibt keine universellen Software-Engineering-Methoden, die für alle Systeme und Unternehmen geeignet sind. Stattdessen hat sich während der letzten 50 Jahre eine facettenreiche Menge von Software-Engineering-Methoden und -Werkzeugen herausgebildet. Die SEMAT-Initiative (Jacobson et al., 2013) schlägt jedoch vor, einen elementaren Metaprozess zu definieren, der instanziiert werden kann, um verschieden Arten von Prozessen zu erzeugen. Dieses Vorgehen befindet sich in einem frühen Entwicklungsstadium und könnte eine Basis dafür werden, unsere aktuellen Software-Engineering-Methoden zu verbessern.
Der vielleicht bedeutendste Faktor bei der Frage, welche Software-Engineering-Methoden und -Techniken die wichtigsten sind, ist die Art der zu entwickelnden Anwendung. Es gibt viele unterschiedliche Anwendungsarten, dazu zählen:

1 *Eigenständige* (*stand-alone) Anwendungen:* Dies sind Anwendungssysteme, die auf einem PC oder Apps, die auf einem mobilen Gerät laufen. Sie besitzen alle

nötigen Funktionalitäten und müssen nicht mit einem Netzwerk verbunden sein. Beispiele solcher Anwendungen sind Office-Anwendungen auf einem PC, CAD-Programme, Software zur Fotobearbeitung, Reise-Apps, Produktivitäts-Apps usw.

2 *Interaktive transaktionsbasierte Anwendungen:* Diese Anwendungen werden auf einem entfernten Computer ausgeführt. Die Benutzer können entweder von ihren eigenen PCs, Telefonen oder Tablets aus darauf zugreifen. Hierzu gehören sicherlich Webanwendungen wie E-Commerce-Anwendungen, bei denen man mit einem entfernten System verbunden ist, um Waren und Dienstleistungen zu kaufen. Diese Anwendungsklasse enthält auch Geschäftssysteme, wobei der Zugang zu diesen Systemen über einen Webbrowser oder über spezielle Client-Programm- und Cloud-basierte Dienste wie E-Mail und Foto-Sharing erfolgt. Interaktive Anwendungen benötigen häufig einen großen Datenspeicher, auf den bei jeder Transaktion zugegriffen wird und der dabei jeweils aktualisiert wird.

3 *Eingebettete Steuerungssysteme:* Dies sind Softwaresteuerungssysteme, die Hardwaregeräte steuern und verwalten. Zahlenmäßig gibt es wahrscheinlich mehr eingebettete Systeme als irgendeine andere Art von System. Beispiele für eingebettete Systeme sind die Software in einem Mobiltelefon, Software zur Steuerung des Antiblockiersystems im Auto und Mikrowellensoftware zum Steuern des Kochvorgangs.

4 *Stapelverarbeitende* (*batch processing*) *Systeme:* Dies sind Geschäftssysteme, die zur Verarbeitung großen Datenmengen entworfen wurden. Sie bearbeiten viele individuelle Eingaben, um die dazugehörigen Ausgaben zu erzeugen. Beispiele für Stapelverarbeitungssysteme sind Systeme zur Abrechnung von regelmäßigen Zahlungen wie Telefonabrechnungssysteme und Lohnauszahlungssysteme.

5 *Unterhaltungssysteme:* Dies sind Systeme, für die private Nutzung, die zur Unterhaltung ihrer Nutzer dienen. Die meisten dieser Systeme sind Spiele die eventuell auf speziellen Spielekonsolen ausgeführt werden können. Die Qualität der angebotenen Benutzerinteraktion ist das wichtigste Unterscheidungsmerkmal von Unterhaltungssystemen.

6 *Systeme für die Modellierung und Simulation:* Dies sind Systeme, die von Wissenschaftlern und Ingenieuren entwickelt wurden, um physikalische Vorgänge oder Situationen zu modellieren, in denen viele separate, interagierende Objekte auftreten. Diese Modelle sind oft rechenintensiv und benötigen zur Ausführung parallele Systeme mit hoher Performanz.

7 *Systeme zur Datenerfassung und -analyse:* Datenerfassungssysteme sind Systeme, die Daten aus ihrer Umgebung sammeln und diese Daten an andere Systeme zur Verarbeitung senden. Die Software muss eventuell mit Sensoren interagieren und wird oft in einer lebensfeindlichen Umgebung oder unter extremen Bedingungen installiert wie beispielsweise innerhalb eines Motors oder an einem unzugänglichen Ort. „Big Data"-Analyse kann Cloud-basierte Systeme einbeziehen, die statistische Analyse ausführen und nach Verbindungen in den gesammelten Daten suchen.

8 *Systeme von Systemen:* Diese sind Systeme, die in Unternehmen und anderen großen Organisationen eingesetzt werden und die aus vielen anderen Soft-

waresystemen zusammengesetzt sind. Einige davon können allgemeine Softwareprodukte wie ein ERP-System sein. Andere Systeme in dem Verbund sind eventuell speziell für diese Umgebung geschrieben worden.

Natürlich sind die Grenzen zwischen diesen Systemtypen unscharf. Wenn Sie ein Spiel für ein Mobiltelefon entwickelt, dann müssen Sie dieselben Beschränkungen (Energieversorgung, Zusammenspiel mit der Hardware) beachten wie die Entwickler der Telefonsoftware. Stapelverarbeitungssysteme werden häufig in Verbindung mit webbasierten Transaktionssystemen eingesetzt. Zum Beispiel könnten in einer Firma die Reisekostenabrechnungen über eine Webanwendung eingereicht werden, die monatliche Auszahlung findet jedoch über eine Stapelverarbeitung statt.

Jede Systemart erfordert spezialisierte Software-Engineering-Techniken, weil die Software ganz unterschiedliche Eigenschaften hat. Beispielsweise ist ein eingebettetes Steuerungssystem in einem Auto sicherheitskritisch und wird bei der Installation in das Fahrzeug ins ROM (Read-Only Memory) gebrannt. Daher ist ein Austausch sehr teuer. Solch ein System erfordert eine umfangreiche **Verifikation und Validierung**, um die Wahrscheinlichkeit eines Rückrufs der Autos nach dem Verkauf zum Beheben von Softwareproblemen zu minimieren. Benutzerinteraktion ist minimal (oder existiert nicht), also sollte auch kein Vorgehensmodell gewählt werden, bei dem zunächst ein Prototyp für die Benutzerschnittstellen entwickelt wird.

Für ein interaktives webbasiertes System oder eine App ist **iterative Entwicklung und Auslieferung** der beste Ansatz, da das System aus wiederverwendbaren Komponenten zusammengesetzt ist. Solch ein Vorgehen ist hingegen unpraktisch für ein System von Systemen, für das detaillierte Spezifikationen der Systeminteraktionen im Voraus spezifiziert werden müssen, damit jedes System separat entwickelt werden kann.

Dennoch gibt es einige Grundsätze des Software-Engineerings, die auf alle Arten von Softwaresystemen angewandt werden können:

1. Ein Softwaresystem sollte mithilfe eines gelenkten und verständlichen Entwicklungsprozesses erstellt werden. Das Unternehmen, das die Software entwirft, sollte den Entwicklungsprozess planen und klare Vorstellungen davon haben, was hergestellt wird und wann es fertiggestellt sein wird. Natürlich muss der im Einzelfall eingesetzte Prozess auf die zu entwickelnde Software abgestimmt sein.

2. Verlässlichkeit und Performanz sind für alle Systemarten wichtig. Software sollte sich wie erwartet verhalten, keine Fehler machen und verfügbar sein, wenn sie benötigt wird. Sie sollte intern sicher in ihren Operationen sein und einen weitestgehenden sicheren Schutz gegen äußere Angriffe bieten. Das System sollte effizient arbeiten und keine Betriebsmittel verschwenden.

3. Einen hohen Stellenwert haben die Vereinbarung und die Verwaltung der Softwarespezifikationen und -anforderungen (was die Software tun sollte). Man muss wissen, was die verschiedenen Kunden und Nutzer des Systems von diesem erwarten, und mit ihren Erwartungen umgehen können, damit ein brauchbares System innerhalb des Budgets und gemäß Zeitplan ausgeliefert werden kann.

4. Die vorhandenen Betriebsmittel sollten effizient eingesetzt werden. Dies bedeutet, dass man immer wenn es angebracht ist, bereits bestehende Software wiederverwenden sollte anstatt neue Software zu schreiben.

Diese Grundbegriffe von Prozess, Verlässlichkeit, Anforderungen, Verwaltung und Wiederverwendung sind wichtige Themen dieses Buches. Unterschiedliche Methoden gehen auf unterschiedliche Weise mit diesen Begriffen um, aber sie liegen allen professionellen Softwareentwicklungen zugrunde.
Diese Grundsätze sind unabhängig von der für die Softwareentwicklung verwendeten Programmiersprache. Ich behandle in diesem Buch keine spezifischen Programmiertechniken, weil diese von einem Systemtyp zum anderen stark variieren. Beispielsweise ist eine dynamische Sprache wie **Ruby** der richtige Sprachtyp für interaktive Systemprogrammierung, sie wäre aber für eingebettete Systementwicklung völlig ungeeignet.

1.1.3 Software-Engineering und das Internet

Die Entwicklung des Internets und Word Wide Web hat unser Leben stark beeinflusst. Anfänglich war das Web hauptsächlich ein universell zugänglicher Informationsspeicher und hatte geringe Auswirkungen auf Softwaresysteme. Diese Systeme liefen auf lokalen Computern und waren nur innerhalb eines Unternehmens zugänglich. Um das Jahr 2000 herum begann das Web sich weiterzuentwickeln und den Browsern wurde immer mehr Funktionalität hinzugefügt. Dies bedeutete, dass webbasierte Systeme entwickelt werden konnten, auf die – anstatt über eine speziell zugeschnittene Benutzerschnittstelle – über einen Webbrowser zugegriffen wurde. Das führte zur Entwicklung einer breiten Palette neuer Systemprodukte, die innovative Dienste lieferten und auf die über das Web zugegriffen wurde. Diese Dienste werden oft durch Werbung finanziert, die auf dem Bildschirm des Nutzers angezeigt wird; damit erübrigt sich ein direktes Bezahlen durch den Nutzer.
Auch die Entwicklung von Webbrowsern, die kleine Programme laufen lassen und einige Funktionen lokal ausführen können, führte zu einer Weiterentwicklung in der Geschäfts- und Unternehmenssoftware. Anstatt Software zu schreiben und sie auf den PCs der Benutzer zu installieren, wurde die Software auf einem Webserver eingerichtet. Dadurch wurde es viel billiger, die Software zu verändern und zu aktualisieren, denn es war nicht nötig, sie auf jedem PC zu installieren. Auch die Gesamtkosten wurden reduziert, da die Entwicklung von Benutzerschnittstellen besonders teuer ist. Unternehmen haben ihre betriebseigenen Softwaresysteme überall, wo dies möglich war, auf webbasierte Interaktion verlegt.
Der Begriff „Software as a Service“ (Software als Service, *Kapitel 17*) entstand Anfang des 21. Jahrhunderts. Heute ist dieser Ansatz Standard bei der Auslieferung webbasierter Systemprodukte wie G Suite, Microsoft Office 365 und Adobe Creative Suite. Immer mehr Software läuft auf entfernten „Clouds“ anstatt auf lokalen Servern, der Zugriff darauf erfolgt über das Internet. Eine Computing-Cloud ist eine riesige Anzahl von miteinander verbundenen Rechnern, die von vielen Anwendern gemeinsam benutzt werden. Die Benutzer kaufen die Software nicht, sondern bezahlen einen Betrag, der davon abhängt, wie oft sie die Software tatsächlich nutzen. Eine andere Möglichkeit ist es, Anwendern freien Zugriff auf die Software als Gegenleistung dafür zu gewähren, dass sie sich Werbung auf ihren Bildschirmen anzeigen lassen. Wenn man einen Dienst wie webbasierte E-Mail verwendet, dann benutzt man ein Cloudbasiertes System.
Das Aufkommen des Webs hat zu einer dramatischen Veränderung in der Gestaltung der Unternehmenssoftware geführt. In der Zeit vor dem Web waren Geschäfts-

anwendungen überwiegend monolithisch, einzelne Programme liefen auf einzelnen Computern oder Computerclustern. Die Kommunikation fand nur lokal, innerhalb eines Unternehmens statt. Heute ist Software hochgradig verteilt, manchmal über die ganze Welt. Geschäftsanwendungen werden nicht von Grund auf neu programmiert, sondern verwenden ausgiebig bestehende Komponenten und Programme.
Dieser Wandel in der Softwareorganisation hatte einen gewaltigen Einfluss auf das Software-Engineering von webbasierten Systemen. Zum Beispiel:

1. Der vorherrschende Ansatz bei der Konstruktion von webbasierten Systemen ist die Wiederverwendung von Software. Bei der Konstruktion von webbasierten Systemen wird darüber nachgedacht, wie man sie aus bereits vorhandener Software zusammenbauen kann. Häufig werden sie innerhalb eines Rahmenwerks gebündelt.

2. Es ist heute eine allgemein anerkannte Tatsache, dass es unpraktikabel ist, alle Anforderungen für solch ein System im Voraus zu spezifizieren. Webbasierte Systeme werden immer inkrementell entwickelt und ausgeliefert werden.

3. Software kann mithilfe von serviceorientiertem Software-Engineering implementiert werden, wobei die Softwarekomponenten eigenständige Webdienste sind. Ich stelle diesen Ansatz des Software-Engineerings in *Kapitel 18* vor.

4. Es sind Technologien zur Schnittstellenentwicklung wie AJAX (Holdener, 2008) und HTML5 (Freeman, 2011) entstanden, mit denen umfangreiche Benutzeroberflächen innerhalb eines Webbrowsers erzeugt werden können.

Die grundlegenden Ideen des Software-Engineerings, die im vorhergehenden Abschnitt besprochen wurden, lassen sich auf webbasierte Software ebenso anwenden wie auf andere Softwarearten. Webbasierte Systeme werden immer größer, daher sind Software-Engineering-Techniken, die mit Skalierung und Komplexität umgehen können, für diese Systeme relevant.

1.2 Ethik des Software-Engineerings

Wie bei anderen technischen Disziplinen auch, wird Software-Engineering in einem sozialen und rechtlichen Rahmen ausgeführt, der die Freiheit der Leute einschränkt, die in diesem Bereich arbeiten. Als Softwareentwickler müssen Sie akzeptieren, dass Ihre Arbeit eine größere Verantwortung umfasst als die reine Anwendung technischer Fähigkeiten. Sie müssen sich ethisch und moralisch verantwortungsbewusst verhalten, wenn Sie als professioneller Softwareentwickler respektiert werden wollen.
Es muss nicht extra erwähnt werden, dass die üblichen Regeln von Ehrlichkeit und Integrität eingehalten werden sollten. Sie sollten Ihr Geschick und Ihr Können nicht einsetzen, um sich unehrlich oder auf eine Weise zu verhalten, die die Softwareentwickler als Berufsgruppe in Misskredit bringt. Es gibt jedoch Gebiete, auf denen das Verhalten nicht von Gesetzen geregelt wird, sondern von dem wesentlich abstrakteren Begriff der beruflichen Verantwortung. Dazu gehören:

1. *Vertraulichkeit:* Sie sollten die Vertraulichkeit gegenüber ihren Arbeitgebern oder ihren Kunden bewahren, unabhängig davon, ob eine formelle Vertraulichkeitsvereinbarung unterzeichnet wurde oder nicht.

2 *Kompetenz:* Sie sollten Ihre Kompetenz nicht falsch darstellen. Sie sollten wissentlich keine Aufträge annehmen, die Ihre Kompetenz übersteigen.

3 *Schutz des geistigen Eigentums:* Sie sollten sich mit den örtlich geltenden Gesetzen zum geistigen Eigentum auskennen, wie Patenten und Urheberrechten. Sie sollten unbedingt sicherstellen, dass das geistige Eigentum Ihrer Arbeitgeber und Ihrer Kunden geschützt wird.

4 *Computermissbrauch:* Sie sollten Ihre technischen Fähigkeiten nicht einsetzen, um die Computer anderer zu missbrauchen. Computermissbrauch umfasst die gesamte Palette von relativ harmlosen Begebenheiten (zum Beispiel Computerspiele auf Firmencomputern) bis hin zu extrem ernsten Verstößen (Verbreitung von Viren oder anderer Malware).

Beim Aufstellen ethischer Standards spielen Berufsvereinigungen und Institutionen eine wichtige Rolle. Organisationen wie die ACM (Association for Computing Machinery) und das IEEE (Institute of Electrical and Electronic Engineers) in den USA, die GI (Gesellschaft für Informatik) und der VDI (Verein Deutscher Ingenieure) in Deutschland sowie die British Computer Society veröffentlichen einen **Kodex für professionelles Verhalten** oder ethische Verhaltensregeln (GI, 1997; VDI, 2000). Die Mitglieder dieser Organisationen verpflichten sich, diesen Leitlinien zu folgen, wenn sie die Mitgliedschaft beantragen. Die Verhaltensregeln beschäftigen sich mit grundlegendem ethischen Verhalten im Allgemeinen.
Fachverbände, insbesondere die ACM und das IEEE, haben kooperiert, um einen gemeinsamen ethischen Kodex zu entwerfen. Dieser Kodex besteht sowohl in einer verkürzten Form, die in ▶ Abbildung 1.3 gezeigt wird, als auch in einer längeren Form (Gotterbarn, Miller und Rogerson, 1999), die die verkürzte Darstellung um Details und Inhalt ergänzt. Die Intention dieses Kodex wird in den ersten beiden Absätzen der langen Form in folgender Weise zusammengefasst:

> *Computer spielen eine zentrale und wachsende Rolle in Handel, Industrie, Verwaltung, Medizin, Bildung, Unterhaltung und der Gesellschaft im Allgemeinen. Softwareentwickler sind diejenigen, die durch direkte Beteiligung oder als Dozenten zu Analyse, Spezifikation, Entwurf, Entwicklung, Zertifizierung, Wartung und Tests von Softwaresystemen beitragen. Aufgrund ihrer Rolle bei der Entwicklung von Softwaresystemen haben Softwareentwickler Gelegenheit, Gutes zu tun oder Schaden zu verursachen, andere in die Lage zu versetzen, Gutes zu tun oder Schaden zu verursachen oder andere dahin gehend zu beeinflussen, Gutes zu tun oder Schaden zu verursachen. Um so sicher wie möglich zu sein, dass ihre Anstrengungen benutzt werden, um Gutes zu tun, müssen Softwareentwickler sich verpflichten, Software-Engineering zu einem nützlichen und geachteten Berufsbild zu machen. In Übereinstimmung mit dieser Verpflichtung sollten sich Softwareentwickler an den folgenden ethischen Kodex halten.*
>
> *Der Kodex enthält acht Prinzipien, die mit dem Verhalten und den Entscheidungen professioneller Softwareentwickler zusammenhängen, zu denen Praktiker, Ausbilder, Manager, Aufsichtführende und Herausgeber von Richtlinien sowie Auszubildende und Studierende dieses Berufs zu zählen sind. Diese Prinzipien beschreiben die ethisch verantwortlichen Beziehungen, die einzelne Personen, Gruppen und Organisationen miteinander eingehen, und die Haupt-*

verpflichtungen, die aus diesen Beziehungen entstehen. Die Klauseln jedes Prinzips illustrieren einige der Verpflichtungen, die mit diesen Beziehungen verbunden sind. Diese Verpflichtungen gründen sich auf die Menschlichkeit der Softwareentwickler, wobei sie den Menschen, die durch ihre Arbeit und die besonderen Elemente des Software-Engineerings einbezogen werden, besondere Aufmerksamkeit schulden. Der Kodex schreibt diese Verpflichtungen als bindend für jeden vor, der sich als Softwareentwickler versteht oder danach strebt, ein Softwareentwickler zu werden.[5]

Ethischer Kodex und professionelles Verhalten des Software-Engineerings
ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

Präambel

Diese verkürzte Version der Verhaltensregeln fasst die Bestrebungen auf einer hohen Abstraktionsebene zusammen. Die vollständige Version dieser Erklärung gibt Beispiele und erläutert im Detail, wie diese Bestrebungen unser Handeln als professionelle Softwareentwickler verändert. Ohne diese Bestrebungen können die Details spitzfindig und langwierig werden; ohne die Details können die Aussagen vielversprechend klingen, aber doch inhaltslos bleiben. Zusammen bilden die Bestrebungen und die Details ein zusammenhängendes Regelwerk.

Softwareentwickler sollen sich verpflichten, Analyse, Spezifikation, Entwurf, Entwicklung, Test und Wartung von Software zu einem nützlichen und geachteten Beruf zu machen. In Übereinstimmung mit ihren Verpflichtungen gegenüber der Gesundheit, Sicherheit und dem Wohlergehen der Öffentlichkeit sollen Softwareentwickler sich an die folgenden acht Prinzipien halten:

1. ÖFFENTLICHKEIT – Softwareentwickler sollen in Übereinstimmung mit dem öffentlichen Interesse handeln.
2. KUNDE UND ARBEITGEBER – Softwareentwickler sollen auf eine Weise handeln, die im Interesse ihrer Kunden und ihres Arbeitgebers ist und sich mit dem öffentlichen Interesse deckt.
3. PRODUKT – Softwareentwickler sollen sicherstellen, dass ihre Produkte und damit zusammenhängende Modifikationen den höchstmöglichen professionellen Standards entsprechen.
4. BEURTEILUNG – Softwareentwickler sollen bei der Beurteilung eines Sachverhalts Integrität und Unabhängigkeit wahren.
5. MANAGEMENT – Für das Software-Engineering verantwortliche Manager und Projektleiter sollen sich bei ihrer Tätigkeit ethischen Grundsätzen verpflichtet fühlen und in diesem Sinne handeln.
6. BERUF – Softwareentwickler sollen die Integrität und den Ruf des Berufs in Übereinstimmung mit dem öffentlichen Interesse fördern.
7. KOLLEGEN – Softwareentwickler sollen sich ihren Kollegen gegenüber fair und hilfsbereit verhalten.
8. SELBST – Softwareentwickler sollen sich einem lebenslangen Lernprozess in Bezug auf ihren Beruf unterwerfen und anderen eine ethische Ausübung des Berufes vorleben.

Abbildung 1.3: Ethischer Kodex von ACM/IEEE (ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, Short Version, Preamble. *http://ethics.acm.org/code-of-ethics/software-engineering-code*) (©IEEE/ACM 1999).

In jeder Situation, in der verschiedene Personen unterschiedliche Ansichten und Ziele haben, ist es wahrscheinlich, dass Sie sich einem ethischen Dilemma gegenübersehen. Wenn Sie beispielsweise der Firmenpolitik des oberen Managements nicht zustimmen, wie sollen Sie dann reagieren? Das hängt sicherlich von den betroffenen Personen und der Art der Unstimmigkeit ab. Ist es besser, innerhalb der Firma für Ihre Ansichten zu argumentieren, oder sollten Sie aus prinzipiellen Überlegungen kündi-

5 Original: ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, Full Version, Preamble. *http://ethics.acm.org/code-of-ethics/software-engineering-code.*

gen? Wenn Sie das Gefühl haben, dass es ein Problem mit einem Softwareprojekt gibt, wann melden Sie dies dem Management? Wenn Sie über die Probleme diskutieren, solange sie nicht mehr als ein Verdacht sind, könnten Sie in dieser Situation überreagieren; wenn Sie es zu lange verschieben, könnte es hingegen zu spät sein, die Probleme zu lösen.

Solchen ethischen Dilemmata stehen wir alle in unserem Berufsleben gegenüber und zum Glück sind sie meist unbedeutend oder können ohne große Schwierigkeiten gelöst werden. Wenn sie nicht gelöst werden können, stehen Entwickler vielleicht vor einem neuen Problem. Aus prinzipiellen Gründen sollten sie ihren Job vielleicht kündigen, aber das ist eine Entscheidung, die auch andere betrifft, wie zum Beispiel die Partner oder die Kinder.

Eine für einen Entwickler schwierige Situation entsteht, wenn der Arbeitgeber sich auf unethische Art und Weise verhält. Dies wäre zum Beispiel der Fall, wenn eine Firma für die Entwicklung eines sicherheitskritischen Systems verantwortlich ist und unter Zeitdruck die Berichte aus der Sicherheitsvalidierung fälscht. Ist der Entwickler verpflichtet, sich an die Vertraulichkeit zu halten, oder muss er den Kunden benachrichtigen oder auf die eine oder andere Weise bekannt machen, dass das betreffende System unsicher sein könnte?

Das Problem besteht in dieser Situation darin, dass es keine absoluten Maßstäbe für Sicherheit gibt. Selbst wenn das System nicht gemäß vordefinierter Kriterien validiert wurde, könnten diese Kriterien auch zu streng sein. Das System könnte tatsächlich ein Leben lang sicher arbeiten. Genauso wäre es möglich, dass das System zusammenbricht, obwohl es korrekt validiert wurde. Ein frühes Aufdecken von Problemen kann den Arbeitgeber und andere Mitarbeiter schädigen; das Verschweigen von Problemen kann andere in Mitleidenschaft ziehen.

Sie müssen sich zu diesen Fragen Ihre eigene Meinung bilden. Der angemessene ethische Standpunkt hängt von den jeweils handelnden Personen und ihren Überzeugungen ab. Das Schadenspotenzial, das Ausmaß des Schadens und die Menschen, die dieser Schaden betrifft, sollten die Entscheidung beeinflussen. Wenn die Situation sehr ernst ist, kann es gerechtfertigt sein, zum Beispiel die Presse zu verständigen oder es in sozialen Netzwerken zu veröffentlichen. Sie sollten jedoch immer versuchen, eine Lösung zu finden, die auch die Rechte Ihres Arbeitgebers schützt.

Ein anderes ethisches Problem ist die Entwicklung von militärischen und nuklearen Systemen. Einige Menschen sind strikt dagegen und wollen an keiner Systementwicklung teilnehmen, die mit Abwehrsystemen zusammenhängt. Andere sind bereit, an militärischen Systemen zu arbeiten, aber nicht an Waffensystemen. Wieder andere glauben, dass die nationale Verteidigung ein höheres Gut ist, und haben somit keine ethischen Bedenken, an Waffensystemen mitzuarbeiten.

In dieser Situation ist es wichtig, dass sowohl Arbeitgeber als auch Arbeitnehmer sich schon im Vorfeld gegenseitig über ihre jeweiligen Standpunkte aufklären. Eine Organisation, die mit militärischen oder nuklearen Projekten zu tun hat, sollte verlangen können, dass die Arbeitnehmer alle möglichen Aufgaben akzeptieren. Genauso sollten Arbeitnehmer, die eingestellt wurden und klargestellt haben, dass sie nicht an solchen Systemen arbeiten wollen, zu keinem späteren Zeitpunkt von ihrem Arbeitgeber deswegen unter Druck gesetzt werden.

Das allgemeine Gebiet der Ethik und der professionellen Verantwortung wird immer wichtiger, da softwareintensive Systeme jeden Aspekt der Arbeit und des alltäglichen Lebens durchdringen. Es kann auch von einem philosophischen Standpunkt betrachtet werden, der die Grundprinzipien der Ethik heranzieht und die Ethik des Software-

Engineerings im Hinblick auf diese Grundprinzipien diskutiert. Das ist der Ansatz, den Laudon (1995) und Johnson (2001) verfolgen. Aktuellere Texte wie Tavani (2013) führen den Begriff Cyberethik ein und behandeln sowohl den philosophischen Hintergrund als auch praktische und legale Aspekte. Sie enthalten ethische Problemstellungen für die Anwender von Technologien ebenso wie für Entwickler.
Ich halte einen philosophischen Ansatz für zu abstrakt und denke, dass er nur schwer mit meinen Erfahrungen im Alltag zu vereinbaren ist. Deshalb bevorzuge ich den konkreteren Ansatz, dem man in professionellen Verhaltensregeln Ausdruck verleiht (Bott, 2005; Duquenoy, 2007). Ich glaube, dass Ethik am besten im Zusammenhang mit Software-Engineering diskutiert wird und nicht als eigenes Thema. Daher bespreche ich ethische Fragen im Software-Engineering nicht auf einer abstrakten Ebene, sondern gebe Beispiele in den Übungen, die als Ausgangspunkt einer Gruppendiskussion dienen können.

1.3 Fallstudien

Um die Konzepte des Software-Engineerings zu illustrieren, verwende ich Beispiele von vier unterschiedlichen Systemtypen. Ich benutze absichtlich nicht nur eine einzelne Fallstudie, weil eine der Kernaussagen dieses Buches ist, dass die Verfahren im Software-Engineering von den herzustellenden Systemtypen abhängen. Ich wähle deshalb jeweils ein passendes Beispiel bei der Besprechung von Konzepten wie Betriebssicherheit und Verlässlichkeit, Systemmodellierung, Wiederverwendung usw.
Die vier Systemarten, die ich als Fallstudien benutze, sind:

1. *Ein eingebettetes System:* Dies ist ein System, bei dem Software ein Hardwaregerät steuert und in diesem Gerät eingebettet ist. Kernfragen bei eingebetteten Systemen beschäftigen sich typischerweise mit der physischen Größe, Antwortverhalten, Energieverwaltung usw. Das hier gewählte Beispiel eines eingebetteten Systems ist ein Softwaresystem zur Steuerung einer Insulinpumpe für Diabetiker.

2. *Ein Informationssystem:* Der Hauptzweck dieses Systemtyps ist es, Zugang zu einer Datenbank von Informationen zu verwalten und zur Verfügung zu stellen. Kernfragen bei Informationssystemen schließen Informationssicherheit, Benutzerfreundlichkeit, Datenschutz und Erhalt der Datenintegrität ein. Als Beispiel für ein Informationssystem wird ein Patientendatenmanagementsystem verwendet.

3. *Ein sensorbasiertes Datenerfassungssystem:* Dies ist ein System, dessen Hauptzweck es ist, Daten von einer Menge von Sensoren zu sammeln und diese Daten in irgendeiner Weise zu verarbeiten. Die Schlüsselanforderungen eines solchen Systems sind Zuverlässigkeit – selbst unter feindlichen Umweltbedingungen – und Wartbarkeit. Das Beispiel eines Datenerfassungssystem, das ich hier verwende, ist eine Wetterstation in Wildnisgebieten.

4. *Eine Support-Umgebung:* Dies ist eine integrierte Sammlung von Softwarewerkzeugen, die bestimmte Aktivitäten unterstützen. Programmierumgebungen wie Eclipse (Vogel, 2012) sind für die Leser dieses Buchs sicher die vertrautesten Arten dieser Umgebungen. Ich beschreibe hier ein Beispiel einer digitalen Lernumgebung, die Schüler beim Lernen unterstützt.

Ich stelle im Folgenden jedes dieser Systeme kurz vor, weitere Informationen darüber sind im Web verfügbar.

1.3.1 Ein Steuerungssystem für Insulinpumpen

Eine **Insulinpumpe** ist ein medizinisches System, das die Funktion der Bauchspeicheldrüse (einem inneren Organ) nachahmt. Die Softwaresteuerung dieses Systems ist ein eingebettetes System, das Informationen von einem Sensor sammelt und eine Pumpe steuert, die eine kontrollierte Insulindosis an den Benutzer abgibt.

Das System wird von Menschen benutzt, die an Diabetes leiden. Diabetes ist eine häufige Erkrankung, bei der die menschliche Bauchspeicheldrüse nicht in der Lage ist, eine ausreichende Menge des Hormons Insulin zu produzieren. Insulin baut Glukose (Zucker) im Blut ab. Die übliche Behandlung von Diabetes umfasst regelmäßige Injektionen von künstlich hergestelltem Insulin. Diabetiker messen ihren Blutzuckerspiegel regelmäßig, indem sie ein externes Messgerät benutzen und dann die Insulindosis schätzen, die sie injizieren sollen.

Das Problem ist, dass der benötigte Insulinspiegel nicht nur vom Blutzuckerspiegel, sondern auch von dem Zeitpunkt der letzten Insulininjektion abhängt. Eine unregelmäßige Überprüfung kann zu sehr niedrigen Blutzuckerspiegeln führen (falls zu viel Insulin im Blut ist) oder zu sehr hohen Blutzuckerspiegeln (wenn zu wenig Insulin vorhanden ist). Kurzfristig gesehen ist ein niedriger Blutzuckerspiegel ein ernsterer Zustand, da er zu zeitweiligen Gehirnfehlfunktionen und schließlich zur Bewusstlosigkeit und zum Tod führen kann. Auf der anderen Seite kann ein ständig zu hoher Blutzuckerspiegel längerfristig zu Augenschädigungen, Nierenschäden und Herzproblemen führen.

Fortschritte bei der Entwicklung von miniaturisierten Sensoren haben zu der Entwicklung von automatischen Insulinabgabesystemen geführt. Diese Systeme überwachen den Blutzuckerspiegel und geben bei Bedarf eine angemessene Insulindosis ab. Insulinabgabesysteme wie dieses sind heute verfügbar und werden von Patienten genutzt, die Schwierigkeiten haben, ihren Insulinspiegel zu regeln. Zukünftig kann es für Diabetiker möglich sein, solche Systeme dauerhaft an ihrem Körper zu tragen.

Ein softwarekontrolliertes Insulinabgabesystem benutzt einen Mikrosensor, der im Patienten implantiert ist, um einige Blutparameter zu messen, die proportional zum Zuckerspiegel sind. Diese Werte werden dann zur Pumpensteuerung gesendet. Die Steuerung berechnet den Zuckerspiegel und die benötigte Insulinmenge. Dann sendet sie Signale an eine Miniaturpumpe, um das Insulin über eine permanent angebrachte Nadel abzugeben.

▶ Abbildung 1.4 zeigt die Hardwarekomponenten und -organisation der Insulinpumpe. Um die Beispiele in diesem Buch zu verstehen, müssen Sie lediglich wissen, dass der Blutsensor die elektrische Leitfähigkeit des Bluts unter verschiedenen Bedingungen misst und dass diese Werte zum Blutzuckerspiegel in Bezug gesetzt werden können. Die Insulinpumpe gibt eine Insulineinheit als Reaktion auf einen einzelnen kurzen Stromstoß der Steuerungseinheit ab. Deshalb sendet die Steuerung zehn Impulse an die Pumpe, um zehn Insulineinheiten abzugeben. ▶ Abbildung 1.5 ist ein UML-Aktivitätsmodell, das illustriert, wie die Software eine Blutzuckerspiegeleingabe in eine Folge von Befehlen umwandelt, die wiederum die Insulinpumpe ansteuern.

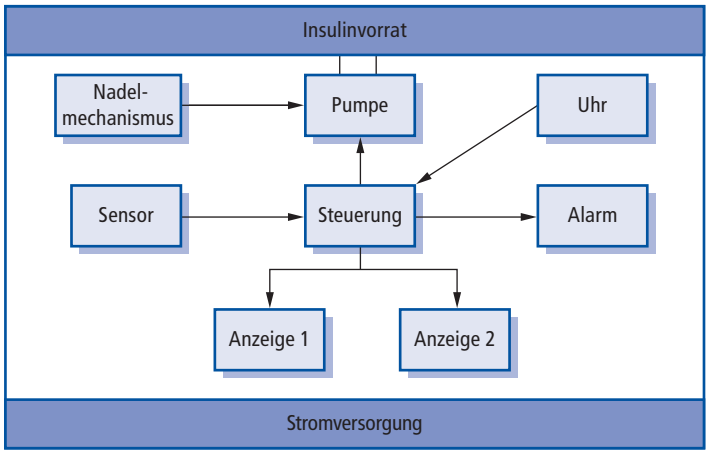

Abbildung 1.4: Hardwarearchitektur der Insulinpumpe.

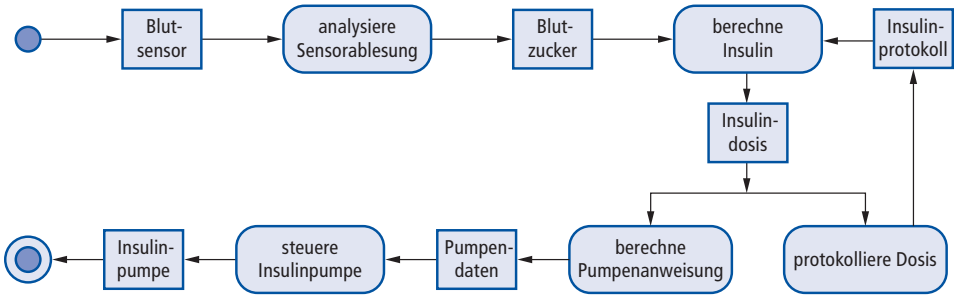

Abbildung 1.5: Aktivitätsmodell der Insulinpumpe.

Dies ist natürlich ein sicherheitskritisches System. Falls die Pumpe ausfällt oder nicht korrekt arbeitet, kann die Gesundheit der Benutzer beeinträchtigt werden oder sie können in ein Koma fallen, weil ihre Blutzuckerspiegel zu hoch oder zu niedrig sind. Dieses System muss deshalb zwei wesentliche Anforderungen auf höherer Ebene erfüllen:

1. Das System soll zur Insulinabgabe bereit sein, wenn diese angefordert wird.
2. Das System soll zuverlässig arbeiten und die richtige Insulinmenge abgeben, um dem aktuellen Blutzuckerspiegel entgegenzuwirken.

Der Entwurf und die Implementierung des Systems müssen daher sicherstellen, dass diese Anforderungen immer erfüllt werden. Detailliertere Anforderungen und Diskussionen darüber, wie die Betriebssicherheit des Systems gewährleistet wird, folgen in späteren Kapiteln.

1.3.2 Ein Patienteninformationssystem für die psychiatrische Ambulanz

Ein Patienteninformationssystem zur Unterstützung einer psychiatrischen Ambulanz (das **Mentcare-System**) ist ein medizinisches Informationssystem, das Informationen über Patienten mit psychischen Problemen und über die durchgeführten Behandlungen verwaltet. Die meisten dieser Patienten benötigen keine stationäre Krankenhausbehandlung, sondern es reicht in der Regel aus, wenn sie regelmäßig in die ambulante Sprechstunde kommen, um einen Arzt zu konsultieren, der genaue Kenntnisse von ihren Problemen hat. Um diesen Patienten den Arztbesuch zu erleichtern, werden diese Sprechstunden nicht nur in Krankenhäusern abgehalten. Sie können auch in lokale Arztpraxen oder Polikliniken durchgeführt werden.

Das Mentcare-System (▶ Abbildung 1.6) ist ein Patienteninformationssystem, das zur Verwendung in solchen ambulanten Sprechstunden vorgesehen ist. Das System verwendet eine zentralisierte Datenbank mit Patienteninformationen, es kann aber auch auf einem Laptop laufen, sodass man darauf zugreifen und das System benutzen kann, auch wenn keine sichere Netzwerkverbindung vorhanden ist. Falls die lokalen Systeme einen sicheren Netzzugang haben, so benutzen sie die Patienteninformation in der Datenbank. Ansonsten können sie lokale Kopien von Patientendatensätzen herunterladen und mit diesen arbeiten, wenn sie nicht im Netz sind. Das System ist kein vollständiges Patientendatenmanagementsystem, es enthält daher keine Informationen über sonstige Krankheiten des Patienten. Es könnte jedoch mit anderen klinischen Informationssysteme interagieren und Daten austauschen.

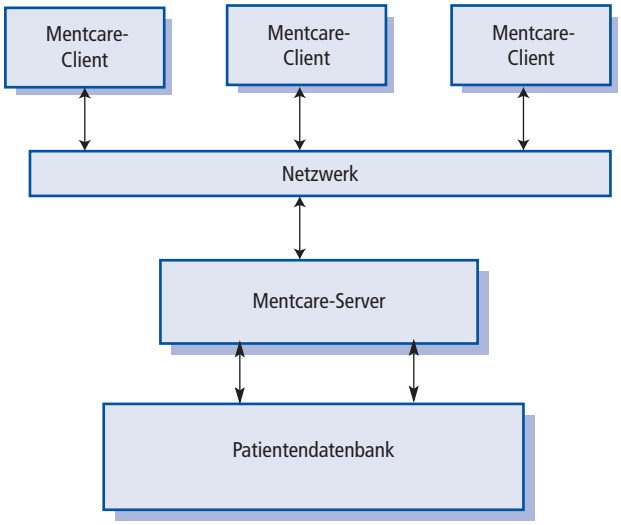

Abbildung 1.6: Die Organisation des Mentcare-Systems.

Das System hat zwei Ziele:

1. Es soll Verwaltungsinformationen erzeugen, die es Verwaltungsangestellten im Gesundheitswesen erlauben, die Leistung gegenüber lokalen und Regierungszielen zu bewerten.

2 Das medizinische Personal soll zeitnah mit Informationen versorgt werden, um die Behandlung der Patienten zu unterstützen.

Patienten mit psychischen Gesundheitsproblemen sind manchmal irrational und desorganisiert und verpassen Termine, verlieren absichtlich oder unabsichtlich Rezepte und Medikamente, vergessen Anweisungen und stellen unangemessene Forderungen an das medizinische Personal. Sie könnten unangemeldet in die Sprechstunde kommen. In einigen wenigen Fällen stellen die Patienten eine Gefahr für sich selbst oder für andere Menschen dar. Es kann passieren, dass sie regelmäßig ihre Adresse ändern oder kurzzeitig oder länger obdachlos sind. Wenn Patienten eine Gefahr darstellen, wird es eventuell notwendig, sie in eine geschlossene psychiatrische Klinik zur Behandlung und Beobachtung einzuweisen.
Benutzer des Systems sind das klinische Personal wie Ärzte, Krankenpfleger und ambulante Gesundheitsdienste (Pfleger, die Patienten zu Hause besuchen, um ihre Behandlung zu überprüfen). Nichtmedizinische Nutzer sind das Empfangspersonal, das die Termine macht, die Mitarbeiter, die das Patientendatensystem pflegen, und Verwaltungsangestellte, die Berichte erzeugen.
Das System wird eingesetzt, um Informationen über Patienten (Name, Adresse, Alter, nächste Angehörige usw.), Konsultationen (Datum, behandelter Arzt, subjektive Eindrücke des Patienten usw.), Krankheitsbilder und Behandlungen aufzuzeichnen. Berichte werden in regelmäßigen Abständen für das medizinische Personal und für die Verwaltungen der Gesundheitsbehörden erstellt. Typischerweise konzentrieren sich die Berichte für das medizinische Personal auf einzelne Patienten, wohingegen Verwaltungsberichte anonymisiert sind und sich mit Bedingungen, Kosten der Behandlung usw. beschäftigen.
Die Schlüsselmerkmale des Systems sind:

1 *Individuelle Patientenbetreuung:* Fachärzte können Datensätze für Patienten erstellen, die vorhandenen Informationen im System bearbeiten, die Krankengeschichte ansehen usw. Das System unterstützt das Zusammenfassen von Daten, sodass sich auch Ärzte, die einen Patienten zuvor noch nicht behandelt haben, schnell über die Hauptprobleme und bisher durchgeführten Maßnahmen informieren können.

2 *Patientenüberwachung:* Das System überwacht regelmäßig die Datensätze der Patienten, die sich in Behandlung befinden, und warnt, wenn mögliche Probleme entdeckt werden. Falls also ein Patient länger nicht zur Sprechstunde gekommen ist, könnte eine Warnmeldung ausgegeben werden. Einer der wichtigsten Elemente des Überwachungssystem ist es, den Überblick über die Patienten zu behalten, die in eine psychiatrische Klinik eingewiesen wurden, und sicherzustellen, dass die gesetzlich vorgeschriebenen Kontrollen zum richtigen Zeitpunkt ausgeführt wurden.

3 *Verwaltungsberichte:* Das System erstellt monatlich Verwaltungsberichte, die die Anzahl der Patienten zeigt, die in jeder Sprechstunde behandelt wurden, die Anzahl der Patienten, die in das Pflegesystem ein- und aus diesem wieder ausgebucht wurden, die Anzahl der Patienten, die in die Psychiatrie eingewiesen wurden, die verschriebenen Arzneimittel und ihre Kosten usw.

Gesetze beeinflussen das System: Gesetze zum Datenschutz, die die Vertraulichkeit von persönlichen Informationen regeln, und Gesetze zum Umgang mit psychisch Kranken, die die Zwangseinweisung von Patienten regeln, die eine Gefahr für sich oder andere darstellen. Psychische Krankheiten sind in dieser Hinsicht einzigartig, denn es ist das einzige medizinische Spezialgebiet, das die Einweisung von Patienten gegen ihren Willen anordnen kann. Diese Einweisung unterliegt sehr strengen gesetzlichen Schutzvorschriften. Eines der Ziele des Mentcare-Systems ist es sicherzustellen, dass das Personal immer im Einklang mit dem Gesetz handelt und dass ihre Entscheidungen für eine eventuelle richterliche Überprüfung aufgezeichnet werden.
Wie in allen medizinischen Systemen ist die Vertraulichkeit der Daten eine kritische Systemforderung. Es ist unbedingt notwendig, dass Patienteninformationen vertraulich behandelt werden und niemals an andere Personen außer dem autorisierten medizinischen Personal und dem Patienten selbst weitergegeben werden. Das Mentcare-System ist ebenfalls ein sicherheitskritisches System. Einige psychische Krankheiten bringen Patienten dazu, an Selbstmord zu denken oder zu einer Gefahr für andere Menschen zu werden. Wo immer es möglich ist, sollte das System das medizinische Personal über potenziell selbstmordgefährdete oder gefährliche Patienten informieren.
Der Gesamtentwurf des Systems muss Datenschutz- und Betriebssicherheitsanforderungen berücksichtigen. Das System muss bei Bedarf zur Verfügung stehen, ansonsten könnte die Betriebssicherheit gefährdet sein und es könnte unmöglich sein, dem Patienten bei Bedarf die richtigen Medikamente zu verschreiben. Hier zeigt sich ein potenzieller Konflikt. Der Datenschutz ist einfacher zu erhalten, wenn es nur eine einzige Kopie der Systemdaten gibt. Um jedoch die Verfügbarkeit im Fall eines Serverausfalls oder bei Unterbrechung der Netzwerkverbindung zu erhalten, sollten mehrere Kopien der Daten vorgehalten werden. Ich bespreche die Abwägung zwischen diesen Anforderungen in späteren Kapiteln.

1.3.3 Eine Wetterstation in Wildnisgebieten

Um bei der Beobachtung von Klimaveränderungen zu helfen und um die Genauigkeit von Wettervorhersagen in abgelegenen Gebieten zu verbessern, entscheidet sich die Regierung eines Landes mit großen Wildnisflächen, mehrere Hundert Wetterstationen in abgelegenen Gebieten einzurichten. Diese Wetterstationen sammeln Daten von einer Reihe von Instrumenten, die Temperatur und Druck, Sonnenschein, Niederschlag, Windgeschwindigkeit und Windrichtung messen.

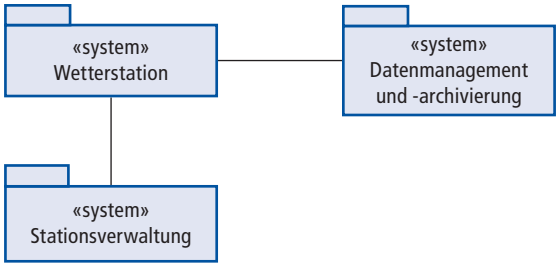

Abbildung 1.7: Die Umgebung der Wetterstationen.

Die **Wildnis-Wetterstationen** sind Teil eines größeren Systems (▶ Abbildung 1.7) – ein Wetterinformationssystem, das Daten von Wetterstationen sammelt und diese anderen Systemen zur Verarbeitung zugänglich macht. Die Systeme in Abbildung 1.7 sind:

1. *Das Wetterstationssystem:* Dieses System ist verantwortlich für das Sammeln der Wetterdaten, für die Ausführung einiger initialer Datenverarbeitungen und für die Übertragung dieser Daten zum Datenmanagementsystem.

2. *Das Datenmanagement- und Archivierungssystem:* Dieses System sammelt die Daten von allen Wildnis-Wetterstationen, verarbeitet Daten und analysiert sie und archiviert die Daten in einer Form, sodass sie anderen Systemen abgerufen werden können, wie beispielsweise Wettervorhersagesystemen.

3. *Das Stationsverwaltungssystem:* Dieses System kann über Satellit mit allen Wildnis-Wetterstationen kommunizieren und so die Funktionstüchtigkeit dieser Systeme überwachen und Berichte über Probleme liefern. Es kann die eingebettete Software in diesen Systemen aktualisieren. Im Falle eines Systemproblems kann dieses System auch benutzt werden, um ein Wettersystem aus der Ferne zu steuern.

In Abbildung 1.7 habe ich das UML-Paketsymbol benutzt um anzuzeigen, dass jedes System eine Sammlung von Komponenten ist. Zur Kennzeichnung der einzelnen Systeme wird das UML-Stereotype «system» verwendet. Die Assoziationen zwischen den Paketen zeigen, dass dort ein Informationsaustausch stattfindet, doch im Moment ist es nicht nötig, diesen genauer zu definieren.

Die Wetterstationen enthalten eine Anzahl von Instrumenten, die Wetterparameter wie Windstärke und -richtung, Boden- und Lufttemperatur, Barometerdruck und Niederschlag innerhalb von 24 Stunden messen. Jedes dieser Instrumente wird von einem Softwaresystem gesteuert, das periodisch Parameterablesungen vornimmt und die Daten verwaltet, die von den Instrumenten gesammelt werden.

Das Wetterstationssystem sammelt Wetterbeobachtungen in kurzen Abständen – Temperaturen werden beispielsweise jede Minute gemessen. Da jedoch die Bandbreite zum Satellit relativ schmal ist, werden einige Daten von der Wetterstation lokal verarbeitet und zusammengefasst. Diese aggregierten Daten werden dann übertragen, wenn das Datenerfassungssystem sie anfordert. Falls es nmöglich ist, eine Verbindung herzustellen, dann pflegt die Wetterstation die Daten lokal, bis der Austausch fortgesetzt werden kann.

Jede Wetterstation ist batteriebetrieben und muss vollständig unabhängig sein – es gibt keine externe Energieversorgung oder Netzwerkkabel. Die gesamte Kommunikation läuft über eine relativ langsame Satellitenverbindung und die Wetterstation muss einige Mechanismen (Solar- oder Windenergie) vorhalten, um ihre Batterien wieder aufzuladen. Da die Stationen in Wildnisgebieten eingesetzt werden, sind sie harten Umweltbedingungen ausgesetzt und könnten von Wildtieren beschädigt werden. Die Stationssoftware ist daher nicht nur mit der Datenerfassung befasst. Sie muss ebenso

1. die Instrumente, Energieversorgung und Kommunikationshardware überwachen und Fehler an das Managementsystem melden;

2. die Systemenergie verwalten um sicherzustellen, dass zum einen die Batterien aufgeladen werden, wenn die Umweltbedingungen es zulassen, und dass zum anderen die Generatoren bei potenziell schädlichen Wetterverhältnissen wie beispielsweise starkem Wind abgeschaltet werden;

3 die dynamische Rekonfigurierung erlauben, wobei Teile der Software mit neuen Versionen ersetzt werden und im Fall eines Systemausfalls auf Backup-Instrumente umgeschaltet wird.

Weil Wetterstationen unabhängig und unbeaufsichtigt funktionieren müssen, ist die installierte Software recht komplex, selbst wenn die Tätigkeit des Datensammelns vergleichsweise einfach ist.

1.3.4 Eine digitale Lernumgebung für Schulen

Viele Lehrer führen an, dass die Benutzung interaktiver Softwaresysteme zur Unterstützung der Lehre sowohl die Motivation verbessern als auch den Wissensstand und das Verständnis der Schüler vertiefen kann. Es gibt jedoch keine grundsätzliche Einigung auf die „beste“ Strategie für computerunterstütztes Lernen und in der Praxis verwenden die Lehrer eine breite Palette unterschiedlicher interaktiver, webbasierter Werkzeuge zur Lernförderung. Die eingesetzten Werkzeuge hängen von dem Alter der Lernenden, ihrem kulturellen Hintergrund, ihrer Erfahrung mit Computern, der verfügbaren Ausstattung und den Vorlieben der beteiligten Lehrer ab.

Eine **digitale Lernumgebung** ist ein Rahmenwerk, in dem eine Reihe allgemeiner sowie speziell entworfener Lernwerkzeuge integriert werden können, zusätzlich zu einigen Anwendungen, die auf die Bedürfnisse der Lernenden abgestimmt sind, die das System benutzen. Das Rahmenwerk stellt allgemeine Dienste zur Verfügung, wie einen Authentifizierungsservice, synchrone und asynchrone Kommunikationsdienste oder einen Speicherdienst.

Die Werkzeuge, die in der jeweiligen Umgebung enthalten sind, werden von Lehrern und Lernenden so gewählt, dass sie zu ihren individuellen Bedürfnissen passen. Dies können allgemeine Anwendungen wie Tabellenkalkulationen, Lernplattformen wie eine virtuelle Lernumgebung, welche die Abgabe und Bewertung von Hausaufgaben verwaltet, oder Spiele und Simulationen sein. Es können aber auch spezielle Inhalte enthalten sein, wie Material zum amerikanischen Bürgerkrieg, sowie Anwendungen, mit denen man diese Inhalte ansehen und kommentieren kann.

▶ Abbildung 1.8 zeigt ein abstraktes Architekturmodell einer digitalen Lernumgebung (**iLearn**), das für den Einsatz in Schulen für Schüler von 3 bis 18 Jahre entwickelt wurde. Bei diesem Ansatz handelt es sich um ein verteiltes System und alle Komponenten der Umgebung sind Dienste, auf die von überall via Internet zugegriffen werden kann. Es ist nicht nötig, dass alle Lernwerkzeuge an einem Ort gebündelt sind.

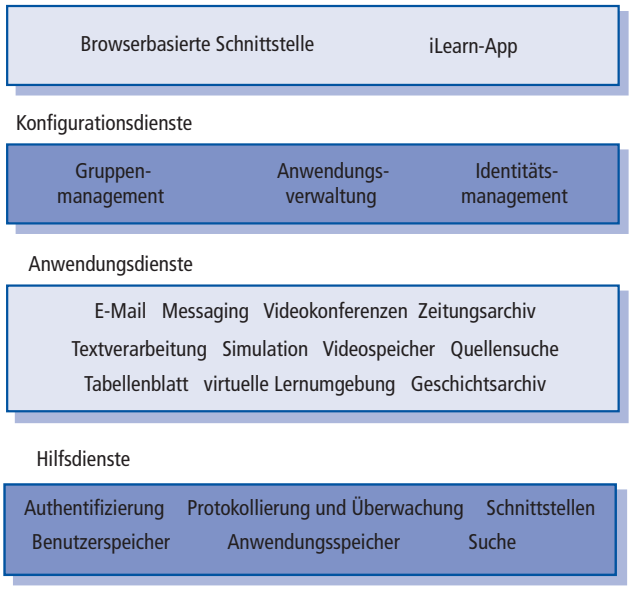

Abbildung 1.8: Die Architektur einer digitalen Lernumgebung (iLearn).

Das System ist ein dienstorientiertes System, bei dem alle Komponenten als austauschbare Dienste betrachtet werden. Drei Arten von Diensten kommen im System vor:

1. *Hilfsdienste*, welche die grundlegende, anwendungsunabhängige Funktionalität liefern und die von anderen Diensten im System eingesetzt werden können. Hilfsdienste werden in der Regel speziell für dieses System entwickelt oder angepasst.

2. *Anwendungsdienste*, die spezifische Anwendungen wie E-Mail, (elektronische) Konferenzführung, Foto-Sharing bieten und auf spezifische didaktische Inhalte wie wissenschaftliche Filme oder historische Quellen zugreifen. Anwendungsdienste sind externe Dienste, die entweder speziell für das System angeschafft wurden oder frei über das Internet verfügbar sind.

3. *Konfigurationsdienste*, die verwendet werden, um die Umgebung auf spezifische Anwendungsdienste anzupassen und um zu definieren, wie Dienste von Schülern, Lehrern und deren Eltern gemeinsam genutzt werden.

Die Umgebung wurde so entworfen, dass Dienste ausgetauscht werden können, sobald neue Dienste verfügbar sind. Außerdem werden unterschiedliche Versionen des Systems zur Verfügung gestellt, die dem Alter des jeweiligen Benutzers entsprechen. Dies bedeutet, das System muss zwei Ebenen von Dienstintegration unterstützen:

1. *Integrierte Dienste* sind Dienste, die eine **Programmierschnittstelle** (API, *Application Programming Interface*) anbieten und auf die von anderen Diensten durch die API zugegriffen werden kann. Direkte Kommunikation von Dienst zu Dienst ist daher möglich. Ein Authentifizierungsdienst ist ein Beispiel eines integrierten Dienstes. Anstatt ihre eigenen Authentifizierungsmechanismen zu verwenden, kann ein Authentifizierungsdienst von anderen Diensten aufgerufen werden, um Benutzer zu authentifizieren. Falls die Benutzer bereits authentifiziert sind, könnte der Authentifizierungsdienst die Informationen über eine API direkt an einen anderen Dienst weitergeben, ohne dass die Benutzer sich erneut authentifizieren müssen.

2. *Unabhängige Dienste* sind Dienste, auf die einfach über einen Browser zugegriffen werden kann und die unabhängig von anderen Diensten arbeiten. Informationen können mit anderen Diensten nur durch explizite Benutzeraktionen wie „Kopieren und Einfügen“ geteilt werden; eventuell ist für jeden unabhängigen Dienst eine erneute Authentifizierung erforderlich.

Sollte sich herausstellen, dass ein unabhängiger Dienst stark genutzt wird, kann das Entwicklerteam diesen Dienst dann integrieren, sodass daraus ein integrierter und unterstützter Dienst wird.

Zusammenfassung

- Software-Engineering ist eine Fachdisziplin, die sich mit allen Aspekten der Softwareherstellung beschäftigt.
- Software besteht nicht nur aus einem oder mehreren Programmen, sondern umfasst auch die gesamte elektronische Dokumentation, die von den Benutzern des Systems, von den Mitarbeitern der Qualitätssicherung und den Entwicklern benötigt wird. Wesentliche Eigenschaften von Softwareprodukten sind Wartbarkeit, Verlässlichkeit und Informationssicherheit, Effizienz und Benutzerfreundlichkeit.
- Der Softwareprozess umfasst alle Aktivitäten, die mit der Entwicklung von Software zusammenhängen. Grundlegende Aktivitäten sind Spezifikation, Entwicklung, Validierung und Evolution.
- Es gibt viele verschiedene Systemarten und jede benötigt entsprechende Software-Engineering-Werkzeuge und -Techniken für ihre Entwicklung. Es gibt wenige – falls überhaupt – spezifische Entwurfs- und Implementierungstechniken, die auf alle Systemarten anwendbar sind.
- Die grundlegenden Konzepte des Software-Engineerings sind auf alle Typen von Softwaresystemen anwendbar. Diese Grundlagen beinhalten geführte Softwareprozesse, Softwareverlässlichkeit und Informationssicherheit, Requirements-Engineering und Wiederverwendung von Software.
- Softwareentwickler haben eine Verantwortung gegenüber ihrem Beruf und der Gesellschaft. Sie sollten sich nicht nur mit technischen Problemen beschäftigen, sondern sollten sich auch der ethischen Aspekte bewusst sein, die ihre Arbeit betreffen.
- Berufsverbände veröffentlichen Verhaltensregeln, die ethische und professionelle Standards beinhalten. Diese legen die Standards für das Verhalten ihrer Mitglieder fest.

Ergänzende Literatur

„Software-Engineering Code of Ethics is approved". Ein Artikel, der den Hintergrund der Entwicklung des Ethikkodex von ACM/IEEE betrachtet und der sowohl die lange als auch die verkürzte Version der Leitlinien enthält. (D. Gotterbarn, K. Miller und S. Rogerson, *Comm. ACM*, Oktober 1999.) *http://dx.doi.org/10.1109/MC.1999.796142*

„A View of 20th and 21st Century Software Engineering". Ein Vorschau und Rückblick auf Software-Engineering von einem der ersten und hervorragendsten Softwareentwickler. Barry Boehm identifiziert zeitlose Prinzipien des Software-Engineerings, aber zeigt auch auf, dass einige der gemeinhin angewendeten Vorgehensweisen überflüssig sind. (B. Boehm, *Proc. 28th Software Engineering Conf.*, Shanghai, 2006.)

http://dx.doi.org/10.1145/1134285.1134288

„Software Engineering Ethics". Spezielle Ausgabe von IEEE Computer mit vielen Artikeln zum Thema. (*IEEE Computer*, 42(6), Juni 2009.)

Ethics for the Information Age. Dies ist ein breitgefächertes Buch, das alle ethischen Aspekte im Bereich Informationtechnologie (IT) behandelt, nicht nur im Software-Engineering. Ich denke, dass dies der richtige Ansatz ist, da Sie die Ethik im Software-Engineering innerhalb eines weiter gefassten ethischen Rahmens sehr gut verstehen müssen.

(M. J. Quinn, 2013, Addison-Wesley).

The Essence of Software Engineering: Applying the SEMAT kernel. Dieses Buch diskutiert die Vorstellung eines universellen Rahmenwerks, das allen Software-Engineering-Methoden zugrunde liegt. Es kann angepasst und für alle Systemarten und Organisationen verwendet werden. Ich persönlich bin skeptisch, ob ein universeller Ansatz in der Praxis realistisch ist, aber das Buch beinhaltet einige interessante Ideen, die überprüfenswert sind.

(I. Jacobsen, P.-W. Ng, P. E. McMahon, I. Spence und S. Lidman, 2013, Addison-Wesley)

Website

- PowerPoint-Folien für dieses Kapitel:
  http://software-engineering-book.com/slides/chap1/
- Links zu begleitenden Videos:
  http://software-engineering-book.com/videos/software-engineering/
- Links zu den Beschreibungen der Fallstudien:
  http://software-engineering-book.com/case-studies/

Lösungshinweise

Übungen

1 Erklären Sie, warum professionelle Software, die für einen Kunden entwickelt wurde, nicht nur die erstellten und ausgelieferten Programme umfasst.

2 Was ist der wichtigste Unterschied zwischen generischer Softwareproduktentwicklung und kundenspezifischer Softwareentwicklung? Was könnte das in der Praxis für Benutzer von generischen Softwareprodukten bedeuten?

3 Welches sind die vier wichtigsten Merkmale, die alle professionellen Softwareprodukte aufweisen sollten? Schlagen Sie vier andere Merkmale vor, die gegebenenfalls auch wichtig sein könnten.

4 Zeigen Sie, abgesehen von den Herausforderungen der Heterogenität, des unternehmerischen und sozialen Wandels, des Vertrauens und der Sicherheit, andere Probleme und Herausforderungen auf, denen Software-Engineering im 21. Jahrhundert wahrscheinlich begegnen wird. (Hinweis: Denken Sie an die Umwelt.)

5 Erklären Sie mit Beispielen auf der Grundlage Ihrer eigenen Kenntnisse von einigen der Anwendungstypen, die in *Abschnitt 1.1.2* besprochen wurden, warum unterschiedliche Anwendungstypen spezialisierte Software-Engineering-Techniken erforderlich machen, um deren Entwurf und Entwicklung zu unterstützen.

6 Erklären Sie, warum die grundlegenden Konzepte des Software-Engineerings – Prozess, Verlässlichkeit, Anforderungsmanagement und Wiederverwendung – für alle Typen von Softwaresystemen relevant sind.

7 Erklären Sie, wie die universelle Benutzung des Webs Softwaresysteme und das Software-Engineering dieser Systeme verändert hat.

8 Diskutieren Sie, ob professionelle Entwickler ebenso wie Ärzte und Anwälte lizenziert sein sollten.

9 Schlagen Sie für jeden Punkt der ethischen Regeln von ACM/IEEE in Abbildung 1.3 ein passendes Beispiel vor, das den entsprechenden Punkt veranschaulicht.

10 Zur Terrorbekämpfung planen zahlreiche Länder die Entwicklung von Computersystemen (oder haben diese bereits entwickelt), die Bürger und deren Aktivitäten überwachen sollen. Dies hat eindeutig Auswirkungen auf den Datenschutz. Erläutern Sie die ethische Überlegungen, die man anstellen könnte, wenn man an der Entwicklung eines solchen Systems beteiligt ist.

Softwareprozesse

2

ÜBERBLICK

Einführung

Ziel dieses Kapitels ist es, Sie mit dem Konzept des Softwareprozesses – einer zusammenhängenden Menge von Aktivitäten in der Softwareherstellung – bekannt zu machen. Wenn Sie dieses Kapitel gelesen haben, werden Sie

- die Konzepte von Softwareprozessen und Vorgehensmodellen für Software verstehen;
- drei allgemeine Vorgehensmodelle für Software kennengelernt haben und wissen, wann diese angewendet werden können;
- die grundlegenden Prozessaktivitäten der Anforderungsanalyse, der Entwicklung, des Test und der Weiterentwicklung (Evolution) von Software kennen;
- verstehen, warum Prozesse geregelt sein sollten, um mit Veränderungen in den Softwareanforderungen und im Softwareentwurf umgehen zu können;
- den Begriff der Softwareprozessverbesserung verstehen sowie die Faktoren kennen, die die Qualität des Softwareprozesses beeinflussen.

Ein Softwareprozess ist eine Menge von zusammengehörigen Aktivitäten, die zur Produktion eines Softwaresystems führen. Wie ich in *Kapitel 1* dargelegt habe, gibt es viele unterschiedliche Arten von Softwaresystemen, und es gibt keine universelle Software-Engineering-Methode, die auf alle Arten anwendbar ist. Folglich gibt es keinen universellen Softwareprozess. Der in einer Firma eingesetzte Prozess hängt von der Art der zu entwickelnden Software ab, von den Anforderungen der Softwarekunden und von den Fähigkeiten der Personen, die die Software schreiben.
Obwohl es jedoch viele unterschiedliche Softwareprozesse gibt, müssen alle in irgendeiner Form die vier grundlegende Software-Engineering-Aktivitäten enthalten, die ich in *Kapitel 1* eingeführt habe und die grundlegend für das Software-Engineering sind:

1. *Softwarespezifikation:* Die Funktionen der Software und die Beschränkungen ihrer Benutzung müssen definiert werden.
2. *Softwareentwicklung:* Die Software, die diese Spezifikation erfüllt, muss erstellt werden.
3. *Softwarevalidierung:* Die Software muss validiert werden um sicherzustellen, dass sie die Kundenwünsche erfüllt.
4. *Softwareevolution:* Die Software muss sich weiterentwickeln, um mit den sich verändernden Bedürfnissen des Kunden Schritt zu halten.

Diese Aktivitäten sind selbst zusammengesetzte Aktivitäten und umfassen Unteraktivitäten wie Anforderungsvalidierung, Architekturentwurf und Modultests. Prozesse beinhalten außerdem andere Aktivitäten wie die Verwaltung der Softwarekonfiguration und Projektplanung, die Produktionsaktivitäten unterstützen.
Wenn wir Prozesse beschreiben und diskutieren, dann meinen wir gewöhnlich die Aktivitäten in diesen Prozessen, wie das Spezifizieren eines Datenmodells und das Entwerfen einer Benutzerschnittstelle sowie die Reihenfolge dieser Aktivitäten. Wir können alle nachvollziehen, was Softwareentwickler tun. Bei der Beschreibung von Prozessen ist es jedoch auch wichtig anzugeben, wer beteiligt ist, was produziert wird sowie welche Bedingungen die Abfolge der Aktivitäten beeinflussen:

1. Produkte oder Liefergegenstände sind das Ergebnis einer Prozessaktivität. Zum Beispiel könnte das Ergebnis der Aktivität des Architekturentwurfs ein Modell der **Softwarearchitektur** sein.

2. Rollen spiegeln die Verantwortlichkeiten der Leute wider, die in den Prozess eingebunden sind. Beispiele für Rollen sind Projektmanager, Konfigurationsmanager und Programmierer.

3. Vor- und Nachbedingungen sind Bedingungen, die gelten müssen, bevor und nachdem eine Prozessaktivität durchgeführt oder ein Produkt hergestellt wurde. Beispielsweise kann es vor Beginn des Architekturentwurfs eine Vorbedingung sein, dass alle Anforderungen vom Kunden genehmigt wurden. Nach dem Ende dieser Aktivität könnte eine Nachbedingung sein, dass die UML-Modelle zur Beschreibung der Architektur überprüft wurden.

Softwareprozesse sind sehr komplex und, wie alle intellektuellen und kreativen Vorgänge, abhängig von menschlichen Entscheidungen und Urteilsvermögen. Da es keinen universellen Prozessgibt, der für alle Arten von Software passend ist, haben die meisten Unternehmen bzw. Organisationen haben ihre eigenen Softwareentwicklungsprozesse entwickelt. Die Prozesse haben sich so verändert, dass sie die Fähigkeiten der Softwareentwickler in einer Organisation und die Eigenschaften der Systeme ausnutzen, die entwickelt werden. Bei sicherheitskritischen Systemen ist ein klar strukturierter Entwicklungsprozess erforderlich, bei dem detaillierte Aufzeichnungen gespeichert werden. Bei Geschäftssystemen, bei denen sich die Anforderungen rasch ändern können, ist ein flexibler, agiler Prozess wahrscheinlich besser.
Wie in *Kapitel 1* schon angesprochen, ist professionelle Softwareentwicklung eine verwaltete Aktivität, Planung gehört demnach zu allen Prozessen dazu. **Plangesteuerte Prozesse** sind Verfahren, bei denen alle Prozessaktivitäten im Voraus geplant werden und bei denen Fortschritt an diesem Plan gemessen wird. In agilen Prozessen, die ich in *Kapitel 3* bespreche, ist das Planen **inkrementell** und kontinuierlich, solange die Software entwickelt wird. Es ist daher leichter, den Prozess zu verändern, um die Änderungen der Kunden- oder Produktanforderungen widerzuspiegeln. Wie Boehm und Turner (2003) darlegen, ist jeder Ansatz für unterschiedliche Softwarearten geeignet. In der Regel muss eine Balance zwischen plangesteuerten und agilen Prozessen gefunden werden.
Obwohl es keinen universellen Softwareprozess gibt, können viele Unternehmen ihren Softwareprozess noch deutlich verbessern. Der Prozess beruht mitunter auf veralteten Techniken oder ignoriert die Vorteile anerkannter Praktiken beim industriellen Software-Engineering. Tatsächlich benutzen viele Unternehmen bei der Entwicklung ihrer Software noch immer keinerlei Methoden aus dem Bereich des Software-Engineerings.

Die Prozesse könnten durch die Einführung von Techniken wie UML-Modellierung und **testgetriebene Entwicklung** verbessert werden. Ich bespreche die Verbesserung von Softwareprozessen kurz in diesem Kapitel und werde ausführlicher in *Kapitel 26* darauf zurückkommen.

2.1 Vorgehensmodelle

Wie ich in *Kapitel 1* erläutert habe, handelt es sich bei einem Vorgehensmodell (manchmal auch als Softwareentwicklungslebenszyklus oder SDLC-Modell bezeichnet) um eine vereinfachte Darstellung eines Softwareprozesses. Jedes Vorgehensmodell stellt einen Prozess aus einer bestimmten Sicht dar und liefert somit nur einen Teil der Informationen über diesen Prozess. Zum Beispiel zeigt ein Prozessaktivitätsmodell die Aktivitäten und ihre Abfolge, aber möglicherweise nicht die Rollen der Personen, die an diesen Aktivitäten beteiligt sind. In diesem Abschnitt führe ich einige sehr allgemeine Vorgehensmodelle (auch Prozessparadigmen genannt) ein und stelle sie aus architektonischer Sicht vor. Das heißt, wir sehen die Struktur des Prozesses, aber keine Details über die Prozessabläufe.

Diese allgemeinen Modelle sind abstrakte Beschreibungen der Softwareprozesse auf einer höheren Ebene, die benutzt werden können, um verschiedene Ansätze der Softwareentwicklung darzustellen. Sie können sich diese als Prozessrahmen vorstellen, die erweitert und angepasst werden können, um spezifischere Software-Engineering-Prozesse zu erzeugen.

Die allgemeinen Vorgehensmodelle, die ich in diesem Kapitel erläutere, sind:

1. *Das Wasserfallmodell:* Dieses Modell stellt die grundlegenden Prozessabläufe wie Spezifikation, Entwicklung, Validierung und Evolution als eigenständige Phasen des Prozesses dar, wie zum Beispiel Anforderungsspezifikation, Softwareentwurf, Implementierung und Tests.

2. *Inkrementelle Entwicklung:* Dieser Ansatz verknüpft die Aktivitäten der Spezifikation, der Entwicklung und der Validierung. Das System wird als eine Folge von Versionen (Inkremente) entwickelt, wobei jede Version neue Funktionalität zu der vorherigen hinzufügt.

3. *Integration und Konfiguration:* Dieses Modell basiert auf der Verfügbarkeit von wiederverwendbaren Komponenten oder Systemen. Der Systementwicklungsprozess beschäftigt sich mehr damit, diese Komponenten für den Einsatz in einer Umgebung zu konfigurieren und in ein System zu integrieren.

Wie bereits gesagt, gibt es kein universelles Prozessmodell, das für alle Arten von Softwareentwicklung geeignet ist. Der passende Prozess hängt vom Kunden und von regulatorischen Anforderungen ab, von der Umgebung, in der die Software eingesetzt werden wird, und vom Typ der zu entwickelnden Software. Beispielsweise wird sicherheitskritische Software in der Regel mithilfe eines Wasserfallprozesses entwickelt, da viel Analyse und Dokumentation erforderlich ist, bevor die Implementierung beginnt. Softwareprodukte werden heutzutage immer mithilfe eines inkrementellen Prozessmodells entwickelt. Die Entwicklung von Geschäftssystemen geht zunehmend den Weg, bestehende Systeme zu konfigurieren und diese zu integrieren, um so ein neues System mit der geforderten Funktionalität zu erzeugen.

Die Mehrheit der praktischen Softwareprozesse basiert auf einem allgemeinen Modell, doch es werden häufig Funktionen anderer Modelle einbezogen. Dies gilt insbesondere für großes Systems-Engineering.
Diese Modelle schließen sich nicht gegenseitig aus und werden häufig zusammen verwendet, vor allem bei der Entwicklung großer Systeme. Für große Systeme ist es sinnvoll, einige der besten Eigenschaften von allen allgemeinen Prozessen zu kombinieren. Man benötigt Informationen über die wesentlichen Systemanforderungen, um eine geeignete Softwarearchitektur zu entwerfen. Hier kann man nicht inkrementell vorgehen. Doch Subsysteme innerhalb eines größeren Systems könnten mit unterschiedlichen Ansätzen entwickelt werden. Teile des Systems, die bereits gut verstanden sind, können mithilfe eines wasserfallbasierten Prozesses spezifiziert und entworfen werden oder können als fertiges System gekauft und dann konfiguriert werden. Andere Teile des Systems jedoch, die im Voraus schwierig zu spezifizieren sind, sollten immer unter Benutzung des inkrementellen Ansatzes entwickelt werden. In beiden Fällen ist es wahrscheinlich, dass Softwarekomponenten wiederverwendet werden.
Es wurden verschiedene Versuche unternommen, um „universelle" Prozessmodelle zu entwickeln, die auf alle diese allgemeinen Modelle zurückgreifen. Eines der bekanntesten dieser universellen Modelle ist RUP (*Rational Unified Process*; Krutchen, 2003), das von der US-amerikanischen Softwarefirma Rational entwickelt wurde. RUP ist ein flexibles Modell, das auf unterschiedliche Art und Weise instanziiert werden kann, um Prozesse zu erzeugen, die jedem der hier diskutierten allgemeinen Prozessmodelle ähneln. **RUP** wurde von einigen großen Unternehmen (insbesondere IBM) übernommen, konnte aber keine breite Akzeptanz gewinnen.

Rational Unified Process (RUP)

Der RUP vereinigt Elemente aus allen allgemeinen Vorgehensmodellen, die hier besprochen werden, und unterstützt die Entwicklung von Prototypen sowie die inkrementelle Auslieferung (Krutchen, 2003). Der RUP wird normalerweise aus drei Perspektiven heraus beschrieben: eine dynamische Perspektive, die die Phasen des Modells zeitlich darstellt; eine statische Perspektive, die die Prozessaktivitäten darstellt; und eine praxisbezogene Perspektive, die die im Prozess empfohlenen Vorgehensweisen vorschlägt. Die Phasen im RUP sind: Konzeption, in der ein Geschäftsfall für das System ausgearbeitet wird; Entwurf, in der Anforderungen und Architektur entwickelt werden; Konstruktion, in der die Software implementiert wird; und Übergabe, in der das System ausgeliefert wird.

http://software-engineering-book.com/web/rup/

2.1.1 Das Wasserfallmodell

Das erste veröffentlichte Modell für die Softwareentwicklung wurde von Prozessmodellen abgeleitet, die in der Entwicklung großer militärischer Systeme verwendet wurden (Royce, 1970). Dieses Modell stellt den Softwareentwicklungsprozess als eine Reihe von Phasen dar, wie in ▶ Abbildung 2.1 veranschaulicht. Wegen der Kaskade von einer Phase zur nächsten wird dieses Modell **Wasserfallmodell** oder der **Softwarelebenszyklus** genannt. Das Wasserfallmodell ist ein Beispiel eines plangesteuer-

ten Prozesses. Im Prinzip plant man alle Prozessaktivitäten inhaltlich und zeitlich, bevor man mit der Softwareentwicklung beginnt.

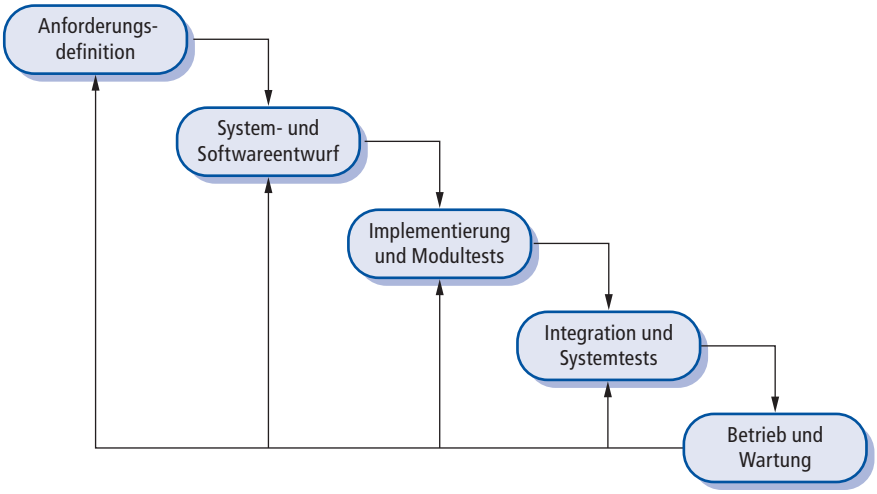

Abbildung 2.1: Das Wasserfallmodell.

Die Phasen des Wasserfallmodells spiegeln direkt die grundlegenden Aktivitäten der Softwareentwicklung wider:

1. *Analyse und Definition der Anforderungen:* Die Dienstleistungen, Einschränkungen und Ziele des Systems werden in Zusammenarbeit mit den Systembenutzern aufgestellt. Dann werden sie detaillierter definiert und dienen so als Systemspezifikationen.

2. *System- und Softwareentwurf:* Der Systementwurfsprozess weist die Anforderungen entweder Hard- oder Softwaresystemen zu. So wird eine übergeordnete Systemarchitektur festgelegt. Beim Softwareentwurf geht es um das Erkennen und Beschreiben der grundlegenden abstrakten Softwaresysteme und ihrer Beziehungen zueinander.

3. *Implementierung und Modultests:* In dieser Phase wird der Softwareentwurf durch eine Menge von Programmen oder Programmeinheiten umgesetzt. Das Testen der Module stellt sicher, dass jede Einheit ihre Spezifikation erfüllt.

4. *Integration und Systemtest:* Die einzelnen Programmeinheiten oder Programme werden integriert und als Ganzes getestet, um sicherzustellen, dass die Softwareanforderungen erfüllt werden. Nach den Tests wird das Softwaresystem an den Kunden ausgeliefert.

5. *Betrieb und Wartung*: Normalerweise ist dies die längste Phase innerhalb des Lebenszyklus. Das System wird installiert und zum Gebrauch freigegeben. Zur Wartung gehören das Korrigieren von Fehlern, die in den früheren Phasen nicht entdeckt wurden, die Verbesserung der Implementierung von Systemeinheiten und die Verbesserung des Systems, falls neue Anforderungen aufgedeckt werden.

Das Spiralprozessmodell von Boehm

Barry Boehm, einer der Pioniere im Software-Engineering, hat ein inkrementelles Prozessmodell vorgeschlagen, das risikogesteuert war. Der Prozess wird als eine Spirale dargestellt anstatt als Folge von Aktivitäten (Boehm, 1988).

Jede Windung der Spirale steht für eine Phase des Prozesses. So beschäftigt sich die innere Windung mit der Machbarkeit des Systems, die nächste mit der Definition der Systemanforderungen, die folgende mit dem Systementwurf usw. Das **Spiralmodell** kombiniert die Vermeidung von Änderungen mit Änderungstoleranz. Es geht davon aus, dass Änderungen ein Ergebnis von Projektrisiken sind, und beinhaltet explizite Risikomanagementaktivitäten, um diese Risiken zu reduzieren.

http://software-engineering-book.com/web/spiral-model/

Im Prinzip gehen aus jeder Phase des Wasserfallmodells ein oder mehrere Dokumente hervor, die genehmigt oder abgenommen werden. Die nächste Phase sollte nicht beginnen, bevor nicht die vorherige abgeschlossen wurde. Bei der Hardwareentwicklung, wo höhere Herstellungskosten anfallen, ist dies sinnvoll. Bei der Softwareentwicklung überlappen sich die Phasen jedoch und tauschen Informationen untereinander aus. So werden während des Entwurfs Probleme mit den Anforderungen entdeckt; während des Programmierens fallen Fehler im Entwurf auf usw. Der Softwareprozess ist in der Praxis nie ein einfacher linearer Prozess, sondern umfasst Rückkopplungen von einer Phase zur vorherigen. Wenn in einer Phase neue Informationen auftauchen, sollten die Dokumente, die in früheren Phasen erzeugt wurden, abgewandelt werden, um die geforderten Systemänderungen zu reflektieren. Falls beispielsweise entdeckt wird, dass die Implementierung einer Anforderung zu teuer wird, dann sollte das Pflichtenheft abgeändert werden, um diese Anforderung zu entfernen. Dazu ist jedoch die Zustimmung des Kunden erforderlich, was den gesamten Entwicklungsprozess verzögert.

Daher können sowohl der Kunde als auch die Entwickler die Softwarespezifikation vorzeitig einfrieren, sodass keine weiteren Änderungen mehr vorgenommen werden. Leider bedeutet dies, dass Probleme zurückgestellt, ignoriert oder im Programm umgangen werden. Das verfrühte Einfrieren von Anforderungen kann allerdings bedeuten, dass das System nicht das tun wird, was der Benutzer will. Es kann auch zu einem schlecht strukturierten System führen, weil Probleme im Entwurf durch Tricks in der Implementierung umgangen werden.

Während der letzten Phase des Lebenszyklus (Betrieb und Wartung) wird die Software in Betrieb genommen. Dabei werden Fehler und Lücken in den ursprünglichen Anforderungen entdeckt. Es tauchen Programm- und Entwurfsfehler auf und der Bedarf an neuer Funktionalität wird festgestellt. Das System muss sich also weiterentwickeln, um nützlich zu bleiben. Das Durchführen dieser Veränderungen (die Softwarewartung) kann zur Folge haben, dass einige oder alle der vorherigen Prozessphasen wiederholt werden müssen.

Tatsächlich muss Software während der Entwicklungszeit flexibel sein und auf Änderungen reagieren. Die Notwendigkeit der frühen Festlegung und der Systemüberarbei-

tung bei Änderungen führt dazu, dass das Wasserfallmodell nur für einige Systemarten angemessen ist:

- Eingebettete Systeme, bei denen die Software mit Hardwaresystemen verbunden sein muss. Aufgrund der Inflexibilität von Hardware ist es in der Regel nicht möglich, Entscheidungen zur Softwarefunktionalität bis zur Implementierung hinauszuschieben.
- Kritische Systeme, bei denen eine umfangreiche Betriebs- und Datensicherheitsanalyse von Softwarespezifikation und -entwurf nötig sind. In diesen Systemen müssen die Spezifikations- und Entwurfsdokumente vollständig sein, damit diese Analyse durchgeführt werden kann. Sicherheitsrelevante Probleme bei Spezifikation und Entwurf sind in der Regel sehr teuer, wenn sie in der Implementierungsphase korrigiert werden.
- Große Softwaresysteme, die Teil von größeren technischen Systemen sind, welche von mehreren Partnerfirmen entwickelt werden. Die Hardware in den Systemen wird vielleicht mithilfe eines ähnlichen Modells entwickelt und es ist für die Unternehmen leichter, ein gemeinsames Modell für Hardware und Software einzusetzen. Wenn darüber hinaus mehrere Firmen involviert sind, kann es sein, dass vollständige Spezifikationen benötigt werden, um die unabhängige Entwicklung von unterschiedlichen Teilsystemen zu ermöglichen.

Das Wasserfallmodell ist kein geeignetes Prozessmodell, wenn informale Teamkommunikation möglich ist und sich Softwareanforderungen schnell ändern. Für diese Systeme sind iterative Entwicklung und agile Methoden besser.
Eine wichtige Variante des Wasserfallmodells ist die formale Systementwicklung, wobei ein mathematisches Modell einer Systemspezifikation erzeugt wird. Dieses Modell wird dann mithilfe von mathematischen Transformationen, die konsistenzerhaltend sind, zu ausführbarem Code verfeinert. Formale Entwicklungsprozesse wie solche, die auf der B-Methode (Abrial, 2005, 2010) basieren, werden hauptsächlich für die Entwicklung von Softwaresystemen mit strengen Betriebssicherheits-, Zuverlässigkeits- und Informationssicherheitsanforderungen eingesetzt. Der formale Ansatz erleichtert es den Entwicklern, den Nachweis zur Betriebs- und Informationssicherheit zu erstellen. Dies zeigt dem Kunden oder den Zertifikationsstellen, dass das System tatsächlich die Anforderungen an die Betriebs- oder Informationssicherheit erfüllt. Aufgrund der hohen Entwicklungskosten einer formalen Spezifikation wird dieses Entwicklungsmodell jdeoch außer für kritische Systeme selten verwendet.

2.1.2 Inkrementelle Entwicklung

Die inkrementelle Entwicklung basiert darauf, eine Anfangsimplementierung zu entwickeln, Feedback von den Benutzern und anderen zu erhalten und die Software über mehrere Versionen hinweg zu verbessern, bis das erwünschte System entstanden ist (▶ Abbildung 2.2). Die Spezifikation, die Entwicklung und die Validierung werden nicht als separate Abläufe betrachtet, sondern werden gleichzeitig ausgeführt, wobei sie untereinander Rückmeldungen zügig austauschen.

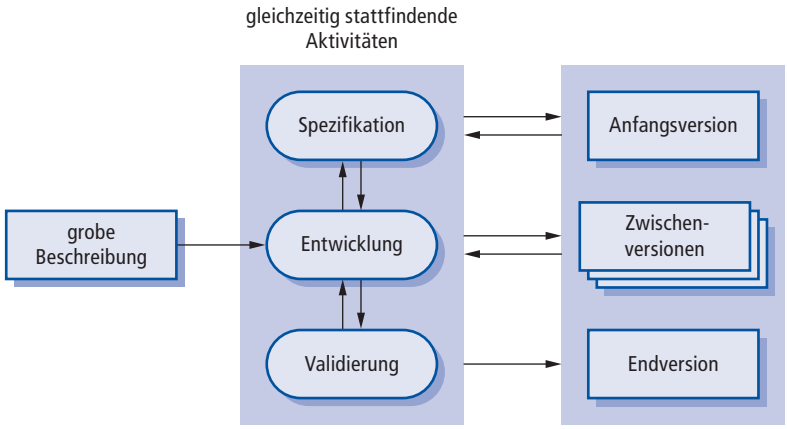

Abbildung 2.2: Inkrementelle Entwicklung.

Inkrementelle Entwicklung in der einen oder anderen Form ist heute die am häufigsten eingesetzte Vorgehensweise für die Entwicklung von Anwendungssystemen und Softwareprodukten. Dieser Ansatz kann entweder plangesteuert, agil oder – am häufigsten jedoch – eine Mischung dieser Methoden sein. In einem plangesteuerten Ansatz werden die Systeminkremente im Voraus bestimmt. Bei einem agilen Vorgehen werden zwar die frühen Inkremente ermittelt, die Entwicklung der späteren Inkremente hängt jedoch vom Projektfortschritt und von den Prioritäten des Kunden ab.
Inkrementelle Softwareentwicklung ist ein fundamentaler Teil von agilen Entwicklungsmethoden; sie ist besser als ein Wasserfallansatz für Systeme, deren Anforderungen sich vermutlich während des Entwicklungsprozesses ändern. Dies trifft für die meisten Geschäftssysteme und Softwareprodukte zu. Inkrementelle Entwicklung bildet die Art und Weise ab, wie wir Probleme lösen. Wir erarbeiten selten eine vollständige Problemlösung im Voraus, sondern bewegen uns schrittweise in Richtung einer Lösung und gehen einen Schritt zurück, wenn wir feststellen, dass wir einen Fehler gemacht haben. Beim inkrementellen Entwickeln der Software ist es billiger und einfacher Änderungen einzuarbeiten als bei schon fertiger Software.
Jedes Inkrement oder jede Version des Systems besitzt einen Teil der Funktionalität, die vom Kunden gebraucht wird. In der Regel enthalten die frühen Systeminkremente die wichtigsten oder am dringendsten benötigten Funktionen. Dies bedeutet, dass der Kunde oder Anwender das System zu einem relativ frühen Zustand in der Entwicklung evaluieren kann, um festzustellen, ob es den Anforderungen entspricht. Falls dies nicht der Fall ist, dann muss nur das aktuelle Inkrement geändert werden und es müssen eventuell neue Funktionen für spätere Inkremente definiert werden.
Inkrementelle Entwicklung hat gegenüber dem Wasserfallmodell drei wesentliche Vorteile:

1. Die Kosten für das Implementieren von Anforderungsänderungen werden reduziert. Der Umfang der wiederholt durchzuführenden Analyse und Dokumentation ist deutlich geringer als beim Wasserfallmodell.

2. Es ist einfacher, Rückmeldungen der Kunden zu bereits fertiggestellten Teilen der Entwicklungsarbeit zu bekommen. Sie können sich bei Softwaredemonstrationen dazu äußern und sehen, wie viel implementiert wurde. Den Fortschritt anhand von Softwareentwurfsdokumenten zu beurteilen, fällt Kunden dagegen schwer.

3 Eine frühe Auslieferung und Installation von verwendungsfähiger Software an den Kunden ist selbst dann möglich, wenn noch nicht die gesamte Funktionalität enthalten ist. Die Kunden können die Software früher verwenden und daraus Nutzen ziehen, als es mit einem Wasserfallmodell möglich wäre.

Die inkrementelle Entwicklung hat allerdings aus Managementsicht zwei Schwachpunkte:

1 Der Prozess ist nicht sichtbar: Manager brauchen in regelmäßigen Abständen Zwischenversionen, an denen sie den Fortschritt messen können. Wenn Systeme schnell entwickelt werden, ist es nicht kosteneffektiv, jede Version zu dokumentieren.

2 Die Systemstruktur wird tendenziell schwächer, wenn neue Inkremente hinzugefügt werden. Stetige Veränderungen führen zu unsauberem Code, wenn neue Funktionalität auf alle möglichen Arten hinzukommt. Es wird zunehmend schwierig und teuer, einem System neue Funktionen hinzuzufügen. Um strukturelle Degradierung und allgemeines Codechaos zu reduzieren, empfehlen agile Methoden, die Software regelmäßig zu refaktorieren (also zu verbessern und neu zu strukturieren).

Bei großen, komplexen Systemen mit einer langen Lebensdauer werden die Probleme der inkrementellen Entwicklung schnell akut, wenn verschiedene Teams verschiedene Teile des Systems entwickeln. Große Systeme benötigen einen stabilen Rahmen oder Architektur und die Verantwortlichkeiten von unterschiedlichen Teams, die an Teilen des Systems arbeiten, müssen bezüglich dieser Architektur klar festgelegt werden. Dies muss im Voraus geplant statt inkrementell entwickelt werden.
Inkrementelle Entwicklung bedeutet nicht, dass jedes Inkrement an den Kunden des Systems ausgeliefert werden muss. Man kann ein System inkrementell entwickeln und es den Kunden und anderen Beteiligten zum Kommentieren vorführen, ohne es notwendigerweise auszuliefern und in der Umgebung des Kunden einzurichten. Inkrementelle Auslieferung (*Abschnitt 2.3.2*) bedeutet, dass die Software in realen betrieblichen Prozessen verwendet wird, sodass das Feedback der Anwender wahrscheinlich realistisch ist. Rückmeldungen zu bekommen ist nicht immer möglich, da das Ausprobieren der neuen Software die normalen Geschäftsabläufe stören kann.

Probleme bei der inkrementellen Entwicklung

Obwohl die inkrementelle Entwicklung viele Vorteile hat, so ist sie doch nicht ganz ohne Probleme. Der Hauptgrund für die Schwierigkeiten ist die Tatsache, dass große Organisationen bürokratische Handlungsabläufe besitzen, die sich im Laufe der Zeit entwickelt haben und die unter Umständen nicht mit einem informelleren iterativen oder agilen Prozess harmonieren.

Manchmal gibt es einen guten Grund für diese Handlungsabläufe – zum Beispiel um sicherzustellen, dass die Software gesetzliche Regelungen ordnungsgemäß umsetzt (z. B. die Vorschriften zur Rechnungslegung gemäß dem Sarbanes-Oxley Act in den Vereinigten Staaten). Diese Geschäftsabläufe zu verändern ist nicht immer möglich, sodass Konflikte zwischen den Prozessen unvermeidbar sind.

http://software-engineering-book.com/web/incremental-development/

2.1.3 Integration und Konfiguration

In einem Großteil aller Softwareprojekte wird Software wiederverwendet. Häufig geschieht dies informell, wenn die Mitarbeiter eines Projekts von Code wissen oder danach suchen, der dem ähnelt, der gebraucht wird. Sie suchen ihn heraus, verändern ihn nach ihren Bedürfnissen und integrieren ihn mit den neuen Code, den sie geschrieben haben.
Diese informelle Wiederverwendung findet unabhängig von dem eingesetzten Entwicklungsprozess statt. Seit dem Jahr 2000 wurden jedoch Vorgehensmodelle der Softwareentwicklung immer populärer, die den Schwerpunkt auf die Wiederverwendung bereits vorhandener Software legten. Wiederverwendungsorientierte Ansätze beruhen auf einer großen Menge wiederverwendbarer Softwarekomponenten und auf einem Integrationsrahmen für die Zusammenstellung diese Komponenten.
Drei Arten von Software werden häufig wiederverwendet:

1. Eigenständige Anwendungen, die für die Benutzung in einer bestimmten Umgebung konfiguriert wurden. Dies sind Allzwecksysteme mit vielen Funktionen, aber sie müssen für den Einsatz in einer speziellen Anwendung angepasst werden.

2. Sammlungen von Objekten, die als eine Komponente oder als ein Paket entwickelt werden, um mit Komponenten-Framworks wie Java Spring integriert zu werden (Wheeler und White, 2013).

3. Webdienste, die im Hinblick auf Servicestandards entwickelt werden und die für entfernte Aufrufe über das Internet verfügbar sind.

▶ Abbildung 2.3 zeigt ein allgemeines Prozessmodell für wiederverwendungsbasierte Entwicklung, das auf Integration und Konfiguration beruht. Die Phasen in diesem Prozess sind:

1. *Anforderungsspezifikation:* Die anfänglichen Anforderungen für das System werden vorgeschlagen. Diese müssen nicht im Detail ausgearbeitet sein, aber sollten kurze Beschreibungen der wesentlichen Anforderungen und der erwünschten Systemfunktionen enthalten.

2. *Softwareerkennung und -evaluierung:* Ausgehend von einem Entwurf der Softwareanforderungen wird eine Suche nach Komponenten und Systemen durchgeführt, die die geforderte Funktionalität bieten. Mögliche Komponenten und Systeme werden bewertet, um festzustellen, ob sie die wesentlichen Anforderungen erfüllen und ob sie grundsätzlich für die Verwendung in dem System geeignet sind.

3. *Anforderungsnachbesserung:* Während dieser Phase werden die Anforderungen nachgebessert mithilfe von Informationen über die wiederverwendbaren Komponenten und Anwendungen, die gefunden wurden. Die Anforderungen werden modifiziert, damit sie auf die verfügbaren Komponenten angepasst sind, und die Systemspezifikation wird neu definiert. Wo Veränderungen unmöglich sind, kann die Aktivität der Komponentenanalyse erneut durchgeführt werden, um nach alternativen Lösungen zu suchen.

4 *Konfiguration des Anwendungssystems*: Falls ein passendes System verfügbar ist, das die Anforderungen erfüllt, kann dieses entsprechend konfiguriert werden und so das neue System erzeugen.

5 *Anpassen und Integration von Komponenten:* Gibt es kein passendes System, dann können einzelne wiederverwendbare Komponenten modifiziert und neue Komponenten entwickelt werden. Diese werden dann integriert, um das System zu erzeugen.

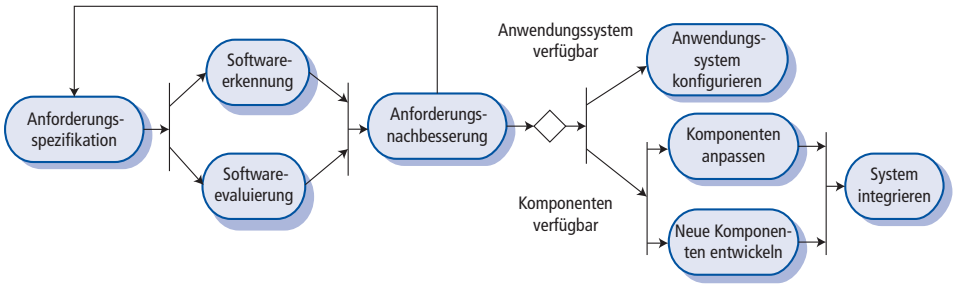

Abbildung 2.3: Wiederverwendungsorientiertes Software-Engineering.

Wiederverwendungsorientiertes Software-Engineering, das auf Konfiguration und Integration basiert, hat den offensichtlichen Vorteil, dass es die Menge an zu entwickelnder Software und somit auch die Kosten und Risiken verringert. Außerdem wird die Software schneller geliefert. Kompromisse bei den Anforderungen sind jedoch unvermeidbar, und das kann zu einem System führen, das die wirklichen Bedürfnisse des Benutzers nicht erfüllt. Außerdem geht ein Teil der Kontrolle über die Weiterentwicklung des Systems verloren, da sich neue Versionen wiederverwendeter Komponenten der Kontrolle der Organisation entziehen, die sie benutzt.
Die Wiederverwendung von Software ist sehr wichtig, daher habe ich mehrere Kapitel im dritten Teil des Buches diesem Thema gewidmet. Allgemeine Probleme der Wiederverwendung von Software werden in *Kapitel 15* behandelt, komponentenbasiertes Software-Engineering in *Kapitel 16* und *17* und serviceorientierte Systeme in *Kapitel 18*.

2.2 Prozessaktivitäten

Reale Softwareprozesse sind überlappende Abfolgen von technischen, auf Zusammenarbeit basierenden und betriebswirtschaftlichen Aktivitäten mit dem Gesamtziel der Spezifikation, des Entwurfs, der Implementierung und des Testens eines Softwaresystems. Heutzutage werden Prozesse in der Regel durch Werkzeuge unterstützt. Das heißt, dass Softwareentwickler eine Vielzahl verschiedener Softwarewerkzeuge als Hilfe benutzen können, zum Beispiel Anforderungsmanagementsysteme, Editoren zur Entwurfsmodellierung, Programmeditoren, automatische Testwerkzeuge und Debugger. Solche Werkzeuge unterstützen besonders das Bearbeiten verschiedener Dokumenttypen und die Verwaltung des immensen Umfangs von detaillierten Informationen, die in einem großen Softwareprojekt erzeugt werden.
Die vier grundlegenden Prozessaktivitäten der Spezifikation, Entwicklung, Validierung und Evolution sind in verschiedenen Entwicklungsprozessen unterschiedlich

aufgeteilt. Im Wasserfallmodell werden sie nacheinander, während sie in der inkrementellen Entwicklung gleichzeitig ausgeführt werden. Wie diese Aktivitäten ausgeführt werden, hängt vom Typ der zu erstellenden Software ab, der Erfahrung und Kompetenz der Entwickler und der Art der Organisation, die die Software entwickelt.

Werkzeuge für die Softwareentwicklung

Werkzeuge zur Softwareentwicklung sind Programme, die eingesetzt werden, um Softwareprozesse zu unterstützen. Zu diesen Werkzeugen zählen Anforderungsmanagementwerkzeuge, Entwurfseditoren, Werkzeuge zur Unterstützung von Refactoring, Compiler, Debugger, Fehlerdatenbanken und Werkzeuge für die Systemerstellung.

Softwarewerkzeuge unterstützen Prozesse, indem sie einige Abläufe automatisieren und Informationen über die zu entwickelnde Software verfügbar machen. Zum Beispiel:

- die Entwicklung grafischer Systemmodelle als Teil der Anforderungsspezifikation oder des Softwareentwurfs;
- die Codeerzeugung aus diesen grafischen Modellen;
- die Erzeugung von Benutzeroberflächen durch eine grafische Oberflächenbeschreibung, die interaktiv vom Benutzer erstellt wird;
- die Behebung von Programmfehlern durch die Bereitstellung entsprechender Informationen über ein laufendes Programm;
- die automatische Übersetzung von Programmen, die in einer alten Version einer Programmiersprache geschrieben sind, in eine neuere Version.

Werkzeuge können mit einem Framework namens Interactive Development Environment oder IDE kombiniert werden. Eine IDE stellt Funktionalität zur Verfügung, die Werkzeuge benutzen können, um einfacher miteinander kommunizieren und arbeiten zu können.

http://software-engineering-book.com/web/software-tools/

2.2.1 Softwarespezifikation

Unter der Softwarespezifikation bzw. Anforderungsanalyse versteht man den Prozess zum Verstehen und Definieren der vom System verlangten Funktionen sowie der Beschränkungen, denen der Betrieb und die Entwicklung des Systems unterliegen. Die Anforderungsanalyse ist eine besonders kritische Phase des Softwareprozesses, da die hier gemachten Fehler unweigerlich zu späteren Problemen beim Entwurf und der Implementierung des Systems führen.

Bevor der Prozess der Anforderungsanalyse beginnt, möchte ein Unternehmen eventuell eine Durchführbarkeits- oder Marketingstudie vornehmen, um festzustellen, ob es einen Bedarf oder einen Markt für die Software gibt und ob es technisch und finanziell realstisch ist, die Software zu entwickeln. Durchführbarkeitsstudien sind kurze, relativ kostengünstige Studien, die eine fundierte Entscheidung darüber ermöglichen, ob eine detailliertere Analyse sinnvoll ist oder nicht.

Die Anforderungsanalyse (▶ Abbildung 2.4) hilft, ein vereinbartes Anforderungsdokument zu erstellen, das ein System spezifiziert, welches die Anforderungen der Projektbeteiligten erfüllt. Die Anforderungen werden normalerweise auf zwei Detailstufen

dargestellt. Die Endbenutzer und Kunden brauchen eine grobe Aufstellung der Anforderungen, während die Systementwickler eine detailliertere Systemspezifikation benötigen.

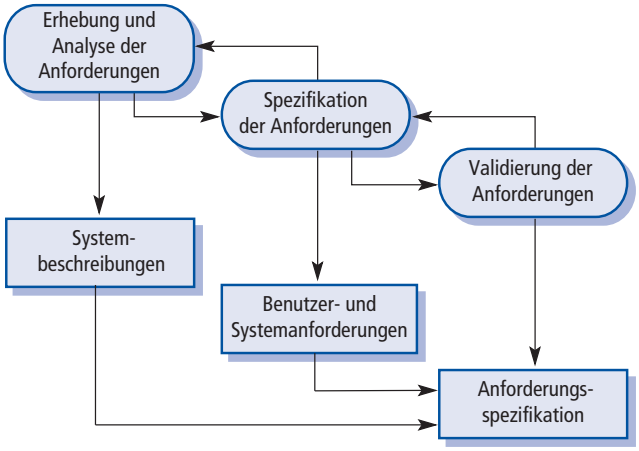

Abbildung 2.4: Ablauf der Anforderungsanalyse.

Die Anforderungsanalyse besteht aus drei wesentlichen Aktivitäten:

1. *Erhebung und Analyse der Anforderungen:* In dieser Phase werden die Systemanforderungen aus der Beobachtung bestehender Systeme, Diskussionen mit potenziellen Benutzern und Käufern, der Aufgabenanalyse usw. abgeleitet. Dazu kann die Entwicklung eines oder mehrerer unterschiedlicher Systemmodelle und Prototypen gehören. Diese helfen Ihnen, das zu spezifizierende System besser zu verstehen.

2. *Spezifikation der Anforderungen:* Dies ist die Aktivität, bei der die während der Anforderungsanalyse gesammelten Informationen in ein Dokument umgesetzt werden, das einen Satz von Anforderungen definiert. In diesem Dokument können zwei Arten von Anforderungen aufgenommen werden: Benutzeranforderungen sind abstrakte Beschreibungen der Systemanforderungen für den Kunden und den Endnutzer, während es sich bei den eigentlichen Systemanforderungen um eine detailliertere Auflistung der Funktionen handelt, die zur Verfügung gestellt werden sollen.

3. *Validierung der Anforderungen:* Diese Aktivität prüft, ob die Anforderungen realistisch, konsistent und vollständig sind. Während dieses Prozesses werden Fehler im Anforderungsdokument unweigerlich erkannt. Es muss dann verändert werden, um diese Probleme zu korrigieren.

Die Analyse der Anforderungen wird während der Definition und der Spezifikation weitergeführt, dabei kommen oft neue Anforderungen zum Vorschein. Die Analyse, die Definition und die Spezifikation sind daher eng miteinander verknüpft.

Bei agilen Methoden ist die Anforderungsspezifikation keine separate Aktivität, sondern wird als Teil der Systementwicklung angesehen. Anforderungen werden für jedes Systeminkrement erst kurz vor Beginn dessen Entwicklung informell spezifiziert. Die Anforderungen werden entsprechend den Benutzerprioritäten definiert. Die Bestimmung der Anforderungen erfolgt durch Benutzer, die Teil des Entwicklerteams sind oder eng mit diesem zusammenarbeiten.

2.2.2 Softwareentwurf und -implementierung

Die Implementierungsphase der Softwareentwicklung ist ein Prozess, in dem ein ausführbares System zur Auslieferung an den Kunden entwickelt wird. Manchmal sind dabei die Aktivitäten Softwareentwurf und Promgrammierung getrennt voneinander, doch wenn ein agiler Ansatz benutzt wird, sind Entwurf und Implementierung miteinander verwoben. Es wird dann kein formales Entwurfsdokument während des Prozesses erzeugt. Selbstverständlich gibt es trotzdem einen Entwurf der Software, aber dieser wird informell auf Whiteboards oder in Form von persönlichen Notizen der Programmierer festgehalten.

Ein Softwareentwurf ist eine Beschreibung der Struktur der zu implementierenden Software, der Datenmodelle und -strukturen, die vom System benutzt werden, der Schnittstellen zwischen den Systemkomponenten und manchmal der verwendeten Algorithmen. Die Entwickler stellen nicht sofort einen fertigen Entwurf auf, sondern entwickeln diesen Entwurf schrittweise. Während der Entwicklung wird der Entwurf um Details erweitert, wobei ständig modifizierend auf frühere Entwürfe zurückgegriffen wird.

▶ Abbildung 2.5 ist ein abstraktes Modell des Entwurfsprozesses, in dem die Eingaben des Entwurfsprozesses, Prozessaktivitäten und Prozessausgaben zu sehen sind. Die Aktivitäten des Entwurfsprozesses sind sowohl miteinander verwoben als auch voneinander unabhängig. Es werden ständig neue Informationen über den Entwurf erzeugt, wodurch bisherige Entwurfsentscheidungen betroffen werden. Die Überarbeitung des Entwurfs ist bei allen Entwurfsprozessen unvermeidbar.

Die meisten Softwareprodukte haben Schnittstellen zu anderen Softwaresystemen. Zu diesen anderen Systemen gehören das Betriebssystem, Datenbanken, Middleware und andere Anwendungssysteme. Diese bilden die „Softwareplattform“, die Umgebung, in der die Software ausgeführt wird. Informationen über diese Plattform sind eine unverzichtbare Eingabe in den Entwurfsprozess, da die Entwickler entscheiden müssen, wie die Software am besten mit ihrer Umgebung integriert wird. Wenn das System vorhandene Daten verarbeiten soll, dann kann die Beschreibung dieser Daten innerhalb der Plattformspezifikation erfolgen. Anderenfalls muss die Datenbeschreibung eine Eingabe des Entwurfsprozesses sein, sodass die Struktur der Systemdaten definiert werden kann.

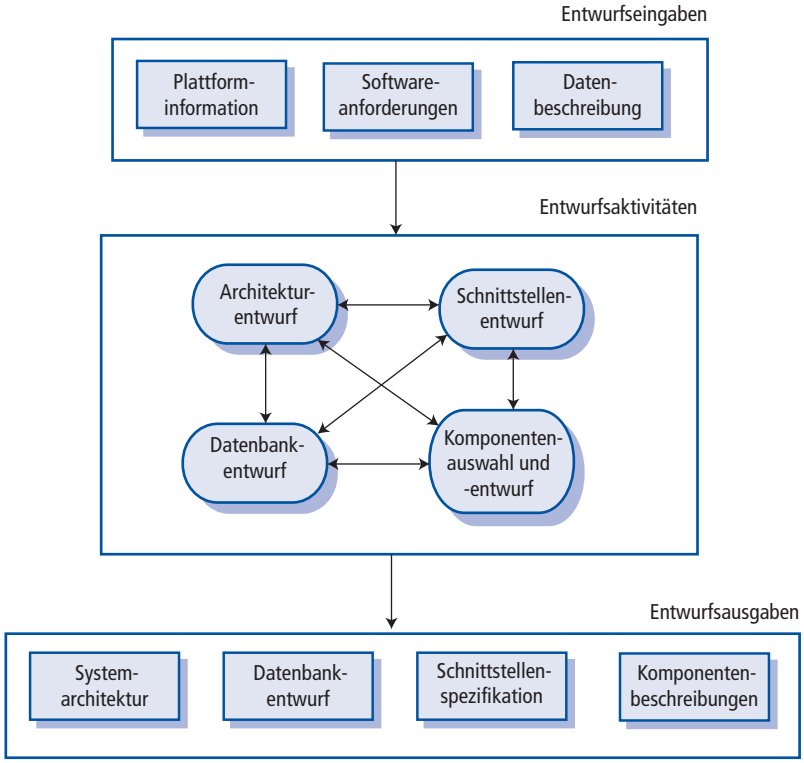

Abbildung 2.5: Ein allgemeines Modell des Entwurfsprozesses.

Die Aktivitäten in dem Entwurfsprozess variieren je nach Art des zu entwickelnden Systems. Bei **Echtzeitsystemen** muss beispielsweise das Zeitverhalten im Entwurf berücksichtigt werden, dagegen wird möglicherweise keine Datenbank und somit kein Datenbankentwurf benötigt. Abbildung 2.5 zeigt vier Aktivitäten, die Teil des Entwurfsprozesses für Informationssysteme sein können:

1. *Architekturentwurf:* Hier werden die Gesamtstruktur des Systems, die Hauptkomponenten (manchmal Subsysteme oder Module genannt) und ihre Verbindungen sowie deren Verteilung bestimmt.

2. *Datenbankentwurf:* Hier werden die Strukturen der Systemdaten entworfen und festgelegt, wie diese in einer Datenbank abgebildet werden sollen. Auch hier hängt der Arbeitsaufwand davon ab, ob eine vorhandene Datenbank wiederverwendet oder ob eine neue Datenbank erzeugt werden soll.

3. *Schnittstellenentwurf:* Hier definiert man die Schnittstellen zwischen den Systemkomponenten. Diese Schnittstellenspezifikation muss absolut eindeutig sein. Eine präzise Schnittstelle führt dazu, dass eine Komponente von anderen benutzt werden kann, ohne Kenntnisse über die Implementierung dieser Komponente zu haben. Wenn eine Schnittstellenspezifikation vereinbart wurde, können die Komponenten einzeln entworfen und entwickelt werden.

4 *Komponentenauswahl und -entwurf:* Hier wird nach wiederverwendbaren Komponenten gesucht und – falls keine passenden verfügar sind – neue Softwarekomponenten entworfen. In dieser Phase kann der Entwurf eine einfache Beschreibung der Komponenten sein – die Implementierungsdetails werden dem Programmierer überlassen. Das Entwurfsmodell kann dann zur automatischen Erzeugung einer Implementierung verwendet werden.

All diese Aktivitäten führen zu einer Reihe von Ausgaben, die ebenfalls in Abbildung 2.5 zu sehen sind. Bei kritischen Systemen sind die Ausgaben des Entwurfsprozesses detaillierte Entwurfsdokumentationen, welche präzise und akkurate Beschreibungen des Systems liefern. Wenn ein modellgetriebener Ansatz benutzt wird (*Kapitel 5*), sind diese Ausgaben Entwurfsdiagramme. Wenn agile Entwicklungsmethoden verwendet werden, erfolgt die Ausgabe des Entwurfsprozesses nicht in Form separater Spezifikationsdokumente, sondern zunehmend als Darstellung im Programmcode.

Wenn ein System implementiert werden soll, folgt die Entwicklung eines Programms ganz natürlich aus dem Systementwurf. Obwohl einige Klassen von Programmen wie sicherheitsrelevante Systeme detailliert entworfen werden, bevor die Implementierung beginnt, kommt es häufiger vor, dass Entwurf und Programmentwicklung verknüpft werden. So können Softwareentwicklungswerkzeuge zum Einsatz kommen, um aus einem Entwurf ein Programmgerüst zu erzeugen. Dabei müssen Schnittstellen definiert und implementiert werden, und in vielen Fällen braucht der Programmierer nur Details zur Funktionalität jeder Programmkomponente beizusteuern.

Die Programmierung ist eine persönlich gestaltete Aktivität und es gibt dafür keinen allgemeingültigen Prozessverlauf. Manche Programmierer fangen mit Komponenten an, die sie bereits vollständig verstehen, entwickeln diese und gehen dann zu weniger eindeutigen Komponenten über. Andere gehen den entgegengesetzten Weg und heben sich bekannte Komponenten bis zuletzt auf, weil sie wissen, wie sie entwickelt werden. Einige Entwickler definieren ihre Daten gerne so früh wie möglich und benutzen sie, um die Programmentwicklung voranzutreiben, während andere die Daten so lange wie möglich unspezifiziert lassen.

Normalerweise testen die Programmierer den von ihnen entwickelten Code zumindest oberflächlich selbst. Das enthüllt oft Fehler (Bugs) im Programm, die behoben werden müssen. Das Finden und Korrigieren von Programmfehlern wird Fehlerbehebung (*debugging*) genannt. Fehlertest und Fehlerbehebung sind zwei verschiedene Prozesse. Die Fehlertests stellen das Vorhandensein von Fehlern fest, wohingegen sich die Fehlerbehebung damit beschäftigt, diese Fehler zu finden und zu entfernen.

Bei der Fehlerbehebung stellen Sie Hypothesen über das sichtbare Verhalten des Programms auf und testen diese dann in der Hoffnung, den Fehler zu finden, der die Unregelmäßigkeit in der Ausgabe verursacht hat. Das Testen von Hypothesen kann bedeuten, dass der Programmcode von Hand überprüft werden muss. Mitunter sind auch neue Testfälle nötig, um das Problem zu lokalisieren. Interaktive Werkzeuge zur Fehlerbehebung (Debugger), die die Zwischenwerte von Programmvariablen anzeigen und die auszuführenden Befehle verfolgen, unterstützen den Prozess der Fehlerbehebung.

2.2.3 Softwarevalidierung

Die Softwarevalidierung oder allgemeiner die Verifikation und Validierung (V&V) soll zeigen, dass ein System sowohl seine Spezifikationen erfüllt als auch den Erwartungen des Kunden des Systems entspricht. Der Programmtest, bei denen das System unter Benutzung von simulierten Testdaten ausgeführt wird, ist die Hauptvalidierungstechnik. Validierung könnte auch Überprüfungsprozesse wie **Inspektionen** und Reviews in jeder Phase des Softwareprozesses vorsehen, von der Definition der Benutzeranforderungen bis hin zur Programmentwicklung. Durch den breiten Raum, den das Testen einnimmt, entsteht der Hauptteil der Validierungskosten während und nach der Implementierung. Der Großteil an Zeit und Aufwand bei V & V wird allerdings mit Testen verbracht.
Außer bei kleinen Programmen sollten Systeme nicht als eine einzige große Einheit getestet werden. ▶ Abbildung 2.6 zeigt einen Testprozess in drei Phasen, in dem zuerst die Systemkomponenten individuell und dann das integrierte System getestet wird. Für kundenspezifische Software werden die Kundentests mit realen Daten des Kunden durchgeführt. Bei Produkten, die als Anwendungen verkauft werden, heißen die Kundentests auch manchmal Betatests: ausgewählte Benutzer probieren die Software aus und geben Kommentare dazu ab.

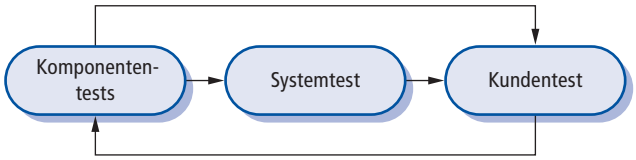

Abbildung 2.6: Testphasen.

Die Phasen des Testprozesses sind die folgenden:

1. *Komponententests:* Die Komponenten, die das Gesamtsystem bilden, werden von den gleichen Personen getestet, die auch das System entwickeln. Jede Komponente wird unabhängig von anderen Systemkomponenten getestet. Komponenten können einfache Einheiten sein, wie z. B. Funktionen oder **Objektklassen**, oder aber zusammenhängende Gruppen dieser Einheiten. Werkzeuge zur Testautomatisierung wie JUnit für **Java**, die Tests erneut ausführen können, wenn neue Versionen der Komponenten erzeugt werden, sind weitverbreitet (Koskela, 2013).

2. *Systemtest:* Die Systemkomponenten werden integriert und bilden ein vollständiges System. Dieser Prozess beschäftigt sich mit dem Auffinden von Fehlern, die aus unerwarteten Interaktionen zwischen den Komponenten entstehen, sowie mit Schnittstellenproblemen zwischen den Komponenten. Er befasst sich auch mit dem Nachweis, dass das System seine **funktionalen** und **nichtfunktionalen Anforderungen** erfüllt. Außerdem werden die grundlegenden Systemeigenschaften getestet. Bei großen Systemen kann es sich hierbei um einen mehrstufigen Prozess handeln, bei dem Komponenten integriert werden, um Subsysteme zu bilden, die wiederum einzeln getestet werden, bevor sie integriert werden und somit das fertige System bilden.

3. *Kundentest:* Dies ist die letzte Phase des Testprozesses, bevor das System für den Betrieb freigegeben wird. Dieser Test wird statt mit simulierten Testdaten mit Daten durchgeführt, die der Kunde (bzw. der potenzielle Kunde) zur Verfügung

stellt. Bei nach Kundenwunsch erstellter Software kann der Kundentest Fehler und Lücken in der Definition der Systemanforderungen enthüllen, wenn die echten Daten zu anderen Systemabläufen führen als die Testdaten der Systementwickler. Außerdem kann der Test auch Anforderungsprobleme aufzeigen, wenn die Funktionen des Systems die Bedürfnisse des Benutzers nicht vollständig erfüllen oder die Systemleistung inakzeptabel ist. Bei Produkten zeigt der Kundentest, wie gut das Softwareprodukt die Bedürfnisse der Kunden erfüllt.

Im Idealfall werden Fehler in den Komponenten früh wärend des Testens und Schnittstellenprobleme spätestens bei der Systemintegration gefunden. Wann immer jedoch Fehler entdeckt werden, müssen sie im Programm behoben werden, und das kann bedeuten, dass andere Testphasen zu wiederholen sind. Fehler in Programmkomponenten können zum Beispiel während der Integrationstests ans Tageslicht kommen. Der Testprozess ist also iterativ, wobei Informationen aus den späteren Phasen als Feedback für die früheren Prozessabschnitte verwendet werden können.

Normalerweise sind Komponententests einfach Teil des typischen Entwicklungsprozesses. Die Programmierer stellen ihre eigenen Testdaten her und testen den Code, während er entwickelt wird. Der Programmierer kennt die Komponente und ist somit am besten für das Erzeugen von Testdaten qualifiziert.

Bei einem inkrementellen Entwicklungsansatz wird jedes Teilsystem während der Entwicklungsphase getestet, wobei die Tests auf Grundlage der für dieses Teilsystem geltenden Anforderungen erfolgen. Bei der testgetriebenen Entwicklung, die ein normaler Teil von agilen Prozessen ist, werden Tests zusammen mit den Anforderungen entwickelt, bevor die Entwicklung beginnt. Dies hilft den Testern und Entwicklern, die Anforderungen zu verstehen, und garantiert, dass es beim Aufstellen der Testfälle zu keinen Verzögerungen kommt.

Wenn ein plangesteuerter Softwareprozess benutzt wird (z. B. für die Entwicklung von kritischen Systemen), wird das Testen von einer Menge von Testplänen gesteuert. Ein unabhängiges Team von Testern arbeitet mit diesen Testplänen, die aus der Spezifikation und dem Entwurf des Systems abgeleitet wurden. ▶ Abbildung 2.7 zeigt, wie die Testpläne die Testaktivitäten mit den Entwicklungsaktivitäten verbinden. Dies wird manchmal das V-Modell der Entwicklung genannt (drehen Sie die Abbildung und Sie sehen das V). Das V-Modell zeigt die jeweiligen Aktivitäten der Softwarevalidierung, die den einzelnen Phasen des Wasserfallprozessmodells entsprechen.

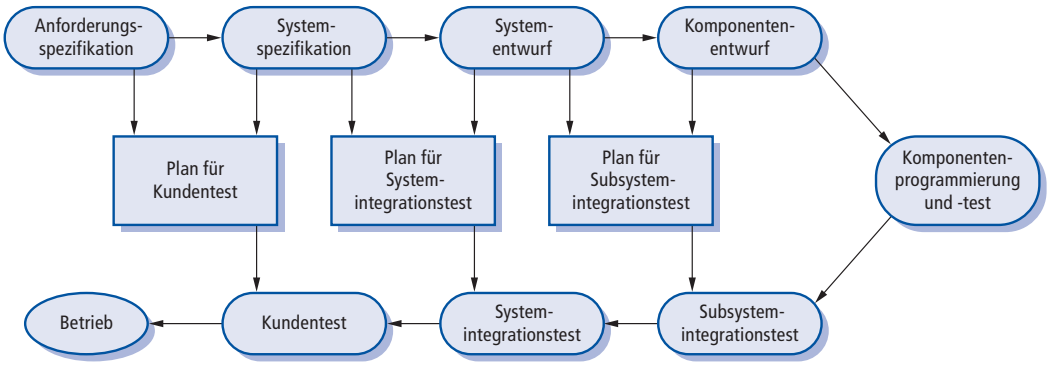

Abbildung 2.7: Testphasen in einem plangesteuerten Softwareprozess.

Wenn ein System als Softwareprodukt auf den Markt kommen soll, werden oft sogenannte Betatests durchgeführt, bei denen das System an eine Anzahl zukünftiger Kunden ausgeliefert wird, die sich verpflichten, das System zu benutzen und den Softwareentwicklern Probleme zu melden. Dadurch wird das System echter Nutzung ausgesetzt und es können Probleme gefunden werden, die die Produktentwickler nicht voraussehen konnten. Das Softwareprodukt kann den Rückmeldungen entsprechend verändert und für weitere Betatests oder für den Verkauf freigegeben werden.

2.2.4 Weiterentwicklung von Software

Die Flexibilität von Software ist einer der Hauptgründe, warum immer mehr Software in große, komplexe Systeme eingebunden wird. Sobald die Entscheidung getroffen ist, Hardware herzustellen, ist es sehr teuer, am Hardwareentwurf Veränderungen vorzunehmen. An der Software können jedoch jederzeit während oder sogar nach der Systementwicklung Veränderungen vorgenommen werden. Selbst gravierende Änderungen sind immer noch billiger als entsprechende Änderungen an der Systemhardware.

In der Vergangenheit hat es immer eine Trennung zwischen dem Prozess der Softwareentwicklung und dem der Softwareweiterentwicklung (Softwarewartung) gegeben. Die Softwareentwicklung wird als ein kreativer Vorgang angesehen, in dem ein Softwaresystem von einem Anfangskonzept bis hin zu einem lauffähigen System entwickelt wird. Die Softwarewartung wird hingegen manchmal als langweilig und uninteressant eingestuft. Sie wird als weniger herausfordernd betrachtet als die ursprüngliche Softwareentwicklung.

Diese Trennung zwischen Entwicklung und Wartung wird immer unwichtiger. Nur wenige Softwaresysteme sind völlig neue Systeme und daher ist es wesentlich sinnvoller, Entwicklung und Wartung als einen kontinuierlichen Vorgang zu betrachten. Anstatt sich zwei getrennte Prozesse vorzustellen, ist es realistischer, das Software-Engineering als einen evolutionären Vorgang (▶ Abbildung 2.8) zu betrachten, in dem die Software während ihrer Lebenszeit ständig verändert wird, um sich wandelnden Anforderungen und Kundenbedürfnissen anzupassen.

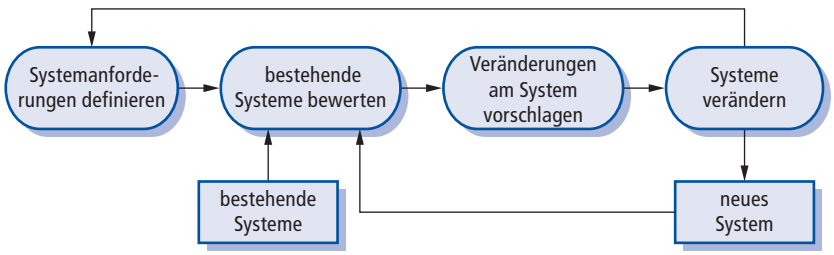

Abbildung 2.8: Weiterentwicklung eines Softwaresystems.

2.3 Umgang mit Änderungen

Bei allen großen Softwareprojekten sind Veränderungen unvermeidlich. Die Systemanforderungen ändern sich, wenn Unternehmen auf externe Anforderungen und auf Wettbewerber reagieren müssen oder wenn sich die Prioritäten der Geschäftsführung wandeln. Mit dem Einsatz neuer Technologien ergeben sich neue Entwurfs- und Implementierungsmöglichkeiten. Welches Softwareprozessmodell daher auch immer benutzt wird – es ist unabdingbar, dass Änderungen an der sich in Entwicklung befindenden Software eingepflegt werden können.
Änderungen erhöhen die Kosten der Softwareentwicklung, weil dann in der Regel bereits erledigte Arbeitsschritte noch einmal durchgeführt werden müssen. Diesen Vorgang bezeichnet man als Rework. Wenn beispielsweise die Beziehungen zwischen den Anforderungen in einem System untersucht und danach neue Anforderungen festgelegt wurden, dann muss die Anforderungsanalyse teilweise oder ganz wiederholt werden. Um den neuen Anforderungen gerecht zu werden, könnte es daraufhin notwendig werden, das System neu zu entwerfen, bereits entwickelte Programme zu ändern und das System noch einmal zu testen.
Es können verwandte Ansätze benutzt werden, um die Kosten des Rework zu reduzieren:

1. *Vorwegnahme von Änderungen:* Der Softwareprozess enthält Aktivitäten, die mögliche Änderungen vorwegnehmen oder vorhersagen können, bevor nennenswertes Rework erforderlich wird. Zum Beispiel könnte ein Prototypsystem entwickelt werden, um dem Kunden einige Hauptmerkmale des Systems vorzuführen. Dieser kann dann mit dem Prototyp experimentieren und seine Anforderungen verfeinern, bevor er höheren Softwareproduktionskosten zustimmt.

2. *Toleranz der Änderungen:* Der Prozess und die Software sind so ausgelegt, dass Änderungen einfach in das System eingearbeitet werden können. Dies setzt in der Regel irgendeine Form von inkrementeller Entwicklung voraus. Die Implementierung von geplanten Änderungen kann in solche Inkremente ausgelagert werden, die bis jetzt noch nicht entwickelt wurden. Wenn dies nicht möglich ist, dann muss vielleicht nur ein einziges Inkrement (ein kleiner Teil des Systems) modifiziert werden, um die Änderungen einzuarbeiten.

In diesem Abschnitt stelle ich zwei Möglichkeiten vor, mit Änderungen und sich ändernden Systemanforderungen umzugehen.

1. *Entwicklung von Systemprototypen:* Eine Version des Systems oder eines Systemteils wird schnell entwickelt, um die Anforderungen des Kunden und die Machbarkeit von Entwurfsentscheidungen zu überprüfen. Dies ist eine Methode der Vorwegnahme von Änderungen, da es den Benutzern erlaubt, mit dem System vor der Auslieferung zu experimentieren, und sie so ihre Anforderungen verfeinern können. Die Anzahl der Anforderungen, die nach der Auslieferung verändert werden müssen, wird damit wahrscheinlich reduziert.

2. *Inkrementelle Auslieferung:* Die inkrementell erstellten Teilsysteme werden an den Kunden ausgeliefert, der damit experimentieren und Kommentare abgeben kann. Dieses Vorgehen unterstützt sowohl die Vermeidung von Änderungen als auch die Änderungstoleranz. Die vorzeitige Festlegung der Anforderungen für das gesamte System wird vermieden und Änderungen können in spätere Inkremente zu relativ geringen Kosten eingearbeitet werden.

Das Konzept des **Refactoring**, vor allem der Verbesserung der Struktur und Organisation eines Programms, ist ebenfalls ein wichtiger Mechanismus, der Änderungstoleranz unterstützt. Ich komme darauf in *Kapitel 3* zurück, wo agile Methoden behandelt werden.

2.3.1 Softwareprototypen

Ein Prototyp ist eine frühe Version eines Softwaresystems, der dazu verwendet wird, Konzepte zu demonstrieren, Entwurfsmöglichkeiten auszuprobieren und Erkenntnisse über das Problem und seine möglichen Lösungen zu gewinnen. Eine schnelle, iterative Entwicklung des Prototyps ist von entscheidender Bedeutung, um die Kosten zu begrenzen und den Systembeteiligten schon in einem frühen Stadium des Softwareprozesses etwas zum Experimentieren zu geben.
Ein Prototyp kann im Softwareentwicklungsprozess eingesetzt werden, um Änderungen vorwegzunehmen, die nötig werden könnten:

1. Beim Requirements-Engineering-Prozess hilft ein Prototyp bei der Ermittlung und Validierung der Systemanforderungen.
2. Beim Systementwurf kann der Prototyp verwendet werden, um Softwarelösungen zu untersuchen und den Entwurf der Benutzeroberfläche für das System zu unterstützen.

Anhand von Systemprototypen können die potenziellen Benutzer erkennen, wie gut das System sie bei der Arbeit unterstützt. Sie gewinnen dadurch möglicherweise neue Ideen für Anforderungen und können Stärken und Schwächen in der Software erkennen. Sie können dann neue Systemanforderungen vorschlagen. Darüber hinaus können bei der Entwicklung des Prototyps fehlerhafte und übersehene Anforderungen aufgedeckt werden. In einer Spezifikation mag eine Funktion klar und nützlich erscheinen. Wenn diese Funktion jedoch mit anderen kombiniert wird, stellen die Benutzer häufig fest, dass ihre anfängliche Auffassung falsch oder unvollständig war. Die Systemspezifikation kann in diesem Falle angepasst werden, um die neuen Erkenntnisse in den Anforderungen widerzuspiegeln.
Ein Prototyp kann während des Systementwurfs verwendet werden, um Entwurfsalternativen zu realisieren und die Machbarkeit des vorgeschlagenen Entwurfs zu überprüfen. So kann zum Beispiel vom Datenbankentwurf ein Prototyp erstellt werden, um zu testen, ob er einen möglichst effizienten Datenzugriff für die häufigsten Benutzerabfragen unterstützt. Schnelles Prototyping (*rapid prototyping*) unter Einbeziehung der Endbenutzer ist der einzig sinnvolle Weg, um Benutzeroberflächen zu entwickeln. Wegen der dynamischen Eigenschaften von Benutzeroberflächen sind textuelle Beschreibungen und Diagramme nicht ausreichend, um den Entwurf und die Anforderungen an Benutzeroberflächen zu beschreiben.
Ein Vorgehensmodell für die Prototypentwicklung zeigt ▶ Abbildung 2.9. Das Ziele der Prototypenentwicklung sollten am Beginn des Prozesses deutlich gemacht werden. Dabei kann es sich um die Entwicklung der Benutzeroberfläche, zur Bewertung funktionaler Systemanforderungen oder zur Demonstration der Anwendung für das Management handeln. Ein und derselbe Prototyp kann nicht alle Anforderungen erfüllen. Wenn die Ziele nicht klar definiert sind, kann es beim Management oder Endbenutzer zu Missverständnissen über den Einsatz des Prototyps kommen. Folglich

werden diese Personen möglicherweise nicht den Nutzen erhalten, den man sich von der Prototypentwicklung versprochen hatte.

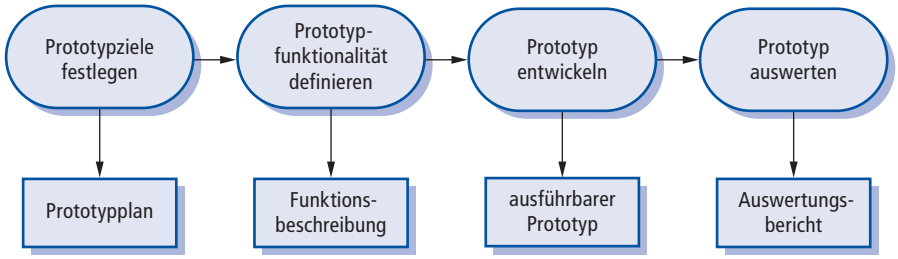

Abbildung 2.9: Prototypentwicklung.

Im nächsten Schritt wird entschieden, was der Prototyp beinhalten und – vielleicht noch wichtiger – was nicht darin enthalten sein sollte. Um die Kosten bei der Prototypentwicklung zu senken und die Auslieferung zu beschleunigen, könnten Sie einige Funktionen beim Prototyp weglassen. Sie könnten nichtfunktionale Anforderungen an die Reaktionszeit oder die Speicherverwaltung abschwächen. Fehlerbehandlung und -bearbeitung können ignoriert werden, wenn der Prototyp nicht zur Demonstration der Benutzeroberfläche verwendet wird. Die Anforderungen an Zuverlässigkeit und Programmqualität können ebenfalls verringert werden.
Die letzte Phase des Prozesses ist die Erprobung und Auswertung des Prototyps. In dieser Phase müssen Vorbereitungen für die Benutzerschulung getroffen werden und die gesetzten Ziele für den Prototyp sollten zur Erstellung eines Bewertungsplans herangezogen werden. Potenzielle Benutzer brauchen Zeit, um sich an ein neues System zu gewöhnen und in ein normales Anwenderverhalten hineinzufinden. Sobald sie das System im praktischen Betrieb benutzen, werden sie Anforderungsfehler und -lücken erkennen. Ein generelles Problem bei der Entwicklung eines Prototyps ist, dass Anwender den Prototyp eventuell nicht in der gleichen Weise wie das endgültige System verwenden. Die Tester des Prototyps sind möglicherweise keine typischen Systembenutzer. Die Zeit zum Kennenlernen des Prototyps während der Bewertungsphase reicht unter Umständen nicht aus. Wenn der Prototyp langsam ist, wird man bei der Bewertung die Arbeitsweise darauf abstimmen und Systemfunktionen mit langen Rechenzeiten vermeiden. Wenn man dann beim endgültigen System über bessere Reaktionszeiten verfügt, wird man es auf andere Weise nutzen.

2.3.2 Inkrementelle Auslieferung

Die inkrementelle Auslieferung (▶ Abbildung 2.10) ist ein Ansatz des Software-Engineerings, bei dem einige der inkrementell entwickelten Teilsysteme an den Kunden ausgeliefert und zur Benutzung in ihrer Arbeitsumgebung eingesetzt werden. In einem inkrementellen Auslieferungsprozess definieren die Kunden, welche Leistungen das System zur Verfügung stellen soll. Sie legen fest, welche Leistungen ihnen am wichtigsten und welche unwichtig sind. Dann wird eine Anzahl von lieferbaren Teilsystemen vereinbart, wobei jedes Inkrement eine Teilmenge der Funktionen des Systems bereitstellt. Die Zuordnung der Dienste zu den Teilsystemen hängt von der Priorität der Dienste ab, wobei Dienste mit der höchsten Priorität zuerst implementiert und ausgeliefert werden.

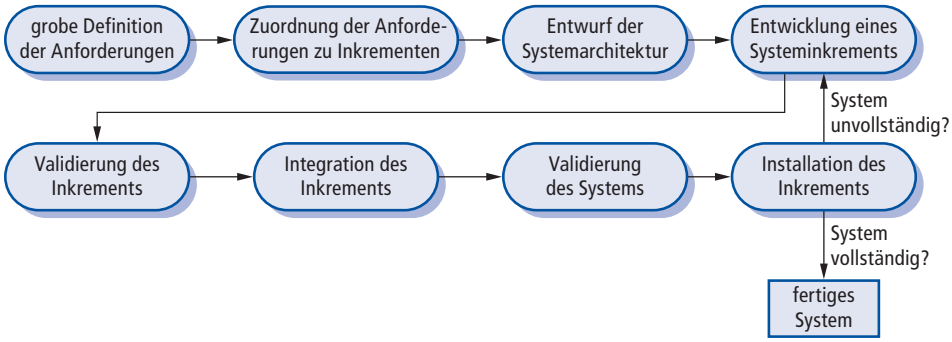

Abbildung 2.10: Inkrementelle Auslieferung.

Sobald die inkrementell zu entwickelnden Teilsysteme festgelegt sind, werden die Anforderungen für die im ersten Teilsystem bereitzustellenden Dienste im Detail definiert und dieses Inkrement wird entwickelt. Während dieser Entwicklungsarbeit können die Anforderungen für die Inkremente analysiert werden, Veränderungen am aktuellen Teil sind jedoch nicht mehr möglich.
Sobald ein Teilsystem fertiggestellt und ausgeliefert wurde, wird es in der normalen Arbeitsumgebung des Kunden installiert. Er kann die Funktionen ausprobieren und auf diese Weise Anforderungen für spätere Teilsysteme abklären. Sind neue Inkremente fertiggestellt, werden sie in die bereits vorhandenen Teilsysteme integriert, sodass sich die Funktionalität des Systems mit jeder neuen Erweiterung verbessert.
Dieser inkrementelle Auslieferungsprozess hat eine Reihe von Vorteilen:

1. Die Kunden können frühe Teilsysteme als eine Art Prototyp benutzen und Erfahrungen sammeln, die in die Anforderungen für spätere Erweiterungen einfließen. Anders als Prototypen sind dies Teile des realen Systems, sodass sie sich nicht erneut umstellen müssen, wenn das vollständige System verfügbar ist.

2. Die Kunden brauchen nicht zu warten, bis das gesamte System geliefert wird, bevor sie es nutzen können. Das erste Teilsystem erfüllt ihre wichtigsten Anforderungen, sodass die Software sofort eingesetzt werden kann.

3. Der Prozess bewahrt den Vorteil der inkrementellen Entwicklung dahingehend, dass es relativ leicht sein sollte, Änderungen in das System einzuarbeiten.

4. Da die Funktionen mit der höchsten Priorität zuerst ausgeliefert werden und spätere Inkremente danach integriert werden, sind diese wichtigen Systemfunktionen am intensivsten getestet. Das bedeutet, dass in den wichtigsten Teilen des Systems weniger Softwareausfälle auftreten sollten.

Es gibt bei der inkrementellen Entwicklung jedoch auch einige Probleme. In der Praxis funktioniert dieses Konzept nur, wenn ein völlig neues System eingeführt wird und wenn den Personen, die das System bewerten, Zeit gegeben wird, um mit dem neuen System zu experimentieren. Die Hauptprobleme bei diesem Ansatz sind:

1. Iterative Auslieferung ist problematisch, wenn das neue System ein bestehendes System ersetzen soll. Die Benutzer benötigen die gesamte Funktionalität des alten Systems und sind in der Regel nicht willens, mit einem unvollständigen neuen System zu experimentieren. Es ist häufig unpraktisch, das alte und das neue System nebeneinander zu verwenden, da sie wahrscheinlich unterschiedliche Datenbanken und Benutzeroberflächen besitzen.

2. Die meisten Systeme verlangen eine Anzahl grundlegender Funktionen, die von verschiedenen Teilen des Systems benutzt werden. Da die Anforderungen nicht detailliert festgelegt werden, bevor ein Teilsystem implementiert werden soll, ist es daher schwer, Funktionen zu erkennen, die alle Teile des Systems brauchen.

3. Das Wesentliche bei iterativen Prozessen ist, dass die Spezifikation zusammen mit der Software entwickelt wird. Das steht jedoch im Widerspruch zu den Beschaffungsmodellen vieler Unternehmen, in denen eine vollständige Systemspezifikation Teil des Vertrages ist. Bei einem inkrementellen Ansatz gibt es jedoch so lange keine vollständige Systemspezifikation, bis die letzte Erweiterung spezifiziert ist. Das erfordert eine neue Art von Vertrag bzw. Übereinkommen, dem sich große Kunden wie Behörden nur schwer anpassen können.

Für einige Systemarten ist die inkrementelle Entwicklung und Auslieferung nicht das beste Vorgehensmodell. Dies betrifft sehr große Systeme, bei denen an der Entwicklung häufig Teams beteiligt sind, die an verschiedenen Orten arbeiten, sowie einige eingebettete Systeme, bei denen die Software von der Hardwareentwicklung abhängt, und kritische Systeme, bei denen alle Anforderungen analysiert sein müssen, um auf Interaktionen zu testen, die die Informationssicherheit und die Betriebssicherheit des Systems gefährden könnten.
Diese großen Systeme leiden natürlich an demselben Problem der unsicheren und sich ändernden Anforderungen. Um dieses Problem anzugehen und einige der Vorteile der inkrementellen Entwicklung auszunutzen, könnte daher ein Systemprototyp entwickelt und der als Plattform für Experimente mit den Systemanforderungen und dem Systementwurf verwendet werden. Mit der Erfahrung, die aus diesem Prototyp gewonnen wird, können endgültige Anforderungen vereinbart werden.

2.4 Prozessverbesserung

Gegenwärtig besteht eine ständige Nachfrage der Industrie nach billigerer, besserer Software, die immer kurzfristiger abgeliefert werden muss. Infolgedessen haben sich zahlreiche Softwarefirmen der Verbesserung von Softwareprozessen als Methode zur Verbesserung ihrer Software, zur Reduzierung der Kosten oder zur Beschleunigung ihrer Entwicklungsprozesse zugewandt. **Prozessverbesserung** bedeutet, die bestehenden Prozesse zu verstehen und sie zu verändern, um die Produktqualität zu erhöhen und/oder Kosten und Entwicklungszeit zu reduzieren.

Es werden zwei verschiedene Ansätze zur Prozessverbesserung und -änderung verfolgt:

1. Der Prozessreifeansatz, der sich darauf konzentriert, Prozess- und Projektmanagement zu verbessern und bewährte Vorgehensweisen des Software-Engineerings in ein Unternehmen einzuführen. Der **Prozessreifegrad** spiegelt wider, in welchem Umfang bewährte technische und Managementverfahren in betriebsspezifische Softwareentwicklungsprozesse eingeflossen sind. Die primären Ziele dieses Ansatzes sind verbesserte Produktqualität und Prozessvorhersagbarkeit.

2. Der agile Ansatz, der sich der iterativen Entwicklung und der Senkung des Overheads im Softwareprozess verschrieben hat. Die Primärmerkmale von agilen Methoden sind die schnelle Auslieferung der Funktionalität und Reaktion auf sich ändernde Kundenanforderungen. Die Philosophie zur Verbesserung ist hier, dass die besten Prozesse diejenigen mit dem geringsten Overhead sind, und agile Ansätze können dies erreichen. Ich beschreibe agile Ansätze in *Kapitel 3*.

Anhänger des einen Ansatzes stehen im Allgemeinen dem anderen Ansatz skeptisch gegenüber. Der Prozessreifeansatz hat seine Wurzeln in der plangesteuerten Entwicklung und erfordert normalerweise einen erhöhten „Overhead“, insofern als Aktivitäten anfallen, die nicht direkt mit der Programmentwicklung verbunden sind. Agile Ansätze konzentrieren sich auf den zu entwickelnden Code und minimieren absichtlich Formgebundenheit und Dokumentation.
Die Prozessverbesserung, der der Prozessreifeansatz zugrunde liegt, ist im Allgemeinen ein zyklischer Prozess (▶ Abbildung 2.11). Die Phasen in diesem Prozess sind:

1. *Prozessmessung:* Man misst ein oder mehrere Attribute des Softwareprozesses oder -produktes. Diese Messungen bilden die Grundlage für die Entscheidung, ob die Prozessverbesserungen effektiv waren. Sobald die Verbesserungen eingeführt wurden, werden dieselben Attribute erneut gemessen - in der Hoffnung, dass sie sich auf die ein oder andere Weise verbessert haben.

2. *Prozessanalyse:* Die Prozessanalyse umfasst die Bewertung des aktuellen Prozesses und die Ermittlung von Schwachstellen und Engpässen. Während dieses Stadiums können Prozessmodelle (manchmal auch als Prozesslandkarte bezeichnet) entwickelt werden, die den Prozess beschreiben. Die Analyse kann auf bestimmte Prozessmerkmale wie Schnelligkeit und Stabilität ausgerichtet sein.

3. *Prozessänderung:* Es werden Prozessänderungen vorgeschlagen, um einige der ermittelten Prozessschwächen zu beseitigen. Diese werden umgesetzt und der Zyklus dann mit der Erhebung von Daten über die Effektivität der Änderungen fortgesetzt.

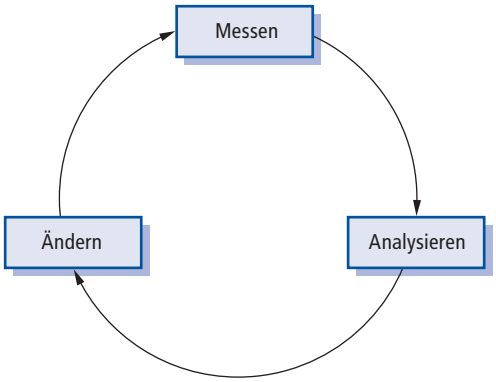

Abbildung 2.11: Der Prozessverbesserungszyklus.

Ohne konkrete Daten über den Prozess oder die Software, die mit diesem Prozess entwickelt wurde, ist es nicht möglich, den Nutzen der Prozessverbesserung zu bewerten. Aber Firmen, die den Prozessverbesserungsprozess starten, haben wahrscheinlich keine Prozessdaten als Ausgangsbasis für die Verbesserung zur Verfügung. Deshalb müssen Sie eventuell als Teil des ersten Änderungszyklus Daten über den Softwareprozess sammeln und die Merkmale des Softwareprodukts messen.
Prozessverbesserung ist eine langfristige Aktivität, d. h., jedes Stadium im Verbesserungsprozess kann mehrere Monate dauern. Außerdem ist es eine fortlaufende Aktivität, da sich unabhängig vom Prozess die Geschäftsumgebung ändert und die neuen Prozesse ebenfalls weiterentwickelt werden müssen, um diesen Änderungen Rechnung zu tragen.
Der Begriff der Prozessreife wurde in den späten 1980er Jahren eingeführt, als das Software Engineering Institute (SEI) ihr Modell der **Capability Maturity** (CM) vorstellten (Humphrey, 1988). Der Reifegrad der Prozesse eines Softwareunternehmens spiegelt das Prozessmanagement, die Prozessmessung sowie den Einsatz von guten Software-Engineering-Praktiken in der Firma wider. Diese Idee wurde vom US-Verteidigungsministerium aufgegriffen, um die CM ihrer Vertragspartner zu bewerten, und zwar im Hinblick darauf, Verträge nur mit Partnern einzugehen, die eine geforderte Prozessreife erreicht hatten. Es wurden fünf Prozessreifegrade vorgeschlagen, wie in ▶ Abbildung 2.12 zu sehen ist. Diese haben sich im Laufe der letzten 25 Jahre weiterentwickelt (Chrissis, Konrad und Shrum, 2011), aber die grundlegenden Ideen von Humphreys Modell bilden immer noch die Basis der Bewertung von Softwareprozessreifegraden.
Die Grade im Prozessreifemodell sind:

1. *Initial:* Die mit dem Prozessgebiet verknüpften Ziele wurden erreicht. Für alle Prozesse wird der Umfang der zu leistenden Arbeit explizit angegeben und den Teammitgliedern mitgeteilt.

2. *Geführt:* Auf dieser Stufe werden die Ziele des Prozessgebiets erreicht und es bestehen organisationsweite Richtlinien, die festlegen, wann die einzelnen Prozesse einzusetzen sind. Es muss dokumentierte Projektpläne geben, die die Projektziele festlegen. Für die gesamte Organisation müssen Ressourcenmanagement- und Prozessüberwachungsverfahren gelten.

3 *Definiert:* Diese Stufe konzentriert sich auf Standardisierung und Umsetzung von Prozessen innerhalb des Unternehmens. Für jedes Projekt gibt es einen verwalteten Prozess, der aus einem definierten Bestand organisationsweiter Prozesse an die Projektanforderungen angepasst wurde. Prozess-Assets und -messungen müssen gesammelt und für die spätere Prozessverbesserung genutzt werden.

4 *Quantitativ verwaltet:* Auf dieser Stufe ist das Unternehmen zur Nutzung statistischer und anderer quantitativer Methoden zur Steuerung von Teilprozessen verpflichtet. Die gesammelten Prozess- und Produktmessungen müssen also bei der Prozessverwaltung verwendet werden.

5 *Prozessoptimierung:* Auf der höchsten Stufe muss das Unternehmen die Prozess- und Produktmessungen zur Steuerung der Prozessverbesserung einsetzen. Trends müssen analysiert und die Prozesse an veränderte geschäftliche Erfordernisse angepasst werden.

Die Arbeit an den den Prozessreifestufen hatte einen großen Einfluss auf die Softwareindustrie. Es fokussierte die Aufmerksamkeit auf die Software-Engineering-Prozesse und -Praktiken, die verwendet wurden, und führte zu signifikanten Verbesserungen im Bereich des Software-Engineerings. Für kleine Unternehmen bringt jedoch formale Prozessverbesserung zu viel Mehraufwand mit sich und bei agilen Prozessen ist eine Reifegradschätzung schwierig. Daher benutzen heute nur große Unternehmen diesen auf Reifegrade fokussierten Ansatz zur Verbesserung der Softwareprozesse.

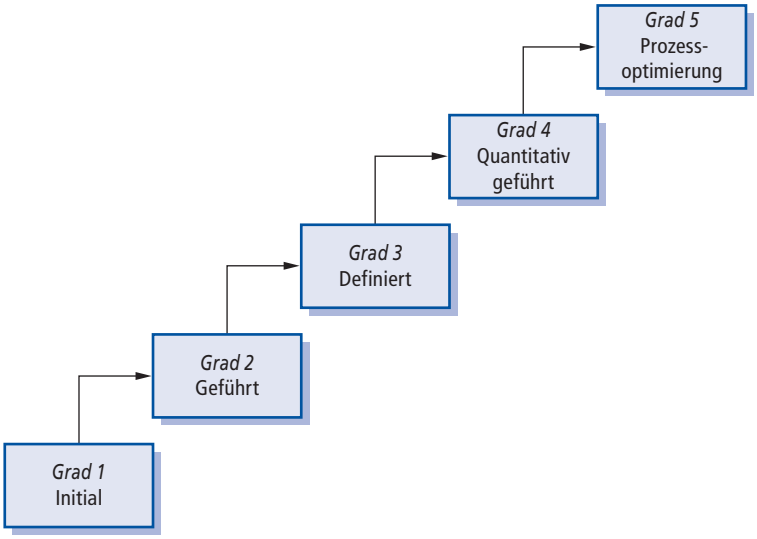

Abbildung 2.12: Stufen der Capability Maturity.

Zusammenfassung

- Softwareprozesse sind die Aktivitäten, die bei der Erstellung von Softwaresystemen beteiligt sind. Vorgehensmodelle sind abstrakte Darstellungen dieser Prozesse.
- Allgemeine Vorgehensmodelle beschreiben den Aufbau von Softwareprozessen. Beispiele dieser allgemeinen Modelle sind das Wasserfallmodell, inkrementelle Entwicklung sowie Konfiguration und Integration von wiederverwendbaren Komponenten.
- Die Anforderungsanalyse ist der Prozess, bei dem die Softwarespezifikation entwickelt wird. Spezifikationen dienen dazu, den Systementwicklern die Systemanforderungen des Kunden mitzuteilen.
- Der Entwurfs- und Implementierungsprozess beschäftigt sich mit der Umwandlung der spezifizierten Anforderungen in ein ausführbares Softwaresystem.
- Die Softwarevalidierung ist der Prozess, bei dem überprüft wird, ob das System mit seiner Spezifikation übereinstimmt und ob es die wirklichen Bedürfnisse des Benutzers erfüllt.
- Die Weiterentwicklung von Software findet statt, wenn bestehende Softwaresysteme verändert werden müssen, um an neue Anforderungen angepasst zu werden. Veränderungen kommen ständig vor und die Software muss sich weiterentwickeln, um nützlich zu bleiben.
- Prozesse sollten Aktivitäten enthalten, die den Umgang mit Änderungen unterstützen. Dazu könnte eine Prototypphase gehören, die hilft, falsche Entscheidungen bei Anforderungen und Entwurf zu vermeiden. Prozesse können für iterative Entwicklung und Auslieferung strukturiert sein, sodass Änderungen vorgenommen werden können, ohne das Gesamtsystem zu stören.
- Prozessverbesserung ist der Prozess, bestehende Softwareprozesse zu verbessern, um die Softwarequalität zu erhöhen, die Entwicklungskosten zu reduzieren oder die Entwicklungszeit zu senken. Es ist ein zyklischer Prozess, der die Messung, die Analyse, die Modellierung und die Änderung von Prozessen umfasst.

Ergänzende Literatur

„Process Models in Software Engineering". Dieser Artikel bietet einen ausgezeichneten Überblick über eine breite Palette von möglichen Vorgehensmodellen im Software-Engineering. (W. Scacchi in *Encyclopaedia of Software Engineering*, J.J. Marciniak (Hrsg.), John Wiley & Sons, 2001.)

http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf.

Software Process Improvement: Results and Experience from the Field. Dieses Buch ist eine Zusammenfassung von Aufsätzen, die sich schwerpunktmäßig mit Fallstudien zur Prozessverbesserung in kleinen bis mittleren norwegischen Firmen beschäftigen. Es enthält außerdem eine gute Einführung in die allgemeinen Probleme von Prozessverbesserung. (R. Conradi, T. Dybå, D. Sjøberg, T. Ulsund (Hrsg.), Springer, 2006.)

„Software Development Life Cycle Models and Methodologies." Dieser Blog-Artikel ist eine prägnante Zusammenfassung mehrerer Vorgenensmodelle, die eingesetzt werden. Es werden hier die Vor- und Nachteile jedes dieser Modelle besprochen. (M. Sami, 2012). *http://melsatar.wordpress.com/2012/03/15/software-development-life-cycle-models-and-methodologies/*

Website

- PowerPoint-Folien für dieses Kapitel:
  http://software-engineering-book.com/slides/chap2/
- Links zu begleitenden Videos:
  http://software-engineering-book.com/videos/software-engineering/

Lösungshinweise

Übungen

1 Schlagen Sie das am besten geeignete allgemeine Vorgehensmodell vor, das als Grundlage für die Entwicklung der folgenden Systeme dienen kann. Erläutern Sie Ihre Antwort entsprechend des Systemtyps, der entwickelt wird:

a. ein System, das bei einem Auto das Antiblockiersystem steuert

b. ein Virtual-Reality-System zur Unterstützung der Softwarewartung

c. ein Buchhaltungssystem, das in einer Universität das bestehende System ersetzt

d. ein interaktives Reiseplanungssystem, das den Nutzern hilft, Reisen mit den geringsten Auswirkungen auf die Umwelt zu planen

2 Erklären Sie, warum inkrementelle Entwicklung der effektivste Ansatz für die Entwicklung von Geschäftssystemen ist. Warum ist dieses Modell weniger geeignet für Echtzeitsysteme?

3 Betrachten Sie noch einmal das Vorgehensmodell mit Integration und Konfiguration aus Abbildung 2.3. Erklären Sie, warum es entscheidend ist, die Aktivitäten zur Anforderungsanalyse in dem Prozess zu wiederholen.

4 Erklären Sie, warum es wichtig ist, während der Anforderungsanalyse zwischen der Entwicklung der Benutzeranforderungen und der Entwicklung der Systemanforderungen zu unterscheiden.

5 Erklären Sie anhand eines Beispiels, warum die Entwurfsaktivitäten von Architekturentwurf, Datenbankentwurf, Schnittstellenentwurf und Komponentenentwurf unabhängig voneinander sind.

6 Erklären Sie, warum das Testen von Software immer eine inkrementelle, stufenförmige Aktivität sein sollte. Sind Programmierer die geeignetsten Leute, die Programme zu testen, die sie entwickelt haben?

7 Erklären Sie, warum Änderungen in komplexen Systemen unvermeidlich sind und geben Sie Beispiele von Softwareprozessaktivitäten (abgesehen von Prototypen und inkrementeller Auslieferung), die helfen, mögliche Änderungen vorherzusagen und die die zu entwickelnde Software robuster gegenüber Änderungen zu machen.

8 Sie haben einen Prototyp eines Softwaresystems entwickelt und Ihre Vorgesetzte ist davon sehr beeindruckt. Sie schlägt vor, es als Produktionssystem einzusetzen, bei dem neue Funktionen bei Bedarf hinzugefügt werden. Damit werden die Kosten der Systementwicklung umgangen und das System ist direkt benutzbar. Schreiben Sie einen kleinen Bericht für Ihre Vorgesetzte, in dem Sie erklären, warum Prototypen normalerweise nicht als Produktionssysteme verwendet werden sollten.

9 Nennen Sie zwei Vor- und zwei Nachteile des Ansatzes zur Prozessbeurteilung und -verbesserung, die in das Capability-Maturity-Rahmenwerk der SEI aufgenommen wurden.

10 Historisch gesehen hat die Einführung von Technologien tief greifende Auswirkungen auf den Arbeitsmarkt gehabt und, zumindest zeitweilig, Menschen ihren Arbeitsplatz weggenommen. Diskutieren Sie, ob die Einführung von umfassender Prozessautomatisierung dieselben Auswirkungen für Softwareentwickler haben wird. Wenn Sie das nicht glauben, begründen Sie Ihre Meinung. Wenn Sie hingegen glauben, dass diese Technologie Karrierechancen einschränken wird, ist es dann moralisch vertretbar, wenn die betroffenen Entwickler sich aktiv und passiv dagegen wehren?

Agile Softwareentwicklung

3

ÜBERBLICK

Einführung

Das Ziel dieses Kapitels ist, Sie in die Methoden der agilen Softwareentwicklung einzuführen. Wenn Sie dieses Kapitel gelesen haben, werden Sie

- die Grundprinzipien der Methoden in der agilen Softwareentwicklung, das agile Manifest und den Unterschied zwischen agiler und plangesteuerter Entwicklung verstehen;
- die wichtigsten Methoden der agilen Entwicklung wie User-Storys, Refactoring, Paarprogrammierung und Test-First-Entwicklung kennen;
- den Scrum-Ansatz zum agilen Projektmanagement verstehen;
- die Probleme der skalierbaren agilen Methoden verstehen und agile Ansätze mit plangesteuerten Ansätzen bei der Entwicklung von großen Softwaresystemen kombinieren können.

Unternehmen sind heute in globalen, sich schnell verändernden Umfeld tätig. Sie müssen auf neue Gelegenheiten und Märkte, veränderte wirtschaftliche Bedingungen und das Erscheinen von Konkurrenzprodukten und -dienstleistungen reagieren. Software ist nahezu Teil aller Geschäftstätigkeiten, daher muss neue Software schnell entwickelt werden, um neue Gelegenheiten zu nutzen und auf Konkurrenzdruck zu reagieren. Heutzutage sind deswegen die schnelle Softwareentwicklung und Auslieferung die entscheidendsten Anforderungen an die meisten Geschäftssysteme. Tatsächlich sind viele Unternehmen bereit, Softwarequalität und die Einhaltung von Anforderungen gegen die schnelle Installation von wichtiger neuer Software einzutauschen.

Da diese Unternehmen in einem sich ändernden Umfeld tätig sind, ist es unmöglich, eine vollständige Menge stabiler Softwareanforderungen zu gewinnen. Anforderungen ändern sich, da die Kunden kaum vorhersagen können, wie ein System die Arbeitsabläufe beeinflussen und wie es mit anderen Systemen zusammenarbeiten wird und welche Benutzerprozesse automatisiert werden sollten. Es kann sein, dass die wirklichen Anforderungen erst dann deutlich werden, wenn ein System ausgeliefert wurde und die Benutzer Erfahrungen damit gesammelt haben. Selbst dann werden externe Faktoren Veränderungen der Anforderungen anstoßen.

Plangesteuerte Verfahren, die auf der vollständigen Spezifikation der Anforderungen und dem anschließenden Entwerfen, Erstellen und Testen des Systems basieren, zielen nicht auf schnelle Softwareentwicklung ab. Wenn sich die Anforderungen ändern oder Probleme mit den Anforderungen entdeckt werden, muss der Systementwurf oder die Implementierung angepasst und erneut getestet werden. Folglich dauert ein konventioneller Wasserfallprozess oder ein spezifikationsbasiertes

Verfahren gewöhnlich sehr lange, sodass die endgültige Software lange Zeit nach der ursprünglichen Spezifikationserstellung an den Kunden ausgeliefert wird.
Für einige Softwarearten wie sicherheitskritische Steuerungssysteme, bei denen eine vollständige Analyse des Systems von entscheidender Bedeutung ist, ist dieser plangesteuerte Ansatz die richtige Wahl. In einer schnell wechselnden Geschäftsumgebung jedoch kann dieser Ansatz zu großen Problemen führen. Bis die Software zur Verfügung steht, kann sich der ursprüngliche Grund für ihre Bestellung so grundlegend geändert haben, dass die Software letztlich nutzlos ist. Daher sind vor allem für Geschäftssysteme Entwicklungsverfahren wichtig, die sich auf schnelle Softwareentwicklung konzentrieren.
Es ist schon vor einigen Jahren erkannt worden, dass eine schnelle Softwareentwicklung sowie Verfahren, die mit sich ändernden Anforderungen umgehen können, benötigt werden (Larman und Basili, 2003). Einen wahrhaften Aufschwung erlebte die schnelle Softwareentwicklung jedoch in den späten 1990er Jahren mit der Entwicklung von agilen Methoden wie **DSDM** (Stapleton, 2003), **Scrum** (Schwaber und Beedle, 2001) und Extreme Programming (Beck, 1999).
Schnelle Softwareentwicklung wurde als agile Entwicklung oder agile Methoden bekannt. Diese agilen Methoden sind dazu gedacht, in kurzer Zeit nützliche Software zu entwerfen. Alle agilen Methoden, die bisher vorgeschlagen wurden, weisen einige Gemeinsamkeiten auf:

1. Die Prozesse für Spezifikation, Entwicklung und Implementierung erfolgen verzahnt. Es gibt keine detaillierte Systemspezifikation, die Entwurfsdokumentation ist minimal oder wird automatisch von der zur Implementierung verwendeten Programmierumgebung erstellt. Die Dokumentation der Benutzeranforderungen definiert lediglich die wichtigsten Systemeigenschaften.

2. Das System wird in einer Reihe aufeinander aufbauender Inkremente entwickelt. Die Endbenutzer und andere am System Beteiligte werden in die Entwicklung und Evaluation aller Inkremente eingebunden. Sie können Softwareänderungen und neue Anforderungen vorschlagen, die in spätere Versionen der Software einfließen können.

3. Der Entwicklungsprozess wird durch umfangreiche Werkzeuge unterstützt. Zu den verwendeten Werkzeugen gehören solche zum automatischen Testen, zur Unterstützung der Konfigurationsverwaltung und der Systemintegration sowie zum Automatisieren der Produktion von Benutzeroberflächen.

Agile Methoden sind inkrementelle Entwicklungsmethoden, bei denen die Inkremente klein sind und bei denen typischerweise alle zwei oder drei Wochen neue Versionen des Systems erzeugt und dem Kunden zur Verfügung gestellt werden. Sie beziehen Kunden in den Entwicklungsprozess ein, um schnelle Rückmeldungen auf sich ändernde Anforderungen zu bekommen. Indem eher informelle Kommunikation eingesetzt wird anstatt formelle Sitzungen mit schriftlichen Unterlagen abzuhalten, wird die Dokumentation minimiert.
Bei agilen Methoden der Softwareentwicklung sind Entwurf und Implementierung die zentralen Aktivitäten im Softwareprozess. Andere Aktivitäten wie die Anforderungserhebung und das Testen in der Entwurfs- und Implementierungsphase werden integriert. Bei einem plangesteuerten Ansatz werden dagegen einzelne Phasen im Soft-

wareprozess identifiziert sowie die damit verbundenen Ergebnisse. Die Ergebnisse einer Phase werden als Basis für die Planung der folgenden Prozessaktivität benutzt.
▶ Abbildung 3.1 veranschaulicht die wesentliche Unterschiede zwischen plangesteuerten und agilen Ansätzen zur Systemspezifikation. Bei einem plangesteuerten Softwareentwicklungsprozess findet die Iteration innerhalb solcher Aktivitäten statt, bei denen formale Dokumente zur Kommunikation zwischen den Prozessphasen benutzt werden. Zum Beispiel entwickeln sich die Anforderungen stetig weiter, bis schlussendlich eine Anforderungsspezifikation erstellt wird. Diese wird dann zur Eingabe des Entwurfs- und Implementierungsprozesses. Bei einem agilen Ansatz findet die Iteration in allen Aktivitäten statt. Daher werden die Anforderungen und der Entwurf zusammen statt getrennt entwickelt.

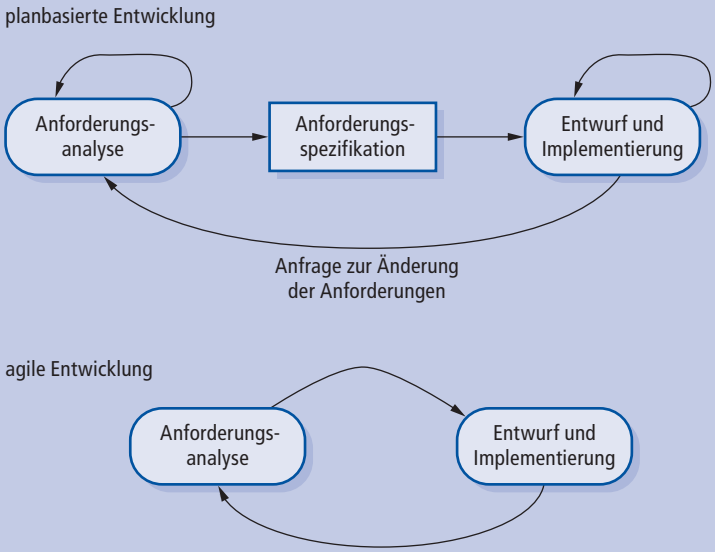

Abbildung 3.1: Plangesteuerte und agile Entwicklung.

Plangesteuerte Prozesse werden in der Praxis häufig zusammen mit agiler Programmierung eingesetzt und agile Methoden können – abgesehen von Programmierung und Testen – Aktivitäten des plangesteuerten Ansatzes enthalten. Ich werde darauf in *Abschnitt 3.4.1* zurückkommen. Es ist durchaus möglich, in einem plangesteuerten Prozess die Zuteilung von Anforderungen sowie die Entwurfs- und Entwicklungsphase als eine Folge von Inkrementen zu planen. Ein agiler Prozess ist nicht zwangsläufig codezentriert und er könnte einige Entwurfsdokumente erzeugen. Agile Entwickler könnten entscheiden, dass eine Iteration keinen neuen Code hervorbringen soll, sondern stattdessen Systemmodelle und Dokumentation.

3.1 Agile Methoden

In den 1980er und frühen 1990er Jahren war die Ansicht weitverbreitet, dass die beste Möglichkeit, bessere Software zu erreichen, in sorgfältiger Projektplanung, formalisierter Qualitätssicherung, Analyse- und Entwurfsmethoden mithilfe von Softwarewerkzeugen und kontrollierten und strikten Softwareentwicklungsprozessen bestand. Diese Ansicht wurde von der Software-Engineering-Gemeinschaft aufgestellt, die verantwortlich war für die Entwicklung von großen, langlebigen Softwaresystemen wie Systemen für die Raumfahrt und die Regierung.

Dieser plangesteuerte Ansatz wurde für Software entwickelt, die von großen Teams erstellt wurde, die für unterschiedliche Unternehmen arbeiteten. Die Teams waren häufig weiträumig verstreut und arbeiteten jeweils lange Zeit an der Software. Ein Beispiel für diese Art von Software sind Steuerungssysteme für moderne Flugzeuge, bei denen von der ursprünglichen Spezifikation bis zur Bereitstellung bis zu zehn Jahre vergehen können. Plangesteuerten Ansätze erfordern einen deutlichen zusätzlichen Aufwand für die Planung, den Entwurf und die Dokumentation des Systems. Dieser Mehraufwand ist gerechtfertigt, wenn die Arbeit von mehreren Entwicklerteams koordiniert werden muss, wenn es sich um ein kritisches System handelt und wenn an der Wartung der Software während ihrer Einsatzzeit viele verschiedene Menschen beteiligt sind.

Wenn dieser schwergewichtige, plangesteuerte Entwicklungsansatz jedoch auf kleine und mittlere Geschäftssysteme angewendet wird, erweist sich der Aufwand als so hoch, dass er den gesamten Softwareentwicklungsprozess beherrscht. Es wird mehr Zeit für die Planung der Systementwicklung aufgewendet als für die eigentliche Programmentwicklung und die Tests. Bei Änderungen der Systemanforderungen sind Umarbeitungen erforderlich. Dabei müssen – zumindest im Prinzip – die Spezifikation und der Entwurf zusammen mit dem Programm geändert werden.

Die Unzufriedenheit mit diesen aufwendigen Ansätzen des Software-Engineerings führte zur Entwicklung von **agilen Methoden** in den späten 1990er Jahren dazu. Diese Methoden erlaubten es dem Entwicklerteam, sich auf die Software selbst zu konzentrieren statt auf ihren Entwurf und die Dokumentation. Sie sind bei der Entwicklung von solchen Geschäftsanwendungen am besten geeignet, bei denen sich die Systemanforderungen während der Entwicklung schnell ändern. Der Sinn ist, in kurzer Zeit funktionierende Software an den Kunden auszuliefern, der dann neue und veränderte Anforderungen vorschlagen kann, die in spätere Iterationen des Systems aufgenommen werden sollen. Agile Methoden helfen, die Prozessbürokratie zu reduzieren, indem Arbeit vermieden wird, die einen zweifelhaften Langzeitwert hat, und indem Dokumentation abgeschafft wird, die vermutlich niemals benutzt wird.

Die Philosophie hinter agilen Methoden spiegelt sich im **agilen Manifest** (*http://agilemanifesto.org*) wider, das von den führenden Entwickler dieser Vorgehensweise ausgegeben wurde. Dies Manifest besagt:

> *Wir suchen nach besseren Wegen, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen. Durch diese Arbeit haben wir schätzen gelernt:*
>
> > *Individuen und Interaktionen haben Vorrang vor Prozessen und Werkzeugen.*
> >
> > *Funktionierende Software hat Vorrang vor umfangreicher Dokumentation.*