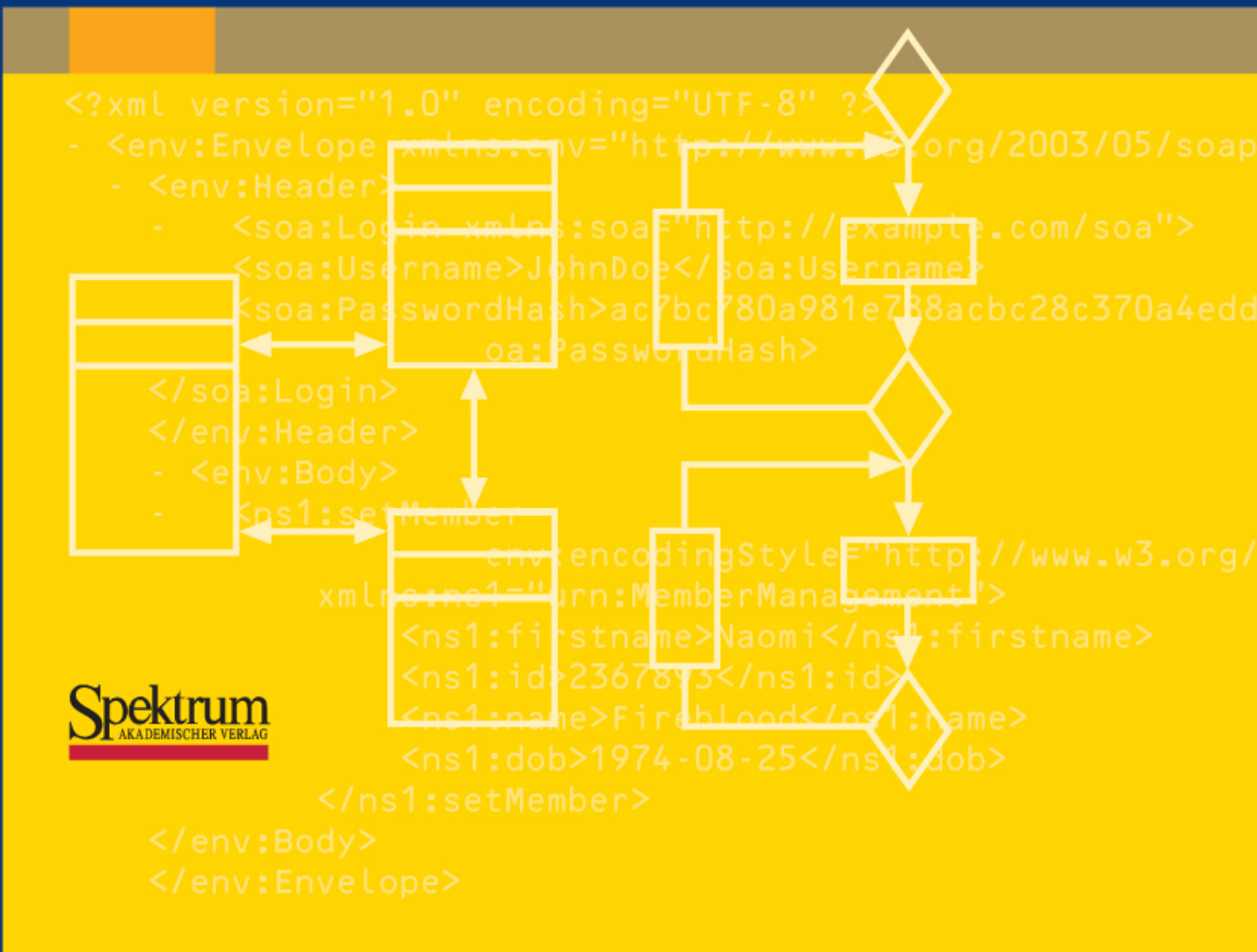
A. Schatten S. Biffl M. Demolsky
E. Gostischa-Franta Th. Östreicher D. Winkler

Best Practice Software-Engineering

Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen



Spektrum
AKADEMISCHER VERLAG

Alexander Schatten / Markus Demolsky / Dietmar Winkler / Stefan Biffl /
Erik Gostischa-Franta / Thomas Östreicher

Best Practice Software-Engineering

Eine praxiserprobte Zusammenstellung von
komponentenorientierten Konzepten,
Methoden und Werkzeugen

Spektrum
AKADEMISCHER VERLAG

Autoren
Alexander Schatten
Markus Demolsky
Dietmar Winkler
Stefan Biffl
Erik Gostischa-Franta
Thomas Östreicher

Weitere Informationen zum Buch unter:
http://bpse.ifs.tuwien.ac.at

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

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Spektrum Akademischer Verlag ist ein Imprint von Springer

10 11 12 13 14 5 4 3 2 1

Planung und Lektorat: Dr. Andreas Rüdinger, Bianca Alton
Redaktion: Bernhard Gerl
Herstellung und Satz: Crest Premedia Solutions (P) Ltd, Pune, Maharashtra, India
Umschlaggestaltung: SpieszDesign, Neu–Ulm

ISBN 978-3-8274-2486-0

Inhaltsverzeichnis

1 | Einleitung

Software-Engineering, also die systematische, ingenieurmäßige Herstellung von Software [46], erfolgt in vielfältigen Anwendungsfeldern, technischen Plattformen und Projektgrößen. In einem Feld, das von raschem technologischem Wandel geprägt ist, stellt sich für Einsteiger mit Grundkenntnissen in der Programmierung und für Wiedereinsteiger die Frage, welche Auswahl an Methoden, Notationen und Techniken eine gute Grundlage für die Durchführung eines mittelgroßen Projekts sein kann.

„Best-Practice“ Software-Engineering

„Best-Practice Software-Engineering“ liefert eine aufeinander abgestimmte Zusammenstellung von Konzepten, Methoden und Werkzeugen für Software-Engineering-Projekte, die sich in der Praxis bewährt haben. Der Begriff „Best-Practice“ basiert auf der Idee, dass es für eine bestimmte Aufgabe ein Bündel von Techniken, Methoden, Prozessen, Aktivitäten und Anreizen gibt, das effektiver oder effizienter ist als andere gebräuchliche Vorgehensweisen. Effektiv bedeutet, dass das angestrebte Ergebnis mit höherer Wahrscheinlichkeit bzw. geringeren Risiken erreicht wird. Effizient heißt, dass für ein bestimmtes Ergebnis weniger Ressourcen gebraucht werden oder mit gleichen Ressourcen ein besseres Ergebnis erreicht wird als mit einer alternativen Vorgehensweise.

Um sich als „Best-Practice“ zu qualifizieren, muss eine Vorgehensweise wiederholbar sein und in ihrer Wirksamkeit im Zielkontext ausreichend evaluiert werden. In der Arbeit mit Projektteams und in der Analyse von Open-Source-Projekten hat sich gezeigt, welche Ansätze effektiv und robust genug sind, um als Best-Practice zu gelten, und welche sich als zu umständlich oder als Modeerscheinungen herausgestellt haben.

Komponentenorientierung

Ziel dieses Buches ist eine durchgängige Darstellung der Fähigkeiten, die für die erfolgreiche Abwicklung eines Software-Entwicklungsprojekts notwendig sind. Gutes Software-Engineering erfordert einen ganzheitlichen Ansatz und Verständnis in den Bereichen Anforderungsbeschreibung, Software-Prozess und Projektmanagement, Architektur und Design Patterns, Komponentenorientierung, Qualitätssicherung, integrierte Werkzeuge, Kommunikation sowie eine systematische Evaluierung bis zur Dokumentation. Als Einführung in diese breite Materie finden Sie in den folgenden Abschnitten eine Darstellung der wesentlichsten Komponenten software-intensiver Systeme, einen Überblick zu Ansätzen, um Software-Entwicklungsprojekte zu planen und zu steuern sowie eine Zusammenfassung der thematischen Schwerpunkte der Buchkapitel.

Anwendungsbereiche

Die Anwendungsbereiche der Projekte befassen sich mit kommerziellen Web-/Open-Source-Systemen und *explizit nicht* mit eingebetteten und sicherheitskritischen Systemen.

1.1 Projektarten und -aspekte

Prozesse, Technologien und Benutzer

Software-Systeme sind die Basis für umfassende Abläufe des Geschäftslebens (Abwicklung von Geschäftsprozessen wie einer Reisebuchung), der Industrie (Anlagensteuerung) und Kommunikation & Unterhaltung (etwa Musikplattformen wie iTunes). Das komplexe Zusammenspiel einer Menge von Prozessen, Technologien und Benutzern ist notwendig, um effektiv und effizient Ergebnisse ausreichend hoher Qualität zu erzielen.

Schnellere Entwicklung komplexer Systeme mit Komponenten

Die Herstellung komplexer Software ist anspruchsvoll und braucht professionelles Herangehen, von der Erfassung der Anforderungen, über den Entwurf der Systemarchitektur, bis hin zur detaillierten Umsetzung und Überprüfung. Ein wesentlicher Aspekt in vielen Projekten ist, die Zeit von der Planung zur Umsetzung einer Produktidee zu verkürzen und gleichzeitig Produkte mit ausreichend hoher Qualität zu liefern. Die Herstellung von Software-Systemen durch Zusammensetzen von bestehenden Komponenten unterschiedlicher Quellen (etwa Open-Source-Plattformen) unterstützt eine raschere Entwicklung.

Vorhersehbares Verhalten

Fehler und Qualitätsmängel von Software-Systemen betreffen immer mehr Menschen in immer größerem Umfang. Ziel professioneller Software-Hersteller ist es daher, „No-surprise-Software“ zu entwickeln, also Software, deren Verhalten und Eigenschaften vorhersehbar sind, insbesondere auch im Zusammenspiel mit anderen Systemen.

Durch die zunehmende Verbreitung von Software-Systemen kommen immer mehr Menschen mit Software-Programmen in Kontakt. Das Know-how von Benutzern reicht dabei von Laien bis hin zu „Power Usern“. Abgesehen von dem vorhandenen Wissen der Benutzer muss das Software-System leicht zu bedienen sein und bei auftretenden Fehler eine bestmögliche Unterstützung bieten.

Software-Engineering

Ziel der Disziplin Software-Engineering ist, für große Software-Systeme ähnliche Qualitätsmaßstäbe zu erreichen, wie es in klassischen Ingenieurdisziplinen üblich ist. Kostengünstige Entwicklung innerhalb des geplanten Zeitrahmens (vorhersagbare Herstellungsdauer und -aufwand), hohe Produktqualität (geringe Fehlerquote bei Auslieferung), verfügbare Professionisten für Anpassung und Wartung werden dabei angestrebt.

Typische Probleme bei großen Software-Entwicklungsprojekten resultieren aus der Komplexität von Software-Systemen, insbesondere auch aus der Integration mit anderen Systemen und der Veränderung des Umfelds im Lauf der Zeit. Diese Komplexität wird, besonders am Anfang, häufig unterschätzt. Eine unzureichende Erfassung der Anforderungen sowie Definition und Überprüfung der Qualität von Zwischenergebnissen, führt zu mangelnder Qualität bzw. Verwendbarkeit der gelieferten Software.

Entwicklung und Validierung des „Bauplans“

Software-Projekte haben als Ziel ein neues Software-System zu entwickeln oder existierende Systeme weiterzuentwickeln bzw. zu integrieren. Im Gegensatz zu materiellen Produkten hat Software besondere Eigenschaften in

der Herstellung (wie geringe Kosten der Vervielfältigung, kein Qualitätsverlust bei Vervielfältigung, sehr geringe Einschränkung durch physikalische Gesetze), sodass der Hauptaufwand in der Entwicklung und Validierung des „Bauplans“ für das Software-System entsteht.

Unrealistische Aufgabenstellungen, deren Komplexität mit dem Stand der Technik nicht zu bewältigen sind, bzw. mangelnde Ausbildung und Erfahrung der Projektmitarbeiter führen zu Schwächen im Projekt- und Risikomanagement und zu Zeitverzögerungen bzw. zum Überschreiten des geplanten Budgets.

Domäne & Technologien

Die Technologie eines Software-Projekts – etwa *Rich-Clients*, *Distributed Systems*, *Web Applications* – hat bei Weitem nicht so viel Einfluss auf die Komplexität eines Software-Produkts wie die Domäne, für welche das Produkt hergestellt wird. Natürlich werden die verwendeten Technologien für die einzelnen Projekttypen auch auf die Anforderungen abgestimmt, dies ist sogar eine bekannte Vorgehensweise von Kunden: „Welche Software ist möglich?“ Erfahrung im Team wird benötigt, um die Domäne zu analysieren, die entsprechenden Technologien zu wählen und mittels Software-Engineering und Projektmanagement-Methoden den Aufwand zu schätzen und den Projektverlauf zu steuern.

Treiber des Projektaufwands

Typ und Komplexität des geplanten Produkts haben starken Einfluss auf die Herangehensweise (Prozesse, Methoden und Technologien) und den Umfang des zu planenden Projekts. In der Praxis wurden Projektkomplexität und Produktgröße als Treiber für den Projektaufwand identifiziert [13].

Produktkomplexität

Produktkomplexität beschreibt die Funktionen und die Anzahl bzw. den Umfang der beteiligten Software-Komponenten, wie z. B. Anzahl der Klassen, Module, Datenbanken und verwendete Technologien sowie Code-Zeilen. Verwendete Technologien und insbesondere bestehende Frameworks tragen ebenfalls zur Projektkomplexität bei, und es werden erfahrene Entwickler und Spezialisten für deren Einsatz benötigt.

Projektrisiken

Neue Projekte bringen oft viele Neuheiten für die Teammitglieder mit sich, was wiederum Projektrisiken birgt, die durch passende Planung und Hinzuziehen von Experten besser vorhersehbar und damit kontrollierbar werden. Häufigstes Risiko ist die Volatilität der Rahmenbedingungen (Änderungen an Technologien, Projektzielen) und der Anforderungen.

Methodik und Werkzeuge

Mangelhafte Methodik und Werkzeuge in der Software-Entwicklung können zu Design-Fehlern führen (*late design breakage*). Schwierige und teure Wartung kann die Folge sein. Software-Entwickler sollten daher die entsprechende Vorgehensweisen kennen, um die Wünsche des Kunden als Anforderungen einer Domäne festzuhalten und weiter das Fachwissen besitzen, die entsprechenden Technologien für die erforderte Funktionalität auszuwählen und einzusetzen.

Modellierung und Vorgehensmodelle

Als „Sprachen“ dienen in diesem Buch unter anderem die *Unified Modeling Language (UML)* sowie Vorgehensmodelle (RUP, SCRUM) und an-

dere Projektmanagement-Methoden zur Anforderungserhebung. In einem Entwicklerteam sowie in der Kommunikation mit dem Management und mit Kunden ist die frühzeitige Einigung auf die Vorgehensweise und die gewählten Werkzeuge sehr wichtig. Zudem haben nicht alle Teammitglieder Kundenkontakt – die geplante Arbeit muss festgehalten und vom Projektmanagement verteilt werden.

Kleines Projekt

Ein kleines Projekt könnte beispielsweise die Entwicklung einer Rich-Client Applikation zur Verwaltung von Kundendaten sein. In der Umsetzung sind ein bis zwei Personen beteiligt, die sich am gleichen Standort befinden. Das primäre Ziel ist hier, ein kleines, funktionierendes Produkt zu erzeugen, das auf einer soliden und erweiterbaren Architektur aufbaut. Anwendung und Datenbank befinden sich auf demselben Rechner.

Mittleres Projekt

Bei einem mittleren Projekt sind etwa 3-20 Personen an der Umsetzung beteiligt. Es ist wahrscheinlich, dass sich nicht alle Personen am gleichen Standort befinden. Persönliche Treffen mit allen wesentlichen Beteiligten sind aber regelmäßig möglich. Es wird beispielsweise eine Webanwendung entwickelt. Die Anwendung und die Datenbank befinden sich auf unterschiedlichen Systemen, beide müssen eine hohe Verfügbarkeit haben. Bei mittleren Projekten findet wesentlich mehr Modellierung statt, da mehrere Entwickler beteiligt sind und diese sich auf ein gemeinsames Design einigen müssen, um eine einheitliche Projektstruktur zu gewährleisten.

Großes Projekt

An einem großen Projekt sind weit über 20 Personen beteiligt, die sich an verschiedenen Standorten (vielleicht sogar unterschiedlichen Kontinenten) befinden. Aufgrund der hohen Anzahl an Beteiligten sind regelmäßige persönliche Treffen nicht leicht möglich und würden vermutlich auch nicht sehr produktiv sein. Das Projekt besteht beispielsweise aus einer verteilten Anwendung im Enterprise-Umfeld, das hohe Verfügbarkeit aufweisen muss und eine sehr hohe Anzahl an Benutzern aufweist. Die Anwendung integriert verschiedene bestehende Systeme und soll so Unterstützung bei der Abwicklung des Kerngeschäfts bieten.

Projektkomplexität

Nicht nur die Größe des Projekts ist ein kritischer Faktor. Auch die Kenntnis der Domäne, die Qualifikation des Entwicklerteams sowie die technische Komplexität spielen eine wesentliche Rolle. Projekte, die beispielsweise wenig erprobte *Cutting-edge*-Technologien einsetzen (müssen) oder als sehr innovativ zu betrachten sind, bergen unabhängig von der Menge der Anforderungen naturgemäß höhere Risiken als Routineprojekte, die mit einem erfahrenen Team und bekannten Technologien umgesetzt werden.

Methodisches Vorgehen

Mit der Zunahme der Projektgröße und -komplexität aber werden andere Ansätze zur Bewältigung der zusätzlichen Komplexität in den Bereichen Software-Engineering, Architektur und Design sowie Projektmanagement und Qualitätssicherung notwendig. Im Software-Engineering bedeutet dies in erster Linie die systematische Verwendung von Methoden mit Werkzeugunterstützung wie z. B. Versionsverwaltung bzw. Konfigurationsmanagement und Build- und Test-Automatisierung. Wenn der Projektmana-

gement-Bedarf an Organisation, Planung und Überblick steigt, kann eine Werkzeugunterstützung zwar helfen, jedoch muss prinzipiell noch intensiver an dem jeweiligen Projektmanagementprozess festgehalten werden. Eine genaue Definition der Kommunikationspfade bzw. wer mit wem kommuniziert ist notwendig. Der Testaufwand, ein großer Teil der Qualitätssicherung, steigt meist überproportional. Hier ist eine saubere Dokumentation wichtig, damit das Projekt nachvollziehbar bleibt.

Neben erfahrenen Mitarbeitern helfen also verbesserte Prozesse, Methoden und Werkzeuge des Software-Engineering, den Aufwand für die Entwicklung und Validierung zu verringern. Ziel sind kleinere, handhabbarere Projekte zu dimensionieren (wie etwa in der agilen Software-Entwicklung angestrebt). Eine zu hohe Produkt-/Projektkomplexität ist ein besonderes Risiko, das auch durch das Hinzufügen von Mitarbeitern (besonders in späten Phasen) nicht zu bewältigen ist [15].

1.2 Überblick zu Kapiteln und Themen

Lebenszyklus eines Software-Produkts

Kapitel 2 beschäftigt sich mit dem Lebenszyklus eines mittelgroßen Software-Produkts aus Expertensicht. Diese chronologische Beschreibung kann den Entwicklerteams als Leitlinie dienen, um ein Software-Projekt im Team erfolgreich zu strukturieren und abzuwickeln. Je nach Anwendungsbereich der Software-Lösung sind die jeweiligen Phasen eines Produktlebenszyklus unterschiedlich ausgeprägt und für den Anwender unterschiedlich wahrnehmbar, wie beispielsweise Wartungsprojekte, Migrationsprojekte, Projekte mit öffentlichen Ausschreibungen und klassische Software-Entwicklungsprojekte. Im realen Projektverlauf ist natürlich eine Anpassung an den jeweiligen Anwendungsfall im jeweiligen Projektkontext erforderlich. Dieses Kapitel behandelt alle Phasen eines Software-Produktes und beschreibt ausgewählte und wichtige Aspekte in den einzelnen Abschnitten.

Vorgehensmodelle

Kapitel 3 stellt traditionelle Prozesse vor, wie beispielsweise *Wasserfallmodell*, *V-Modell XT* und *Rational Unified Process* sowie flexible und agile Ansätze, wie beispielsweise *SCRUM*. Typische Schritte im Entwicklungsprozess, die im Rahmen des Produktlebenszyklus erläutert wurden, sind Anforderungserhebung, Entwurf und Design, Implementierung und Integration, Verifikation und Validierung sowie Betrieb, Wartung und Evolution. Diese Schritte benötigen eine passende Strategie, um die Produktentwicklung effektiv vorantreiben zu können. Je nach Projekt und Anwendungsdomäne haben sich zahlreiche konkrete Vorgehensmodelle etabliert, aus denen die Strategie für einen Projektkontext auszuwählen und anzupassen ist.

Projektmanagement

Kapitel 4 beschreibt klassische Projektplanungsmethoden, versucht aber auch den Zusammenhang mit agiler Entwicklung herzustellen. Das Projektmanagement entwickelt, ausgehend von einem Projektauftrag, einen

Projektplan, der die Arbeit strukturiert und realistische Schätzungen der erforderlichen Ressourcen sowie der Projektdauer erlaubt. Während Manager Schätzverfahren verwenden können, um einen Gesamtprojektplan zu erstellen, ist es selbst für erfahrene Projektmanager oft schwer, die Dauer von technischen Arbeiten zu schätzen. Dies trifft vor allem dann zu, wenn die Implementierung auf neuen Technologien beruht. Entsprechend sind Ansätze zur Projektverfolgung und -steuerung notwendig, um den Plan im Projektverlauf mit der Realität abzustimmen.

Qualitätssicherung und Test-Driven Development

Kapitel 5 beschreibt grundlegende Konzepte der Qualitätssicherung in der modernen Software-Herstellung, wie Software-Reviews, Inspektionen, Architekturreviews und Konzepte des Software-Testens. *Test-Driven Development* ist ein bewährtes Konzept im Rahmen der komponentenorientierten Software-Herstellung, um qualitativ hochwertige Produkte herstellen zu können. Konzepte, wie *Continuous Integration* und *Daily Builds*, tragen zur Verbesserung der zu erstellenden Software-Lösung und insbesondere zur frühzeitigen Sichtbarkeit des realen Entwicklungsfortschritts bei.

Notationen und Methoden der Modellierung

Kapitel 6 gibt einen Überblick zu Notationen und Methoden der Modellierung für die Entwicklung kommerzieller Software-Produkte. Im administrativen Bereich hat sich in den letzten Jahren die Unified Modeling Language (UML) als Notation und Basis für Methoden weitgehend durchgesetzt. Die Sammlung von Notationen und Methoden in diesem Kapitel werden durchgehend in diesem Buch eingesetzt, um Konzepte und Strukturen zu illustrieren und zu beschreiben.

Software-Architektur

Kapitel 7 beschäftigt sich mit der Strukturierung von Software-Systemen und stellt einen Überblick verschiedener Architekturstile vor. Mit der zunehmenden Komplexität moderner Software-Systeme gewinnen die Software-Architektur und die Rolle eines Software-Architekten immer mehr an Bedeutung. Die Architektur bringt klare Grenzen und Struktur in die Anwendung, befasst sich aber nicht nur mit der statischen Systemstruktur, sondern legt auch besonderes Augenmerk auf nichtfunktionale Anforderungen, wie Skalierbarkeit, Performanz oder Verfügbarkeit.

Entwurfs-, Architektur-, und Integrationsmuster

Kapitel 8 soll als Referenz auf bestehende Lösungsansätze für die im vorherigen Kapitel beschriebenen Architekturen für ein kleines bis mittelgroßes Projekt dienen. Die hier beschriebene Einführung in Entwurfs-, Architektur- und Integrationsmuster kann helfen, wiederkehrende Aufgaben mit erprobten Ansätzen zu implementieren.

Komponentenorientierte Software-Entwicklung

Kapitel 9 befasst sich mit der Frage, in welche Teile (Komponenten) sich die Software zerlegen lässt, und ob manche Teile mit bereits bestehenden Software-Produkten realisiert werden können. Wir betrachten in diesem Buch eine Komponente als austauschbaren Programmteil gröberer Granularität mit wohl definierten Schnittstellen nach außen. Generell zielen Komponenten auf die Verbesserung der Wiederverwendung von Software-Investitionen ab und ermöglichen die Komposition komplexer Software-Systeme aus einfacheren Teilkomponenten sowie den einfacheren Austausch

von Systemteilen im Lauf der Wartung und Evolution eines Anwendungssystems.

Techniken und Werkzeuge

Kapitel 10 stellt konzeptionelle Aspekte von Werkzeugen und Praktiken vor, die den Software-Entwicklungsprozess wesentlich unterstützen. In heutigen Projekten ist es üblich eine ausgereifte Entwicklungsumgebung (IDE) mit umfangreichen Funktionen zu verwenden und darüber hinaus Werkzeuge für Versionierung (SCM), Issue Tracking, Build Management und Test Reporting, sowie Kollaborationswerkzeuge einzusetzen. Diese Werkzeuge sollen das strukturierte Arbeiten in einem Team unterstützen.

1.3 Empfohlene Vorkenntnisse

Zielgruppe

Zielgruppe dieses Buches sind Entwickler mit Grundkenntnissen der Programmierung, die einen Einstieg in moderne komponentenorientierte Entwicklung suchen.

Es wird vorausgesetzt, dass der Leser gute Kenntnisse zu algorithmischen Strukturen und Abläufen hat. Des Weiteren ist Erfahrung mit Konzepten der objektorientierten Programmierung für das Verständnis notwendig. Die meisten Beispiele im Buch sind in Java programmiert, sind aber so generisch gehalten, dass man sie auch ohne spezielle Java-Kenntnisse verstehen bzw. die Konzepte leicht auf andere Plattformen übertragen kann.

Fokus

Der Fokus dieses Buches liegt auf übergreifenden Aspekten des Software-Engineering. Technologien oder Konzepte, zu denen es bereits Spezialliteratur gibt, können hier nicht im Detail erklärt werden. Im Besonderen trifft dies auf folgende Themen zu:

Java

Die Beispiele in diesem Buch sind weitgehend in Java gehalten, Java-Kenntnisse sind daher von Vorteil. Eine der besten Einführungen in die wichtigsten Aspekte der Programmiersprache Java ist das „Handbuch der Java-Programmierung“ von Guide Krüger und Thomas Stark [62]. Dieses Handbuch ist auch online[1] verfügbar.

UML

Modelle werden in diesem Buch zumeist in der Unified Modeling Language (UML) dargestellt. In Kapitel 6 werden einige wichtige Diagramme und Konzepte ausreichend vorgestellt, um die Diagramme in diesem Buch zu verstehen. Für eine Vertiefung ist der UML User Guide von Rumbaugh, Jacobson und Booch [78] empfehlenswert.

Relationales Modell

Das relationale Datenbankmodell wird kurz in Abschnitt 9.4.7 eingeführt. In der Entwicklungspraxis ist ein wesentlich detaillierteres Verständnis sowohl des relationalen Modells als auch der Verwendung relationaler Datenbanken (Schema, SQL, Administration, Interfaces etc.) erforderlich. Die

[1] `http://www.javabuch.de`

Modellierung mit Entity-Relationship Diagrammen wird in Abschnitt 6 ebenfalls nur kurz erklärt. Als weiterführende Literatur wird „Relationale Datenbanksysteme: Eine praktische Einführung“ [59] empfohlen

XML

Die extensible Markup Language (XML) steht im Zentrum einer Vielzahl von Standards wie XML DTDs, Schemasprachen, XSL(T), XLink. Eine ganz knappe Einführung wird in Abschnitt 9.4.9 gegeben. In den meisten Projekten wird man an vielen Stellen mit Standards aus dem XML-Umfeld konfrontiert, eine Beschäftigung mit diesem Themenbereich ist daher ebenfalls ratsam. Die meisten XML-Standards sind offen spezifiziert und im Rahmen des World Wide Web Konsortium (W3C)[2] publiziert. Auch die Webservice Standards, die z. B. in Abschnitt 8.6.1 oder 9.1.4 erwähnt werden, sind XML-basierte W3C-Standards.

Software-Testen

In Kapitel 5 wird ein Überblick von Software-Testen sowie eine kurze Beschreibung von Test-Driven Development geboten. Vor allem für die Anwendung im Software-Engineering empfiehlt es sich, weitere vertiefende Literatur zu studieren. Hier ist das Buch „xUnit Test Patterns“ [69] von Gerard Meszaros sehr empfehlenswert.

1.4 Weitere Ressourcen im Web

Dieses Buch adressiert den Bedarf an einer Zusammenstellung praxiserprobter Werkzeuge und Methodiken für Entwickler im Team. Das Buch beschreibt langfristig gültige Konzepte und wird weiterhin durch Ressourcen im Web ergänzt: unter der Adresse `http://bpse.ifs.tuwien.ac.at` sind größere zusammenhängende Beispiele zu finden, die die Beispiele im Buch ergänzen und erweitern. Weiterhin finden sich detailliertere technische Anleitungen zu Frameworks und Patterns sowie Dokumentvorlagen und Beispiele für das Projektmanagement zum Download. Im Best-Practice Software-Engineering Blog `http://best-practice-software-engineering.blogspot.com` diskutieren wir aktuelle Themen und Trends im Software-Engineering. Sie sind herzlich eingeladen, dort an der Diskussion zu teilzunehmen.

[2] `http://www.w3c.org`

1.5 Die Autoren

Alexander Schatten

ist Projektleiter des Best-Practice-Software-Engineering Projekts und verantwortlich für Design- und Integrationsmuster, Techniken und Werkzeuge, Aspekte der Software-Architektur, komponentenorientierte Entwicklung. Alexander Schatten ist Senior-Researcher am Institut für Software-Technik der TU Wien, Berater im Bereich von Software-Architekturen und Software-Engineering mit Schwerpunkt auf Open-Source-Systeme sowie Autor zahlreicher Artikel in Zeitschriften wie Infoweek oder iX[3].

Markus Demolsky

ist Spezialist für Architektur- und Integrationsthemen mit langjähriger Erfahrung in Analyse und Umsetzung internationaler Software-Projekte. Sein besonderes Interesse gilt komponentenorientierten Open-Source-Technologien im Java-EE-Umfeld. In seiner selbstständigen Tätigkeit unterstützt er Unternehmen bei der Auswahl von Open-Source-Technologien im Java-Umfeld und entwickelt Individualsoftware für KMUs. Außerdem schreibt er regelmäßig Fachartikel für das Javamagazin, Eclipsemagazin und die iX[4].

Dietmar Winkler

ist Lehrbeauftragter und Forscher am Institut für Softwaretechnik an der TU Wien mit den Schwerpunkten Software- und Systementwicklung, Qualitätssicherung und Qualitätsmanagement sowie empirischer Softwaretechnik. Aus diesen Tätigkeiten resultierten zahlreiche Publikationen bei internationalen Konferenzen. Weiteres ist er als selbstständiger Berater für Qualitätssicherung, Projekt-, Qualitäts- und Prozessmanagement in Konzeptionierung und Umsetzung in den Industriebereichen Automobilzulieferung und öffentliche Verwaltung tätig. Er ist verantwortlich für die Themenbereiche Software-Lebenszyklus, Vorgehensmodelle, Projektmanagement sowie Qualitätssicherung [5][6].

Stefan Biffl

lehrt und forscht an der Fakultät für Informatik der TU Wien in den Bereichen Software-Engineering, Qualitätssicherung und Projektmanagement.

[3] `http://www.schatten.info`
[4] `http://www.demolsky.at/de/team/mdy`
[5] `http://qse.ifs.tuwien.ac.at/~winkler`
[6] und `http://www.dwq-consulting.at`

Grundlegendes Interesse ist, im steten Wandel der Methoden und Werkzeuge brauchbare Bündel von Ansätzen im Praxiskontext zu erproben und erfolgreiche „Best-Practices“ für Studierende und Praktiker zugänglich zu machen. Die Ansätze im vorliegenden Buch basieren auf Konzepten, die sich in den letzten 20 Jahren in der Arbeit mit kleinen bis mittleren Projektteams als erfolgreich erwiesen haben[7].

Erik Gostischa-Franta

studiert an der TU Wien und ist Tutor in den Bereichen Software-Engineering, Projektmanagement und Qualitätssicherung am Institut für Software-Technik. Besonderes Interesse besteht an agilen Methoden sowie strukturiertem Vorgehen in der Software-Entwicklung mithilfe von definierten Prozessen. Diese Erkenntnisse gibt er an Studierende im Rahmen von Lehrveranstaltungen weiter. Er hat Erfahrung mit Content-Management und missions-kritischen Systemen in der Industrie sowie Software-Demonstrationen bei Messen und Präsentationen bei Kundenstellen. Erik Gostischa-Franta ist Hauptverantwortlich für die Projektmanagement- und Dokumentationsaspekte in diesem Buch mit Schwerpunkt auf der UML und dessen Einsatz im Software-Engineering-Umfeld.

Thomas Östreicher

studiert an der TU Wien und forscht am Institut für Software-Technik. Besonderes Interesse besteht an der Modellierung softwareintensiver Systeme und testgetriebenen agilen Methoden. Thomas Östreicher ist verantwortlich für die Modellierungsaspekte in diesem Buch mit Schwerpunkt auf der UML.

1.6 Danksagung

Dieses Buch ist aus der Praxis unserer Erfahrung mit zahlreichen Software-Teams entstanden. Unser herzlicher Dank gilt insbesondere Michael Zöch, Lukas Lang, Dominik Dorn, Andreas Pieber, Christian Mastnak und Adam Zielinski für interessante Diskussionen sowie zahlreiche konkrete konstruktive Hinweise zur Verbesserung.

[7] `http://qse.ifs.tuwien.ac.at/~biffl`

2 | Lebenszyklus eines Software-Produkts

Software-Produkte folgen – ebenso wie Produkte des täglichen Lebens, wie beispielsweise Mobiltelefone und Autos – einem Lebenszyklus, der bei der ersten Idee beginnt, den Entwicklungsprozess durchläuft, in die Betriebs- und Wartungsphase übergeht und schlussendlich mit der kontrollierten Stilllegung des Produkts endet.

Dieses Kapitel behandelt alle Phasen eines mittelgroßen Software-Produkts aus Expertensicht und beschreibt ausgewählte und wichtige Aspekte in den einzelnen Abschnitten. Diese chronologische Beschreibung kann den Entwicklerteams als Leitlinie dienen, um ein Software-Projekt im Team erfolgreich zu strukturieren und abzuwickeln. Je nach Anwendungsbereich der Software-Lösung sind die jeweiligen Phasen eines Produktlebenszyklus unterschiedlich ausgeprägt und für den Anwender unterschiedlich wahrnehmbar, wie beispielsweise Wartungsprojekte, Migrationsprojekte, Projekte mit öffentlichen Ausschreibungen und klassische Software-Entwicklungsprojekte. Im realen Projektverlauf ist natürlich eine Anpassung an den jeweiligen Anwendungsfall im jeweiligen Projektkontext erforderlich.

Übersicht

2.1 Grundlegende Phasen des Software-Lebenszyklusses

Prozessmodell

Ein Software-Prozessmodell bildet ein einheitliches Rahmenwerk für die Ablaufplanung eines Entwicklungsprojekts, wie sie beispielsweise durch die ISO/IEC 12207 [47] oder durch das „Software-Engineering Body of Knowledge“ (SWEBOK) [88] definiert sind. Grundsätzlich ist es natürlich auch möglich, Software ohne Plan, Rahmen oder Konzept zu erstellen. In diesem sogenannten *Build-and-Fix*-Verfahren werden Anforderungen quasi *„auf Zuruf“* umgesetzt und bestehende Lösungen geändert oder erweitert. Diese unsystematische und unstrukturierte Vorgehensweise mag bei sehr kleinen Software-Lösungen zwar mehr oder weniger gut funktionieren, das Produkt wird jedoch durch sehr viele unkontrollierte Änderungen im Lauf der Zeit schnell unübersichtlich und dadurch unwartbar, unprüfbar, unplanbar und schlicht unbrauchbar. Aufgrund dieser Nachteile ist dieses Vorgehen für Software-Projekte (auch bei scheinbar kleinen Lösungen) denkbar ungeeignet und entwickelt sich schnell zu einer Kostenfalle.

Build-and-Fix

Systematische und strukturierte Software-Entwicklung

Software-Engineering – als Ingenieursdisziplin – beschäftigt sich mit der systematischen und strukturierten Herstellung von Software-Produkten [83]. Ein wesentlicher Aspekt ist dabei der geregelte (also systematische und strukturierte) Ablauf eines Software-Entwicklungsprojekts. Im einfachsten Fall umfasst dieser Ablauf den gesamten Lebenszyklus eines Software-Produkts, von der Konzeptionierung über Analyse, Design, Implementierung und Test bis zur Betriebs- und Wartungsphase. Der Lebenszyklus endet mit dem kontrollierten Außerbetriebsetzen des Software-Produkts.

Projekt- und Qualitätsmanagement begleiten ein Software-Projekt

Die einzelnen Phasen eines Software-Projekts werden dabei vom *Projektmanagement* (PM) und dem *Qualitätsmanagement* (QM) begleitet. Sie unterstützen die Steuerung des Projektablaufs unter Berücksichtigung von Zeit-, Kosten- und Qualitätsaspekten im Hinblick auf die zu erstellenden Produkte. Definierte Zeitpunkte innerhalb des Projektablaufs, sogenannte Entscheidungspunkte *(Decision Gates)* oder Meilensteine *(Milestones)* trennen die einzelnen Phasen in einem Projektplan ab. Diese Zeitpunkte stellen Fixpunkte für das Projekt- und Qualitätsmanagement dar, um den Projektfortschritt zu beobachten und gegebenenfalls steuernd einzugreifen, falls das angestrebte Qualitätsniveau zu diesem Zeitpunkt nicht den definierten Erwartungen entspricht. Dazu ist es notwendig, entsprechende Kennzahlen und Metriken zur Erhebung des Projektstatus aus der Sicht des Projektmanagements (siehe Kapitel 4) und des Qualitätsmanagements (siehe Kapitel 5) einzuführen. Diese Kennzahlen und Metriken dienen als Entscheidungsgrundlage für den Projekt- und Qualitätsleiter, um beispielsweise im Rahmen eines Vorgehensmodells (siehe Kapitel 3) zusätzliche Iterationen einzuplanen (etwa in einem inkrementellen Vorgehensmodell) oder weitere Maßnahmen zur Qualitätssicherung (siehe Kapitel 5) anzustoßen.

Kennzahlen und Metriken zur Projektverfolgung

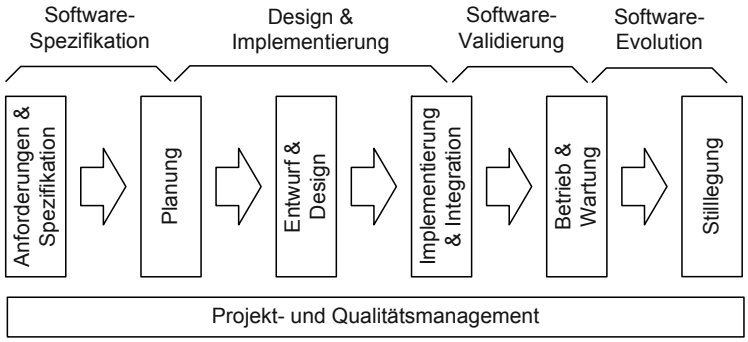


Abbildung 2.1
Basiskonzept des Software-Lebenszyklus.

Vier grobe Phasen

Abbildung 2.1 visualisiert die vier grundlegenden Schritte des Lebenszyklusses sowie eine detailliertere Gliederung anhand konkreter Phasen [83]. Die Schritte *Software-Spezifikation* (Projektdefinition und -planung), *Design und Implementierung* (Detailplanung und konkrete technische Umsetzung), *Software-Validierung* (Überprüfung des Produkts im Hinblick auf die Spezifikation und auf die Kundenwünsche) und *Software-Evolution* (Weiterentwicklung des Produkts) bilden eine sehr grobe Struktur eines Software-Projekts und finden sich nahezu in allen Projekten wieder. Eine detailliertere Betrachtung des Software-Lebenszyklusses beinhaltet eine feinere Untergliederung dieser vier groben Phasen, die in weiterer Folge detaillierter behandelt werden. Die technischen Phasen beinhalten dabei *Anforderungen & Spezifikation*, *Planung*, *Entwurf & Design*, *Implementierung & Integration*, *Betrieb & Wartung* sowie *Stilllegung*.

Projekt- und Qualitätsmanagement sind übergreifende Aktivitäten mit definierten Schnittstellen zu allen technischen Phasen. In weiterer Folge werden wichtige Aspekte in den jeweiligen Phasen des Software-Lebenszyklusses im Sinn von Best-Practice Software-Engineering identifiziert und im Überblick dargestellt.

2.2 Übergreifende Aktivitäten

Begleitend zu den unterschiedlichen technischen Phasen im Rahmen eines Software-Entwicklungsprojekts sind übergreifende Aktivitäten erforderlich, um einen erfolgreichen Projektablauf zu ermöglichen. Dies sind im Wesentlichen *Projektmanagement* und *Qualitätsmanagement*.

2.2.1 Projektmanagement

Planung, Kontrolle, Steuerung

Das Projektmanagement (PM) beschäftigt sich mit der Planung, Kontrolle und Steuerung von Projekten und stellt einen organisatorischen Rahmen für eine erfolgreiche Projektabwicklung zur Verfügung. Gemäß DIN 69901 ist ein Projekt ein *„Vorhaben, das im Wesentlichen durch die Einmaligkeit der*

Bedingungen in ihrer Gesamtheit gekennzeichnet ist" [24]. Diese Bedingungen umfassen sowohl Zielvorgaben des Projekts und konkrete Anforderungen an das zu erstellende Produkt, als auch finanzielle, zeitliche und personelle Restriktionen und sich daraus ergebende Risiken. Diese Aktivitäten werden von zahlreichen Faktoren beeinflusst: So ist die Planung von Projekten nicht nur vom anfallendem Aufwand und den zur Verfügung stehenden Ressourcen und Personen abhängig, sondern unter anderem auch von ihren technischen und fachlichen Fähigkeiten, sowie ihrem sozialem Umgang in einem Team. *„Software-Engineering is a value-based contact sport"*, wie es Barry Boehm treffend beschreibt [13]. Das bedeutet, dass jedes Teammitglied seinen Beitrag zum Gesamtprojekt leistet und somit auch einen Anteil am Projekterfolg hat. Aufgrund der Bedeutung des Projektmanagements in einem Software-Entwicklungsprojekt ist diesem Themenschwerpunkt das Kapitel 4 gewidmet.

„Software-Engineering is a value-based contact sport"

2.2.2 Qualitätsmanagement

Unter Qualitätsmanagement (QM) versteht man alle zu erfüllenden Aufgaben, die zur Herstellung eines qualitativ hochwertigen Produkts notwendig sind. Qualitätsmanagement und Projektmanagement ergänzen einander und begleiten das Software-Entwicklungsprojekt über den gesamten Lebenszyklus. In der Praxis zielt Projektmanagement eher auf technische, finanzielle und planerische Aspekte ab, Qualitätsmanagement rückt den qualitativen Kundennutzen in den Vordergrund. Eine konkrete Definition von *„Qualität"* findet sich beispielsweise in der ISO 9000: *„Die Beschaffenheit einer Einheit bezüglich ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen"* [49]. In Anlehnung an diese Definition verwenden wir den Qualitätsbegriff in diesem Buch aus zwei unterschiedlichen Sichten:

Herstellen hochwertiger Produkte

Sichten auf Qualität

1. *Erfüllung der spezifizierten Eigenschaften eines Produkts*: Diese Sichtweise beleuchtet das Produkt im Hinblick auf die spezifizierten Vorgaben, also ob die erstellte Lösung auch der Spezifikation entspricht, das Produkt also *richtig erstellt* wurde. Diese Sichtweise wird generell auch als *Verifikation* bezeichnet.
2. *Erfüllung der vom Kunden gewünschten Eigenschaften eines Produkts*: Diese Sichtweise rückt den Kundenwunsch und die Kundenanforderungen in den Vordergrund, also ob das *richtige Produkt* erstellt wurde. Diese Sichtweise wird auch als *Validierung* bezeichnet.

Zentrale Fragestellungen sind, (a) welche Eigenschaften ein Software-Produkt erfüllen muss und wie man dazu kommt, (b) wie diese Eigenschaften effizient umgesetzt werden können und (c) wie die umgesetzten Eigenschaften überprüft werden können. Produkteigenschaften lassen sich beispielsweise aus den Kundenanforderungen und anderen übergeordneten Vorgaben, etwa gesetzliche Regelungen, ableiten (siehe dazu Ab-

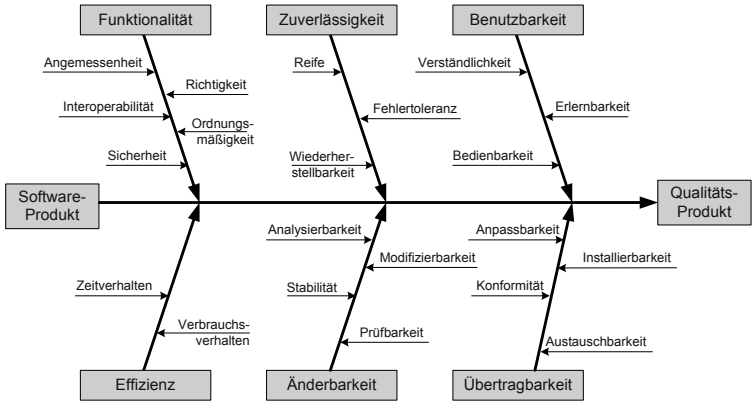


Abbildung 2.2
Qualitätsmerkmale nach ISO/IEC 9126-1 [50].

schnitt 2.3). Die Umsetzung erfolgt typischerweise – je nach Anwendungsdomäne – über ein konkretes Vorgehensmodell (siehe dazu Kapitel 3). Zur Überprüfung (Verifikation und Validierung) der erstellten (Teil-)Lösung können geeignete Maßnahmen aus dem Bereich Qualitätssicherung eingesetzt werden (siehe dazu Kapitel 5).

Ordnungsrahmen von Produkteigenschaften

Die Eigenschaften eines Produkts lassen sich zwar direkt aus den Kundenanforderungen ableiten, allerdings empfiehlt sich hier die Verwendung eines Ordnungsrahmens, um eine gewisse Vollständigkeit zu erreichen. Wichtig ist hier nicht nur die Betrachtung der funktionalen Anforderungen, sondern auch begleitender Anforderungen, die zwar nicht direkt ersichtlich sind, aber für ein erfolgreiches Produkt unumgänglich sind.

Qualitätsmerkmale nach ISO/IEC 9126-1

Beispielsweise stellt die ISO *International Organization for Standardization* einen Klassifikationsansatz zur systematischen Einteilung von Qualitätseigenschaften oder -merkmalen zur Verfügung. Abbildung 2.2 fasst wichtige Qualitätsmerkmale in Anlehnung an die ISO/IEC 9126-1 zusammen [50]. Diese Klassifikation ermöglicht eine systematische Betrachtung der Kundenanforderungen im Rahmen der Anforderungsanalyse (siehe Abschnitt 2.3). Dieser Klassifikationsansatz beschreibt eine grobe Einteilung von Qualitätsmerkmalen:

- Das Qualitätsmerkmal *Funktionalität* umfasst alle (funktionalen) Anforderungen an das System, d. h., welche Aufgaben durch die zu erstellende Lösung erfüllt werden sollen, z. B. bezüglich Korrektheit und Sicherheit.
- Die *Zuverlässigkeit* beschreibt die Fähigkeit eines Systems, die verlangte Funktionalität unter gegebenen Randbedingungen in gegebener Zeit zu erfüllen, z. B. Produktreife und Fehlerverhalten.
- Die *Benutzbarkeit* definiert die Verwendbarkeit durch den Endanwender, wie z. B. Verständlichkeit der Benutzerführung und Erlernbarkeit durch den Anwender. In diese Kategorie fällt typischerweise die Gestaltung der Benutzeroberfläche.

- Die *Effizienz* umfasst primär das Zeitverhalten bzw. das Ausmaß, in dem ein System seine Leistungen mit einem Minimum an Ressourcen erbringen kann.
- *Änderbarkeit* ist eine Eigenschaft, vorgegebene Änderungen oder Erweiterungen an dem Software-Produkt durchführen zu können.
- *Übertragbarkeit* bezeichnet die Fähigkeit, das Software-System in eine andere Umgebung (z. B. auf eine andere Plattform) zu übertragen bzw. dort einzusetzen.

Jedes Projekt ist unterschiedlich

Aufgrund der unterschiedlichen Anforderungen an die Produkte ist meist nur eine Auswahl der genannten Qualitätsmerkmale relevant. Im Rahmen des Projekts sollte man aber zumindest über alle Qualitätsmerkmale nachdenken, um den notwendigen Bedarf zu hinterfragen. Die Auswahl der relevanten Qualitätsmerkmale hängt auch von der Anwendungsdomäne ab. Je nachdem, ob ein administratives Software-System (etwa ein Informationssystem) oder eine sicherheitskritische Anwendung (etwa eine Kraftwerkssteuerung) erstellt werden soll, sind andere Aspekte der Qualitätsmerkmale zu betrachten und eine geeignete Vorgehensweise zu definieren.

Sind die Anforderungen und Merkmale definiert, kann das Projekt sauber strukturiert, geplant und umgesetzt werden – sofern die Anforderungen stabil bleiben. In diesem Fall bieten sich systematische Software-Prozesse an, wie sie im Kapitel 3 beschrieben werden. In einigen Fälle (beispielsweise im öffentlichen oder im sicherheitskritischen Bereich) sind diese systematischen Vorgehensweisen auch explizit vorgeschrieben.

In der gängigen Praxis stehen die Anforderungen und erwarteten Merkmale leider nicht bereits zu Beginn fest und bleiben auch nicht stabil. Daher muss man davon ausgehen, dass sich Anforderungen und Merkmale laufend (auch häufig) ändern können und auch werden. Vor diesem Hintergrund sind agile Ansätze gefragt, die auf diese Änderungen geeignet reagieren können. Beispielsweise empfiehlt sich hier ein agiler Ansatz, der das Projekt inkrementell aufbaut. Beispiele für diese Modelle, wie etwa SCRUM, werden in Kapitel 3.8 erläutert. Obwohl in zahlreichen Projekten eine agile Vorgehensweise verfolgt wird, sind dennoch Anforderungen (beispielsweise Plattformabhängigkeit oder Skalierbarkeit), die einen direkten Einfluss auf die Architektur haben, frühzeitig zu berücksichtigen. In solchen Fällen gilt es, die Produktqualität frühzeitig einzuschätzen und zu verbessern. Dabei ist es wichtig, Fehler möglichst frühzeitig zu erkennen und zu korrigieren, da die Kosten dramatisch ansteigen, je später ein Fehler im Produkt erkannt wird. Reviews und Inspektionen als statische Methoden und Software-Testen als dynamische Methode sind weitverbreitete Methoden der Qualitätssicherung (siehe dazu Kapitel 5).

Frühzeitige Fehlererkennung und Korrektur

2.2.3 Reviews und Inspektionen

Reviews und Inspektionen sind analytische Methoden der Qualitätssicherung, die eine systematische Fehlererkennung (und Lokalisierung) während des gesamten Entwicklungsprozesses ermöglichen. Besonders bewährt haben sich Reviews in Inspektionen speziell in frühen Phasen der Entwicklung, da – im Gegensatz zu Tests – kein ausführbarer Softwarecode benötigt wird. Reviews und Inspektionen können beispielsweise für Anforderungen, Design, Testfälle aber natürlich auch Softwarecode eingesetzt werden. Je nach Schwerpunkt existieren Ansätze, die unterschiedliche Fehler- und Risikoklassen oder Dokumenttypen adressieren. Neben speziellen Anwendungen, wie beispielsweise Design-Inspections oder Code-Reviews, werden sie typischerweise auch für die Freigabe von Dokumenten bei Meilensteinen oder an Phasengrenzen eingesetzt. Reviews und Inspektionen als wichtige Vertreter analytischer Methoden der Qualitätssicherung sowie die konkrete Vorgehensweisen werden in Abschnitt 5.3 detailliert beschrieben.

Analytische Methoden der QS

2.2.4 Software-Tests

Traditionelle Prozesse

Im Gegensatz zu Reviews und Inspektionen benötigen Software-Tests ausführbaren Code, der in traditionellen Prozessen, wie beispielsweise im Wasserfallmodell (siehe Abschnitt 3.2), typischerweise erst recht spät im Entwicklungsprojekt vorliegt. Diese Tests finden auf unterschiedlichen Ebenen (beispielsweise Komponenten- oder Unittests, Integrationstests und System- oder Abnahmetests) Anwendung und erfüllen unterschiedlich definierte Aufgaben. Testansätze werden ebenfalls im Kapitel *„Qualitätssicherung und Test-Driven Development“* in Abschnitt 5.6 detailliert beschrieben.

Test-Driven Development

Test-Driven Development (oder auch *Test-First Development*) ist ein Ansatz aus dem Bereich der agilen Software-Entwicklung, bei dem Implementierung und Test quasi parallel ablaufen. Die Grundidee dabei ist, zuerst die Testfälle (zur Beschreibung der Funktionalität) zu erstellen und erst im Anschluss daran die eigentliche Funktionalität zu implementieren und zu testen. Diese Vorgehensweise hat unter anderem den Vorteil, dass die Anforderungen in Form von Testfällen klar und nachvollziehbar dokumentiert (und auch verstanden) sind und entsprechend umgesetzt werden können (siehe dazu Kapitel 5).

2.3 Anforderungen und Spezifikation

„The hardest single part of building a system, is deciding what to build.“ (Boehm, 1997)

Anforderungen *(Requirements)* stehen am Beginn jedes Software-Entwicklungsprojekts. Im Rahmen der Anforderungen wird definiert, welches Problem gelöst werden soll und welche Leistung das geplante Produkt erbringen soll. Durch die Berücksichtigung möglichst aller beteiligten Personen

(Stakeholder) gestaltet sich beispielsweise der erste Kontakt zwischen Auftraggeber (Kunde) und Entwickler sehr facettenreich. Die Sicht des Kunden auf das Produkt wird von seinem eigenen Nutzen geprägt sein – allerdings weiß er häufig nicht genau, was er wirklich will und wie das Vorhaben realisiert werden kann. Der Entwickler hat seine eigene (oft technische) Sicht auf das Projekt und hat ebenfalls keine genaue Vorstellung, was der Kunde tatsächlich braucht. Er verfügt aber über das notwendige Know-how möglicher technischer Lösungen, die – aus dem Blickwinkel des Entwicklers – den Vorstellungen des Kunden entsprechen könnten. In der Anforderungsphase *(Requirements Definition)* muss es also einen Kompromiss aller beteiligten Personen geben, sodass einerseits klar ist, was der Auftraggeber benötigt und erwartet und andererseits, was der Entwickler wie verwirklichen kann und soll.

Systematische Anforderungsanalyse

Die *Anforderungsanalyse* oder *Anforderungsspezifikation* beschreibt den Kompromiss aller beteiligten Stakeholder im Hinblick auf das zu erstellende Produkt aus der aktuellen Sicht aller Beteiligten. Typische Kompromisse betreffen beispielsweise die Funktionalität, natürlich die Kosten, einen realisierbaren Abgabetermin und andere Leistungsparameter, wie beispielweise die Zuverlässigkeit des Produkts. Je nach Projekt erstellt das Team, das sowohl Vertreter von Kunden und Anwendern als auch Vertreter des Entwicklungsteams umfassen soll, im Rahmen der Anforderungsanalyse Produkte, wie beispielsweise Systembeschreibungen, Begriffsverzeichnisse zur Verbesserung des gemeinsamen Verständnisses, Schnittstellendefinitionen, Story-Boards, Anwendungsfalldiagramme und entsprechende Beschreibungen und Prototypen. Nach Fertigstellung der Anforderungsspezifikation erfolgt typischerweise eine qualitätssichernde Maßnahme – meist ein Review oder eine Inspektion, um die erstellten Produkte im Hinblick auf Richtigkeit, Vollständigkeit und Konsistenz zu prüfen. In vielen Fällen, beispielsweise in öffentlichen oder sicherheitskritischen Projekten, ist diese Anforderungsspezifikation Vertragsbestandteil und dient als Basis für die weitere Projektplanung. Das setzt jedoch voraus, dass die Anforderungen weitgehend stabil sind.

Iterationen

In der gängigen Praxis können sich diese Anforderungen jedoch häufig ändern, vor allem, wenn sich der Kunde noch nicht im Klaren ist, was konkret umgesetzt werden soll, oder dem Entwickler nicht klar ist, was der Kunde genau benötigt. In diesem Fall empfiehlt sich ein agiler Ansatz mit mehreren Iterationen, die auf einer stabilen Architektur aufbauen. In derartigen Projekten ist es durchaus üblich, nur wichtige Teile (beispielsweise Anforderungen mit hoher Priorität) eines Projekts in einer Iteration zu betrachten und im Rahmen der Anforderungsanalyse zu bearbeiten. Die Anforderungsanalyse kann daher als Teil einer Iteration betrachtet werden, die bei jedem neuen Durchlauf an aktuelle Gegebenheiten angepasst wird. SCRUM ist ein typischer Vertreter eines agilen Prozesses, der diesem Ansatz folgt (siehe Abschnitt 3.8). In jedem Fall ist es notwendig, die Anforderungen des Kunden an das zu erstellende System explizit zu betrachten.

Warum sind Anforderungen aber so wichtig?

Warum sind Anforderungen wichtig?

- Anforderungen repräsentieren die „reale Welt“ und drücken das gewünschte Verhalten aus Nutzersicht aus. Der Anwender muss mit dem Produkt arbeiten können.
- Unterschiedliche Stakeholder (etwa Kunde/Anwender und Entwickler) haben unterschiedliche Interessen und Erwartungshaltungen und betrachten das Projekt aus ihren individuellen Blickwinkeln (abhängig vom individuellen Fachwissen).
- Durch Anforderungen wird ein *gemeinsames Verständnis* über die Leistung des Produkts hergestellt.
- Typischerweise werden Anforderungen in Textform beschrieben bzw. grafisch dargestellt. Dazu bieten sich beispielsweise Story-Boards, User-Stories oder eine Modellierung der Anforderungen an (beispielsweise mittels UML Use-Cases, wie sie in Abschnitt 6.1.2 beschrieben werden).
- *Anforderungen müssen testbar und nachvollziehbar sein*! Werkzeuge für das Anforderungsmanagement wie etwa „Doors“ oder „Rational Requisite Pro“ unterstützen das Team bei der Bearbeitung, Nachverfolgung und Organisation von Anforderungen.

Eine zentrale Herausforderung in der Software-Entwicklung sind *instabile*, *unvollständige* und *unklare Anforderungen*. Im Idealfall gilt es, diese Änderungen an den Anforderungen zu reduzieren oder gänzlich zu vermeiden. Die Praxis sieht allerdings anders aus: Änderungen treten während des gesamten Projekts auf. Dementsprechend ist ein kontrollierter Umgang mit sich ändernden Anforderungen unumgänglich. Eine Grundvoraussetzung dafür ist aber, dass das „Fundament“, wie beispielsweise die Architektur, stabil bleibt. Kritische Änderungen, die beispielsweise die Architektur und das Design betreffen, können hohe Kosten für Korrekturen hervorrufen und bis zum Projektabbruch führen. Untersuchungen im industriellen Umfeld zeigen diese Tatsache deutlich auf.

Chaos-Report

Der Chaos-Report dokumentiert Ergebnisse aus einer Untersuchung von über 8300 Anwendungen bei 365 Unternehmen im Hinblick auf Projektfehlschläge. Die häufigsten Gründe für den Projektabbruch betreffen den Problembereich *Anforderungen* [17]. Die Abbildung 2.3 beschreibt die Ergebnisse des Chaos-Report im Überblick. Anforderungen beschreiben den Kundenwunsch an ein Software-System, um ein konkretes Problem zu lösen. Daher sollte möglichst frühzeitig feststehen, was der Kunde konkret benötigt, um eine qualitativ hochwertige Lösung erstellen zu können. Da das in vielen Fällen nicht möglich ist und die Anforderungen instabil sind, empfiehlt sich ein iterativer Prozess um – aufbauend auf einem stabilen

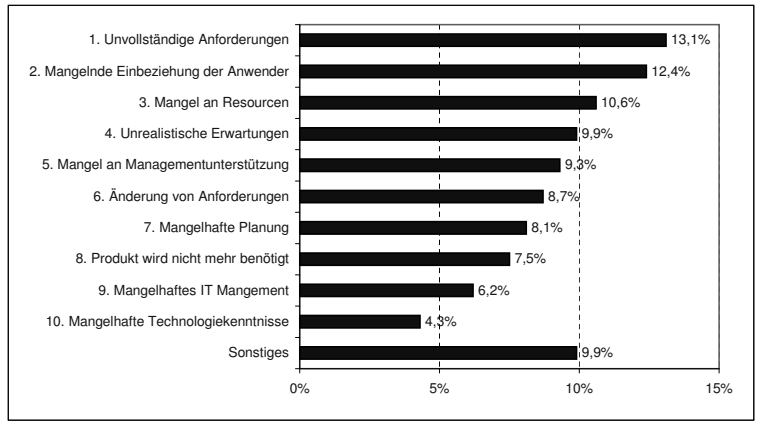


Abbildung 2.3
Chaos-Report [17].

Fundament (der Architektur) – das System schrittweise zu entwickeln. Unabhängig davon, sind neben den Endanwendern alle involvierten Stakeholder (Rollen innerhalb des Entwicklungsteams, Kunden, Endanwender und Wartungspersonal) zu berücksichtigen, da unterschiedliche Rollen typischerweise unterschiedliche Sichten auf das Projekt haben. Generell stellt sich aber die Frage, welche Arten von Anforderungen überhaupt existieren und wie diese möglichst vollständig erfasst und beschrieben werden können.

2.3.1 Arten von Anforderungen

Klassifikation von Anforderungen

Zur vollständigen Bearbeitung von Anforderungen ist es sinnvoll, Anforderungen in unterschiedliche Kategorien zu klassifizieren. Generell lassen sich Anforderungen in vier grobe Kategorien einteilen [71]:

- *Funktionale Anforderungen* bilden das eigentliche Systemverhalten und die jeweiligen Funktionen des zu erstellenden Produkts ab.
- *Nichtfunktionale Anforderungen* zielen auf Qualitätsmerkmale ab, die beispielsweise auf die Leistungsfähigkeit des Systems abzielen. Eine mögliche Klassifizierung dieser nicht funktionalen Anforderungen ist in Abbildung 2.2 im Rahmen eines Qualitätsmodells dargestellt.
- *Designbedingungen* legen die technischen Rahmenbedingungen, wie beispielsweise Entwicklungsumgebung und Zielplattform fest.
- *Prozessbedingungen* definieren die Rahmenbedingungen für die Entwicklung des Software-Produkts in dem ein konkretes Vorgehen definiert wird.

Funktionale Anforderungen

Funktionale Anforderungen (Functional Requirements) umfassen und beschreiben das erwartete Systemverhalten aus der Sicht unterschiedlicher Stakeholder, etwa aus der Sicht eines Kunden *(Customer)*, Endanwenders

(User) oder eines Entwicklers *(Developer)*. Eine zentrale Herausforderung im Rahmen der Anforderungserhebung und -definition ist die Herstellung eines gemeinsamen Verständnisses der zu erstellenden Software-Lösung, da die Projektteilnehmer typischerweise aus unterschiedlichen Domänen stammen und ein unterschiedliches Vokabular, Begriffsverständnis und ein domänenspezifisches Hintergrundwissen mitbringen. Erst ein gemeinsames Verständnis der zu erstellenden Software-Lösung ermöglicht auch eine erfolgreiche Projektdurchführung. UML Anwendungsfalldiagramme *(Use-Cases)* sind ein geeigneter Ansatz zur Visualisierung der Anforderungen (speziell für funktionale Anforderungen) in einem System. In der agilen Software-Entwicklung haben sich sogenannte User-Stories, Story-Boards und Feature-Lists etabliert, die für die Beschreibung von Anforderungen eingesetzt werden. Diese Darstellungsarten werden im Kapitel 6 vorgestellt.

Nicht funktionale Anforderungen

Während funktionale Anforderungen direkt das Systemverhalten beschreiben und die jeweiligen Funktionen in den Vordergrund rücken, beschreiben *nicht funktionale Anforderung (Non-Functional Requirements)* zusätzliche Systemattribute und Qualitätsmerkmale, die nicht unmittelbar die Funktionalität des Systems adressieren. Derartige nicht funktionale Anforderungen umfassen beispielsweise Anwenderfreundlichkeit *(Usability)*, Effizienz oder Performance. Die ISO/IEC 9126-1 stellt etwa einen Ordnungsrahmen für die Klassifikation von unterschiedlichen Anforderungsklassen bereit (siehe Abbildung 2.2) [50]. Während die Überprüfung der funktionalen Anforderungen recht einfach zu sein scheint, müssen auch nicht funktionale Anforderung überprüfbar sein (siehe dazu Kapitel 5).

Designbedingungen

Bei der Erhebung von Anforderungen müssen nicht nur die produktrelevanten Eigenschaften, sondern auch Eigenschaften, die auf den Herstellungsprozess abzielen, adressiert werden. Diese Design- und Prozessbedingungen beeinflussen den Prozessablauf und somit auch den Projektverlauf. *Designbedingungen (Design Constraints)* beschreiben die technischen Rahmenbedingungen, die bei der Entwicklung zu berücksichtigen sind. Designbedingungen umfassen beispielsweise die zu verwendende Software-Architektur, Zielplattform oder definieren eine verteilte Nutzung der Anwendung. Bei der Überprüfung müssen diese Designbedingungen ebenfalls berücksichtigt werden.

Prozessbedingungen

Prozessbedingungen (Process Contraints) definieren die grundlegende Vorgehensweise bei der Entwicklung eines Software-Produkts. Sie legen also den konkreten Software-Prozess fest, der für das aktuelle Projekt anzuwenden ist und definieren die einzusetzenden Methoden zur Erstellung der Produkte im Rahmen des Projekts. Beispielsweise können hier Vorgehensmodelle (siehe Kapitel 3) zum Einsatz kommen. Je nach Projektumfeld eignen sich systematische und strukturierte Prozesse (beispielsweise für sicherheitskritische Anwendungen oder Projekte im öffentlichen Bereich) aber auch agile Ansätze (beispielsweise bei instabilen Anforderungen oder bei inkrementellen Entwicklungen) zur Durchführung des Projekts. In der in-

dustriellen Praxis ist ein Trend von den „schwergewichtigen“ hin zu flexiblen (agilen) Prozessmodellen feststellbar. In jedem Fall sollten Prozesse eingesetzt werden, die auf den jeweiligen Anwendungsfall oder das Unternehmen abgestimmt sind *(Process Tailoring)*. Tailoring ermöglicht die Anpassung eines konkreten Vorgehensmodells an individuelle Projektgegebenheiten (siehe Abschnitt 3.9). Im öffentlichen Bereich hat sich beispielsweise das V-Modell XT [43] etabliert, das unter anderem auch Mechanismen für Prozesstailoring beinhaltet (siehe Abschnitt 3.4). Das V-Modell XT wird bei der Vergabe von Projekten im öffentlichen Bereich auch zunehmend im Rahmen der Ausschreibung gefordert, wobei das V-Modell XT auch agile Projektdurchführungsstrategien unterstützt – also auch dem aktuell feststellbaren Trend in der industriellen Praxis folgt.

2.3.2 Anforderungsprozess

Anforderungsprozess

Aufgrund der Vielzahl an unterschiedlichen Anforderungen ist es sinnvoll, bei der Erhebung von Anforderungen einem definiertem Prozess zu folgen. Dadurch wird einerseits die Nachvollziehbarkeit und Vollständigkeit als auch ein gewisses Maß an Qualität der Anforderungen ermöglicht.

Stabilität von Anforderungen

Sowohl die möglichst vollständige Erhebung von Anforderungen (oder von Teilen von Anforderungen im Rahmen einer Iteration) als auch die Überprüfung der definierten Anforderungen unter Berücksichtigung aller relevanter Rollen *(Stakeholder)* ist zeitaufwändig und nicht trivial. Je nach Stabilität der Anforderungen muss eine geeignete Vorgehensweise *(Software-Prozess)* gewählt werden, um die Anforderungen zielgerichtet umsetzen zu können. Bei stabilen und sich kaum mehr ändernden Anforderungen kann ein systematisches und phasenorientiertes Prozessmodell, wie etwa das V-Modell (siehe Abschnitt 3.3), gewählt werden; sind die Anforderungen instabil und ändern sich während des Entwicklungsprozesses häufig, sollte eher ein agiler Ansatz, wie etwa SCRUM (siehe Abschnitt 3.8), eingesetzt werden.

Priorisierung von Anforderungen

Je nach Projektgröße existieren unzählige Anforderungen in verschiedenen Kategorien, die im Projekt umgesetzt werden sollen. Allerdings sind nicht alle Anforderungen gleich wichtig, daher sollten Überlegungen über die Reihenfolge der Umsetzung angestellt werden. Eine Priorisierung von Anforderungen in *must-be*-, *expected*- und *nice-to-have*-Kritieren ist im praktischen Umfeld sehr empfehlenswert. *Must-be*- und *nice-to-have*-Kriterien sind im Regelfall einfach fassbar, da diese typischerweise transparent sind und klar kommuniziert werden. Speziell in agilen Prozessen wie SCRUM ist diese Priorisierung sowie der Umgang mit Änderungen in der Priorisierung ein wesentlicher Teil des Prozesses.

Erwartete Anforderungen (expected) an ein System werden zwar implizit angenommen, aber in seltenen Fällen auch explizit bekannt gegeben. Diese Erwartungen sind in unterschiedlichen Anwendungsdomänen auch

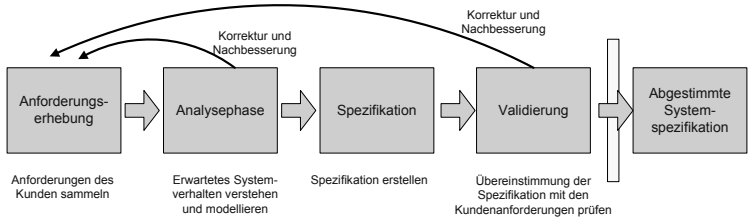


Abbildung 2.4 *Prozess zur Anforderungs-erhebung [71].*

unterschiedlich – je nach Erwartungshaltung der involvierten Stakeholder. Die Herausforderung besteht darin, implizite Anforderungen explizit zu machen, um sie entsprechend umsetzen und auch überprüfen zu können. Je nach Hintergrund und Erfahrung der beteiligten Stakeholder existieren unterschiedliche (eventuell auch sich widersprechende) Erwartungshaltungen. Beispielsweise kann die *Einfachheit der Benutzerführung* unterschiedlich interpretiert werden: Der Endkunde könnte erwarten, mithilfe eines Wizzards oder Assistenten durch einen Bestellvorgang durchgeführt zu werden. Ein Entwickler könnte eine einfache Benutzerführung durch eine klare Strukturierung der Menüs ausdrücken. Daher ist es erforderlich, alle (auch implizit angenommene) Anforderungen explizit zu definieren und gegebenenfalls zu diskutieren. In vielen Fällen kann es auch sinnvoll sein, *schnelle Prototypen* zu erstellen, um einerseits die Kommunikation zwischen den unterschiedlichen Stakeholdern zu verbessern und andererseits dem Kunden etwas *Greifbares* zur Verfügung zu stellen. Durch Prototypen kann sich der Kunde und Anwender schnell ein Bild davon machen, wie beispielsweise der Bestellvorgang umgesetzt werden könnte.

Implizite Anforderungen explizit definieren

Andere Arten der Priorisierung betrachten beispielsweise den zu erwartenden (Kunden-)Nutzen der konkreten Anforderung *(value-based)* oder das erwartete Risiko bei der Umsetzung *(risk-based)* [9].

Daher ist es sinnvoll, speziell in der frühen Phase der Anforderungserhebung, einem definiertem Prozess zu folgen (siehe Abbildung 2.4), an dessen Ende abgestimmte Systemspezifikationen oder klar definierte User-Stories stehen. Dieser Prozess besteht aus vier wesentlichen Schritten:

Prozess zur Anforderungserhebung

- *Anforderungserhebung (Requirements Elicitation)*. Ziel ist es, abgestimmte Anforderungen in einem definierten Projektumfeld zu sammeln, zu klassifizieren und zu priorisieren. Meist wird dieser Schritt in Form eines Brainstormings im Team umgesetzt.
- *Analysephase*. Die Modellierung von Anforderungen (etwa mit UML Use-Cases) ermöglicht durch den Einsatz von Visualisierungen ein besseres Verständnis des erwarteten Systemverhaltens. Das Kapitel 6 beschreibt die wesentlichen Modellierungsvarianten im Detail.
- *Spezifikation*. Modellierte und bewertete Anforderungen fließen in Spezifikationen oder User-Stories ein, die das Systemverhalten in Bezug auf die Anforderungen beschreiben. Diese Dokumente sind die

Grundlage für die spätere Planung und technische Entwicklung (siehe dazu Abschnitt 2.5).

- *Validierung*. Ein wichtiger Schritt am Ende der Anforderungsdefinition ist Verifikation und die Validierung (also die Überprüfung im Hinblick auf die Erwartungen des Auftraggebers). Dazu können beispielsweise Reviews und Inspektionen verwendet werden (siehe Abschnitt 5.3). Nach Abschluss einer erfolgreichen Validierung liegt die Anforderungsspezifikation für das zu entwickelnde Produkt vor.

Werkzeuge zur Anforderungserhebung

Durch die Vielzahl an Produktanforderungen und die Anzahl der beteiligten Personen kann es sinnvoll sein, Werkzeuge für die Anforderungserhebung einzusetzen. EasyWinWin ist beispielsweise ein konkretes Werkzeug zur Unterstützung des Anforderungserhebungsprozesses [35], das sowohl die Erhebung als auch die Klassifikation und Priorisierung von Anforderungen und Risiken ermöglicht.

2.3.3 Wesentliche Artefakte der Anforderungsphase

Artefakte der Anforderungsphase

Aus der Anforderungs- und Spezifikationsphase entstehen Dokumente, die als wesentliche Grundlage für das Projekt verwendet werden und sehr häufig auch die Basis für die Abnahme eines Projekts darstellen. Eines der wichtigsten Artefakte ist der Projektauftrag.

Projektauftrag

Der *Projektauftrag* ist die Grundlage für den Start eines Projekts. Er enthält vor allem die Problemstellungen, eine Liste der Risiken und die Vereinbarungen mit dem Auftraggeber und definiert die konkreten Ziele (und auch Nicht-Ziele), die durch das Projekt verfolgt (bzw. nicht verfolgt) werden. Details zum Projektauftrag sind im Abschnitt 4.2.3 zu finden.

Domänenmodell

Das *Domänenmodell* definiert die statische Struktur des geplanten Systems in der jeweiligen Domäne, einschließlich deren Systemabgrenzungen. Das Domänenmodell wird detailliert in Abschnitt 6.1.3 beschrieben.

Anwendungsfälle und User-Stories

Anwendungsfälle und User-Stories ermöglichen die Modellierung und Beschreibung der Kundenanforderungen in einer Form, die für alle beteiligten Stakeholder verständlich ist. Ziel dabei ist es, gute und aussagekräftige Anwendungsfälle zu ermitteln, die auf derselben Abstraktionsebene liegen, also einen vergleichbaren Detailgrad zu erreichen. Es macht beispielsweise nur wenig Sinn, einen Anwendungsfall sehr detailliert zu erstellen, andere dagegen auf einem sehr hohen Abstraktionsniveau. Dadurch entsteht schnell ein falscher Eindruck der Komplexität, und es droht ein Verlust des Überblicks über die Systemanforderungen. Beispielsweise eignen sich Anwendungsszenarien, UML Anwendungsfalldiagramme, Story-Boards oder User-Stories gut für diesen Zweck (siehe Abschnitt 6.1.2).

Wesentlich dabei ist, dass die Anwendungsfälle das Verhalten (von außen nach innen) beschreiben, während das Domänenmodell die Struktur des

Systems (von innen nach außen) abbildet. Kernaufgabe der Analysephase ist, die beiden Modelle „in der Mitte“ aufeinander abzustimmen, um die Spezifikation des geplanten Systems korrekt, vollständig und konsistent zu halten.

Besprechungsprotokolle

Besprechungsprotokolle sind auch bereits in der Anforderungsphase relevant, um im Zweifelsfall auf dokumentierte Entscheidungen, beispielsweise für die Priorisierung von Anforderungen, zurückgreifen zu können. Diese Rückverfolgbarkeit erlangt auch eine hohe Bedeutung, falls eine getroffene Technologieentscheidungen, etwa im Hinblick auf die Architektur, nachvollzogen werden muss (Begründung für die Auswahl und an der Entscheidung beteiligte Personen). Aus den Protokollen kann speziell auch abgeleitet werden, warum eine spezielle Technologie zwar diskutiert, aber nicht eingesetzt wurde. Protokolle stellen also eine „lebende“ Dokumentation des Projekts dar. Weitere Details zu Protokollen und weiteren effizienten Kommunikationsmöglichkeiten im Team sind in Abschnitt 10.8 zu finden.

2.4 Projektplanung und -steuerung

Die Hauptaufgaben des Projektmanagements sind die Planung des Projekts, der Arbeitspakte und Ressourcen sowie die Steuerung des Projekts während der gesamten Projektlaufzeit. Typischerweise wird am Beginn eines Projekts eine Planung durchgeführt und laufend überwacht und unter Umständen angepasst.

2.4.1 Projektplanung

Projektplanung, Monitoring und Controlling

Die Projektplanungsphase *(Planning Phase)* für die initiale Projektplanung kann sowohl als eigenständige Phase durchlaufen, aber auch in die benachbarten Phasen integriert werden. Beispielsweise ermöglicht die Integration in die Spezifikations- oder Designphase bereits genauere Aufwandsschätzungen, da bereits detailliertere Informationen der tatsächlichen Umsetzung sowie der benötigten Resourcen vorliegen. Unabhängig von dieser initialen Projektplanung muss die Projektplanung in regelmäßigen Abständen überprüft *(Project Monitoring)* und in Abhängigkeit vom jeweiligen Projektstatus gegebenenfalls angepasst werden *(Project Controlling)*.

Plan-getriebene und agile Planung

Vor Beginn der (initialen) Planung sollte jeder (Auftraggeber und Entwickler) möglichst genau wissen, was gemäß Anforderungen und Spezifikation umgesetzt werden muss. Je detaillierter diese Informationen verfügbar sind, desto genauer und plan-getriebener kann die Entwicklung erfolgen. Diese plan-getriebene Entwicklung wird durch systematische Vorgehensmodelle, wie beispielsweise Wasserfallmodell, V-Modell oder Rational-Unified Process, unterstützt. Bei eher vagen Vorstellungen oder ungenauen Projek-

taufträgen kann ebenfalls ein geeigneter Software-Prozess – etwa ein agiler Ansatz – ausgewählt und die Planung entsprechend, beispielsweise iterativ, ausgerichtet werden (siehe dazu Kapitel 3).

Berücksichtigung von Projektabhängigkeiten

Bei der Planung müssen die weiteren Schritte des Entwicklungsprozesses bis zum Ende des Projekts geplant werden, wobei entsprechende Abhängigkeiten bzw. notwendige Reihenfolgen berücksichtigt werden müssen. *Prozessabhängigkeiten* sind über das eingesetzte Vorgehensmodell definiert und orientieren sich an den jeweiligen Phasen während des Projektablaufs, beispielsweise folgt auf die Analysephase eine Entwurfs- und Designphase. *Produktabhängigkeiten* definieren die Abhängigkeiten von Projektergebnissen, also konkreten Produkten, die während einer definierten Phase entstehen. Produkte umfassen dabei nicht nur ausführbaren Code, sondern alle Ergebnisse, die im Lauf eines Projekts entstehen, also beispielsweise auch Anforderungsdokumente, Spezifikationen, Testfälle oder Protokolle. „Produktabhängigkeit“ bedeutet dabei, dass ein Produkt erst erstellt werden kann, wenn ein anderes in einem definiertem Fertigstellungsgrad vorliegt: beispielsweise kann ein Designdokument erst erstellt werden, wenn die Anforderungsdokumente vorliegen, oder ein Integrationstest kann erst durchgeführt werden, wenn implementierte Komponenten vorliegen.

2.4.2 Projektmonitoring und -controlling

Projektsteuerung

Die Projektplanung wird zwar am Beginn des Projekts initial definiert, kann sich aber – in Abhängigkeit vom Projektverlauf – entsprechend ändern und muss gegebenenfalls angepasst werden. Beispielsweise kann bei unzureichender Qualität eines Produkts bei einem Meilenstein eine weitere Iteration eingeplant und durchgeführt werden. Details zur Projektverfolgung sind in Abschnitt 4.4 zu finden.

2.4.3 Wesentliche Artefakte der Planungsphase

Artefakte des Projektmanagements

Im Rahmen des Projektmanagements werden zahlreiche Artefakte verwendet, um eine initiale Planung des gesamten Projekts durchzuführen und den Projektverlauf zu begleiten und steuernd einzugreifen. Die Artefakte der Planungsphasen befassen sich mit rein planerischen und steuernden Dokumenten. In der Praxis ist es sinnvoll und notwendig, sich über den Nutzen einzelner Artefakte Gedanken zu machen. Es sollten nur diejenigen Dokumente erstellt werden, die wesentlich zum Erfolg eines Projekts beitragen, ohne einen unerwünschten und unnötigen zusätzlichen Ballast zu erzeugen. Diese Entscheidung über die Auswahl der Dokumente und Methoden wird typischerweise durch das Projektmanagement getroffen.

Netzplan, PERT

Netzpläne oder *PERT-Diagramme* stammen aus der Netzplantechnik und stellen *Abhängigkeiten* im zeitlichen Ablauf von Projekten dar. Durch die

Darstellung dieser Abhängigkeiten ist die Ermittlung des *kritischen Pfades* möglich (siehe Abschnitt 4.3.4 und Abschnitt 6.3.2).

WBS

Die *Work-Breakdown-Structure* oder der *Projektstrukturplan* ermöglichen eine Gliederung des gesamten Projekts in kleinere und planbarere Teilaufgaben. Diese Gliederung kann sich an (a) der Organisationseinheit (z. B. Abteilungen und Teams), (b) am Produkt (wie etwa Teilprodukte oder Komponenten) oder (c) am Prozess (etwa Aktivitäten, Phasen oder Funktionen) orientieren (siehe Kapitel 4).

Projektplan

Projektpläne werden vom Projektmanager aus den Informationen der Work-Breakdown-Structure (siehe auch Abschnitt 6.3.1) und dem Netzplan (siehe auch Abschnitt 6.3.2) während der Designphase erstellt und im weiteren Projektverlauf ständig aktualisiert. Der Projektplan muss von allen Teammitgliedern verstanden und verwendet werden, da er zu erfüllende Arbeitspakete definiert, die für alle am Projekt beteiligten Personen relevant sind.

GANTT

Ein weiteres Instrument der Projektplanung ist ein sogenanntes *GANTT-Diagramm*, das die *zeitliche Abfolge und Dauer* von Aktivitäten grafisch aufbereitet. Die Informationen aus der WBS sowie die Abhängigkeiten der Aktivitäten aus dem PERT-Diagramm bzw. dem Netzplan können mittels eines GANTT-Diagramms in kompakter Form dargestellt werden (siehe auch Abschnitt 6.3.3).

Meilensteintrend-analyse

Eine *Meilensteintrendanalyse (MTA)* setzt eine klare Definition von Meilensteinen (Terminen) und definierten Zielen von Arbeitspaketen voraus. Durch eine regelmäßige Beobachtung des Projektstatus kann sowohl der Projektfortschritt überwacht und gegebenenfalls angepasst als auch die Stabilität der Terminschätzungen bewertet werden. Gemeinsam mit dem Netzplan ist somit eine Bewertung des Gesamtprojekts möglich.

2.5 Entwurf und Design

Umsetzung von Anforderungen

Nach der Festlegung der Anforderung in der *Anforderungs- und Spezifikationsphase* muss definiert werden, wie diese Anforderungen im Detail umgesetzt werden. Aufbauend auf dem Anforderungsdokument (oder der Anforderungsspezifikation) werden in der Entwurfs- und Designphase alle Artefakte erstellt, die für die konkrete Erstellung des Produkts notwendig sind. In dieser Phase wird also der Grundstein für die konkrete Implementierung gelegt.

Definition von Design

Die IEEE 610.12-90 definiert Software-Design folgendermaßen [46]: *„Software design is the process of defining the architecture, components, interfaces, and other characteristics of a system or component ... and the results of that process.“* Unter dem Software-Design versteht man also den Prozess, der die Definition der Architektur, der Komponenten, Interfaces und anderer Charakteristika eines Systems oder einer Komponenten zur

Folge hat. Designspezifikationen enthalten also neben detaillierten Informationen über die funktionalen und nicht funktionalen Anforderungen an das Software-Produkt auch das Umfeld sowie Test- und Dokumentationsanforderungen. Sie beschreiben detailliert, was das Produkt leisten muss und wie die konkreten Kundenanforderungen umgesetzt werden sollen.

Konkrete Umsetzung

Das Ziel der Entwurfs- und Designphase ist also die Festlegung der *konkreten Umsetzung des geplanten Produkts*. Diese Festlegungen umfassen unter anderem die Definition der internen Strukturen, der Architektur, der Datenflüsse, der Algorithmen und der Komponenten bzw. Schnittstellen zwischen den Komponenten sowie die Benutzerschnittstellen gemäß den in der Anforderungsspezifikation dokumentierten Anforderungen.

Nachvollziehbarkeit

Jede Designentscheidung muss dokumentiert werden, um die Ideen des Entwicklerteams oder des Designers nachvollziehen zu können. Diese Dokumentation kann Fragen zu eingeschlagenen Umsetzungswegen (beispielsweise die Begründung für die Auswahl einer konkreten Architekturvariante) beantworten und die Nachvollziehbarkeit der Entwicklung ermöglichen.

ATAM

Stehen für eine Lösung mehrere Architekturvarianten zur Verfügung, muss die Entscheidung für eine *Best-Practice-Variante* getroffen und entsprechend dokumentiert werden. Zur Evaluierung von Architekturvarianten existieren zahlreiche Methoden, die diesen Entscheidungsprozess unterstützen. Die *Architecture Tradeoff Analysis Method (ATAM)* kann Architekten und Designer dabei unterstützen, die bestmögliche Architekturvariante zu finden [57]. Weitere Details zu ATAM sind in Abschnitt 5.5 zu finden.

Verifikation und Validierung

Aufgrund der großen Bedeutung der Designspezifikation ist eine gründliche Überprüfung des Dokuments im Hinblick auf die Anforderungen erforderlich. Reviews und Inspektionen sind erprobte Methoden für die Verfikation und Validierung von Designspezifikationen (siehe die Abschnitte 5.3 und 5.4).

Basis für die Umsetzung

Da die Entwurfs- und Designphase eng aneinander gekoppelt sind, ist eine Unterteilung in unterschiedliche Phasen kaum möglich und in der Praxis auch selten anzutreffen. Möchte man diese Unterscheidung treffen, kann das aufgrund des Detailgrades des Lösungsentwurfs erfolgen: Der *Entwurf* beschäftigt sich mit einer groben Planung der Software-Lösung, beispielsweise wird die grobe Architektur der geplanten Software-Lösung oder ein Datenbankmodell skizziert. Diese Grobentwürfe werden durch das *Designteam* verfeinert und im Detail ausgearbeitet (etwa detaillierte Architektur- und Komponentenbeschreibungen und Schnittstellendefinitionen). Die jeweiligen Schnittstellen zwischen Komponenten sind von besonderer Bedeutung, da sie die Kommunikation beschreiben und die korrekte Funktionsweise der einzelnen Komponenten und – in weiterer Folge – des Gesamtsystems sicherstellen müssen. Die Entwurfs- und Designspezifikation definieren alle notwendigen Details des Systems, sodass die Programmierer imstande sind, das System – basierend auf diesen Festlegungen – zu implementieren.

2.5.1 Designprinzipien

Bei der objektorientierten Entwicklung von Software-Produkten sind zahlreiche Grundregeln in der Entwurfs- und Designphase zu beachten. Die an dieser Stelle angesprochenen Prinzipien stellen eine wichtige Auswahl dar, die sich in verschiedenen Kapiteln dieses Buches, beispielsweise bei den Entwurfsmustern (siehe Kapitel 8), wiederfinden lassen.

Abstraktion

In einem komplexen System finden sich in der Regel zahlreiche Komponenten, die ähnliche Aufgaben erfüllen und zum Teil gleiche Datentypen verwenden. In einer Kundenverwaltung finden sich beispielsweise Adressdaten von Kunden, Lieferanten oder Mitarbeitern. Es erscheint daher sinnvoll, von den konkreten Personengruppen zu abstrahieren. Durch Abstraktionen werden irrelevante Details ausgeblendet und Gemeinsamkeiten zusammengefasst. Dadurch wird der Fokus auf die wesentlichen Komponenten und Zusammenhänge gelegt. Bei der Modellbildung können somit Informationen auf relevante Aspekte reduziert werden *(Reduktion der Komplexität)*. Abstraktionen finden sich beispielsweise bei abstrakten Datentypen aber auch bei UML Klassendiagrammen (siehe Abschnitt 6.1.3) wieder.

Kapselung

Kapselung (encapsulation) und *Information Hiding* sind zwei wesentliche Grundprinzipien in der objektorientierten Software-Entwicklung. Kapselung zielt darauf ab, möglichst viele Details *nach außen* zu verstecken und nur die notwendige Funktionalität und Eigenschaften bereitzustellen. Beispielsweise können *private Methoden* nur innerhalb der Klasse verwendet werden, *öffentliche Methoden* sind auch außerhalb einer Klasse verwendbar. Die Kommunikation erfolgt in der Regel über Schnittstellen *(Interfaces)*, die eine Trennung zur Außenwelt darstellen. Solange die Schnittstelle unverändert bleibt, kann die gekapselte Funktionalität ausgetauscht oder verändert werden, ohne das Verhalten des Gesamtsystems zu stören. Diese Grundregel der Trennung von Schnittstelle (Sicht von außen) und der inneren Realisierung (Implementierung) spielt bei der komponentenorientierten Software-Entwicklung eine zentrale Rolle. Die Schnittstelle definiert die Funktionalität für die Verwendung einer Komponente. Die eigentliche Implementierung umfasst die eigentliche Funktionalität der Komponente und die intern notwendige Funktionalität. Durch diese Trennung kann die Implementierung geändert oder auch komplett ausgewechselt werden, ohne die tatsächliche Verwendung zu beeinflussen.

Trennung von Schnittstelle und Implementierung

Modularisierung

Decomposition und Modularisierung zielen darauf ab, die Komplexität durch Zerlegung in kleine und überschaubare Komponenten besser beherrschbar zu machen *(devide and conquer)*. Dabei werden große Systeme in mehrere kleine und unabhängige Teile mit abhängiger Funktionalität zerlegt. Diese Aufteilung ermöglicht die Lösung von Teilproblemen mit relativ geringer Komplexität. Bereits fertige Teillösungen können für die Lösung eines übergeordneten Problems verwendet werden. Bei dieser Aufteilung ist jedoch auf geeignete Kontrollmechanismen (zentrale oder verteilte Kontrolle) und den Zusammenhalt innerhalb der Komponente *(Kohäsion)*

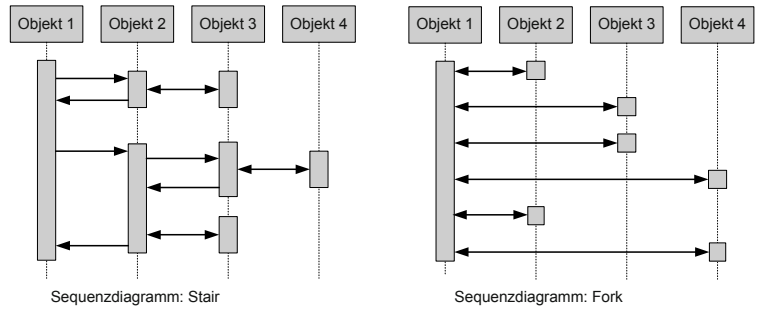


Abbildung 2.5 *Stair und Fork.*

als auch die Abhängigkeit zu anderen Komponenten *(Kopplung)* zu achten. Modularisierung erleichtert also die Wiederverwendbarkeit von Elementen und unterstützt die Entwickler sowohl bei der Entwicklung und Änderung von Komponenten als auch bei der Wartung.

Zentrale oder verteilte Kontrolle?

Ein wesentliches Designprinzip beschäftigt sich mit der Kontrolle im System. Abbildung 2.5 zeigt die Unterschiede anhand von UML Sequenzdiagrammen. Bei einem *fork* liegt eine zentrale Kontrolle vor: ein Objekt kontrolliert das gesamte System. Eine zentrale Kontrolle hat den Vorteil, dass Änderungen und Anpassungen an einem Objekt oder sehr wenigen Objekten vorgenommen werden müssen. Nachteilig wirkt sich eine zentrale Kontrolle aus, falls das zentrale Element ausfällt – dann kommt das gesamte System zum Stillstand. Ein weiterer nicht zu unterschätzender Nachteil ist, dass diese eine Komponente das gesamte Systemwissen vereinigen muss. Dadurch entstehen häufig sehr große und komplexe Kontrollobjekte, die in der Praxis schwierig herzustellen sind und in der Regel die Wartbarkeit deutlich erschweren.

Im Gegensatz dazu wird das System bei einem *stair* verteilt und dezentral – also großteils lokal – kontrolliert. Fallen Teile des Systems aus, sind meist nicht alle Systemteile davon betroffen und Teile des Gesamtsystems funktionieren nach wie vor. Ein weiterer Vorteil dieses Konzepts ist auch die Verteilung des Systemwissens auf unterschiedliche Komponenten. Da Entscheidungen meist lokal getroffen werden können, ist in der Regel keine oder nur eine geringe übergeordnete „Kontrolle“ erforderlich. Nachteilig wirkt sich allerdings aus, dass Änderungen meist an vielen unterschiedlichen Stellen durchgeführt werden müssen, wodurch die Fehleranfälligkeit steigt und die Wartbarkeit leidet.

Speziell bei verteilten Komponenten erlangen ereignisgetriebene Architekturen eine zunehmende Bedeutung, da die Kommunikation zwischen den Komponenten über Ereignisse *(events)*, das sind signifikante Zustandsänderungen des Systems, erfolgt und somit das Konzept einer „lockeren Kopplung“ unterstützt. Ereignisgetriebene Architekturen werden in Abschnitt 7.7 ausführlich behandelt.

Alle Prinzipien haben Vor- und Nachteile, die für den jeweiligen Anwendungsfall im konkreten Projekt abgewogen werden müssen.

Kopplung und Kohäsion

Um eine optimale Entwicklung und Wartung eines Software-Produkts zu ermöglichen, ist ein ausgewogenes Verhältnis zwischen innerem Zusammenhang einer Komponente und der Abhängigkeit zwischen den einzelnen Komponenten anzustreben. Die Kopplung *(coupling)* beschreibt dabei die Abhängigkeit zwischen einzelnen Komponenten, etwa die Anzahl der gegenseitigen Methodenaufrufe. Eine hohe Kopplung bedeutet somit eine hohe Anzahl von Methodenaufrufen und somit eine hohe Abhängigkeit zu anderen Methoden. Anzustreben ist daher meist eine niedere Kopplung. Das hat den Vorteil, dass sich Änderungen nur lokal auf die Komponente auswirken und unerwartete Effekte in anderen Komponenten reduziert werden. Durch eine niedrige Kopplung können Komponenten besser wiederverwendet und besser gewartet werden. Auf der anderen Seite ist die Kohäsion *(cohesion)* ein Maß für den inneren Zusammenhalt einer Komponente. Falls sehr viel (auch ungenutzte) Funktionalität in eine Komponente gepackt wird, steigt die Komplexität, und man spricht man von einer hohen Kohäsion. Um die Wartung zu erleichtern, ist ein sinnvolles Gleichgewicht zwischen Kopplung und Kohäsion wünschenswert, um die Komplexität und somit auch die Fehleranfälligkeit zu reduzieren. Durch UML Sequenz- und Kollaborationsdiagramme (siehe Abschnitt 6.1.6) lassen sich Kopplung und Kohäsion gut visualisieren. Komplexe Diagramme deuten meist auf eine hohe Kopplung und eine niedrige Kohäsion hin.

Gleichgewicht zwischen Kopplung und Kohäsion

Details und konkrete Anwendungsmöglichkeiten von Designprinzipien auf Architekturebene werden in Kapitel 8 ausführlich gezeigt. Neben der Festlegung des Systems im Hinblick auf die Architektur ist es auch notwendig, sich geeignete Mechanismen für Tests auf Design- und Architekturebene zu überlegen. Beispielsweise können Szenarien helfen, geeignete Testfälle zu finden, um beispielsweise Schnittstellen testen zu können (siehe Abschnitt 5.5).

2.5.2 Wesentliche Artefakte der Entwurfs- und Designphase

Artefakte der Entwurfs- und Designphase

Die Artefakte der Entwurfs- und Designphase bilden die Grundlage für die Implementierung, daher ist ein entsprechend hoher Detailgrad erforderlich. Die folgende Liste umfasst einige wichtige Artefakte der Entwurfs- und Designphase, wie sie bei der Softwareentwicklung im Rahmen der betrieblichen IT anzutreffen sind. Je nach Anwendungsbereich können auch domänenspezifische Artefakte notwendig sein. Aber auch hier gilt es, eine geeignete Auswahl von Artefakten zu finden, die durch ihren konkreten Nutzen wesentlich zum Projekterfolg beitragen. Die Auswahl der nutzbringenden Artefakte hängt stark vom Projekt und dem Projektkontext ab.

Komponentendiagramm

Eine Komponente ist ein logischer Teil eines Systems, das die konkrete Realisierung von einem oder mehreren Interfaces abbildet. Ein *Komponentendiagramm (Component Diagram)* visualisiert diese Zusammenhänge (siehe Abschnitt 6.1.4).

Klassendiagramm

Ein *Klassendiagramm (Class Diagram)* beschreibt die jeweiligen verwendeten Klassen und stellt deren Beziehung zueinander dar. Wichtig dabei ist, dass die Struktur der Daten im Vordergrund steht. Dabei werden sowohl Klassen mit ihren Attributen und Methoden als auch Assoziationen auf Datenebene beschrieben (siehe Abschnitt 6.1.3).

Zustands- und Aktivitätsdiagramm

Ein *Zustandsdiagramm (State Chart)* bzw. *Aktivitätsdiagramm (Activity Diagram)* beschreibt das dynamische Verhalten eines Systems oder eines Teilsystems. Ein Aktivitätsdiagramm zeigt eine bestimmte Sicht auf die Dynamik eines Systems, d. h. es beschreibt das konkrete Verhalten des Systems (siehe auch Abschnitt 6.1.5). Zustandsdiagramme oder Zustandsautomaten beschreiben Systemzustände bei definierten Ereignissen (siehe auch Abschnitt 6.1.8).

EER

Ein *Datenbankmodell* beschreibt die semantische Sicht auf die persistierten Daten. In der Praxis hat sich für die Modellierung das *Extended-Entity-Relationship-Diagramm (EER)* bewährt. Ergänzend zum Datenmodell empfehlen sich Datenbeschreibungstabellen, die alle relevanten Attribute, aber auch Datentypen, Integritätsbedingungen und entsprechende Schlüssel beinhalten. Das EER-Modell bildet gemeinsam mit den Datentabellen die Grundlage für die Realisierung der Datenbank (siehe auch Abschnitt 6.2.1).

Benutzerschnittstelle

Die *Benutzerschnittstelle (User-Interface)* stellt die Schnittstelle zur Außenwelt, also zum Anwender des Systems, dar. Im Rahmen der Entwurfs- und Designphase sind daher Überlegungen zur Umsetzung der Benutzerschnittstelle notwendig. Das kann beispielsweise durch Prototypen der Benutzerschnittstelle erfolgen (siehe auch Abschnitt 9.6).

Testen

Testen auf Entwurfs- und Designebene. Im Zusammenhang mit dem Entwurf und Design ist es parallel auch notwendig, sich geeignete Konzepte für den Test auf Integrations- und Systemebene zu überlegen. Dabei steht primär die Interaktion mit anderen Komponenten, die jeweiligen Schnittstellen und die Architektur im Fokus der Testüberlegungen (siehe auch Kapitel 5).

2.6 Implementierung und Integration

Ein Software-Produkt ist mehr als Code

Die Entwurfs- und Designdokumente bilden die Basis für die Implementierungsphase. Der Entwickler programmiert die einzelnen Komponenten und testet sie entsprechend den definierten Testfällen. Nach erfolgreicher Implementierung werden die einzelnen Komponenten zu „größeren“ Einheiten zusammengebaut – sie werden integriert. Neben der technischen Umsetzung der spezifizierten Anforderungen müssen in der Implementierung- und Integrationsphase auch Themen wie *Standardisierung*, *Dokumentation* und *Integration* beachtet werden, um ein Software-Projekt effizient umsetzen zu können.

2.6.1 Standardisierung

Der Begriff „Standardisierung“ hat meistens einen negativen Beigeschmack, da damit in der Regel ein hoher Dokumentationsaufwand verbunden ist. In einigen Anwendungsdomänen (beispielsweise bei sicherheitskritische Anwendungen) gibt es geforderte Normen und Regelungen, die einzuhalten sind und dessen Einhaltung entsprechend zu dokumentieren ist. Allerdings ist es auch bei der Entwicklung traditioneller Software-Produkte notwendig, sich an gemeinsame Richtlinien (also auch an „Standards“) zu halten, um

- den produzierten Softwarecode der anderen Teammitglieder lesen und bearbeiten zu können,
- eine effiziente Weiterentwicklung zu ermöglichen,
- die Zusammenarbeit im Team zu verbessern und
- die Wartung zu erleichtern.

Im Rahmen der Standardisierung in der Software-Entwicklung haben sich drei Gruppen von Standards auf unterschiedlichen Ebenen etabliert: (a) Nationale und Internationale Standards, (b) Unternehmensweite Standards und (c) Projektspezifische Standards.

Nationale und Internationale Standards

Mit Normen und Standards werden vielfach Vorgaben von Normungsgremien, wie der DIN[1], ISO[2] oder IEEE[3] verbunden. Diese Normen definieren Vorgaben, die je nach Domäne und Anwendungsbereich verbindlich anzuwenden sind. Beispielsweise müssen Software-Produkte, die für medizinische Anwendungen oder sicherheitskritische Anwendungen erstellt werden, spezielle Auflagen erfüllen. Diese Anwendungen erfordern typischerweise auch einen erheblichen Dokumentationsaufwand, der als Nachweis über die ordnungsgemäße Herstellung der Produkte oder Nachvollziehbarkeit von Anforderungen von der Anforderungsebene bis hin zu den implementierten Software-Komponenten dient. Neben diesen Vorgaben sind auch andere Standards im jeweiligen Anwendungsbereich relevant, um erfolgreich ein Produkt entwickeln zu können. Beispielsweise beeinflussen standardisierte Kommunikationsprotokolle (beispielsweise Format und Inhalt von Nachrichten), Programmiersprachen (beispielsweise Syntax oder Semantik), unterschiedliche Plattformen (etwa Interfaces für Betriebssystemaufrufe) oder Vorgaben von Werkzeugen (beispielsweise Notationen wie UML) die Software-Entwicklung und müssen entsprechend berücksichtigt und umgesetzt werden. Best-Practices, die sich als *State-of-the-Art*

Best-Practices als Standard

[1] `http://www.din.de`)
[2] `http://www.iso.org`)
[3] `http://www.ieee.org`)

oder *State-of-the-Practice* etabliert haben, definieren ebenfalls eine Art „Standard“. Sofern keine anwendungsspezifischen Normen zu berücksichtigen sind, gilt es, die richtige Mischung auszuwählen und anzuwenden.

Unternehmensweite Standards

Unabhängig von nationalen und internationalen Standards und einer breiten Palette von Best-Practices ist es in Organisationen üblich, *unternehmensweite Standards* zu definieren und anzuwenden. Diese internen Standards orientieren sich unter anderem an der Anwendungsdomäne und internen Vorgaben zur besseren und effizienteren Zusammenarbeit im Projektgeschäft. Die Standardisierung kann dabei beispielsweise konkrete anzuwendende Vorgehensmodelle, ausgewählte Best-Practices und Vorgaben zur Dokumentation von Software-Produkten (etwa Strukturierung und konkreter Inhalt einer Spezifikation) und Softwarecode (beispielsweise Namenskonventionen und Formatierungsrichtlinien) beinhalten. Diese internen Standardisierungen haben den Vorteil, dass sich Projektmitarbeiter in anderen Projekten gut und schnell zurechtfinden und sich mit den Problemlösungen beschäftigen können, statt sich erst mühsam in ein Projekt einarbeiten zu müssen. Gleiches gilt natürlich auch für die Entwicklung in einem Projekt, falls die Software von unterschiedlichen Personen entwickelt wird.

Interne Standards erleichtern die Zusammenarbeit im Team

Projektspezifische Standards

In vielen Fällen kann es notwendig sein, *projektspezifische Standards* einzusetzen. Diese dienen der Verbesserung der team-orientierten Software-Entwicklung für ein konkretes Projekt und sind typischerweise an übergreifende Standards und unternehmensspezifische Vorgaben angelehnt. Die Festlegungen von Namenskonventionen, Formatierungsrichtlinien und Dokumentationsrichtlinien verbessern einerseits die Zusammenarbeit und die Kommunikation der einzelnen Teammitglieder, erleichtern das Zurechtfinden in fremden Codeteilen und ermöglichen somit eine effektive und effiziente Weiterentwicklung des Produkts. Neben allgemeingültigen Konventionen werden hier auch projektspezifische Festlegungen getroffen, die spezifische Anforderungen an das konkrete Projekt betreffen. Das kann beispielsweise ein projektspezifisches Abkürzungsverzeichnis für eine spezielle Anwendungsdomäne sein. In Unternehmen werden typischerweise spezielle Vorgaben definiert, die eine effektive und effiziente Code-Erstellung in Entwicklungsteams ermöglichen sollen. Diese Vorgaben werden meist in sogenannten Programmier- und Dokumentationsrichtlinien zusammengefasst.

Zusammenfassung in Dokumentationsrichtlinien

2.6.2 Dokumentation

Dokumentation

Die Dokumentation von Softwarecode wird häufig unterschätzt und daher vernachlässigt, bietet aber viele Möglichkeiten für eine effizientere Weiterentwicklung und bessere Zusammenarbeit, bringt massive Erleichterungen in der Wartung und kann eine automatisierte Erstellung der Produktdokumentation ermöglichen. Durch einfache Mittel, wie (a) Namenskonventionen, (b) Dokumentations- und Formatierungsrichtlinien, (c) eine intensive Nutzung von Kommentaren, (d) eine definierte Nutzung von Headerblocks

und (e) eine sinnvolle Nutzung von SVN-Commit-Nachrichten kann eine umfassende Verbesserung der Produkt-, Prozess- und Projektqualität erzielt werden.

Erhöhung der Lesbarkeit des Codes

Die Einhaltung von *Namenskonventionen*, das bedeutet eine einheitliche Namensgebung, ist für die Software-Entwicklung im Team unbedingt erforderlich, damit bereits anhand der Namensvergabe auch deren Bedeutung und deren Anwendungsmöglichkeiten hervorgehen. Beispielsweise haben sich Namenskonventionen für die Vergabe von Bezeichnungen für Klassen, Methoden und Variablen sehr bewährt, da die Lesbarkeit und Übersichtlichkeit des Softwarecodes stark verbessert wird. Typischerweise sind Namenskonventionen ein fixer Bestandteil der *Dokumentationsrichtlinien*. Eine häufige Schwierigkeit für das Verstehen eines fremden Softwarecodes liegt in einer uneinheitlichen Formatierung. Daher ist es sinnvoll, einheitliche *Formatierungsrichtlinien* festzulegen. Formatierungsrichtlinien können Festlegungen für die Art und das Ausmaß von Einrückungen bei Methoden und eine einheitliche Klammernsetzung beinhalten. Diese Maßnahmen erscheinen recht einfach (und sind es auch), können aber die Produktqualität und die Lesbarkeit des Codes massiv verbessern sowie das Zurechtfinden in fremdem Code erleichtern. Beispielsweise lässt sich dadurch die „Suchzeit“ nach der letzten offenen Klammer reduzieren. Geeignete und einheitliche Strukturierungen ermöglichen es, rasch einen Überblick über den Code zu erhalten.

Formatierungsrichtlinie

Dokumentationsrichtlinien und Kommentare

Sinnvolle *Kommentierungen* sowie die Verwendung eines einheitlichen *Headerblocks* ermöglichen einen raschen Überblick über den aktuellen Projektstatus in Bezug auf die Komponente. Diese Kommentare können dann – in weiterer Folge – für die automatisierte Produktdokumentation (beispielsweise mit Javadoc) herangezogen werden. Kommentare helfen sowohl dem Entwickler als auch einem Wartungstechniker dabei, den vorliegenden Code besser zu verstehen und die ursprünglichen Gedankengänge für die vorliegende Realisierungsvariante besser und schneller nachzuvollziehen. Daher sollten Kommentare dort eingesetzt werden, wo sie wirklich zum besseren Verständnis beitragen, etwa Erklärung, wie eine komplexere Methode funktioniert oder warum gerade diese Lösung (und keine andere) gewählt wurde. Triviale Kommentare wie „Erhöhung der Variable xy“ sollen in gutem Softwarecode nicht vorkommen.

Headerblock

Dem *Headerblock* kommt – als spezieller Kommentar – eine besondere Bedeutung zu. Informationen über den Status der aktuellen Komponenten (siehe auch Abschnitt 10.7) sind typischerweise im Headerblock zu finden. Durch den Einsatz von Repositories für die Codeverwaltung können diese Informationen auch im Rahmen von Commit-Nachrichten erfasst werden. In der Praxis hat sich gezeigt, dass sich sowohl der Headerblock als auch „sprechende“ Commit-Nachrichten in der modernen teamorientierten Software-Entwicklung bewährt haben. Dadurch stehen einerseits Informationen über die Komponente (aus dem Headerblock) als auch Informationen über die „Geschichte“ der Komponente (Änderungshistorie aus den

Traceability

Commit-Nachrichten) zur Verfügung. Ein wesentlicher Aspekt in der Software-Entwicklung ist auch die Nachvollziehbarkeit *(Traceability)* in den unterschiedlichen Produkten. Informationen, welche Anforderungen in der Komponenten umgesetzt werden oder welche Testfälle der Komponente und den Anforderungen zugeordnet sind, können beispielsweise im Headerblock verankert werden und so diese Nachvollziehbarkeit ermöglichen. In Open-Source-Projekten finden sich immer wieder auch Angaben über die Lizenzbestimmungen der jeweiligen Komponente, die meist ein fixer Bestandteil des Headerblocks sind. Ein weiterer und wesentlicher Vorteil von Headerblockinformationen und integrierten Kommentaren ergibt sich im Rahmen der Automatisierung der Produktdokumentation. Speziell diese Informationen können automatisiert in die Produktdokumentation einfließen, die im Rahmen von regelmäßigen Builds bei der *Continuous Integration* erstellt werden (siehe auch Abschnitt 10.3.6).

Automatisierte Erstellung der Produktdokumentation

2.6.3 Integration von Software-Komponenten

Integrationsstrategien

Wenn alle Komponenten erzeugt und getestet sind, müssen sie zu einem größeren Teilsystem oder System zusammengebaut werden. Eine Integrationsstrategie beschreibt dabei die Art und Weise, wie die einzelnen Komponenten zusammengebaut werden. Ausgehend von einer „sauberen" Architektur mit klar definierten Schnittstellen ist ein gründlicher Test der Teilssysteme und Systeme erforderlich. Diesen Test bezeichnet man häufig als *Integrationstest* auf Architekturebene. Je nach Art der Integration sind „Hilfsmittel" notwendig, um eine noch nicht vorhandene Funktionalität zu simulieren und somit das integrierte Teilsystem zu verifizieren und zu validieren.

Big-Bang-Integration

Abbildung 2.6 zeigt exemplarisch acht Komponenten (D1 bis D8) eines Systems, die zu einem Gesamtsystem integriert werden sollen. Man muss sich also überlegen, wie diese Komponenten zusammengebaut werden. Eine einfache – aber in der modernen Software-Entwicklung nicht mehr sehr verbreitete – Variante ist die sogenannte *Big-Bang-Integration*. Alle Komponenten werden „auf einmal" zu einem definierten Zeitpunkt zusammengebaut. Von der Anwendung dieser „Strategie" ist aber ausdrücklich abzuraten, da sie zu großen Schwierigkeiten führen kann, wenn das Zusammenspiel der Komponenten sich nicht so verhält wie erwartet, Fehler durch die Integration auftreten oder etwaige Seiteneffekte zu „unerklärlichen" Problemen führen. Die Fehlersuche gestaltet sich dabei recht schwierig. Aufgrund dieser Risiken ist die Big-Bang-Integration höchstens für sehr kleine und sehr überschaubare Projekte geeignet. Sobald die Projektgröße ansteigt oder ein Team involviert ist, ist diese Strategie denkbar ungeeignet. Hier empfehlen sich *Continuous-Integration*-Konzepte oder, falls das nicht möglich ist, schrittweise Integrationsstrategien, wie *Top-down*- oder *Bottom-up*-Strategien. In der Praxis der modernen komponentenorientierten Software-Entwicklung hat sich Continuous Integration als *Best-Practi-*

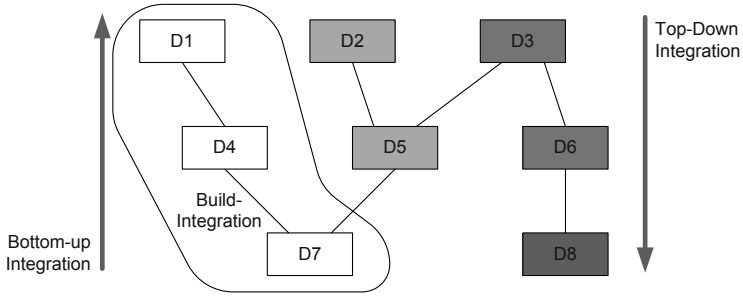


Abbildung 2.6
Integrationsstrategien.

ce etabliert (siehe auch Abschnitt 10.3.6). Die Grundidee der *Continuous Integration* ist es, Systeme häufig, mindestens täglich zu integrieren *(Daily Build)*, um jederzeit den aktuellen Stand des Gesamtsystems verfügbar zu haben und Fehler möglichst rasch (im Rahmen der Integration) feststellen zu können. In Abbildung 2.6 werden beispielsweise die Komponenten D1, D4 und D7 in einem ersten Build – also schrittweise – integriert. Diese Integrationsstrategie ist nur mit entsprechender Automatisierung und Werkzeugunterstützung möglich (siehe dazu Build-Management in Abschnitt 10.3). Ein wesentlicher Aspekt der Continuous Integration ist allerdings, dass bestimmte Systemteile nicht oder noch nicht verfügbar sind. Diese müssen dann simuliert werden, um das integrierte System geeignet testen zu können. Beispielsweise soll eine Komponente für eine Datenbankabfrage integriert und getestet werden. Steht die Datenbank nicht zur Verfügung, bedient man sich bei der Integration einer Simulation, bei der eine Komponente eingesetzt wird, die den Datenbankzugriff simuliert und definierte Datenbankelemente zurückliefert.

Continuous Integration

Es gibt allerdings auch Projekte, bei denen eine laufende und häufige Integration nicht oder nur schwer möglich ist. Beispiele dafür sind Versions-Rollouts, bei denen Systemteile nicht simuliert werden können oder Teilsysteme die von unterschiedlichen Herstellern stammen und nicht in den Continuous-Integration-Prozess integriert werden können. In diesen Fällen kann auf schrittweise Integrationsstrategien zurückgegriffen werden, wobei lokal immer eine Continuous-Integration-Strategie (als Best-Practice-Ansatz) angewandt werden sollte.

Top-down-Integration

Die Grundidee der *Top-down-Integrationsstrategie* besteht darin, die Integration der einzelnen Komponenten bei den Geschäftsfällen *(Business-Cases)* zu beginnen und schrittweise bis zu den hardwarenahen Komponenten fortzusetzen. Abbildung 2.6 zeigt ein Beispiel für die Top-down-Strategie. Die Integration beginnt bei den Komponenten D1, D2 und D3 in einem ersten Schritt; anschließend wird D4, D5 und D6 integriert und am Ende erfolgt die Integration von D7 und D8. Diese Strategie bietet den Vorteil, dass das User-Interface (auf der Business-Case-Seite) sehr früh verfügbar ist und der Auftraggeber beispielsweise recht früh erste Prototypen zur Verfügung hat. Natürlich ist nach dem ersten Integrationsschritt noch kaum Funktionalität verfügbar, die Lösung kann aber als Prototyp und als Test-Rahmen

(Test-Framework) eingesetzt werden. Nachteilig wirkt sich aus, dass die zugrunde liegende Funktionalität simuliert werden muss, was je nach Komplexität einen hohen Mehraufwand bedeuten kann. Ein weiterer, nicht zu unterschätzender Nachteil ist die sehr späte Integration von hardwarenahen Komponenten, was je nach Anwendung ein hohes Risiko bedeuten kann.

Bottom-up-Integration

Um dieses Risiko entsprechend adressieren zu können, könnte man sich überlegen, bei diesen „tiefen" Ebenen mit der Integration zu beginnen – der *Bottom-up-Strategie*. Bei dieser Strategie beginnt man beispielsweise bei den hardwarenahen Komponenten D7 und D8, danach folgt D4, D5 und D6 und am Ende erfolgt die Ankopplung an die Business-Cases durch die Integration von D1, D2 und D3. Dadurch ergibt sich der Vorteil, dass das System auf einem stabilem Fundament aufgebaut ist und die Integration schrittweise bis zu den Business-Cases vorangetrieben wird. Diesen Vorteilen stehen einige nicht unbedeutende Nachteile gegenüber: Ein ausführbares Gesamtsystem ist erst sehr spät verfügbar, d. h., der Kunde sieht erst gegen Projektende, wie sein Produkt aussehen wird (dafür sollte die komplette Funktionalität des Gesamtsystems verfügbar sein). Daher ist in der Regel zusätzlicher Aufwand für einen Prototypen (für Demo-Zwecke) erforderlich. Tests von *low-level*-Komponenten erfordern geeignete *Test-Driver*, durch die die Einbettung der zu testenden Komponenten in das Gesamtsystem simuliert wird. Die Erstellung von Test-Treibern kann ebenfalls einen erheblichen Mehraufwand bedeuten.

Build-Integration

Eine Mischung von Top-down- und Bottom-up-Strategie erscheint als sinnvoller Ansatz, um sowohl auf einem stabilem Fundament aufzubauen und gleichzeitig dem Kunden bereits zumindest eine Teilmenge an Funktionalität zur Verfügung zu stellen. Durch die *Build-Integration* wird eine schrittweise Integration entsprechend den priorisierten Business-Cases (und der jeweiligen Bedeutung für den Kunden) über alle Ebenen des Systems angestrebt. Beispielsweise kann ein erster Integrationsschritt die Komponenten D1, D4 und D7 umfassen, bei einem zweiten Schritt kommen die Komponenten D2 und D5 dazu und in einem letzten Schritt werden die Komponenten D3, D6 und D8 integriert. Diese phasen-orientierte Integrationsstrategie ermöglicht einerseits eine frühe Verfügbarkeit von Prototypen (zumindest der wichtigsten Anwendungsfälle) mit der vollen Funktionalität über alle Layer und kann daher gut für Prototypen und Demos eingesetzt werden. Die Build-Strategie orientiert kann sich an priorisierten Anforderungen des Kunden, wie es beispielsweise im Sprint-Backlog von SCRUM dargestellt ist (siehe Abschnitt 3.8). Diesen Vorteilen stehen aber auch Nachteile gegenüber: Die Wiederverwendung von Komponenten kann erschwert sein, da die Komponenten bereits in anderen Integrationsschritten verwendet werden. Änderungen an diesen Komponenten erfordern ein erneutes Testen *(Regressionstests)* aller bisher integrierten Komponenten, um Seiteneffekte entsprechend prüfen zu können. Diese Regressionstests können allerdings durch Continuous-Integration-Strategien bzw. Daily-Builds (siehe Abschnitt 10.3.6) automatisiert in den Entwicklungsprozess integriert werden.

Regressionstest

2.6.4 Artefakte der Implementierungs- und Integrationsphase

Artefakte der Implementierungs- und Integrationsphase

Die eigentliche Umsetzung der Kundenanforderungen (siehe Abschnitt 2.3) basierend auf den Festlegungen aus der Entwurfs- und Designphase (siehe Abschnitt 2.5) erfolgt in der Implementierungsphase; der Zusammenbau der Komponenten entsprechend den Architekturfestlegungen erfolgt in der Integrationsphase. Dementsprechend sind die wichtigsten Artefakte dieser Phase einerseits der implementierte und integrierte Softwarecode und natürlich die durchgeführten Tests einschließlich der Testdokumentation.

Softwarecode und integriertes System

Softwarecode und integriertes System. Basierend auf den Anforderungen und den Artefakten der Entwurfs- und Designphase werden die einzelnen Komponenten erstellt und getestet. Nach erfolgreichem Test erfolgt der Zusammenbau (die Integration) der Komponenten zu einem kompletteren System.

Testbericht

Testberichte sind erforderlich, um Aussagen über die Qualität des Produkts tätigen zu können. Typischerweise werden Testberichte auf Komponentenebene – meist automatisiert – erstellt. Für die Testfallerstellung und die Durchführung können Unittests eingesetzt werden, die über geeignete Analysewerkzeuge Auswertungen über die Testdurchführung ermöglichen (siehe Abschnitt 5.8). Maße für die Testqualität der Komponenten sind beispielsweise die Testabdeckung *(Test Case Coverage)* und die Ergebnisse der durchgeführten Tests (siehe Abschnitt 5.6.6). Entsprechende Testergebnisse auf Architektur- und Designebene liefern Aussagen über die Qualität des integrierten Systems (siehe Abschnitt 5.6.3). Die Überprüfung der realisierten Gesamtlösung im Hinblick auf die Kundenanforderungen wird beispielsweise in Testberichten auf Akzeptanztestebene dokumentiert. Details zu Software-Tests auf den unterschiedlichen Ebenen werden in den Abschnitten 5.6 und 5.7 beschrieben.

2.7 Betrieb und Wartung

Standard- oder Individualsoftware?

Ist die Integration (siehe Abschnitt 2.6.3) einschließlich der durchzuführenden Tests (siehe Kapitel 5) abgeschlossen, kann das Produkt in den operativen Betrieb übergeführt werden. Spätestens zu diesem Zeitpunkt ist es relevant, ob es sich bei dem Produkt um Standardsoftware für den „breiten“ Markt (beispielsweise um eine Textverarbeitungssoftware) oder um Individualsoftware, die auf den konkreten Bedarf eines einzelnen Kunden oder einer Kundengruppe zugeschnitten ist, handelt. Während Standardsoftware-Produkte in der Regel durch eine Vertriebsfreigabe nach einer Reihe von Pilotinstallationen und Betatests am freien Markt verfügbar sind, ist für Individualsoftware meist ein formeller Abnahmetest durch den Auftraggeber und eine individuelle Einführung des Produkts beim Auftraggeber erforderlich.

2.7.1 Abnahme- und Einführungsphase

Abnahme- und Einführungsphase

Die Abnahme- und Einführungsphase umfasst nicht nur die Durchführung von Akzeptanztests (Teile des Systemtests werden in der operativen Umgebung beim Auftraggeber ausgeführt), sondern auch eine kontrollierte und nachvollziehbare Übergabe des Gesamtprodukts einschließlich der vollständigen Dokumentation und der Projektsourcen an den Auftraggeber. Bei Individualsoftware können meist auch Installationen, individuelle Schulungen und Datenmigrationen (etwa die Übernahme existierender Datenbestände aus Altsystemen) erforderlich sein. Gängige Praxis ist es auch, das System über einen definierten Zeitraum unter Echtbedingungen (Testbetrieb) zu betreiben. Wird diese Strategie angewandt, muss vor der Betriebsphase eine kontrollierte Inbetriebnahme, also eine Überführung vom Test- in den Produktivbetrieb durchgeführt werden. Zu diesem Zeitpunkt startet die eigentliche Betriebs- und Wartungsphase.

Betrieb und Wartung

Betrieb *(Operation)* und Wartung *(Maintenance)* sind feste Bestandteile des Software-Prozesses und werden nicht nur durchgeführt, um nach abgeschlossener Entwicklung Fehler zu beseitigen. Eine weit wichtigere Aufgabe während der Wartung ist die laufende Pflege und Weiterentwicklung des Produkts. Betriebs- und Wartungsaktivitäten müssen bereits während der Entwurfs- und Designphase berücksichtigt werden, um eine möglichst effiziente Wartung zu ermöglichen. Geänderte Anforderungen an das Produkt können bis zu einem gewissen Grad berücksichtigt werden, indem zu erwartende Anpassungen – vorausschauend – eingeplant werden. Beispielsweise kann bereits während der Entwicklung auf Mechanismen zurückgegriffen werden, die Erweiterbarkeit, Skalierbarkeit oder Performance unterstützen (etwa über Architektur- und Designentscheidungen). Die Produktdokumentation ist dabei ein wesentlicher Punkt in der Betriebs- und Wartungsphase. Eine gut dokumentierte Lösung ist deutlich besser wartbar, d. h., wesentliche Designentscheidung sind protokolliert, eine einheitliche Namensgebung und Struktur liegt vor und die Entwurfs- und Designunterlagen sind vollständig und aktuell. Das Vorliegen dieser Informationen erleichtert das Einlesen und Verstehen der Lösung und ermöglicht effiziente Wartungsprozesse.

Der Grundstein für die Wartung liegt im Design und in der Dokumentation

2.7.2 Wartungskategorien

Wartungskategorien

Was versteht man jetzt unter der Wartung eines Software-Systems? In Anlehnung an IEEE 1219 lässt sich die Software-Wartung folgendermaßen zusammenfassen [45]: *Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.* Software-Wartung ist also die Veränderung eines Software-Produkts nach Auslieferung, um Fehler zu korrigieren, die Leistung oder andere Eigenschaften zu verbessern oder das Produkt an geänderte Rahmenbedingungen anzupassen.

Tabelle 2.1 *Wartungskategorien [91].*

	Korrektur	Erweiterung
Reaktive Wartung	korrektive Wartung	adaptive Wartung
Proaktive Wartung	präventive Wartung	perfektive Wartung

Korrektur und Erweiterung

Generell wird zwischen korrektiver Wartung *(Correction)* und Erweiterungen *(Enhancement)* unterschieden. Während die korrektive Wartung direkt auf die Beseitigung von Fehlern abzielt, die im Lauf der Betriebsphase auftreten, beschäftigt sich die erweiternde Wartung mit der Weiterentwicklung des Systems. Eine weitere wesentliche Unterscheidung muss zwischen proaktiver und reaktiver Wartung getroffen werden. Im Rahmen der reaktiven Wartung wird eine Wartungsaktivität von außen, meist durch den Anwender oder Kunden, angestoßen. Dementsprechend muss auf ein Ereignis (beispielsweise auf einen Fehler oder einen Änderungswunsch) reagiert und eine passende Wartungsaktivität durchgeführt werden. Eine weit bessere Strategie verfolgt die proaktive Wartung. Sie zielt darauf ab, Verbesserungen durchzuführen, bevor sie vom Kunden angestoßen werden. Beispielsweise fällt darunter die Ergänzung und Vervollständigung der Produktdokumentation.

Proaktiv und reaktiv

In Anlehnung an van Vliet lassen sich vier unterschiedliche Wartungskategorien ableiten [91], die in Tabelle 2.1 skizziert sind.

Unterschiedliche Wartungskategorien

- Die *Korrektive Wartung (korrektiv)* umfasst die Behebung von Fehlern als Reaktion auf das Auftreten einer Abweichung vom erwarteten Systemverhalten. Diese Wartungsaktivität wird typischerweise vom Anwender oder Kunden angestoßen und erfolgt meist im Umfeld der Gewährleistung.
- Die *Adaptive Wartung (adaptiv)* umfasst die Änderung der Software-Lösung im Hinblick auf geänderte Rahmenbedingungen, beispielsweise geänderte Hardware, Anpassung an eine neue Plattform oder Anpassung der Software durch geänderte Anforderungen.
- Die *Vorbeugende Wartung (präventiv)* dient der Verbesserung des Produkts im Hinblick auf zukünftige Aktivitäten, beispielsweise Ergänzung der Dokumentation, um künftige Wartungsaktivitäten zu erleichtern.
- Als *Produktpflege und Verbesserung (perfektiv)* werden alle Aktivitäten gesehen, die vorbeugend (ohne konkreten Handlungsbedarf) umgesetzt werden. Beispielsweise fallen hierunter präventive Erweiterungen oder die Verbesserung von nicht funktionalen Anforderung wie Performanz und Effizienz.

Wartungskosten

Stellt sich die Frage, wie sich die Aufwände auf die jeweiligen Wartungskategorien überhaupt verteilen. Gemäß Sommerville [83] entfallen 65% der

Wartungskosten auf funktionelle Erweiterungen und die Weiterentwicklung eines Produkts, 18% auf Anpassungen und „nur“ 17% auf Fehlerkorrekturen.

Wartungsaktivitäten erfordern dann einen hohen Aufwand, wenn sie durch Personen erfolgen, die nicht in die Entwicklung des Produkts involviert waren, was in der Praxis häufig vorkommt. Hier ist einerseits Einarbeitungszeit des Wartungspersonals notwendig. Andererseits können Änderungen auch erhebliche Auswirkungen auf Architektur und Design haben und damit hohe Wartungsaufwände verursachen. Allgemein hängen die Kosten auch vom Anwendungstyp, der Neuheit des Produkts und der Qualität des Software-Designs, der Dokumentation und der Art und Qualität der verfügbaren Tests ab.

2.7.3 Vorgehensweise bei Wartungsprojekten

Alle Lebenszyklus-Phasen

Die ursprüngliche Entwicklung eines Software-Produkts orientiert sich am Software-Lebenszyklus bzw. einer konkreten Ausprägung in Form eines Vorgehensmodells. Im Rahmen der Wartung sind grundsätzlich *dieselben* Prozessschritte wie bei einem Entwicklungsprojekt zu durchlaufen – allerdings mit anderer Aufwandsverteilung. Bei jeder Änderung muss man sich über die Auswirkungen der Änderung in allen Phasen des Software-Prozesses Gedanken machen und das Ergebnis entsprechend dokumentieren. Ist eine Phase von der Änderung nicht betroffen, kann diese sehr kurz ausfallen. Jede Änderung beinhaltet allerdings auch die entsprechende Überprüfung des Gesamtsystems (beispielsweise durch Regressionstests).

2.7.4 Außerbetriebnahme und Stilllegung

Stilllegung

Nach (jahrlanger) Nutzung kann es notwendig sein, über die weitere Verwendung der Software nachzudenken und sie gegebenenfalls auszutauschen. Wird eine Software außer Betrieb genommen, spricht man auch von der Stilllegung eines Produktes *(retirement phase)*. Dabei muss eine kontrollierte Außerbetriebnahme des Produkts und gegebenenfalls ein störungsfreier Übergang zu einem Nachfolgeprodukt gewährleistet sein. Mögliche Gründe für die Stilllegung eines Software-Produkts können sein:

Gründe für die Stilllegung eines Software-Produkts

- *Komplexe Änderungen*. Speziell bei komplexen Erweiterungen oder gänzlich geänderten Anforderungen kann ein komplettes Redesign des Produkts erforderlich sein, falls beispielsweise die Erweiterungen mit der bestehenden Architektur nicht umsetzbar sind.
- Bei einer *hohen Anzahl von Änderungen* während der Wartungsphase kann es wirtschaftlich sinnvoller sein, das Produkt neu zu entwickeln.
- *Inkonsistenzen* innerhalb des Programmcodes oder zwischen Code und Dokumentation durch „schnelle“ Änderungen. Viele Änderungen in

der Betriebs- und Wartungsphase führen rasch zu „überfrachtetem“ Softwarecode, speziell wenn die Dokumentation nicht oder unzureichend nachgeführt wurde. Eine kleine Modifikation in eine vermeintlich einfachen Komponente kann zu einem komplett falschen Resultat an einer anderen Stelle führen (Seiteneffekte).

- *Hardware- oder Plattformwechsel*. Bei einem Wechsel des „Fundaments“ kann eine Neuimplementierung erforderlich sein, um die Funktionsfähigkeit des Produkts weiter sicherstellen zu können.

Neben diesen eher technischen Gründen für die Stilllegung eines Software-Produkts spielen auch betriebswirtschaftliche Gründe eine Rolle, falls das Software-Produkt nicht mehr benötigt wird oder der ursprünglich Grund für den Einsatz des Produkts nicht mehr vorhanden ist. Wird ein Software-System durch ein anderes abgelöst, muss man sich eine Strategie zurechtlegen, um die Daten entsprechend übernehmen und in ein neues System zu integrieren. Man spricht dann von einem Migrationsprojekt.

2.7.5 Wesentliche Artefakte der Betriebs- und Wartungsphase

Artefakte der Betriebs- und Wartungsphase

Im Rahmen der Betriebs- und Wartungsphase fallen – je nach Anwendungsgebiet – sehr unterschiedliche Artefakte an. Gebräuchlich und gemeinsam für alle Anwendungsfälle sind meist *Change Requests* (Änderungenwünsche oder konkrete Fehlermeldungen) und die *Dokumentation* umgesetzter Änderungen oder Korrekturen.

Change Request

Change Request. In der Praxis empfiehlt es sich, kontrollierte Methoden zur Erfassung und Nachverfolgung von Änderungen (auch im Rahmen der Betriebs- und Wartungsphase) einzusetzen. Wichtig dabei ist, dass Änderungen transparent und nachvollziehbar gehalten werden. Diese Aufgaben sollten idealerweise auch durch Werkzeuge, wie Issue-Tracking-Systeme, unterstützt werden (siehe auch Abschnitt 10.8). Diese Systeme können für die Verwaltung von Änderungen und Fehlern eingesetzt werden.

Dokumentation

Dokumentation. Die im Rahmen der Wartung durchgeführten Aktivitäten und Produktänderungen müssen in der jeweiligen Produktdokumentation nachgeführt werden, um die Konsistenz der Unterlagen sicherzustellen und die Nachvollziehbarkeit der Änderungen zu gewährleisten.

2.8 Vom Software-Lebenszyklus zum Vorgehensmodell

Phasen des Lebenszyklusses finden sich in allen Produkten

In den vorangegangenen Abschnitten wurden die einzelnen Phasen des „Lebens“ eines Software-Produkts von der Idee über Entwicklung und Betrieb bis zur Stilllegung im Überblick beschrieben. Diese grundlegenden

Phasen des Software-Lebenszyklusses finden sich in allen Software-Projekten wieder.

In den meisten Fällen wird im realen Projekt aber nur ein Teil des Software-Lebenszyklusses – meist die technische Entwicklung eines Produkts – abgedeckt. Auch die rein sequenzielle Abarbeitung in einem Projekt ist nicht immer sinnvoll realisierbar; beispielsweise ist bei unklaren Anforderungen diese strikt sequenzielle Vorgehensweise eher hinderlich. Daher haben sich aus dem Software-Lebenszyklus zahlreiche spezifische Vorgehensmodelle entwickelt, die unterschiedliche Ziele verfolgen und Projekttypen adressieren. Diese spezifischen Vorgehensmodelle sind an den Projektkontext und die jeweiligen Projektbedürfnisse angepasst, wobei der Schwerpunkt der meisten Vorgehensmodelle auf der technischen Seite liegt; sie beginnen bei der Definition der Anforderungen und enden mit der Inbetriebnahme beim Kunden. Zahlreiche Vorgehensmodelle haben sich in einem spezifischen Kontext in der Praxis etabliert und sind quasi „standardisiert". Ein derartiges Vorgehensmodell entspricht einer konkreten Strategie zur kontrollierten Durchführung eines Projekts in einem definiertem Umfeld. Ausgewählte Vorgehensmodelle werden im Detail in Kapitel 3 beschrieben.

Vorgehensmodelle müssen zum Projekt passen

Strategie zur kontrollierten Projektdurchführung

2.9 Zusammenfassung

Das Hauptziel moderner Software-Entwicklung ist die Herstellung qualitativ hochwertiger Produkte, die für den Anwender einen realen Nutzen bringen. Für die systematische Herstellung dieser Produkte ist eine professionelle Vorgehensweise – ein Prozess – erforderlich, der alle wesentlichen Schritte von der Konzeptphase über Entwicklung bis zu Betrieb und Wartung umfasst. Neben diesen eher technischen Schritten sind begleitende Aktivitäten, wie Projekt- und Qualitätsmanagement, erforderlich, um den Projektfortschritt zu steuern und die Qualität eines Software-Projekts sicherstellen und gegebenenfalls beeinflussen zu können. Der Software-Lebenszyklus beschreibt also eine Abfolge von Schritten (Phasen) eines Software-Projekts mit all seinen Aktivitäten, Beziehungen und Ressourcen und stellt das Basiskonzept für professionelles Vorgehen im Rahmen der Software-Entwicklung dar:

- Die *Anforderungs- und Spezifikationsphase* ist der Grundstein für den Erfolg eines Software-Produkts und spiegelt den Bedarf und den Wunsch des Kunden wider.
- *Planung*. In der Planungsphase wird die initiale Projektplanung durch das Projektmanagement durchgeführt. Der Projektfortschritt muss jedoch überwacht, und die Planung muss je nach Projektergebnissen aktualisiert werden.

- *Entwurf und Design* sind die Basis für die eigentliche Implementierung und legen den Aufbau des Systems fest. In dieser Phase sind Designprinzipien zu beachten, um die Ausarbeitung in späteren Phasen zu erleichtern.
- Die *Implementierung und Integration* umfasst die Umsetzung der Anforderungen und des Entwurfs. Standards und die Dokumentation unterstützen Software-Entwicklungsteams nicht nur bei der Erstellung, sondern auch bei der Weiterentwicklung und Wartung.
- Ein Großteil der *Betriebs- und Wartungsphase* umfasst Erweiterungen des Produkts, aber auch Fehlerkorrektur und Anpassungen an geänderte Systemgegebenheiten sind integraler Bestandteil der Betriebs- und Wartungsphase. Die *Stilllegung (retirement phase)* schließt den Software-Lebenszyklus ab, in dem das Software-Produkt kontrolliert außer Betrieb genommen oder ersetzt wird.

Ein reales Projekt orientiert sich am Software-Lebenszyklus und benötigt konkrete Schritte, Produkte, Rollen und Aktivitäten zur erfolgreichen Projektdurchführung. Diese Schritte sind in sogenannten Vorgehensmodellen organisiert, die auf den jeweiligen Projektkontext abgestimmt sind. Dieser Projektkontext umfasst beispielsweise konkrete Projektgegebenheiten, wie Projekttypen und Anwendungsdomäne. Der nächste Abschnitt beschäftigt sich daher mit ausgewählten Vorgehensmodellen für eine konkrete Projektdurchführung.

3 | Vorgehensmodelle

Software-Entwicklungsprozesse strukturieren das zu planende Software-Projekt und geben einen Überblick über die zu erstellenden Produkte. Typische Schritte im Entwicklungsprozess, die im Rahmen des Produktlebenszyklusses erläutert wurden, sind Anforderungserhebung, Entwurf und Design, Implementierung und Integration, Verifikation und Validierung sowie Betrieb, Wartung und Evolution. Diese Schritte benötigen eine passende Strategie, um die Produktentwicklung effektiv vorantreiben zu können. Je nach Projekt und Anwendungsdomäne haben sich zahlreiche konkrete Vorgehensmodelle etabliert, die im jeweiligen Projektkontext anwendbar sind. In diesem Kapitel werden traditionelle Prozesse, wie beispielsweise *Wasserfallmodell*, *V-Modell XT* und *Rational Unified Process*, sowie flexible und agile Ansätze, wie beispielsweise *SCRUM*, vorgestellt. Allgemein definieren Vorgehensmodelle eine konkrete Abfolge von Schritten im Entwicklungsprozess, die dem Entwickler helfen, sich im Projekt zu orientieren, und regeln, welche Produkte in welchem Fertigstellungsgrad und mit welcher Qualität vorliegen sollen. Sie helfen also, im Rahmen des Projektmanagements das Software-Projekt zu planen und zu begleiten.

Übersicht

3.1 Strategie für die Projektdurchführung

Herstellung qualitativ hochwertiger Produkte

Ein vorrangiges Ziel in der Software-Entwicklung im Umfeld von komplexen Systemen ist die Herstellung von qualitativ hochwertigen Produkten. Qualitativ hochwertige Produkte sind beispielsweise gekennzeichnet durch eine minimale Anzahl von verbleibenden Fehlern, einen hohen Erfüllungsgrad der Kundenanforderungen und die Fertigstellung innerhalb vorgegebener Projektparameter, wie beispielsweise Aufwand und Kosten. Dabei umfassen diese Produkte nicht nur das finale Produkt – die Software-Lösung – sondern sämtliche Artefakte, die für die Herstellung dieser Lösung relevant sind, wie beispielsweise Anforderungs- und Spezifikationsdokumente, Softwarecode, aber auch Testfälle und Projektfortschrittsberichte.

Projektstrategie

Software-Prozesse oder *Vorgehensmodelle* legen eine Projektstrategie fest, die sowohl die zeitliche Reihenfolge der zu erstellenden Produkte als auch das notwendige Qualitätsniveau zu einem bestimmten Zeitpunkt (etwa bei einem Meilenstein) definiert. Sie regeln also, wann welche Produkte von wem in welchem Fertigstellungsgrad und auf welchem Qualitätsniveau vorliegen müssen. Vorgehensmodelle bilden also den *organisatorischen Rahmen eines Software-Projekts*. In der industriellen Praxis existieren zahlreiche Vorgehensmodelle, die auf unterschiedliche Projektbedürfnisse und Projektcharakteristika abgestimmt sind (siehe Abschnitt 1.1). Wesentliche Unterscheidungskriterien sind beispielsweise Projektdauer und Projektgröße, Anwendungsdomäne und Komplexität. Der vorliegende Abschnitt beschreibt eine Auswahl wesentlicher Grundkonzepte für Vorgehensmodelle und illustriert, wie diese Prozessmodelle in der Praxis eingesetzt werden können.

3.2 Wasserfallmodell

Wasserfallmodell

Das Wasserfallmodell ist der Klassiker unter den Vorgehensmodellen. Das Modell wurde bereits in den 1970er Jahren von J. Royce veröffentlicht [76] und wird heute nur noch selten für bestimmte Anwendungsgebiete eingesetzt, bei denen ein strikt sequenzieller Projektablauf möglich ist. Namensgebend ist, dass die sequenziellen Schritte des Lebenszyklusses eines Software-Produkts (siehe Kapitel 2) in Form eines Wasserfalls dargestellt werden. Abbildung 3.1 zeigt den schematischen Ablauf des Wasserfallmodells. Alle Schritte werden sequenziell durchlaufen, wobei jeweils die vorangegangene Phase *vollständig abgeschlossen* und *freigegeben* werden muss, bevor die nächste Phase gestartet werden kann. Als Abschluss jeder Phase sind Verifikations- und Validierungsschritte vorgesehen, die quasi als Freigabe der erstellten Produkte verwendet werden. Wird die geforderte Qualität an der Stelle nicht erreicht, kann zur vorangegangenen Phase „zurückgesprungen“ werden.

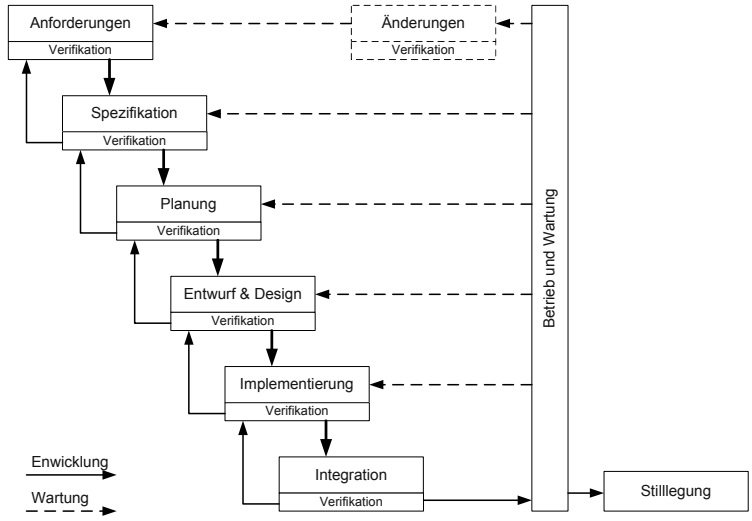


Abbildung 3.1
Schematischer Aufbau des Wasserfallmodells nach [83].

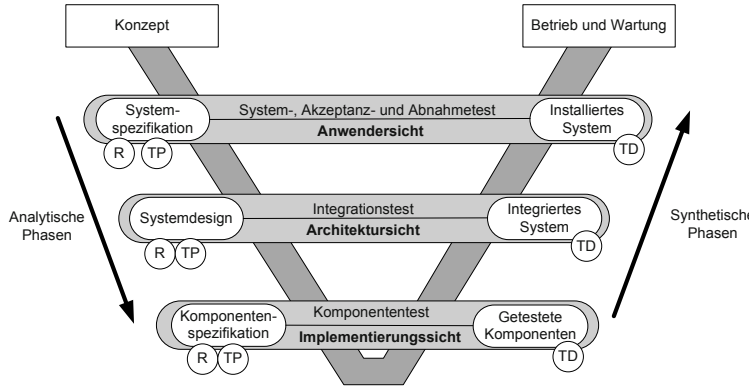
Vor- und Nachteile

Aufgrund der guten Planbarkeit durch klar abgegrenzte Phasen wird das Wasserfallmodell typischerweise bei stabilen und genau beschriebenen Anforderungen und Abläufen eingesetzt. Nachteilig wirkt sich allerdings aus, dass dieses Modell nur wenig Flexibilität aufweist und somit nur schwer auf (späte) Änderungen im Projekt reagieren kann. Als weitere Folge werden Fehler unter Umständen erst sehr spät im Entwicklungsprozess oder erst beim Kunden erkannt. Das hat zur Folge, dass teuere Nacharbeiten notwendig werden und das Projekt in Zeitnot kommen kann. Ein weiterer Nachteil des Wasserfallmodells ist, dass durch den sequenziellen Ablauf eine finale Software-Lösung erst sehr spät zur Verfügung steht, der Kunde also erst gegen Ende des Projekts sieht, was er tatsächlich bekommen wird und dann grundlegende Änderungen nicht mehr leicht machbar sind. In der Praxis der modernen Software-Entwicklung ist das Modell gelegentlich noch anzutreffen, wird aber zunehmend von flexibleren Prozessmodellen verdrängt.

3.3 Das V-Modell

V-Modell

Basierend auf dem Software-Lebenszyklus (siehe Kapitel 2) umfasst das V-Modell [16] ebenfalls die wesentlichen Schritte von der Konzeptphase eines Software-Projekts, über Planung, Entwurf, Design und Implementierung bis zu Test, Wartung und Weiterentwicklung. Das V-Modell deckt somit ebenso wie das Wasserfallmodell einen Großteil des Software-Lebenszyklusses ab. Im Gegensatz zum Wasserfallmodell trennt das V-Modell aber die „spezifizierenden“ Phasen von den „realisierenden“ Phasen; jeder Spezifikationsphase wird eine Realisierungs- und Testphase mit entsprechenden konkreten Produkten gegenübergestellt. Dadurch ergeben sich unterschiedlich detaillierte Sichten auf das Software-Projekt, beispielswei-



Legende: R … Reviews, TP … Testplanung- und Definition, TD … Testdurchführung

Abbildung 3.2
Schematischer Aufbau des V-Modells nach [16].

se aus Anwender-, Architektur- oder aus Implementierungssicht. Abbildung 3.2 illustriert diesen schematischen Aufbau des V-Modells[1].

Analytische und Synthetische Phasen

Das V-Modell teilt also den Entwicklungsprozess grob in zwei wesentliche Phasen, (a) die *analytischen* Phasen und (b) die *synthetischen* Phasen, durch deren Anordnung sich das charakteristische „V“ ergibt. Die analytischen Phasen sind dadurch gekennzeichnet, dass – ausgehend vom Konzept – schrittweise ein immer höherer Detaillierungsgrad der Spezifikation bis auf Komponentenebene erreicht wird. Nach Umsetzung der detaillierten Komponentenspezifikationen wird das System schrittweise zusammengebaut (synthetische Phasen), bis die Software-Lösung vollständig umgesetzt, d. h. implementiert, integriert und getestet, ist.

Unterschiedliche Sichten

Eine horizontale Gliederung ermöglicht die Betrachtung des Projekts aus unterschiedlichen Sichten *(Anwendersicht*, *Architektursicht* und *Implementierungssicht)* und stellt gleichzeitig den Zusammenhang zwischen Spezifikation und getesteter Umsetzung auf der jeweiligen Betrachtungsebene in den Vordergrund. Dadurch wird es beispielsweise möglich, auf den jeweiligen Ebenen Testfälle und Testszenarien im analytischen Zweig zu definieren und im synthetischen Zweig auszuführen.

Anwendersicht

Die *Anwendersicht* betrachtet das Gesamtsystem aus der Perspektive des Anwenders und der Geschäftsprozesse. Dementsprechend stehen die Anforderungen an das Gesamtsystem und die Realisierung der gesamten Software-Lösung im Zentrum der Betrachtung. Während die Anforderungen in den Anforderungs- und Spezifikationsdokumenten (Systemspezifikation) definiert werden, liegt das „installierte System“ nach erfolgreicher Umsetzung auf der Zielplattform des Auftraggebers vor. Am Beginn gibt es zwar noch keine detaillierten Informationen über die eigentliche Realisierung

[1] Umfangreiche Informationen zum V-Modell und daraus abgeleiteten Modellen sind auf der Seite der IABG zu finden (siehe `http://www.v-modell.iabg.de`)

(Entwurf, Design und Implementierung), geforderte Anforderungen können aber im Hinblick auf deren Akzeptanz geprüft werden. Daher werden auf dieser Ebene typischerweise *Akzeptanz- und Abnahmetests* aus Kundensicht definiert und nach der Umsetzung durchgeführt – diese Ebene entspricht im Wesentlichen einer Validierung der Lösung im Hinblick auf die Kundenanforderungen.

Architektursicht

Die nächste Ebene umfasst bereits Details der geplanten Lösung, also wie die Anforderungen umgesetzt werden sollen. Die *Architektursicht* befasst sich mit der grundlegenden Architektur, dem Design und der Realisierung der Software-Lösung und umfasst sowohl die Definition von Komponenten als auch den Zusammenhang der einzelnen Komponenten auf der analytischen Seite des Modells. Auf der synthetischen Seite des Modells liegen konkrete getestete und integrierte Komponenten vor. Aus der Sicht des Testers werden auf dieser Ebene Integrationstests definiert und durchgeführt, die auf die Architektur und das Design abzielen.

Definition von Komponenten

Implementierungssicht

Eine weitere Verfeinerung der Entwurfs- und Designdokumente führt zur Implementierungsebene. Diese Implementierungsebene *(Implementierungssicht)* zeigt konkret spezifizierte Komponenten und stellt die Ausgangsbasis für die detaillierte Umsetzung dar. Auf der analytischen Seite beinhaltet die Phase „Komponentenspezifikation“ konkrete Angaben, wie eine Komponente realisiert werden soll. Auf der synthetischen Seite liegen getestete Komponenten vor. Auch auf dieser Ebene gilt es, Tests bereits in der Spezifikationsphase zu definieren, mit denen dann die implementierte Komponente getestet werden kann. Auf dieser Ebene wird beispielsweise sehr häufig das *Test-Driven Development* (siehe Abschnitt 5.7) eingesetzt.

Integrierte Qualitätssicherung

Neben Software-Tests, die auf den jeweiligen Ebenen definierbar (analytische Phase) und ausführbar (synthetische Phase) sind, sind im V-Modell nach jeder Phase (speziell im analytischen Zweig) Methoden der Qualitätssicherung, wie beispielsweise Reviews und Inspektionen, vorgesehen. Diese Methoden werden eingesetzt, um die Qualität der in der Phase erstellten Artefakte beurteilen zu können. Dadurch ergibt sich eine unmittelbare Rückmeldung zu erstellten Produkten und eine frühe Erkennung und Reduktion von gemachten Fehlern. Diese Art von Reviews wird häufig auch für die Freigabe an einem Meilenstein verwendet.

Vor- und Nachteile

Da das V-Modell eine klar definierte Struktur für das Software-Projekt vorgibt, ist es bei klar definierten Projekten mit stabilen und vollständigen Anforderungen besonders gut anwendbar. Eine Voraussetzung für die Anwendung des V-Modells in der dargestellten Form sind also stabile Anforderungen während der gesamten Projektlaufzeit. Aufgrund der systematischen Struktur ist auch eine gute Planung möglich. Nachteilig wirkt sich, ähnlich wie im Wasserfallmodell, auch hier der Ansatz für den Umgang mit Fehlern und Änderungen in späten Phasen des Projekts aus, da nur bedingt Rückschritte (Maßnahmen der Qualitätssicherung am Ende jeder Phase) in frühere Phasen möglich sind. Fehler können, sofern sie erst spät (beispielsweise während der Integrationsphase) erkannt werden, zu sehr hohen

Aufwänden für die Fehlerkorrektur führen. Methoden und Maßnahmen der Qualitätssicherung, wie Reviews und Inspektionen, adressieren beispielsweise Anforderungen und Fehler in frühen Phasen der Software-Entwicklung. Details zur Produkt- und Prozessverbesserung sowie geeignete und anwendbare Methoden werden in Kapitel 5 behandelt.

Verwendung für Projekte im öffentlichen Bereich

Aufgrund der klaren Struktur und guten Planbarkeit eines Software-Projekts, wird das V-Modell häufig in Projekten des öffentlichen Bereichs eingesetzt. Die Anforderungen sind in Ausschreibungsunterlagen festgehalten und können weitgehend als „stabil" betrachtet werden. Das vorgestellte V-Modell bildet die Basis für das V-Modell XT, das als verpflichtendes Vorgehensmodell in Deutschland für IT-Projekte im öffentlichen Bereich im Jahr 2005 eingeführt wurde [43]. Das V-Modell XT ist im Vergleich zu den Vorgängermodellen modular aufgebaut und flexibel und kann somit leichter an individuelle Projektgegebenheiten angepasst werden (siehe auch Abschnitt 3.9). Dadurch wird auch die Anwendbarkeit in Projekten mit unklaren und sich ändernden Anforderungen ermöglicht.

3.4 V-Modell XT

V-Modell XT

Das V-Modell XT baut auf dem vorgestellten V-Modell bzw. dem vorangegangenen Modell, dem V-Modell 97, auf und zielt unter anderem darauf ab, Projektrisiken zu minimieren, die Qualität der Produkte zu verbessern und die Kommunikation aller Beteiligten – speziell zwischen Auftraggeber und Auftragnehmer – zu verbessern. Ein flexibler und modularer Aufbau des Modells ermöglicht eine Anpassung an unterschiedliche Projektgegebenheiten *(Tailoring)*. Zentrale Elemente des Modells sind modulare und in sich abgeschlossene *Vorgehensbausteine*, die je nach Projekterfordernis flexibel im Projekt berücksichtigt werden können.

Vorgehensbausteine kapseln Produkte, Aktivitäten und Rollen

Produkte und Projektergebnisse, zugeordnete Aktivitäten und verantwortliche Rollen stehen im Mittelpunkt und bilden die Grundlage eines Vorgehensbausteins (siehe Abbildung 3.3). Jedes Produkt kann weitere abhängige Produkte beinhalten und ist einem verantwortlichen Autor zugeordnet. Produkte entstehen durch Aktivitäten, die ebenfalls abhängige Aktivitäten beinhalten können [74]. Vorgehensbausteine kapseln also Produkte, Rollen und Aktivitäten mit definierten Schnittstellen „nach außen". Diese abgeschlossenen Vorgehensbausteine können als unabhängige Einheiten gesehen werden, die – je nach Bedarf – unabhängig voneinander veränderbar und austauschbar sind. Das V-Modell XT kennt 21 Vorgehensbausteine, die sich in verpflichtende Elemente *(Kernelemente)* und *optionale Elemente* einteilen lassen (siehe dazu Tabelle 3.1 und 3.2). Die Kernelemente sind für alle Projekte anzuwenden, während die optionalen Elemente je nach Projektanforderungen eingesetzt werden müssen bzw. können.

Ein Projekt wird in unterschiedliche *Projekttypen* klassifiziert (siehe Abbildung 3.4), wobei sich ein Projekttyp im V-Modell XT an der Sichtwei-

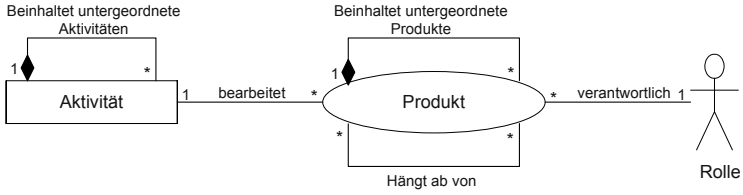


Abbildung 3.3 *Aufbau eines Vorgehensbausteins im VMXT [74].*

se auf das Projekt orientiert. Dieser Projekttyp ist nicht zu verwechseln mit dem *Projektgegenstand*, der den Inhalt eines Projekts bezeichnet (also beispielsweise Hardware-Systeme, Software-Systeme, komplexe Systeme, eingebettete Systeme und Systemintegration im V-Modell XT). Generell werden im V-Modell XT vier Projekttypen unterschieden:

Projektgegenstand

Projekttypen

- Ein *Systementwicklungsprojekt des Auftraggebers* betrachtet das Projekt aus der Sichtweise des Auftraggebers mit allen zugeordneten Aufgaben.
- Ein *Systementwicklungsprojekt des Auftragnehmers* regelt die Vorgehensweise für die Umsetzung aus der Perspektive des Auftragnehmers.
- Systementwicklungsprojekt eines *Auftragnehmers* mit dem *Auftragnehmer in derselben Organisation*. Dieser Projekttyp geht davon aus, dass Auftraggeber und Auftragnehmer häufig innerhalb derselben Organisation angesiedelt sind (beispielsweise Inhouse-Projekte), beide Sichtweisen beinhaltet und keine „externe“ Kommunikation berücksichtigen muss.
- *Einführung und Pflege eines organisationsspezifischen Vorgehensmodells*. Dieser Projekttyp dient – im Sinn der kontinuierlichen Prozessverbesserung – dazu, ein Vorgehensmodell einzuführen, laufend zu verbessern und zu warten.

Auftraggeber und Auftragnehmer

Das bedeutet, dass es aufgabenbezogene und aufeinander abgestimmte Vorgehensweisen sowohl für den Auftraggeber (AG) als auch den Auftragnehmer (AN) gibt. Dadurch wird der Auftraggeber aktiv in das Entwicklungsprojekt integriert und die Zusammenarbeit mit dem Auftragnehmer

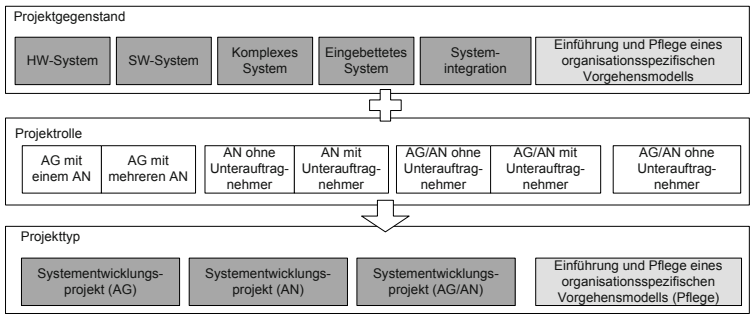


Abbildung 3.4 *Zusammenhang von Projektgegenstand, Projektrolle und Projekttyp im VMXT [74].*

Tabelle 3.1 *Verpflichtende Vorgehensbausteine nach [43].*

	Projekttyp			
Verpflichtende Vorgehensbausteine	**AG**	**AN**	**AG/AN**	**Pflege**
Projektmanagement	X	X	X	X
Qualitätssicherung	X	X	X	X
Konfigurationsmanagement	X	X	X	X
Problem- und Änderungsmanagement	X	X	X	X

verbessert. Vor diesem Hintergrund ergeben sich auch die grundlegenden Projektrollen, die – aus der Sicht des Projektmanagements – relevant sind. Für die konkrete Umsetzung eines Projekts sind im V-Modell XT konkrete Rollen innerhalb eines Vorgehensbausteins definiert [43]. Aus dem Projektgegenstand und der jeweiligen Projektrolle aus der Sicht des Projektmanagements ergibt sich also der V-Modell-XT-Projekttyp. Die Abbildung 3.4 zeigt einen Überblick über verfügbare Projektgegenstände, Projektrollen und daraus abgeleitete Projekttypen.

In den unterschiedlichen Projekttypen sind unterschiedliche Aufgaben wahrzunehmen bzw. definierte Produkte zu erstellen. Daher sind die 21 Vorgehensbausteine nur für die jeweiligen und zugeordneten Projekttypen relevant. Tabelle 3.1 gibt einen Überblick über die verpflichtenden Vorgehensbausteine; Tabelle 3.2 beinhaltet die erforderlichen und optionalen Vorgehensbausteine und illustriert die Zuordnung zu den jeweiligen Projekttypen.

Je nach Projektgegenstand und Projekttyp werden die benötigten Vorgehensbaussteine ausgewählt *(Tailoring)*, wobei die verpflichtenden Bausteine in jedem Projekt zu berücksichtigen sind. Die *erforderlichen und optio-*

Tabelle 3.2 *Erforderliche und optionale Vorgehensbausteine nach [43].*

	Projekttyp			
Erforderliche & optionale Vorgehensbausteine	**AG**	**AN**	**AG/AN**	**Pflege**
Kaufmännisches Projektmanagement	X	X	X	X
Messung und Analyse	X	X	X	X
Einführung und Pflege eines organisationsspezifischen Vorgehensmodells				X
Systementwicklung				
Anforderungsfestlegung		X		
Systemerstellung		X		
HW-Entwicklung		X		
SW-Entwicklung		X		
Logistikkonzeption		X		
Weiterentwicklung und Migration von Altsystemen		X		
Evaluierung von Fertigprodukten		X		
Benutzbarkeit und Ergonomie		X		
Systemsicherheit		X		
Multi-Projektmanagement		X		
Kommunikation zwischen Auftraggeber und Auftragnehmer				
Lieferung und Abnahme (AN)			X	
Lieferung und Abnahme (AG)			X	
Vertragsabschluss (AN)			X	
Vertragsabschluss (AG)			X	

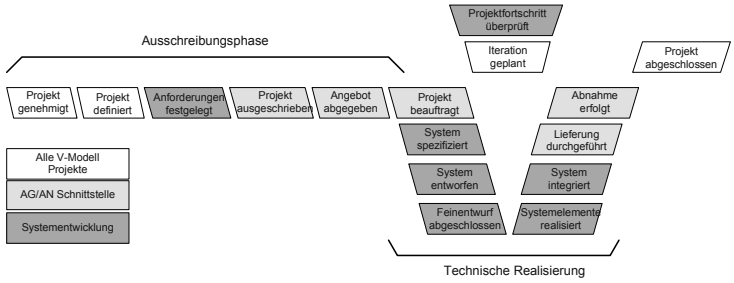


Abbildung 3.5
Projektdurchführungsstrategie mit Entscheidungspunkten nach [74].

nalen Vorgehensbausteine können bzw. müssen je nach Projekt eingebunden werden. Die Vorgehensbausteine sind zwar modular aufgebaut, beinhalten allerdings Abhängigkeiten zu anderen Bausteinen, die in der Gesamtprojektplanung berücksichtigt werden müssen. Beispielsweise gibt es definierte Abhängigkeiten zwischen *Anforderungsfestlegungen* und *Software-Entwicklung*. Bei der Auswahl der Vorgehensbausteine müssen diese Abhängigkeiten geeignet berücksichtigt werden (diese Abhängigkeiten sind in der V-Modell-Dokumentation hinterlegt).

Projektdurchführungsstrategie

Durch die „Aneinanderreihung" von Vorgehensbausteinen mit den entsprechenden Meilensteinen ergibt sich eine *Projektdurchführungsstrategie*, die den Ablauf des Software-Projekts definiert. Aus Projekttyp und Projektmerkmalen, wie beispielsweise Rolle, Risiko oder Ausschnitt aus dem Lebenszyklus, ergeben sich unterschiedliche Durchführungsstrategien, die ebenfalls im Modell hinterlegt sind. Beispielsweise unterstützt das V-Modell XT die Durchführungsstrategien *inkrementelle Entwicklung*, *komponentenbasierte und agile Systementwicklung* aber auch die *Vergabe von Systementwicklungsprojekten* (Auftraggeberrolle im öffentlichen Bereich) und die *Wartung und Pflege von Systemen* (Wartungsprojekte). Diese unterschiedlichen Projektdurchführungsstrategien definieren dabei den grundlegenden Rahmen für die systematische Durchführung eines Projekts. Einzelne Phasen werden durch sogenannte Entscheidungspunkte (vergleichbar mit Meilensteinen) abgeschlossen. Bei jedem Entscheidungspunkt muss eine definierte Menge von Produkten (die im V-Modell XT hinterlegt sind) fertig gestellt sein. Abbildung 3.5 zeigt exemplarisch eine inkrementelle Durchführungsstrategie in Anlehnung an das V-Modell XT. Da das Modell speziell auch bei öffentlichen Projekten eingesetzt wird, ist auch die Ausschreibungsphase ein wesentlicher Bestandteil des V-Modell XT.

Entscheidungspunkt

Werkzeugunterstützung

Der innovative Ansatz des V-Modells XT ermöglicht die Anpassung an projektspezifische Gegebenheiten *(Tailoring)*. Um Projektmanager bei den Tailoring-Aktivitäten und in weiterer Folge bei der Planung des Projekts zu unterstützen, existieren Werkzeuge für viele Entwicklungsumgebungen.

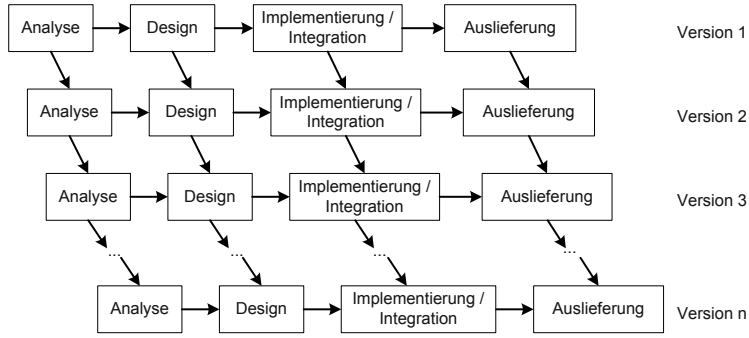


Abbildung 3.6
Schematischer Aufbau einer inkrementellen Entwicklung nach [83].

Weiterhin stellen die Autoren des V-Modells XT[2] Werkzeuge, wie beispielsweise den *Projektassistenten* für Tailoring und Projektplanung zur Verfügung. Anpassungen des V-Modell XT an unternehmensspezifische Gegebenheiten, beispielsweise die Verwendung eigener Logos, Templates und Best-Practices sind über den *V-Modell XT Editor* möglich.

Ein wesentliches Vorgehensmodell – die inkrementelle Entwicklung – wird durch das V-Modell XT unterstützt, ist aber auch als eigenständiges Vorgehensmodell einsetzbar.

3.5 Inkrementelles Vorgehen

Inkrementelle Entwicklung

Eine inkrementelle – also schrittweise – Vorgehensweise wird vor allem bei großen und komplexen Systemen eingesetzt, von denen rasch erste verwendbare Teile ausgeliefert werden sollen. Ein Hauptziel ist dabei, möglichst rasch mit einer (Teil-)Lösung auf den Markt zu kommen und dann das Software-Produkt durch laufende Ergänzungen auszubauen. Abbildung 3.6 illustriert die Vorgehensweise der inkrementellen Entwicklung. Die Darstellung beschreibt dabei nur die Eckpunkte der Entwicklung: Analyse, Design, Implementierung, Integration und Auslieferung. Dabei werden alle Phasen für eine Version (oder ein Release) durchlaufen, bis diese Version abgeschlossen ist und ausgeliefert werden kann. Parallel bzw. zeitlich versetzt kann bereits mit der Planung und Umsetzung der Folgeversion begonnen werden. Das hat den Vorteil, dass Erkenntnisse (auch geänderte Anforderungen) in diese Version einfließen können und in zeitlicher Nähe zur vorangegangenen Version verfügbar sind.

Wesentlich dabei ist, dass die einzelnen Produktteile (Versionen oder Releases) aufeinander abgestimmt sind und zusammenpassen müssen. Ein wesentliches Kriterium dafür ist, dass das grundlegende Design und die Ar-

[2] Die gesamte Dokumentation des V-Modells XT sowie Werkzeuge sind über die Webseite der Autoren unter `http://www.v-modell-xt.de` verfügbar.

chitektur stabil bleiben, da sonst die Gefahr besteht, dass die unterschiedlichen Versionen zu stark voneinander abweichen.

Priorisierung der Anforderungen

Bei der inkrementellen Vorgehensweise gibt es die Möglichkeit, priorisierte Anforderungen in den jeweiligen Versionen und Releases zu berücksichtigen [9]. Beispielsweise können die Anforderungen entsprechend der Relevanz in Bezug auf die finale Lösung für den Kunden geordnet und auf entsprechende Versionen aufgeteilt werden. Dadurch hat der Kunde den Vorteil, die wichtigsten Funktionen relativ rasch zu bekommen und auch einsetzen zu können; weniger wichtige Funktionen werden verschoben und unter Umständen erst in folgenden Versionen bereitgestellt. Weiterhin besteht die Möglichkeit, Erkenntnisse aus dem Einsatz der Software-Lösung bei der Weiterentwicklung zu berücksichtigen.

Vor- und Nachteile

Inkrementelle Modelle sind speziell bei großen und komplexen Produkten mit langen Entwicklungszeiten gut einsetzbar. Sie bieten auch den Vorteil, dass der Kunde bzw. Anwender recht schnell zu einer funktionierenden Lösung kommt, die die wichtigsten Anforderungen beinhaltet. Aufbauend auf dieser Lösung werden weitere Systemkomponenten umgesetzt. Voraussetzung dafür ist aber, dass das System auf einem stabilen Fundament (der Architektur) aufsetzt. Nachteilig kann sich das Modell auswirken, falls die einzelnen Versionen nicht zusammenpassen, die durch massive Änderungen von Architektur, Entwurf und Design, aber auch durch geänderte Kundenanforderungen verursacht werden können. Dies zu erkennen, stellt eine große Herausforderung für das Projektmanagement und den Software-Architekten dar.

3.6 Spiralmodell

Vier grundlegende Schritte im Spiralmodell

Das Spiralmodell stellt eine konkrete Ausprägung der inkrementellen Entwicklung dar, in dem vier grundlegende Schritte solange zyklisch durchlaufen werden, bis das Produkt in einer zufriedenstellenden Qualität vorliegt [12]. Ein Produkt kann dabei auch die Erstellung einer Anforderungsspezifikation oder eines Architekturdokuments sein und muss nicht ein komplettes Projekt umfassen. Je nach Risiko oder Komplexität können diese Schritte auch mehrfach durchlaufen werden. Diese vier Schritte, in Abbildung 3.7 schematisch dargestellt, umfassen:

1. Definition von Zielen und Alternativen.
2. Einschätzung des Risikos.
3. Entwicklung und Durchführung von Tests und Evaluierungen der aktuellen Ergebnisse.
4. Feedback zur erstellten Lösung und Planung für die nächste Iteration.

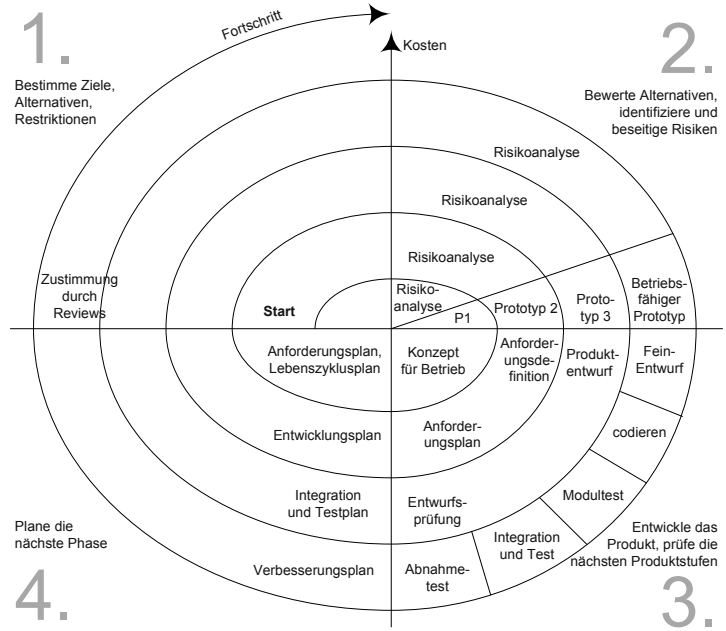


Abbildung 3.7
Schematischer Aufbau des Spiralmodells [12].

Nach der Definition der Ziele folgt jeweils eine Risikoabschätzung der geplanten Iteration. Am Ende steht ein Review der Produkte und Artefakte, die in der jeweiligen Iteration erstellt wurden. Je nach Qualität des Ergebnisses wird eine erneute Iteration (eventuell mit alternativen Lösungen) geplant und umgesetzt, bis das Ergebnis den Erwartungen entspricht, oder die nächste Phase gestartet. Auf der x-Achse ist somit der Projektfortschritt im Sinn des Software-Lebenszyklusses abgebildet, die Entfernung vom Mittelpunkt visualisiert den steigenden Aufwand, die Kosten und die Dauer des Projekts mit jeder neuen Iteration. Je mehr Iterationen durchgeführt werden, umso breiter wird die Spirale und umso länger dauert die Entwicklung und desto teurer wird das Projekt. Eine deterministische Projektplanung ist hier nur für eine überschaubare Menge an Iterationen möglich, für die gesamte Projektdauer ist eine Projektplanung nur sehr schwer machbar.

Risikogetriebene Entwicklung

Dieses risikogetriebene Modell eignet sich speziell für sehr lange und sehr große Software-Projekte mit unklaren Anforderungen. Durch die Planung von Iterationen nähert sich das Projekt an die definierten Ziele an, auch wenn diese Ziele sich im Lauf des Projekts ändern können.

3.7 Rational Unified Process

Rational Unified Process

Der Rational Unified Process (RUP) [61] ist ein phasenorientierter, inkrementeller und iterativer Software-Prozess zur Herstellung von Software-Produkten im kommerziellen Umfeld. Wie in Abbildung 3.8 dargestellt, gliedert sich das Prozessmodell in vier grundlegende Phasen, die jeweils

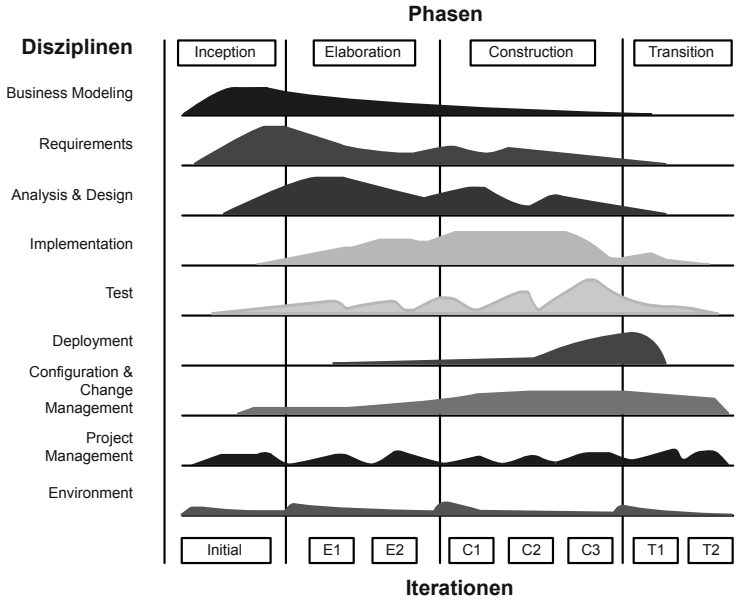


Abbildung 3.8
Schematischer Aufbau des Rational Unified Process [61].

mehrere Iterationen durchlaufen können [61]. Beispielsweise beinhaltet die Phase „Elaboration“ zwei Iterationen E1 und E1. Jede Phase wird mit einem Meilenstein abgeschlossen, bei dem die erstellten Produkte *(Artefakte)* überprüft und freigegeben werden. Artefakte stellen die Ergebnisse von definierten *Aktivitäten* dar und sind einer *verantwortlichen Rolle* zugeordnet. Sind noch nicht alle Artefakte verfügbar oder liegen diese nicht in der benötigten Qualität vor, wird die betroffene Phase um eine weitere Iteration verlängert. Generell sind im Rational Unified Process vier wesentliche Phasen definiert:

Phasen und Iterationen

- *Inception (Beginn)*. Am Beginn eine Software-Projekts werden die wichtigsten Eckpunkte des Projekts definiert. Schwerpunkte liegen dabei auf der Definition der Geschäftsfälle *(Business Modeling)*, der grundlegenden Anforderungen und der Rahmenbedingungen des Projekts.
- *Elaboration (Entwurf und Design)*. Die Aufgaben dieser Phase umfassen die Spezifikation der Anforderungen und die Festlegung der Architektur sowie des Designs der geplanten Software-Lösung. Außerdem werden die jeweiligen Iterationen für die *Construction Phase* (die Implementierung) definiert.
- *Construction (Implementierung)*. Basierend auf den Anforderungen und einer stabilen Architektur werden die jeweiligen Komponenten und Anforderungen umgesetzt und getestet.
- *Transition (Auslieferung)*. Der Schwerpunkt dieser Phase umfasst die Bereitstellung des Software-Produkts und die Auslieferung an den

Kunden. In dieser Phase werden auch Artefakte für die Installation und die Konfiguration erstellt. Ein typischer Abschluss dieser Phase ist der Meilenstein *Product Release*.

Eine Phase kann mehrere Iterationen zur Herstellung eines Produkts umfassen. Die Anzahl der Iterationen wird vom Projektmanagement festgelegt und typischerweise durch das Ergebnis der Meilensteinreviews beeinflusst. Erfüllt beispielsweise ein Produkt die erforderlichen Qualitätskriterien nicht, kann eine weitere Iteration durchgeführt werden.

Disziplinen und Workflows

Unterstützend zu den Phasen und Iterationen stellt der Rational Unified Process eine Reihe von Vorgaben und Richtlinien zur Verfügung, die in *Disziplinen und Workflows* organisiert sind. Die vier Phasen verteilen sich auf insgesamt neun Disziplinen, die definierte Vorgehensweisen für einzelne Aufgaben bereitstellen und durch die *Unified Modeling Language* unterstützt werden (siehe Abschnitt 6.1). Von diesen neun Disziplinen existieren sechs *Engineering Workflows* und drei *Supporting Workflows*. Zu den Engineering Workflows zählen:

Engineering Workflows

- *Business Modeling*. Die Geschäftsfallmodellierung ermöglicht ein einheitliches Verständnis aller beteiligten Stakeholder im Hinblick auf die zu erstellende Software-Lösung und die betriebswirtschaftlichen Gegebenheiten. UML unterstützt diesen Workflow durch Paket-, Use-Case-, Klassen- und Kollaborationsdiagramme.
- *Requirements*. Ziel dieses Workflows ist die detaillierte Erhebung der Anforderungen, die das geplante System erfüllen soll. Aufbauend auf den initialen Use-Cases aus dem Business Modeling Workflow steht in diesem Schritt die Verfeinerung der Use-Cases – bereits im Hinblick auf die technische Umsetzung – im Vordergrund.
- *Analysis & Design*. Die erfassten und modellierten Anforderungen bilden die Basis für die Architektur des Software-Systems und stellen zentrale Artefakte dieses Workflows dar. Dazu sind beispielsweise Architektur-, Design- und Testdokumente auf Architekturebene erforderlich, die unter anderem auch Klassen- und Kollaborationsdiagramme beinhalten.
- *Implementation*. Dieser Workflow definiert, wie die einzelnen Komponenten implementiert, getestet und zu größeren Systemen zusammengebaut (integriert) werden. Neben den Designdokumenten spielen hier speziell Paketdiagramme ein zentrale Rolle.
- *Test*. Qualitätssicherung und Tests sind zentrale Elemente des *Rational Unified Process* und daher in einem eigenständigen Workflow definiert. Die Erstellung von Testfällen findet bereits in frühen Phasen der Entwicklung statt und erhöht unter anderem auch das Verständnis

für das zu entwickelnde Software-System. Die eigentliche Testausführung kann erst bei vorliegenden Komponenten, Subsystemen und System stattfinden. Je nach Testebene dienen die jeweiligen Artefakte und UML-Diagramme als Basis für die Testdefinition und Durchführung.

- *Deployment*. Dieser Workflow regelt die reibungslosen Fertigstellung und Auslieferung des Software-Produkts. Obwohl der Hauptaufwand in der Phase Transistion *(Übergang)* liegt, muss sich der Architekt bereits bei Planung und Design überlegen, wie diese Auslieferung stattfinden soll.

Neben den Engineering Workflows sind im Rahmen der Software-Entwicklung auch Workflows notwendig, die die konstruktiven Entwicklungsaufgaben unterstützen. Zu diesen drei unterstützenden Workflows zählen:

Supporting Workflows

- *Configuration & Change Management*. Dieser Workflow regelt die Organisation des Versions- und Konfigurationsmanagements und definiert, wie mit Änderungen umzugehen ist. Durch dieses breite Anwendungsfeld und die Notwendigkeit der Dokumentenversionierung sind alle projektrelevanten Artefakte und UML-Diagramme über den gesamten Entwicklungsprozess involviert.

- *Project Management*. Die übergreifende Aktivität Projektmanagement beschäftigt sich mit der Planung und Koordination des Projekts unter Berücksichtigung von knappen Ressourcen (etwa Personal, Zeit, Budget und Infrastruktur), Qualität und Quantität. Weiterhin entscheidet das Projektmanagement über die Durchführung weiterer Iterationen aufgrund der Ergebnisse von Meilenstein-Reviews.

- *Environment*. Dieser Workflow definiert, welche Ressourcen dem Entwicklungsteam zur Verfügung stehen und regelt beispielsweise, wie mit Prozessen, Werkzeugen und Methoden umzugehen ist.

Abbildung 3.8 zeigt den Grobaufbau von RUP [61] mit den Phasen, Iterationen und den Disziplinen sowie dem anteiligen Aufwand pro Phase und Disziplin. Wie schon aus der kurzen Beschreibung der Disziplinen hervorgeht, liegt ein besonderer Vorteil des RUP in einer durchgängigen Modell-Unterstützung durch unterschiedliche Diagramme der UML-Diagrammfamilie (siehe dazu Abschnitt 6.1). Diese Diagramme dienen einerseits der Visualisierung von technischen Zusammenhängen zur Verbesserung der Kommunikation zwischen allen Stakeholdern, und sie stellen anderseits die Basis für die schrittweise Entwicklung der Software-Lösung dar.

Vor- und Nachteile

Aufgrund der umfassenden Prozessdefinition mit den vordefinierten Produkten, Rollen und Aktivitäten ist der Rational Unified Process für große Projekte gut geeignet und bietet eine hilfreiche Unterstützung bei der Projektabwicklung. Durch die Anbindung von Modellierungswerkzeugen und

der Unified Modeling Language (UML) können „reale“ Szenarien abgebildet werden, bei denen der Zusammenhang zwischen realem Problem und konkreter Problemlösung im Vordergrund steht. Dem gegenüber stehen als Kritikpunkte, dass das Modell sehr komplex und unflexibel ist, durch eine hohe Anzahl von erforderlichen Dokumenten „überladen“ wird und dadurch in seiner Praxistauglichkeit eingeschränkt ist.

3.8 Agile Software-Entwicklung

Agile Software-Entwicklung

Kritikpunkte an traditionellen Vorgehensmodellen, wie beispielsweise dem V-Modell, dem Spiralmodell oder dem Rational Unified Process, sind häufig unter anderem mangelhafte Flexibilität (durch vorgegebene starre Strukturen), mangelnde Einbeziehung des Kunden in den Entwicklungsprozess sowie ein (scheinbar unnötig) hoher Dokumentationsaufwand. Agile Software-Entwicklungsprozesse sollen diese Nachteile beheben und ein höheres Maß an Flexibilität und Kundennähe bei einem Mindestmaß an Dokumentation ermöglichen. Agile Entwicklungsprozesse gehen auf das Agile Manifest [6], das von 17 Software-Entwicklern im Jahr 2001 veröffentlicht wurde, zurück.

Agiles Manifest

> „Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan.“
> *Agiles Manifest [6]*

Dieses Agile Manifest bildet die Basis für agile Entwicklungsprozesse wie eXtreme Programming [5] oder SCRUM [80]. Das Manifest beinhaltet 12 Grundprinzipien für die moderne agile Software-Entwicklung [6]. Hauptzielsetzung ist, rasch eine funktionsfähige Software-Lösung auszuliefern, um den Kundenwunsch bestmöglich zu erfüllen. Dabei spielt die Interaktion mit dem Kunden eine zentrale Rolle: Eine enge Zusammenarbeit mit dem Kunden ermöglicht eine flexible Handhabung von Anforderungen (auch Änderungen von Anforderungen) und eine unmittelbare Rückmeldung des Kunden in Bezug auf gelieferte Software-Komponenten.

SCRUM

Stellvertretend für agile Software-Prozesse wird in diesem Abschnitt *SCRUM* vorgestellt. SCRUM hat in den letzten Jahren stark an Bedeutung zugenommen [80] und wird auch in der industriellen Praxis häufig eingesetzt. Was steckt hinter diesem Begriff? SCRUM ist keine Abkürzung, sondern wurde in Anlehnung an die Startformation im Rugby-Sport als Begriff für diesen agilen Ansatz gewählt, um die Bedeutung von kleinen und hocheffizienten Teams *(self-organizing teams)* in den Vordergrund zu rücken. Agil bedeutet in diesem Zusammenhang keineswegs „unkontrolliert“, da auch hier Prozesse und Regeln existieren, die eingehalten werden müssen.

Self-Organizing Teams

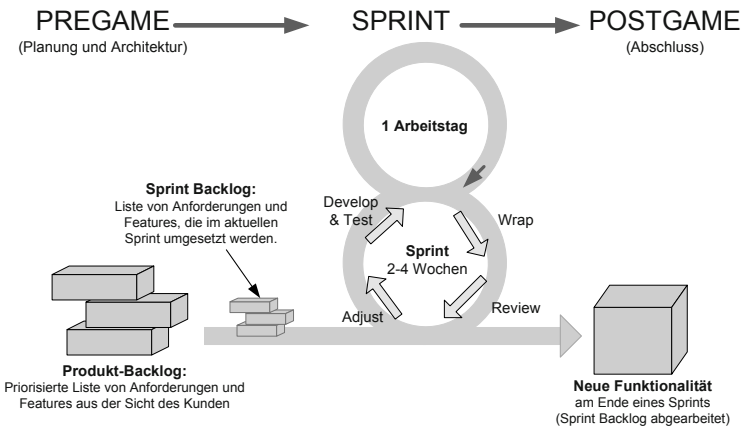


Abbildung 3.9
SCRUM-Phasen und schematischer Aufbau eines Sprints [80].

SCRUM definiert ein agiles Software-Projekt aus der Sicht des Projektmanagements und besteht aus einer Sammlung von Prozeduren, Rollen und Methoden zur erfolgreichen Projektdurchführung. Das Prozessmodell umfasst drei wesentliche Phasen, die in Abbildung 3.9 dargestellt sind.

Phasen

- In einem *Pregame* erfolgt die Festlegung der wesentlichen Produkteigenschaften und der grundlegenden Architektur sowie die Projektplanung. Alle Eigenschaften und gewünschten Features werden gemeinsam mit dem Kunden in einem *Produkt-Backlog* gesammelt und priorisiert. In diesem Produkt-Backlog werden auch geänderte Anforderungen gesammelt.
- Die eigentliche Entwicklungsarbeit wird in einem *Sprint* durchgeführt. Vor einem Sprint werden die wichtigsten und machbaren Features aus dem priorisierten Produkt-Backlog ausgewählt und in das *Sprint-Backlog* übernommen. Wichtig ist dabei, dass die Auswahl und Planung nur so viele Features umfasst, wie das Team im nächsten Sprint auch tatsächlich umsetzen kann. Das Sprint-Backlog beinhaltet also einen machbaren Auszug aus dem Produkt-Backlog.
- Die abschließende Phase – das *Postgame* – umfasst die Bereitstellung und Auslieferung neuer Funktionalität in Form von (neuen) Releases.

Sprint

Die eigentliche technische Entwicklung wird in einem Sprint durchgeführt (siehe Abbildung 3.9) und umfasst vier wesentliche Schritte, die zyklisch durchlaufen werden: (a) Entwicklung eines Stücks neuer Software *(Develop & Test)*, (b) Integration der entwickelten Komponenten *(Wrap)*, (c) Review des integrierten Produkts *(Review)* und (d) Anpassung und Verbesserung der aktuell vorliegenden Lösung *(Adjust)*. Die typische Entwicklungsdauer beträgt – je nach Komplexität des Produkts und der Anzahl der gewählten Arbeitspakete aus dem Produkt-Backlog – zwischen zwei und vier Wochen. Ein wesentliches Merkmal eines Sprints ist, dass das Team normalerweise während eines Sprints ungestört arbeiten kann, ohne durch äußere

Einflüsse gestört zu werden. Beispielsweise fließen geänderte Anforderungen nicht direkt in den Sprint-Backlog ein, sondern werden außerhalb des Sprints (im Produkt-Backlog) gesammelt. Diese Änderungen werden dann beim nächsten Planungsmeeting (vor dem nächsten Sprint) berücksichtigt. Bei kritischen Anforderungsänderungen kann der aktuelle Sprint natürlich abgebrochen werden.

Daily Scrum

Während eines Sprints werden täglich kurze Besprechungen – sogenannte *Daily Scrums* – durchgeführt, um den Status der Entwicklung zu erheben und effizient weiterzutreiben. Typischerweise werden für ein Daily Scrum etwa 15 Minuten vorgesehen. Diese tägliche Besprechung ist durch drei zentrale Fragen gekennzeichnet:

- *Was wurde seit dem letzten Daily Scrum erledigt?* Dadurch wird der aktuelle Projektstatus für alle Teammitglieder sichtbar kommuniziert.
- *Sind während der Entwicklung Probleme aufgetreten und – falls ja – welche waren das?* Dadurch ist es möglich aufgetretene Probleme schnell zu analysieren und zeitnah eine geeignete Lösung zu finden.
- *Was ist das Ziel bis zum nächsten Daily Scrum?* Dieses Element deckt die Planung und Zielsetzung bis zum nächsten Meeting ab. Dadurch wird das Projekt effizient vorangetrieben.

Burndown-Chart

Diese Vorgehensweise hat den Vorteil, dass alle Teammitglieder jeweils über den aktuellen Stand des Projekts Bescheid wissen, aktuelle Probleme direkt im Team diskutiert werden können und somit die Effizienz der Entwicklung deutlich gesteigert werden kann. Für die Projektplanung und Fortschrittskontrolle werden sogenannte *Burndown-Charts* verwendet. Dieses Planungs- und Steuerungsinstrument ist eine grafisch aufbereitete Darstellung, das den Restaufwand eines Sprints darstellt. Bei einer guten und realistischen Aufwandschätzung am Beginn eines Sprints sollte das Burndown-Chart (also der Restaufwand) kontinuierlich sinken. Probleme im Projekt können so recht schnell erkannt werden, da diese Charts täglich aktualisiert werden (siehe auch Abschnitt 4.3.6).

Zusammenfassend bietet eine Projektdurchführung nach SCRUM folgende Vorteile:

Vorteile von SCRUM

- *Flexibilität bezüglich sich ändernder Kundenanforderungen*. Die Sammlung und Priorisierung von Features erfolgt im Produkt-Backlog. Die Auswahl und Übernahme der wichtigsten Features ins Sprint-Backlog erfolgt am Beginn des Sprints. Während eines Sprints sind keine Änderungen zugelassen.
- *Kleine Teams ermöglichen eine hohe Flexibilität des Projekts*. Große Teams sind mit SCRUM zwar ebenfalls möglich, erfordern allerdings einen hohen Kommunikations- und Synchronisierungsaufwand während des Entwicklungsprozesses und stellen eine Herausforderung für

das Projektmanagement dar. Auch mehrere parallel arbeitende Teams sind möglich, was ebenfalls eine Herausforderung an das Projektmanagement darstellt.

- *Qualitätssicherung* ist ein integraler Bestandteil des Sprints *(Develop & Test – Wrap – Review – Adjust)*. Typischerweise kommen in SCRUM-Methoden wie *Test-Driven Development*, *Pair Programming* und *Reviews* zum Einsatz. Diese Maßnahmen ermöglichen ein schnelles Feedback zur Qualität der Software-Lösung und führen zur kontinuierlichen Verbesserung des Produkts.
- *Software-Lösungen stehen rasch zur Verfügung*. Der Kunde erhält rasch (die Dauer eines Sprints beträgt zwei bis vier Wochen) eine funktionierende Teillösung, entsprechend den priorisierten Anforderungen und Features.

3.9 Anpassung von Vorgehensmodellen

Anpassung von Vorgehensmodellen

In der betrieblichen Praxis finden sich zahlreiche Vorgehensmodelle (eine Auswahl wurde in den vorangegangenen Abschnitten 3.2 bis 3.8 vorgestellt), die sich auf den Software-Lebenszyklus zurückführen lassen (siehe Kapitel 2). Allerdings sind Vorgehensmodelle *out of the box* nur selten direkt anwendbar, sondern müssen an spezifische Gegebenheiten einer Organisation oder eines Projekts angepasst werden. Der Themenbereich *Standardisierung*, der bereits in Abschnitt 2.6 kurz diskutiert wurde, spielt auch bei Vorgehensmodellen eine wesentliche Rolle, um Software-Projekte standardisiert und vergleichbar ablaufen zu lassen. Dadurch wird die Zusammenarbeit innerhalb der Organisation erleichtert, die Teammitglieder finden sich leichter im Projekt zurecht und der Einsatz von Best-Practices erfolgt besser und effizienter. Weiterhin stellen standardisierte Vorgehensweisen das Fundament für die kontinuierliche Verbesserung der Software-Entwicklung dar.

Organisation, Domäne und Projekt

Eine Voraussetzung für die Standardisierung von Vorgehensmodellen ist die Fähigkeit, das Modell anzupassen *(Tailoring)*. Anpassungen können notwendig sein, um (a) ein generisches Prozessmodell an eine Organisation *(organisationsspezifisches Vorgehensmodell)*, (b) ein Prozessmodell an eine Domäne *(domänenspezifisches Vorgehensmodell)* oder (c) an ein konkretes Projekt *(projektspezifisches Vorgehensmodell)* anzupassen. Sind in einer Organisation beispielsweise immer ähnliche Projekte (etwa Webapplikationen) zu finden, empfiehlt sich die Verwendung eines standardisierten Vorgehensmodells, das an den konkreten Anwendungsbereich angepasst ist. Die Anpassung eines generischen Software-Prozesses an individuelle Gegebenheiten (etwa an eine Organisation, eine Domäne oder ein Projekt) bezeichnet man auch als *Prozess-Tailoring*. Betrachtet man beispielsweise den Software-Lebenszyklus, ist die strikte Verfolgung eines

Prozess-Tailoring

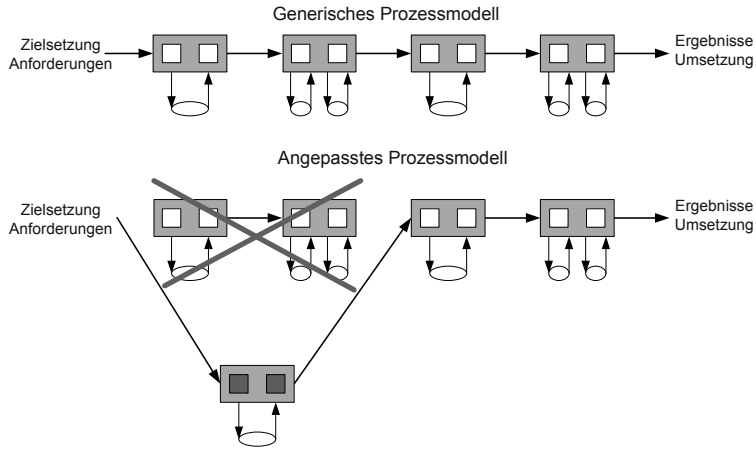


Abbildung 3.10 *Beispiel für Prozess-Tailoring.*

sequentiellen Prozessablaufes nicht immer möglich. Beispielsweise können spezifische Projektgegebenheiten oder instabile Anforderungen Anpassungen des Prozessmodells erfordern. Mögliche Lösungen können etwa die Zusammenlegung einzelner Schritte (etwa Analyse- und Designphase) oder der Austausch generischer Methoden durch unternehmensspezifische Best-Practices (wie beispielsweise die Einführung von Test-Driven Development) sein. Die Abbildung 3.10 zeigt ein Beispiel für die Anpassung eines sequenziellen Software-Prozesses. In diesem Beispiel werden die ersten beiden Phasen durch eine organisationsspezifische Best-Practice-Phase ersetzt. Es ist jedoch notwendig, alle Schritte oder Phasen sowie die relevanten Produkt- und Prozessabhängigkeiten innerhalb des Modells konsistent zu halten.

Aufgabe des Projektmanagements

Prozess-Tailoring ist eine zentrale Aufgabe des Projektmanagements im Rahmen der Planungsphase; es wird also definiert, wie das Projekt organisatorisch umgesetzt werden soll. Da die Abhängigkeiten innerhalb des Vorgehensmodells berücksichtigt werden müssen, ist die Anpassung eines Vorgehensmodell zeitaufwendig und anspruchsvoll und erfordert erfahrene Projektleiter. Modellspezifische Werkzeuge beim Prozess-Tailoring von Software-Prozessen unterstützen die Projektleiter bei dieser Aufgabe. Beispielsweise stellt das V-Modell XT (siehe Abschnitt 3.4) durch den Projektassistenten ein Werkzeug zur effizienten Anpassung des Vorgehensmodells zur Verfügung. Der Projektassistent berücksichtigt die im Vorgehensmodell hinterlegten Abhängigen von Produkten, Aktivitäten und Phasen und erleichtert spezifische Anpassungen.

Process Customization

Da die Anpassung eines generischen Vorgehensmodells zeitaufwendig ist, ist es naheliegend, in einem ersten Schritt ein generisches Modell auf eine Organisation oder eine Domäne anzupassen. Diesen Schritt bezeichnet man als *„Process Customization“*. In einem zweiten Schritt wird dieses angepasste Modell als Ausgangsbasis verwendet und an individuelle Projekte angepasst *(Prozess-Tailoring)*.

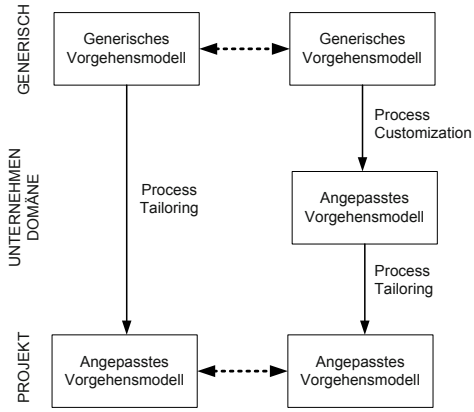


Abbildung 3.11
Anwendung von Prozess-Tailoring.

Dementsprechend gibt es zwei unterschiedliche Anwendungsmöglichkeiten, die in Abbildung 3.11 dargestellt sind:

- *Tailoring eines generischen Prozesses* im Hinblick auf spezifische Projektgegebenheiten.
- *Customization* eines generischen Prozessmodells für eine Organisation oder eine Domäne und – ausgehend von diesem angepassten Prozessmodell – weiteres *Tailoring* für ein individuelles Projekt.

Zusammenhang von Customization und Tailoring

Abbildung 3.11 visualisiert die schematische Anwendung von Customization und Tailoring. Auf der linken Seite ist die direkte Anwendung eines generischen Software-Prozesses auf ein konkretes Projekt dargestellt. Dabei muss – ausgehend vom generischen Modell – der Tailoring-Prozess jeweils für jedes Projekt durchgeführt werden. Auf der rechten Seite ist der Zwischenschritt mit Customization dargestellt. Dabei wird im ersten Schritt die Anpassung auf das Unternehmen, die Domäne oder Produktlinie durchgeführt (Customization), was die Basis für die weitere Anwendung ist. Ausgehend von diesem angepassten Modell erfolgt das individuelle Tailoring auf Projektebene.

Der Schritt *Customization* bietet auch zusätzlich die Möglichkeit, individuelle organisationsspezifische Best-Practices (etwa Unternehmensstandards, typische Anwendungsbereiche und Projekttypen) – im Sinn von *Process Pattern* – geeignet zu berücksichtigen. Derartige Process Pattern bieten speziell für große Organisationen den Vorteil, die Software-Entwicklung durch die Integration von Best-Practices zu standardisieren und die Effizienz der Projektabwicklung zu steigern. Prozess-Customization und Tailoring sind aber auch für Klein- und Mittelbetriebe sinnvoll, da nur relevante und passende Prozess-Schritte, Methoden und Werkzeuge verwendet werden müssen. Wichtig bei Customization und Tailoring ist, dass jeweils die Kompatibilität und Konsistenz zu dem zugrunde liegenden generischen Prozessmodell sichergestellt werden muss. Änderungen im generischen Prozessmodell müssen jeweils in die angepassten Varianten umgesetzt werden.

Kompatibilität zum Basismodell

3.10 Zusammenfassung

Das Wichtigste in Kurzform

Vorgehensmodelle sind konkrete Ausprägungen eines spezifischen Vorgehens für die Durchführung eines Software-Projekts. Typischerweise orientieren sich Vorgehensmodelle am Software-Lebenszyklus und betrachten dabei einen konkreten Ausschnitt daraus. Die meisten Vorgehensmodelle beschäftigten sich mit dem Prozess der technischen Entwicklung eines Software-Produkts.

Wasserfallmodell

Das *Wasserfallmodell* orientiert sich am Software-Lebenszyklus und ist in der Praxis heute noch gelegentlich anzutreffen. Die Phasen werden sequenziell durchlaufen. Jede Phase, die mit einem Review abgeschlossen wird, muss erst vollständig abgeschlossen werden, bevor die nächste Phase begonnen werden kann. Das Modell ist bei Projekten mit sehr klar definierten Anforderungen einsetzbar, da Rückschritte zu vorangegangenen Phasen, beispielsweise für Anpassungen, nicht oder nur eingeschränkt möglich sind.

V-Modell und V-Modell XT

Das *V-Modell* ist durch seine typischer Form (das „V“) gekennzeichnet und gliedert sich in eine analytische und eine synthetische Phase. Während in der analytischen Phase ein immer höherer Detailgrad der Spezifikation bis hin zur konkreten Realisierung auf Komponentenebene vorliegt, befasst sich die synthetische Phase mit dem systematischem Zusammenbau des Systems von einzelnen Komponenten bis zum Gesamtsystem. Das V-Modell stellt – durch definierte Sichten – den Zusammenhang zwischen Spezifikation und Umsetzung bzw. Test in den Vordergrund (Spezifikation in der analytischen Phase und Realisierung in der synthetischen Phase). Das *V-Modell XT* ist eine Weiterentwicklung des V-Modells und ein Standard für öffentliche IT-Projekte in Deutschland. Dementsprechend wird in diesem flexiblen Vorgehensmodell auch die Ausschreibungsphase explizit berücksichtigt. Ein wesentliches Merkmal dieses Vorgehensmodells ist die Miteinbeziehung von Auftraggeber und Auftragnehmer durch definierte Vorgehensweisen zur verbesserten Zusammenarbeit und Kommunikation. Der modulare Aufbau des Modells (Vorgehensbausteine kapseln Produkte, Aktivitäten und verantwortliche Rollen) ermöglicht die Anpassung (Tailoring) an individuelle Projektgegebenheiten. Eine Durchführungsstrategie legt den konkreten Ablauf eines Software-Projekts durch definierte Entscheidungspunkte fest.

Inkrementelles Vorgehen

Speziell bei großen und lang andauernden Projekten kann ein *inkrementelles Vorgehen* gewählt werden. Ein Kennzeichen für ein inkrementelles Vorgehen besteht speziell darin, dass die einzelnen Phasen des Software-Lebensyzklusses zeitlich versetzt und parallel durch mehrere Teams durchlaufen werden. Erkenntnisse aus einem früheren Release können unmittelbar in die Weiterentwicklung einfließen. Jedes Team ist für ein Release zuständig. Für jeden Build werden priorisierte Anforderungen ausgewählt, die in einem Releasezyklus erstellt werden. Dadurch bekommt der Kunde schnell eine funktionierende Teillösung des Systems. Bei extrem großen und un-

überschaubaren Projekten empfiehlt sich ein risikogetriebenes Vorgehen, wie es beispielsweise durch das *Spiralmodell* ermöglicht wird. Dabei wird ein Zyklus – bestehend aus vier Phasen – so lange durchlaufen, bis das Ergebnis das erwartete Qualitätsniveau aufweist.

Spiralmodell

Eine umfassende Modell- und Werkzeugunterstützung bietet beispielsweise der *Rational Unified Process* (RUP). Das Prozessmodell ist ein phasenorientiertes Vorgehensmodell mit vier Phasen, wobei jede Phase mehrere Iterationen beinhalten kann. Neben den jeweiligen Phasen enthält das Modell neun Disziplinen (sechs Engineering und drei unterstützende Disziplinen), die konkrete Abläufe für konkrete Themenbereiche bereitstellen und je nach Phase eine entsprechende Bedeutung (und Anteil am Aufwand) haben. Hervorzuheben ist auch die systematische Einbindung von Modellierungstechniken in allen Phasen und Disziplinen (UML-Unterstützung).

Rational Unified Process

Kritikpunkten an den bisher vorgestellten Vorgehensmodellen (wie mangelnde Flexibilität, zu hoher Komplexitätsgrad oder zu hoher Dokumentationsaufwand) wollen agile Ansätze entgegenwirken. Modelle aus der *agilen Software-Entwicklung*, wie beispielsweise eXtreme Programming und SCRUM, rücken den Kunden und seine konkreten Anforderungen in den Vordergrund. Durch kurze Iterationen, engen Kundenkontakt und den Einsatz von Best-Practices sind diese Modelle sehr flexibel und können auf geänderte Rahmenbedingungen rasch reagieren. Die Dokumentation wird auf das notwendigste Minimum beschränkt.

Agile Software-Entwicklung

Allen Vorgehensmodellen gemeinsam ist, dass die direkte Anwendung eines generischen Vorgehensmodells in der betrieblichen Praxis meist nur schwer möglich ist. Daher sind Anpassungen an organisations-, domänen- und projektspezifische Gegebenheiten notwendig. Prozess-Tailoring ermöglich die individuelle Anpassung an individuelle Projektgegebenheiten. Prozess-Customization stellt einen Ansatz dar, der es ermöglicht, beispielsweise ein organisationsspezifisches Vorgehensmodell – basierend auf einem generischen Prozessmodell – zu erstellen.

Anpassung von Vorgehensmodellen

Vorgehensmodelle stellen also einen organisatorischen Rahmen für die Durchführung von Software-Entwicklungsprojekten dar. Sowohl die Auswahl eines geeigneten Modells als auch dessen Anpassung an individuelle Projektgegebenheiten sind zentrale Aufgaben des Projektmanagements. Kurz zusammengefasst beschäftigt sich der Themenbereich Projektmanagement mit planerischen und steuernden Aufgaben eines Software-Projekts im Hinblick auf die Durchführung des Projekts in Anlehnung an ein konkretes Vorgehensmodell.

4 | Software-Projektmanagement

Das Projektmanagement begleitet das Software-Projekt während des gesamten Entwicklungsprozesses. Am Beginn stehen Aktivitäten zur initialen Festlegung der durchzuführenden Schritte (Projektdefinition, Kick-off und Projektplanung). Während der Projektdurchführung stehen steuernde Aspekte im Vordergrund, um auf geänderte Rahmenbedingungen oder Abweichungen vom initialen Projektplan geeignet reagieren zu können (Projektverfolgung). Dieses Kapitel widmet sich den wichtigsten Kernelementen des Projektmanagements zur erfolgreichen Durchführung eines Software-Projekts. Der Projektauftrag definiert den Rahmen der gewünschten Ergebnisse und der verfügbaren Mittel. Daraus wird ein initialer Projektplan zur Strukturierung des Projekts im Hinblick auf Arbeitspakete und Ressourcen erstellt. Während des Projektverlaufs wird in regelmäßigen Abständen überprüft, ob die definierten Ziele des Projektauftrags nach wie vor gültig sind und der Arbeitsfortschritt mit der ursprünglichen Planung übereinstimmt. Bei Abweichungen sind geeignete Maßnahmen notwendig, die eine Anpassung des Projektplans oder geeignete inhaltlich Projektanpassungen (nach Rücksprache mit dem Auftraggeber) umfassen können. Nach Abschluss des Projekts werden Erfahrungen aus dem Projekt gesammelt und können als „Lessons Learned“ in Folgeprojekten zum Einsatz kommen.

Übersicht

4.1 Einführung ins Projektmanagement

Das Projektmanagement ist eine begleitende Tätigkeit im Rahmen der Software-Entwicklung über den gesamten Entwicklungsprozess. Analog zum Software-Lefbenszyklus (siehe Kapitel 2) beginnt Projektmanagement ebenfalls beim Konzept und der ersten Idee (Projektdefinition oder Projektauftrag), umfasst eine initiale Planung am Beginn eines Projekts oder einer Iteration, eine laufende Projektunterstützung (Projektverfolgung) und endet mit dem Projektabschluss. Software-Prozesse oder Vorgehensmodelle (siehe Kapitel 3) sind konkrete Vorgehensweisen zur strukturierten Abwicklung eines Projekts und dienen als Rahmen für die Produkt- und Komponentenentwicklung und somit auch für das Projektmanagement.

Projektmanagement

Die DIN 69901 definiert Projektmanagement als *Gesamtheit von Führungsaufgaben, -organisation, -techniken und -mitteln für die Abwicklung eines Projekts* [24]. Kernaufgabe des Projektmanagements ist also die Organisation der zu erstellenden Artefakte zu einem definierten Zeitpunkt, die Überprüfung der Artefakte durch geeignete Maßnahmen der Qualitätssicherung und die Ressourcenplanung (etwa Terminplanung, Auswahl der Teammitglieder und Festlegung der Projektinfrastruktur) für eine erfolgreiche Projektdurchführung.

Stellt sich jetzt die Frage, was ein *„Projekt“* im Sinn des Projektmanagements überhaupt ist. Karnovsky definiert ein Projekt beispielsweise folgendermaßen [56]:

Projekt

> Ein Projekt ist (im Gegensatz zum normalen Tagesgeschäft oder zur Produktion) ein einmaliges Vorhaben mit einem definierten Anfang, einem definierten Ende und mehreren beteiligten Personen. *Karnovsky [56]*

Projektgröße und Komplexität

Ein Projekt erfordert meist den Einsatz mehrerer Personen über einen definierten Zeitraum von Wochen bis Monaten oder Jahren mit einem klar festgelegten Ziel. Aufgrund der großen Bandbreite an unterschiedlichen Projekten muss eine geeignete Unterscheidung im Hinblick auf den Umfang des Projektmanagements getroffen werden. Ein typisches Unterscheidungsmerkmal ist die Projektgröße und die Komplexität. Für kleine Projekte reicht ein einfaches und häufig informelles Projektmanagement vielfach aus. Für große und komplexe Projekte mit einem großen Entwicklerteam ist ein starkes und gut organisatorisches Projektumfeld erforderlich. Da sich Projekte stark in Umfang, Komplexität und Risiko unterscheiden, sind je nach Projekttyp (siehe Abschnitt 1.1) geeignete Maßnahmen für das Management notwendig, um eine ausreichende Unterstützung der Planung und Steuerung zu ermöglichen, ohne dabei aber die inhaltliche Arbeit durch übermäßigen organisatorischen Mehraufwand zu belasten. Karnovsky unterscheidet beispielsweise drei *Projektdimensionen* in Bezug auf Größe und Komplexität von Projekten [56]. Durch eine Metapher aus der Schifffahrt lassen sich diese Projektdimensionen gut veranschaulichen: die Dimension

„MINI“ lässt sich beispielsweise mit einem kleinen und leicht steuerbaren „Fischerboot“ vergleichen. Ein *„MIDI“-Projekt* kann etwa als „Küstenwachschiff“ verstanden werden, das bereits eine höhere Komplexität aufweist aber immer noch gut steuerbar ist. Sehr große Projekte *(„MAXI“)* sind mit einem „Supertanker“ vergleichbar, die einen hohen Komplexitätsgrad beinhalten, eher schwer kontrollier und steuerbar sind und somit auch ein entsprechend umfangreiches Projektmanagement benötigen.

Risiko

Risikofaktoren, wie beispielsweise der Einsatz unbekannter Technologien, neue und unbekannte Anwendungsdomänen, hohe Komplexität von Projektinhalten oder mangelnde Erfahrung der Projektmitarbeiter müssen ebenfalls durch das Projektmanagement geeignet berücksichtigt werden. Risiken sind dabei Ereignisse, die mit abschätzbarer Wahrscheinlichkeit das Projekt bedeutend verzögern oder zusätzliche Aufwände oder Kosten verursachen können. Diese Risiken können im fachlichen, technischen oder auch im organisatorischen Bereich liegen. Um diesen Risikofaktoren erfolgreich begegnen zu können, ist ein besonders methodisches Vorgehen anzuraten. Beispielsweise ist in der komponentenorientierten Software-Entwicklung besonders die Abschätzung relevant, mit welchen Vorteilen, Aufwänden und Risiken wiederverwendbare Komponenten im Projekt eingebunden werden sollen und können. Gute und bekannte Komponenten fördern etwa eine effiziente Herstellung von hochwertigen Software-Produkten, während instabile oder unbekannte Komponenten erhebliche Mehraufwände und Qualitätsrisiken im Projekt verursachen können.

Das Entwicklungsteam und die jeweiligen Teammitglieder bilden das Rückgrad eines erfolgreichen Software-Projekts um auch möglichen Risiken geeignet begegnen zu können. Dementsprechend muss in jedem Projekt auch eine definierte Rollenverteilung mit klar verteilten Aufgaben vorhanden sein.

4.1.1 Rollen im Projekt

Eine zentrale Aufgabe des Projektmanagements nimmt die Zusammenstellung des Entwicklerteams mit den dazu gehörenden Rollen ein. Ausgewählte und typische Rollen und Aufgaben, die innerhalb eines Projekts vorhanden sein müssen und zum Einsatz kommen, werden in diesem Abschnitt kurz vorgestellt.

Auftraggeber und Auftragnehmer

Eine zentrale Rolle nimmt der *Auftraggeber* ein. Er formuliert die Anforderungen an das Projekt, benennt Ansprechpersonen und deren Kommunikationsdaten. *Auftragnehmer* sind jene Personen, die den Projektauftrag des Auftraggebers zur Durchführung annehmen, also das *Entwicklerteam* unter der Leitung des *Projektleiters*. Der *Projektleiter* oder *Teamkoordinator* hat die Projektverantwortung und ist für die Projektplanung, das Risikomanagement, die Projektorganisation (insbesondere die Aufgabenverteilung) und Entwicklung von Strategien bzw. Maßnahmen zur Erreichung der Zie-

Projektleiter

le zuständig. Der Projektleiter ist also interner Koordinator des gesamten Entwicklerteams und direkter Ansprechpartner für den Auftraggeber.

Entwicklerteam

Jedem Teammitglied des Entwicklerteams werden konkrete Rollen und Verantwortlichkeiten zugeteilt. Beispielsweise ist die methodische Planung und Abschätzung von Aufwänden und Terminen eine Kernaufgabe des Projektleiters. Oft wird jedoch beim Schätzen von Arbeitspaketen der Implementierungsphasen die Unterstützung der Entwickler und technischen Architekten benötigt, um gute und realistische Schätzungen durchführen zu können. Für Entwickler sind daher ebenfalls bestimmte Fähigkeiten notwendig, um selbst ein Projekt zu managen (zumindest den eigenen Teil) oder um sich mit einem Projektmanager und dessen Plänen kompetent auseinandersetzen zu können. Im Entwicklerteam sollen neben den eigentlichen Entwicklern folgende wichtige Rollen vertreten sein:

Software-Architekt

Der *Software-Architekt* hat die Verantwortung für das technische Projektergebnis und ist für die Einhaltung der Richtlinien für Architektur, Design und Implementierung zuständig. Außerdem ist er auch für Aufgaben der Qualitätssicherung bei der Umsetzung der Architektur im Rahmen der Implementierung verantwortlich. Kapitel 7 beschäftigt sich im Detail mit der Rolle des Software-Architekten und den damit verbundenen Aufgaben. Der Software-Architekt ist außerdem der erste Ansprechpartner für unterschiedliche Fragestellungen im Zusammenhang mit Architektur und Design.

Dokumentenbeauftragter

Der *Dokumentenbeauftragte* ist für qualitätssichernde Maßnahmen in allen Bereichen der Dokumentation zuständig. Er überprüft die Einhaltung der Dokumentationsrichtlinien, stellt Werkzeuge (etwa für das Dokumentenmanagement) und Vorlagen für die gesamte Dokumentation bereit und ist auch für die Aktualität, Richtigkeit und Vollständigkeit der Dokumentation zuständig. Die Hauptaufgaben des Dokumentenbeauftragten umfassen Organisation und Erstellung bzw. Einforderung benötigter Dokumente, die für dem Projektfortschritt notwendig sind.

Testbeauftragter

Der *Testbeauftragte* ist für die Einhaltung der Testpläne und die Umsetzung der Teststrategie zuständig. Im Rahmen der testgetriebenen Entwicklung werden beispielsweise die Tests durch die Entwickler bereits vor bzw. spätestens während der Umsetzung (Implementierung) erstellt. Gemeinsam mit den *Testern* überwacht der Beauftragte den Fortschritt bei der Durchführung der Tests und die Erstellung der Testberichte. Eine zentrale Aufgabe des Testbeauftragten ist auch die Konzeptionierung und Bereitstellung einer geeigneten Testumgebung (Testframework) zur Evaluierung und Verbesserung des Software-Produkts.

Weitere Rollen

Je nach Projekt und Anwendungsbereich können aber auch zahlreiche weitere Rollen notwendig sein, um entsprechende Themenbereiche abdecken zu können. Ein guter Überblick über mögliche Rollen in der Software-Entwicklung sind beispielsweise im Rational Unified Process [61] oder im V-Modell XT [43] ausführlich beschrieben.

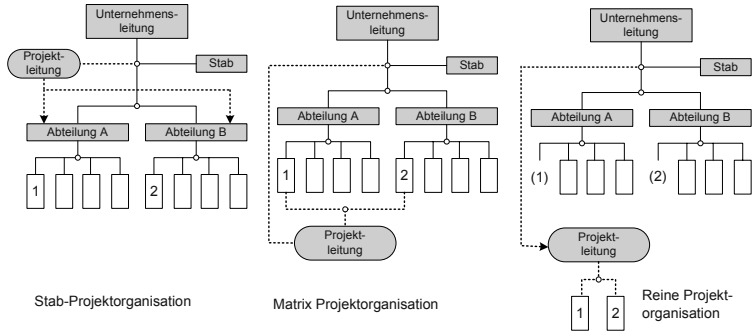


Abbildung 4.1
Projektorganisationsformen [38].

Stellvertreter

Die Auswahl des optimalen Entwicklerteams ist die Hauptaufgabe des Projektleiters. Für ein erfolgreiches Entwicklerteam ist wesentlich, dass alle Rollen, die je nach Projekt variieren können, geeignet besetzt sind. Zur Unterstützung der verantwortlichen Rolle empfiehlt sich in der Praxis auch die Festlegung von Stellvertretern, um ein optimales Projektergebnis zu ermöglichen.

4.1.2 Projektorganisation

Nicht nur die Auswahl von Teammitgliedern sondern auch Einbettung der Entwicklungsteams in die Organisation sind für den Projekterfolg von hoher Bedeutung. Komplexe eher hierarchische Teamstrukturen mit strikt getrennter Rollenverteilung, wie sie häufig in großen Organisationen zu finden sind, sind eher schwerfällig und können schnelle Entscheidungen eher behindern. In der agilen Entwicklung finden sich beispielsweise kleine und effiziente Teams, in dem jedes Teammitglied alle Aufgaben wahrnehmen kann und es einen Hauptverantwortlichen für eine bestimmte Aufgabe gibt. In der Praxis finden sich drei wesentlichen organisatorischen Grundkonzepte (siehe Abbildung 4.1), die sich hinsichtlich der Einbettung in das Unternehmen und Kompetenz des Projektleiters unterscheiden:

Formen der Projektorganisation

- Bei einer *Stab-Projektorganisation* bleiben die organisatorischen Strukturen des Unternehmens erhalten. Die Teammitglieder verbleiben in der ursprünglichen Struktur, der Projektleiter übernimmt koordinierende Aufgaben. Diese Projektorganisation ist eher problematisch, da seitens des Projektleiters keine direkte Entscheidungs- und Weisungsbefugnis (organisatorische Grenzen) vorhanden ist.
- Die *Matrix-Projektorganisation* ist eine Mischform zwischen Projektleitung und organisatorischer Unternehmensstruktur. Die Teammitglieder unterstehen der fachlichen Leitung des Projektleiters, disziplinarisch den Linienvorgesetzten.
- Eine *reine Projektorganisation* ist eine bewährte Form für die Umsetzung eines Software-Projekts. Die Teammitglieder werden aus den ur-

Tabelle 4.1 *Auswahlkriterien für Projektorganisationen [38].*

Kriterien / Projektorganisation	Stab	Matrix	Projekt
Bedeutung für das Unternehmen	gering	groß	sehr groß
Projektumfang	gering	groß	sehr groß
Unsicherheit der Zielerreichung	gering	groß	sehr groß
Technologie	Standard	kompliziert	neu
Zeitdruck	gering	mittel	hoch
Projektdauer	kurz	mittel	lang
Komplexität	gering	mittel	hoch
Bedürfnis nach zentraler Steuerung	mittel	groß	sehr groß
Mitarbeitereinsatz	nebenamtlich	Teilzeit	vollamtlich
Projektleiterpersönlichkeit	wenig relevant	qualifizierte Projektleiter	professionelle Projektleiter

sprünglichen Strukturen herausgelöst und in ein (unabhängiges) Entwicklerteam unter der Leitung des Projektleiters integriert. Nach Abschluss des Projekts werden die Teammitglieder wieder in die ursprünglichen Strukturen „zurück gegeben“ und stehen für neue Aufgaben zur Verfügung.

Je nach Projekt kann eine dieser Grundformen im konkreten Projektumfeld sinnvoll eingesetzt werden. Die Tabelle 4.1 zeigt einen Überblick über diese Organisationsstrukturen und wesentliche Kriterien, die bei der Auswahl einer geeigneten Struktur berücksichtigt werden sollten [38].

Unabhängig von der Organisationsform und Projekt werden im Rahmen des Projektmanagements unterschiedliche Phasen durchlaufen, die im folgenden Abschnitt näher betrachtet werden.

4.1.3 Phasen im Projektmanagement

Phasen

Die Schritte *Projektdefinition*, *Projektplanung*, *Projektverfolgung* und *Projektabschluss* beschreiben die groben Phasen, die – unabhängig von Organisationsform und Projekttyp – durchlaufen werden. Je nach Größe und Komplexität des Projekts finden die Tätigkeiten in diesen Phasen entweder nur informell statt oder erfordern eine einfache, erweiterte oder umfangreiche und formalisierte Methodenunterstützung.

Das Projektmanagement ist bereits am Beginn eines Projekts für die Führung und Koordination der beteiligten Personen zuständig. Nach der Festlegung der konkreten Vorgehensweise (Durchführungsstrategie bzw. Vorgehensmodell) und der Bestimmung der zu erstellenden Artefakte wird ein initialer Terminplan erstellt. Im Lauf des Projekts (Projektverfolgung und -steuerung) erfolgt eine Überprüfung, Nachjustierung und Evaluierung der Ergebnisse zu definierten Zeitpunkten (etwa Meilensteine oder Entscheidungspunkte). Dazu werden beispielsweise Methoden der Qualitätssiche-

rung (siehe Kapitel 5) eingesetzt. Eine zentrale Frage, welche Artefakte in welcher Phase der Produktentwicklung erstellt werden sollen, wird in den Kapiteln 2 und 3 beantwortet. Die beschriebenen Artefakte zeigen, welche Produkte in welcher Phase des Lebenszyklus typischerweise zu finden sind.

Vorgehensmodell

Je nach verwendetem Vorgehensmodell stehen dem Projektmanagement verschiedene Werkzeuge zur Verfügung. Speziell bei mittleren (MIDI) Projekten hat sich in der Praxis eine Mischung aus traditionellen und systematischen Prozessmodellen, wie dem Rational Unified Process (siehe Abschnitt 3.7) und agilen Ansätzen, wie etwa SCRUM (siehe Abschnitt 3.8) bewährt. Die Rollen, Artefakte und Planungswerkzeuge die etwa durch den Rational Unified Process definiert sind, erweisen sich insbesondere zu Beginn eines Projekts als sehr hilfreich. Während in den Implementierungsphasen eine agile Vorgehensweise eine flexiblere Kontrolle und Steuerung des Projekts ermöglicht und den Overhead des Projektmanagements für alle Teammitglieder verringert.

Weitere Informationen zum Software-Projektmanagement sind beispielsweise in Hindel *et al.* [39] zu finden. Speziell im Hinblick auf die agile Software-Entwicklung empfehlen sich Timeboxing-Verfahren als Ansatz für agiles Projektmanagement [95].

4.2 Projektdefinition

Projektidee

Die Projektdefinition umfasst den Abschnitt des Projektmanagements von der *Projektidee* bis zum *Projektstart (Kick-off)*. Startpunkt eines Software-Projekts ist typischerweise der reale Bedarf eines Auftraggebers zur Lösung eines aktuellen Problems durch ein Software-Produkt, das im Rahmen eines Projekts umgesetzt werden soll. Dieser konkrete Bedarf wird im Dokument *Projektvorschlag* festgehalten. Die Projektidee sollte im Projektvorschlag ausreichend konkret beschrieben sein, um die Bewertung der Idee auf Umsetzbarkeit sowie eine ungefähre Abschätzung der Größenordnung des zu erwartenden Nutzens und Aufwands zu erlauben. Zudem sollen daraus in weiterer Folge bereits detaillierte Anforderungsbeschreibungen abgeleitet werden können. Bei umfangreichen Projektvorschlägen kann auch eine Priorisierung der beschriebenen Kernfunktionalität *(Features)* sinnvoll sein, um das Projekt besser einschätzen zu können (siehe dazu auch Priorisierung von Anforderungen in Abschnitt 2.3.2). Nach der Entscheidung zur Umsetzung des Projekts wird der Projektvorschlag zum Projektauftrag erweitert.

4.2.1 Projektvorschlag

Projektvorschlag

Der Projektvorschlag ist im Gegensatz zum späteren Projektauftrag sehr kurz und umreißt wesentliche Anforderungen und Eigenschaften des ge-

planten Software-Produkts, um den konkreten Bedarf in der jeweiligen Domäne zu zeigen. Der Projektvorschlag beinhaltet typischerweise folgende wesentliche Punkte, die in weiterer Folge kurz erläutert werden: (a) Projektbezeichnung und Entwicklerteam, (b) die Ausgangssituation, (c) eine kurze Projektbeschreibung, (d) definierte Zielgruppen für den Einsatz der Lösung, (e) eine grobe Liste von Anforderungen und (f) ein Domänenmodell.

Projektbezeichnung und Entwicklerteam

Das Dokument Projektvorschlag definiert zunächst den *Projektnamen* bzw. die Projektbezeichnung und einen vorläufigen Vorschlag für das *Entwicklerteam* mit einer initialen Rollenverteilung des Entwicklungsteams. Die Projektbezeichnung soll kurz, treffend und geeignet zur Beschreibung der Projektidee als Vorschlag für das Projektmarketing sein. Die Definition des Entwicklerteams soll den groben Bedarf an den benötigten Rollen aufzeigen.

Ausgangssituation

In einem zweiten Schritt wird die aktuelle *Ausgangssituation und das Projektumfeld* kurz skizziert, um (a) den Bedarf an einem neuen Software-Produkt zu illustrieren und (b) die strategische Positionierung des Produkts zu zeigen. Wesentlich dabei ist, dass die konkreten Beiträge des Produkts sowie der zu erwartende Nutzen und Mehrwert für den Anwender ersichtlich sind. Die Ausgangssituation kann daher auch Stärken und Schwächen bestehender alternativen Software-Produkte beinhalten, die den Bedarf an einem neuen Produkt unterstreichen. Aus dieser ersten Analyse ergeben sich beispielsweise grobe Anforderungen (siehe Klassifikation von Anforderungen in Abschnitt 2.3.1) sowie Qualitätsziele.

Projektbeschreibung

Eine Projektbeschreibung dient zur Formulierung der konkreten Projektidee, die aus dem aktuellen Bedarf abgeleitet wird. In den meisten Fällen ist die Projektidee in Prosatext verfasst und beschreibt, was konkret realisiert werden soll. In der Projektbeschreibung stehen die *Hauptergebnisse* sowie der zu erwartende Nutzen für den Anwender (und andere zu definierende Zielgruppen) im Vordergrund. Als Auswahl für die Hauptergebnisse können Kernfunktionen, die für den Kunden kaufentscheidend oder erfolgskritisch sind, herangezogen werden. Beispielsweise sind das die Top-3-Features aus der Liste von Anforderungen, die den größten Nutzen für den Anwender erwarten lassen. Gute Projektbeschreibungen sowie Ausgangssituationen sind kurz und bündig. Für viele kleine bis mittelgroße Software-Produkte sollte es möglich sein, die Projektbeschreibung in sechs bis 15 Sätzen zusammenzufassen. Natürlich ist die Projektbeschreibung an dieser Stelle nur eine grobe Skizze, die im weiteren Projektverlauf und mit steigendem Verständnis für die Anwendungdsdomäne wächst.

Typ des Projekts

Aus der Projektbeschreibung und Analyse der Ausgangssituation ergibt sich der Typ des Software-Projekts, etwa ob eine Neuentwicklung, eine Anpassung oder Erweiterung einer bestehenden Lösung (Konfiguration eines zugekauften Produkts), ein Migrationsprojekt oder ein Wartungsprojekt (Anpassung einer existierenden Lösung) sinnvoll ist. Der Typ des Projekts ergibt sich typischerweise aus den Kernzielen.

Zielgruppe

Das Software-Produkt soll einen konkreten Bedarf abdecken oder ein konkretes Problem lösen. Dementsprechend gibt es unterschiedliche *Zielgruppen (Stakeholder)*, die definierte *Ziele* verfolgen. Die Definition der Stakeholder und der für sie zu erwartende Nutzen bzw. die Ziele sollen im Projektvorschlag ebenfalls definiert sein. Ziele können etwa betriebswirtschaftliche Ziele (Effizienzgewinn), (b) funktionale Ziele (Hauptfunktionen des neuen Produkts) oder (c) soziale Ziele (Erleichterungen im Arbeitsalltag) umfassen. Für jede Zielgruppe soll der Hauptnutzen hervorgehoben und kurz beschrieben werden, etwa, warum ein bestimmtes Feature für eine definierte Zielgruppe besonders wertvoll ist. Ein wesentlicher Punkt ist auch die Festlegung der *„Nicht-Ziele“*, also was durch das Software-System nicht umgesetzt werden soll. Ziele und Nicht-Ziele sollte man sich im gesamten Projekt immer vor Augen halten, da speziell die Erfüllung der Ziele erheblich zur Akzeptanz des Produkts beiträgt.

Ziele und Nicht-Ziele

Anforderungen und Features

Im Projektvorschlag liegen die *Hauptanforderungen* zumeist als grobe Features auf einer bis wenigen Seiten vor, die die Kernanforderungen im Hinblick auf deren zu erwartenden Nutzen des Projektergebnisses darstellen. Gute Anforderungen legen die Grundlage für ein solides Projekt, da damit die gewünschten Ziele und Ergebnisse klar an das Entwicklerteam kommuniziert werden können. In der Software-Entwicklung hat sich die Darstellung von Projektzielen anhand von *Features* bewährt. Diese Features beschreiben Fähigkeiten und Charakteristika eines Produkts oder einer Komponente zur Erfüllung bestimmter Funktionen. Wesentlich dabei ist, dass die Features in ausreichendem Detailgrad vorliegen, um eine grobe Schätzung des Aufwandes und der Projektgröße zu ermöglichen. Die Features des Projektvorschlags können auch bereits konkrete UML-Anwendungsfälle *(Use Cases)* beinhalten, wobei an dieser Stelle primär die *„User Goals“*, also die Ziele des Anwenders betrachtet werden müssen. Details zu den UML-Anwendungsfällen werden im Kapitel 6 beschrieben.

Aufwand und Priorität

Bei der Erstellung einer ersten Feature-Liste im Rahmen des Projektauftrags ist aber auch eine einfache Liste mit Features ausreichend. Die Tabelle 4.2 zeigt exemplarisch eine begonnene *Feature-Liste*. Jedem Feature wird eine eindeutige Nummer zugeordnet. Weiteres ist eine kurze Bezeichnung sowie eine Beschreibung zur Identifikation des Features erforderlich. Wesentlich dabei ist auch eine erste Priorisierung im Hinblick auf den Kundennutzen sowie eine initiale Aufwandsschätzung. Für die Priorisierung kann beispielsweise eine Klassifikation in *„hoch (H)“*, *„mittel (M)“* und *„niedrig (N)“* verwendet werden. Alternativ dazu ist auch eine Unterscheidung in *„must-be“*, *„expected“* und *„nice-to-have“* sinnvoll (siehe dazu Abschnitt 2.3.2). Eine erste grobe Aufwandsschätzung kann beispielsweise mithilfe einer Skalierung von 1 bis 10 erfolgen, wobei 1 einen geringen Aufwand und 10 einen sehr hohen Umsetzungsaufwand bedeutet. Beispielsweise erfordert ein Feature mit dem Aufwand 4 einen etwa doppelt so hohen Aufwand wie ein Feature mit dem Aufwand 2. Diese Informationen dienen als erste Einschätzungen, die im Rahmen der Planung konkretisiert werden müssen (siehe dazu Abschnitt 4.3).

Tabelle 4.2 *Feature-Liste mit Prioritäten und initialer Aufwandsschätzung.*

#	Feature	Beschreibung	Kunden-Priorität	Auf-wand
1	Registrierung der Studenten	Anonyme User können sich registrieren, Bestätigung mit gültiger E-Mail-Adresse (Account anlegen). Accountdaten ändern.	H	8
2	Administrator Login/Verwaltung	Es gibt einen speziellen Benutzer im System, der alle Rechte hat, d. h. vollen Zugriff auf den Datenbestand der Anwendung. Vorerst Termine und LVA's erstellen/ändern/wiederverwenden und in weiterer Folge falsche Eingaben korrigieren und Passwörter neu setzen.	H	8
3	Terminverwaltung	Studenten können sich zu einem Abgabetermin anmelden, dabei werden Anmeldebeschränkungen kontrolliert. Abmeldung vom Abgabetermin ist bis zu einem vorher bestimmten Zeitpunkt möglich. Die LVA-Leitung kann Abgabetermine anlegen und verwalten.	M	3
4	Bewertung durch Tutoren	Am Ende der Einzelphase werden Abgabegespräche durchgeführt, dabei werden die erzielten Punkte und andere, für die Gruppenphase relevante Informationen eingetragen und bestätigt. Es wird festgehalten, welcher Tutor das Abgabegespräch durchgeführt hat, und ob der Student an der Gruppenphase teilnehmen darf.	M	6
5	Export, Administratoren	Am Ende der Einzelphase werden die Daten der Studenten mit ihren Leistungen, Beschreibungen, Gruppenzuteilungen und Präferenzen exportiert und in anderen Werkzeugen weiterverarbeitet.	N	2

Domänenmodell

Wesentlicher Bestandteil des Projektvorschlags ist ein Domänenmodell. Das Modell zeigt die wichtigsten Entitäten der Anwendungsdomäne und ihre Attribute (auch „reale" Entitäten), die mit dem Software-System in Verbindung stehen und beinhaltet bereits grundlegende Ideen zur Umsetzung aus der Sicht der Architektur. Das Domänenmodell basiert im Wesentlichen auf der konzeptionellen Sichtweise des UML-Klassendiagramms (siehe Kapitel 6) und definiert, welche Teile der Domäne im Rahmen der Entwicklung des Produkts abgedeckt werden müssen.

Liegt der Projektvorschlag vor, muss entschieden werden, ob das Projekt auch tatsächlich umgesetzt werden soll. Dazu wird in der Regel eine Kick-off-Besprechung durchgeführt und anhand des Projektvorschlags über die weitere Vorgehensweise entschieden.

4.2.2 Projektentscheidung (Kick-off)

Projektentscheidung

Ziel dieser Kick-off-Besprechung ist etwa die Analyse der Projektidee im Hinblick Umsetzbarkeit, Nutzen, und Vollständigkeit und die Entscheidung ob aus dem Projektvorschlag auch ein Projektauftrag oder ein konkretes Angebot erstellt und in weiterer Folge ein Projekt gestartet werden kann.

Eine positive Entscheidung des Entwicklerteams hat also zur Folge, dass aus dem Vorschlag ein konkreter und detaillierter Projektauftrag erstellt werden kann, der in vielen Fällen auch ein Vertragsbestandteil zwischen einem Software-Hersteller und dem Auftraggeber ist. Ist der Projektvorschlag zu ungenau oder zu wenig greifbar können etwa Nachbesserungen erforderlich sein. Kommt das Team zur Entscheidung, dass das Projekt nicht umsetzbar ist oder sich eine Umsetzung als nicht rentabel erweist, kann das Projekt zu diesem Zeitpunkt gestoppt oder erst gar nicht gestartet werden *(No-Go Entscheidung)*. Die Entscheidungsfindung und die getroffene Entscheidung muss protokolliert werden, um die Argumentation nachvollziehen zu können. Wesentlich bei dieser Besprechung ist es auch, dass alle relevanten und zu diesem Zeitpunkt bekannten Teammitglieder teilnehmen. Um die Entscheidungsfindung zu unterstützen und keine wesentlichen Aspekte unberücksichtigt zu lassen, ist auf der begleitenden „Best-Practice Software-Engineering“ Projekt-Webseite[1] eine entsprechende Checkliste verfügbar.

4.2.3 Projektauftrag

Die Freigabe des Projektvorschlags durch das Team im Rahmen der Kick-off-Besprechung ist die Grundlage für die Erstellung des Projektauftrags.

> Der Projektauftrag ist eine schriftliche (Ziel)vereinbarung zwischen Auftraggeber und Auftragnehmer, ein Projekt zu bestimmten Bedingungen durchzuführen.
> *Grundlagen des Projektmanagements, Karnovsky [56]*

Projektvereinbarung

Der *Projektauftrag* (PA) beinhaltet also die *Vereinbarungen*, die mit dem Auftraggeber getroffen werden. Insbesondere soll der Projektauftrag das Projekt ausreichend abgrenzen, um eine realistische Termin- und Aufwandsplanung zu ermöglichen und ist in weiterer Folge Grundlage für die Projektplanung.

Vom Projektvorschlag zum Projektauftrag

Ergänzend zum Projektvorschlag soll ein vollständiger Projektauftrag folgende Punkte beinhalten, die meist aus dem Projektauftrag übernommen, konkretisiert und ergänzt werden: (a) Projektbezeichnung und Entwicklerteam, (b) Ausgangssituation, (c) Projektbeschreibung, (d) Zielgruppen, (e) Domänenmodell, (f) Komponentendiagramm, (g) Abgrenzungen, (h) funktionalen Anforderungen, (i) nicht funktionalen Anforderungen, (j) Lieferumfang und Abnahme, (k) Arbeitsstruktur
und Rollenverteilung, (l) Projektplan, (m) Informationswesen und (n) Be-

[1] `http://best-practice-software-engineering.ifs.tuwien.ac.at`

sonderheiten. Der Projektauftrag ist in der Regel ein Vertragsbestandteil zwischen Auftraggeber und Auftragnehmer, daher soll er so detailliert wie möglich ausgeführt werden. Es empfiehlt sich auch, den Inhalt mit geeigneten Reviews (siehe Abschnitt 5.3) zu überprüfen, da sich Mängel unangenehm auf den Projektverlauf auswirken können. Beispielsweise ist für unklare oder fehlende Anforderungen keine sinnvolle Aufwandsabschätzung möglich und somit zu massiven Projektverzögerungen führen.

Wie erwähnt, dient der freigegebene Projektvorschlag als Basis für den Projektauftrag. Dementsprechend können die Informationen auch geeignet übernommen werden. Da die Inhalte der übereinstimmenden Themen bereits beim Projektvorschlag beschrieben wurden, konzentrieren sich die folgenden Informationen auf die Besonderheiten, die für den Projektauftrag relevant sind:

Rollen

Im Abschnitt *Projektbezeichnung und Entwicklerteam* wird wie im Projektvorschlag festgehalten, welche Personen mit welchen Rollen und Zuständigkeitsbereichen am Projekt beteiligt sind. Neben der Konkretisierung der *Rollen der Auftragnehmer* ist es für den Projektauftrag besonders wichtig, auch die *Rollen aus der Auftraggebersicht* zu berücksichtigen, um geeignete Ansprechpartner für eine effiziente Zusammenarbeit zu kennen. Daraus können in weiterer Folge die verfügbaren Ressourcen im Rahmen der Planung und Aufwandsschätzung besser abgeschätzt werden. Die Rollen im Projektauftrag beziehen sich auf die bereits im Projektvorschlag definierten Rollen. Jedem Teammitglied soll zumindest eine Rolle zugewiesen werden. Damit ist eine Ansprechperson für den entsprechenden Aufgabenbereiche definiert. Wichtig ist auch, für jede Rolle geeignete Vertretungen festzulegen. In einem Projekt sollte es zumindest einen (a) Projektleiter, (b) Software-Architekten, (c) Tester, und (d) Dokumentbeauftragten als strategische Rollen geben. Diese strategischen Rollen nehmen auch Aufgaben der Qualitätssicherung (etwa Durchführung von Reviews und Dokumentation) als auch Aufgaben des Projektmanagements (etwa Aufwandsschätzungen und Aufwandsaufzeichnungen) wahr.

Verantwortlichkeit

Diese strategischen Rollen definieren Verantwortungen im Kontext des Projektverlaufs, also *„horizontal“* über den Projektverlauf hinweg, aber keinesfalls explizite Zuständigkeiten. Explizite Zuständigkeiten werden durch *„vertikale“* Arbeitspakete definiert (siehe Abschnitt 4.3.3).

Iceberg-Liste

Die Feature-Liste aus dem Projektvorschlag wird im Projektauftrag zur *„Iceberg-Liste“* [19] und enthält – als Planungsinstrument – Informationen für die konkrete Umsetzung von Anforderungen. Im Rahmen einer initialen Projektplanung gilt es, die Anforderungen entsprechenden *„Releases“* zuzuordnen, also konkrete Versionen des Endprodukts für den Kunden festzulegen. Diese Releases werden auch im Lieferumfang, der ebenfalls Bestandteil des Projektauftrags ist, definiert. Details zur Iceberg-Liste werden im Rahmen der Projektplanung (siehe Abschnitt 4.5) beschrieben. Im Projektauftrag ist es notwendig, die funktionalen und nicht funktionalen Anforderungen im Detail zu beschreiben, sodass eine konkretere Pla-

Nicht-Ziele

nung möglich ist. Neben der Definition der konkreten Projektziele ist auch die Festlegung der „Nicht-Ziele“ empfehlenswert bzw. notwendig – es ist also eine geeignete Abgrenzung des Projekts erforderlich. Diese Informationen zeigen dem Auftraggeber, was er von dem Produkt erwarten kann und was nicht. Speziell die Nicht-Ziele können für etwaige Folgeprojekte interessant sein. In jedem Fall können aber mögliche Erweiterungen bei der Festlegung der Architektur und des Design bereits berücksichtigt werden, um die Wartbarkeit im Sinn der Erweiterbarkeit zu erleichtern. Dieser Aspekt spielt auch bei Architekturreviews oder Architekturevaluierungen (siehe dazu Abschnitt 5.5) eine Rolle.

Komponentendiagramm

Aus dem Domänenmodell wird ein *Komponentendiagramm* entwickelt. Das Komponentendiagramm ist der erste Schritt der technischen Entwicklung und das bisher erste im Projektmanagement enthaltene technische Modell, das sich auf tatsächlichen Sourcecode bezieht. Komponentendiagramme werden meist in der Notation des UML-Klassendiagramms in der spezifizierenden Sichtweise gezeichnet (siehe dazu Kapitel 6). In der ersten Version werden meist nur Subsysteme und Komponenten sowie ihre Abhängigkeiten skizziert. Dadurch kann die Machbarkeit, die Zuordnung von Features zu Systemteilen sowie die Komplexität besser abgeschätzt werden. Das Komponentendiagramm wird weiter entwickelt und beinhaltet einen immer höheren Detailgrad. Ein nächster Schritt umfasst die Definition der Schnittstellen zwischen den einzelnen Komponenten. Daraus entstehen in weiterer Folge Klassendiagramme und Verteilungsdiagramme.

Wiederverwendung von Komponenten

Aus der Sicht des Projektmanagements stellt sich im Rahmen der Projektabwicklung immer die Frage, welche Software-Komponenten bereits existieren, welche angepasst werden können und welche neu erstellt werden müssen. Dazu kann das Komponentendiagramm als technischer „Ordnungsrahmen“ geeignete Antworten liefern, da die Software-Produkte stets den jeweiligen Komponenten zugeordnet werden können. Die Wiederverwendbarkeit von Komponenten und Systemteilen spielt eine große Rolle in der modernen und komponentenorientierten Software-Entwicklung, da bestehende Komponenten in der Entwicklung berücksichtigt werden können. Die Integration bestehender Komponenten oder Teilsysteme birgt aber auch Risiken, etwa der Einsatz veralteter Technologien, die auch einen erheblichen Mehraufwand verursachen können. Demnach sollte eine Wiederverwendung bestehender Komponenten bereits im Projektauftrag berücksichtigt werden.

Lieferumfang und Abnahme

Der Abschnitt *Lieferumfang und Abnahmeprozedur* des Projektauftrags beschreibt Projektergebnisse (etwa Releases und Versionen) und definiert die konkret abzuliefernden Produkte und Services. Diese Produkte umfassen beispielsweise das lauffähige Software-Produkt, die Projektquellen, geeignete Anwender- und Projektdokumentationen, Testprotokolle, oder Statusberichte. Es sollte auch definiert werden, wie die Übergabe an den Kunden und gegebenenfalls auch die Inbetriebnahme beim Kunden erfolgen soll. Sind beispielsweise formelle Abnahmeprüfungen erforderlich, ist ein Test-

betrieb vorgesehen oder ist das Projekt mit der Übergabe der Datenträger und der Dokumentation abgeschlossen. In diesem Abschnitt wird also festgelegt, was der Kunde genau bekommt.

Arbeitsstruktur

Die *Arbeitsstruktur* ist ebenfalls ein wichtiger Aspekt im Projektauftrag, da hier – als Grundlage für die Projektplanung – die konkreten Arbeitspakete und Zuständigkeiten definiert werden. Der *Projektstrukturplan* (siehe Abschnitt 4.3.2) ist eine Möglichkeit zur Festlegung der Arbeitsstruktur. In diesem Planungsinstrument werden alle Arbeitspakete durch eine Analyse der erfasst. Arbeitspakete umfassen dabei nicht nur den Sourcecode sondern auch Entwurfs- und Designdokumente sowie die gesamte interne Dokumentation. Wesentlich dabei ist, dass die überprüfbaren Ergebnisse einen klaren Bezug zum technischen Ziel im Projektauftrag herstellen können, etwa die Implementierung von konkreten technischen Komponenten, Anwendungsfällen und Features. Eine derartige Strukturierung von Arbeitspaketen ist beispielsweise in der agilen Software-Entwicklung als *„Backlog“* zu finden.

Da der Projektstrukturplan im Rahmen der Planung einen immer höheren Detailgrad aufweist, ist im Projektauftrag nur eine erste Version sinnvoll. Diese erste Version sollte die wesentlichen Projektmeilensteine und den zugeordneten Arbeitsaufwand (Kosten) beinhalten. Eine grobe Abschätzung des Aufwands für die Herstellung der Ergebnisse und Erreichung der Ecktermine des Projekts erfolgt typischerweise mithilfe von Schätzverfahren, wie sie im Abschnitt 4.3.2 beschrieben werden.

Projektplan

Der *Projektplan* dient als *„Roadmap“* für die Projektdurchführung, die zwischen allen Projektpartner abgestimmt sein muss. Diese Roadmap basiert auf der Feature-Liste bzw. der Iceberg-Liste und der initialen Version des Projektstrukturplans. Der Projektplan wird im Rahmen der Planungsphase konkretisiert *(Projektplanung)* und im Projektverlauf laufend überprüft und gegebenenfalls angepasst *(Projektverfolgung)*.

Informationswesen

Die Zusammenarbeit eines Entwicklungsteams sowie die Kommunikation zwischen den Projektpartnern ist ein erfolgskritischer Faktor in der Software-Entwicklung. Daher ist im Projektauftrag auch die Art der Kommunikation sowie die Frequenz der minimal erforderlichen Projektbesprechungen (etwa bei Meilensteinen) festzulegen. Je nach Bedarf sind weitere Abstimmungen erforderlich, die jedoch dynamisch festgelegt werden und nicht von vornhinein definiert werden können. In diesem Abschnitt sollte auch definiert werden, welches konkrete Vorgehensmodell gewählt wird, wie die informelle Projektkommunikation stattfindet und welche Werkzeuge dafür eingesetzt werden. Nicht nur die Kommunikation sondern auch eine definierte und gemeinsame Datenablage ist für den Projekterfolg notwendig. Dazu ist es notwendig, einen zentralen Verwahrungsort bzw. eine Sammelstelle für alle Projektunterlagen einzurichten. Das wird meist über Repositories, die im Rahmen der Projektinfrastruktur festgelegt werden, realisiert (siehe dazu auch Kapitel 10). Diese Infrastruktur muss dann natürlich ent-

sprechend bereit gestellt und verfügbar sein. Diese Aspekte unter *„Kommunikationsinfrastruktur“* oder *„Kommunikationswesen“* zusammengefasst.

Besonderheiten

Im Abschnitt Besonderheiten können sonstige Anmerkungen zu Rahmenbedingungen, wie etwa benötigte Räumlichkeiten, besondere Hardware und Software, zusätzliche Kompetenzen (etwa externe Berater bei erforderlichen Spezialkenntnissen) und juristische Aspekte festgelegt werden.

Eine Beschreibung sowie ein Beispiel eines vollständigen Projektauftrags sind auf der „Best-Practice Software-Engineering“ Projekt-Webseite[2] zu finden.

Für alle beteiligten Projektpartner steht also nach Fertigstellung des Projektauftrags fest, welches Produkt erstellt wird, welche Anforderungen umgesetzt werden sollen, wie eine grundlegende Architektur aussehen kann, welche Aufwände mit der Herstellung verbunden sind, welche Projektmeilensteine anzustreben sind und wie die Rahmenbedingungen aussehen. In der Praxis hat sich ein gründliches Review des Projektauftrags bewährt, um die Richtigkeit und Vollständigkeit des Projektauftrags zu gewährleisten. Der Projektauftrag ist in vielen Fällen, speziell wenn externe Auftraggeber involviert sind, integraler Vertragsbestandteil. Das bedeutet, der Projekterfolg ist an die Erfüllung der definierten Aspekte gekoppelt.

Nach der Freigabe des Projektauftrags durch die Projektpartner findet in der Regel ein offizielles Kick-off für die Umsetzung des Projekts statt. Im nächsten Schritt wird – ausgehend vom Projektauftrag – eine detaillierte Planung für das Projekt und in weiterer Folge für die Umsetzung erstellt.

4.3 Projektplanung

Kick-off zur Umsetzung

Nach der Freigabe des Projektauftrags und dem offiziellen Start (Kick-off) zur Umsetzung des Software-Produkts im Rahmen eines Projekts muss der gesamte Projektverlauf unter Berücksichtigung möglicher Risiken initial geplant werden. Im Projektverlauf erfolgt eine laufende Überprüfung des jeweiligen Status des Projekts und gegebenenfalls eine Anpassung des initialen Projektplans (Projektverfolgung). Um eine initiale Planung durchführen zu können, sind wichtige Aspekte zur technischen und wirtschaftlichen Planung zu berücksichtigen, die in diesem Abschnitt beschrieben werden. Weiterhin werden Methoden zur Aufwandsschätzung sowie ausgewählte Methoden zur Unterstützung der operativen Planung vorgestellt.

Aufwände, Termine und Risiken

Primär geht es beim Projektplan um die Abschätzung der Aufwände, Termine und Risiken in Bezug auf die definierten Anforderungen, Features

[2] `http://best-practice-software-engineering.ifs.tuwien.ac.at`

und die dazugehörigen Aktivitäten und Ressourcen. Wichtig dabei ist, dass eventuell auftretenden Risiken vermieden oder geeignet adressiert werden können. In der Praxis sind beispielsweise Überschreitungen des Budgetrahmens von Projekten zur Entwicklung von Software eher die Regel als die Ausnahme. Typische Ursachen dafür sind beispielsweise das Übersehen von Risiken oder die Unterlassung der frühzeitigen Klärung von Unsicherheiten über Ziele und den Verlauf des Projekts. Oft bleiben im Projektplan auch viele kleine „Routinearbeiten" unberücksichtigt, die den Zeit- und Aufwandsrahmen beim Auftreten von unerwarteten Komplikationen sprengen. Eine Risikoanalyse bzw. -abschätzung sollte demnach immer ein Bestandteil der Projektplanung sein, die laufend überprüft und aktualisiert wird. Es empfiehlt sich auch, diese Risikoabschätzung bereits im Projektauftrag zu berücksichtigen.

Ressourcen

Die Projektplanung teilt im Projektplan die vorhandenen Ressourcen an Personal, Zeit, Maschinen und Geld ein, um die Projektziele wirtschaftlich zu erreichen. Ein Projekt besteht dabei aus vielen Einzelaktivitäten, die teilweise parallel bearbeitet werden können aber teilweise auch in bestimmten Reihenfolgen durchgeführt werden müssen, da sie entweder aufeinander aufbauen oder von denselben Personen oder Maschinen bearbeitet werden. Es ist die Aufgabe des Projektmanagements, diese Aktivitäten zu identifizieren und in der Planung geeignet zu berücksichtigen. Aufgrund der schnell steigenden Komplexität der Planung empfiehlt sich der Einsatz von Werkzeugen, wie sie etwa ab Abschnitt 4.3.5 und in Kapitel 10 beschrieben werden.

Variablen und Varianten

Der Projektauftrag ist die Basis für den Projektplan und orientiert sich primär an *Projektvariablen*, wie *Projektzielen* (etwa eine priorisierte Anforderungsliste), verfügbaren *Ressourcen* (Teammitglieder und technische Hilfsmittel) und *Projektdauer*. Je nach Projektausprägung folgt die Projektplanung typischerweise zwei grundlegenden Varianten. Bei *Fixpreisprojekten* wird etwa die Zielsetzung meist vertraglich festgelegt (etwa in Bezug auf den Projektauftrag), die benötigen Ressourcen und die Projektdauer muss geeignet geschätzt und geplant werden. Eine andere Variante, die sich in der Praxis gut bewährt hat, fixiert beispielsweise die Zeitplanung und Projektdauer *(Timeboxing)*, etwa in Form von zeitlich abhängigen Releases und Versionen. Dabei werden die Ressourcen und die Projektdauer definiert – bei der Planung werden die erreichbaren Projektziele geschätzt, die unter den definierten Rahmenbedingungen umsetzbar sind. Dieser Ansatz findet sich typischerweise bei agilen Vorgehensweisen (etwa SCRUM) wieder.

Agile Software-Entwicklung

Die agile Software-Entwicklung, wie etwa am Beispiel von SCRUM in Abschnitt 3.8 beschrieben, fokussiert auf technische Projektziele wie etwa Komponenten, funktionale Anforderungen oder Anwendungsfälle. Jedes Ergebnis in der agilen Entwicklung ist also ein lauffähiger und getesteter Sourcecode oder ein auslieferbares Arbeitspaket, wie etwa die Benutzerdokumentation. Wie bereits im Abschnitt 3.8 beschrieben, ist SCRUM ein Vorgehensmodell zur Abwicklung eines agilen Software-Projekts aus

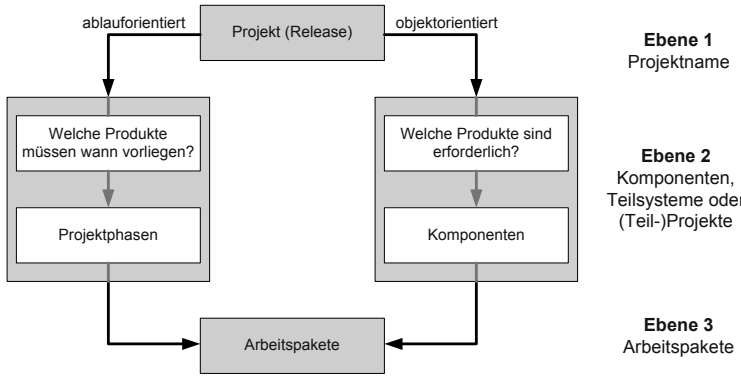


Abbildung 4.2
Projektgliederung mit Strukturplänen [56].

der Sicht des Projektmanagements. Agile Vorgehensmodelle, wie SCRUM, sind zwar Gegenbewegungen zu schwergewichtigeren Vorgehensmodellen wie RUP oder V-Model, können jedoch auch kombiniert werden.

RUP mit SCRUM

Für kleine bis mittelgroße Projekte hat sich etwa eine Kombination zwischen einem schwergewichtigen Prozess wie dem Rational Unified Process (RUP) und eines agilen Prozesses wie SCRUM bewährt. Beispielsweise kann RUP den groben Rahmen im Sinn eines Fixpreisprojekts vorgeben. Die einzelnen Iterationen können aber durchaus mit dem Timeboxing-Verfahren (also eher agil) ablaufen und so eine höhere Flexibilität im Entwicklungsprozess ermöglichen. Die Tabelle 4.3 zeigt beispielsweise die Phasen eines Projektplans der an den Rational Unified Process angelehnt ist. Alle Arbeitspakete einer Iteration werden agil abgearbeitet und gesteuert. Tabelle 4.5 illustriert beispielsweise eine begonnene Iceberg-Liste einer Iteration; auch die Fortschrittskontrolle kann über agile Methoden, wie etwa ein Burn-down-Chart erfolgen (siehe Abschnitt 4.3.6).

4.3.1 Strukturierung eines Projekts

Projektstrukturplan

Ausgehend vom Projektauftrag ist eine geeignete Projekt- und Produktstruktur die Grundlage für eine realistische Planung. Ziel ist es, das System in einer neutralen und flexiblen Struktur aufzubauen, um dadurch die eigentliche Planung zu unterstützen. Ein *Projektstrukturplan (Work Breakdown Structure, WBS)* besteht im Wesentlichen aus drei Ebenen: (a) dem *Projektnamen*, (b) einer Ebene mit *Komponenten, Teilsystemen* oder *Teilprojekten* und (c) den konkreten *Arbeitspaketen*, die entweder auf konkrete Komponenten oder Phasen und Aufgaben abgebildet werden. Ja nach Betrachtungsweise, d.h. Gesamtsystem, Teilsystem oder Komponente, können auch zusätzliche Ebenen nach Bedarf verwendet werden.

In der Praxis sind zwei grundlegende Typen von Strukturplänen zu finden [56]:

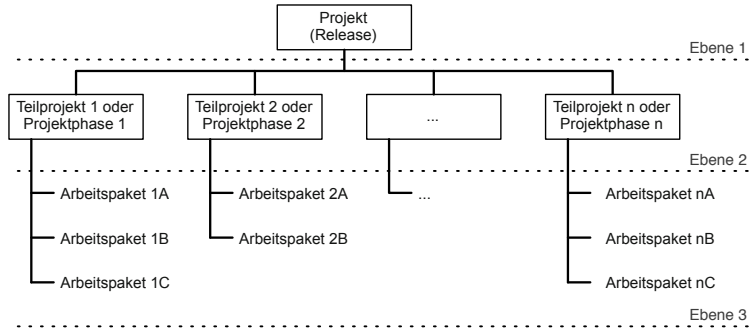


Abbildung 4.3
Objektorientierter Projektstrukturplan.

Grundlegende Projektstrukturen

> Ein *objektorientierter* Projektstrukturplan gliedert das gesamte Projekt in einzelne Teilprojekte und in weiterer Folge in Arbeitspakete, die dem (Teil-)Projekt und den jeweiligen Komponenten funktionell zugeordnet sind. Bei diesem Strukturplan stehen die jeweiligen Funktionen der zu erstellenden Komponenten und Systeme auf allen Ebenen im Vordergrund und ermöglichen eine Planung in Bezug auf diese Produkte.

> Im *ablauforientierten* Projektstrukturplan orientiert sich die Gliederung der Arbeitspakete am Entwicklungsprozess und somit an den jeweiligen Aufgaben im Rahmen der Entwicklung. Im Vordergrund stehen dabei die Arbeitspakete, die einer konkreten Aufgabe zugeordnet sind. Eine Planung ist somit auf Aufgabenebene möglich.

Weitere in der Literatur beschriebenen Projektstrukturen lassen sich im Wesentlichen auf diese beiden Grundformen zurückführen. In der Praxis der komponentenorientierten Software-Entwicklung ist es naheliegend, den objektorientierten Projektstrukturplan, wie in Abbildung 4.3 schematisch dargestellt, für die Planung einzusetzen.

Aufbau eines PSP

Der Projektstrukturplan (PSP) stellt die Arbeitspakete, für die Zeit und Ressourcen benötigt werden, in einem Baum bzw. in einer hierarchischen Struktur dar. Damit erlaubt der PSP einerseits die Betrachtung auf grobem Niveau – etwa im Hinblick auf Projektphasen (oder Teilprojekte) für die Darstellung von Projektmeilensteinen – oder auf detailliertem Niveau für die Zuordnung konkreter Aufwände zu den jeweiligen Arbeitspaketen im Projektverlauf. Zudem enthält der PSP die Datenbasis für die Aufgabenverteilung und Aufwandsschätzung für eine spätere Terminplanung.

Arbeitspaket

Die Struktur des PSPs orientiert sich also an den konkreten Projekterfordernissen und folgt somit auch den wesentlichen Endprodukten und der gewählten Vorgehensmethodik, etwa einem Vorgehensmodell. Beispielsweise setzt sich ein Software-Entwicklungsprojekt aus mehreren Teilprojekten zusammen, die wiederum in einzelne Arbeitspakete aufgeteilt und den Teilprojekten zugeordnet werden. Wichtig ist es auch „Teilprojekte“, wie Integrationstest oder Systemtest geeignet in der Planung zu berücksichtigen. Startpunkt der technischen Planung ist also das Ermitteln der

konkreten Arbeitspakete, die für die Erreichung der im Projektauftrag definierten Ziele erforderlich sind. Ausgehend von diesen Arbeitspaketen müssen die jeweiligen Komplexitäten und Aufwände ermittelt werden, um eine realistische Planung bezüglich Zeit und Ressourcen zu ermöglichen. Ein Arbeitspaket ist dabei die kleinste plan- und verfolgbare Einheit und sollte zur besseren Kontrolle des Projektfortschritts innerhalb von etwa vier Wochen abgeschlossen werden können und einen Aufwand von 12 Personenwochen nicht überschreiten.

Qualitätsplanung

Neben dem Projektmanagement begleitet auch das Qualitätsmanagement das Projekt während des gesamten Lebenszyklusses eines Software-Projekts (siehe Kapitel 2). Dementsprechend ist es auch erforderlich, Maßnahmen der Qualitätssicherung geeignet in der Projektplanung und der Ermittlung der Arbeitspakete zu berücksichtigen. Im Rahmen der Qualitätsplanung werden die definierten (technischen) Arbeitspakete um Aufgaben zur Überprüfung der Qualitätsanforderungen (die etwa aus dem Projektauftrag abgleitet werden) ergänzt. Dazu gehören beispielsweise Testpläne und Testdurchführungen, Reviews oder Benchmarks für Zwischen- und Endprodukte. Eingeplante Maßnahmen der Qualitätssicherung helfen auch, den Projektverlauf basierend auf den erstellten Produkten zu steuern (Projektverfolgung). Entsprechen erstellte Produkte etwa nicht dem geforderten Qualitätsniveau, können weitere Iterationen oder ergänzende Maßnamen der Qualitätssicherung eingeleitet werden. Die Qualitätssicherung verfolgt dabei die zwei grundlegenden Kontrollziele, Verifikation und Validierung zur Überprüfung der Produkte im Hinblick auf die Spezifikation und die Kundenanforderungen (siehe dazu auch Abschnitt 5.2).

Wesentlich dabei ist eine *ausgewogene und sinnvolle* Auswahl an Maßnahmen zur Qualitätssicherung im Hinblick auf die Projektziele. Beispielsweise sollte es eine Festlegung über das Ausmaß an Software-Tests geben, etwa „wann wurde genug getestet“. Diese Festlegung kann etwa über die Definition eines erforderlichen *Überdeckungsmaßes* erfolgen (siehe dazu Abschnitt 5.6.6).

Im Rahmen der Qualitätsplanung können etwa folgende Aspekte hilfreich sein, um geeignete Maßnahmen der Qualitätssicherung auszuwählen und einzuplanen:

Einflussfaktoren

- Gewünschte *„Quantität“* und *„Qualität“* des Endprodukts, etwa durch priorisierte Anforderungen, geeignet ausgewählte Methoden zur Umsetzung und Überprüfung oder Anwendung eines definierten Vorgehensmodells.
- Die *Anzahl der beteiligten Personen* hat erheblichen Einfluss auf die Qualitätssicherung und das Projektmanagement. Beispielsweise erfordert ein großes Projektteam eine effiziente Kommunikation der Projektziele und eine gute Koordination der Änderungen und somit ein strafferes Projektmanagement.

- Der *Umfang des Produkts* hat maßgeblichen Einfluss auf das Konfigurationsmanagement der Zwischen- und Endprodukte und kann zusätzliche qualitätssichernde Maßnahmen erfordern.
- Die *Qualifikation* der Entwickler und die *Fähigkeiten* von Werkzeugen und Methoden müssen bei der Qualitätsplanung ebenfalls berücksichtigt werden.
- Projekte mit *längeren Projektzeiten* erfordern etwa auch detailliertere Aufzeichnungen, um die Nachvollziehbarkeit ermöglichen zu können.

In der Praxis empfiehlt es sich, Maßnahmen der Qualitätssicherung bereits in der initialen Planung zu berücksichtigen, indem dafür definierte Arbeitspakete im Rahmen eines Projektstrukturplans vorgesehen werden. Nachjustierungen im Rahmen der Projektverfolgung hängen dann entsprechend vom Projektverlauf ab.

Die technische Planung, inklusive Qualitätsplanung, ergibt einen PSP, der Basis für weitere Aufwandsschätzungen ist, um die initiale Einschätzung der Projektgröße zu überprüfen und eine detaillierte Planung durchzuführen.

4.3.2 Aufwandsschätzung

Aufwandsschätzung für den Projekterfolg

Eine ausreichend genaue Schätzung des Aufwands für ein Arbeitspaket, Teilprojekt oder Projekt ist ein wesentlicher Erfolgsfaktor für das Gelingen eines Projekts. Wird der Aufwand zu hoch eingestuft, steigen damit auch die veranschlagten Kosten und ein an sich sinnvolles Projekt wird möglicherweise nicht gestartet. Eine zu niedrige Schätzung hat zur Folge, dass Meilensteine unrealistisch und nicht erreichbar sind. In solchen Projekten sind meistens überarbeitete Mitarbeiter und mangelhafte Produktqualität zu finden. Ziel bei der Aufwandsschätzung ist es also, möglichst realistische und realisierbare Aufwände zu ermitteln und die Planung darauf auszurichten. Aber auch bei korrekter Schätzung bedarf es oft einer aufwendigen Argumentation gegenüber dem Auftraggeber/Kunden, um diesen bei einer hohen – aber angemessenen – Aufwandsschätzung vom Nutzen des Projekts zu überzeugen.

Ressourcen, Kosten, Termine

Die Aufwandsschätzung ist die Grundlage für die Ressourcen-, Kosten- und Terminplanung. Diese Schätzung findet während der initialen Planung statt und wird im Verlauf des Projekts aktualisiert, um die Auswirkungen aktueller Projektinformationen auf den erwarteten Aufwand zu überprüfen und gegebenenfalls geeignet reagieren zu können. Informationen aus dem PSP bzw. der erweiterten Feature-Liste *(Iceberg-Liste)* bilden die Grundlage der Aufwandsschätzung auf Arbeitspaketebene. Die geschätzten Aufwände dienen dann als Basis für die Abschätzung von Teilprojekten und das Gesamtprojekt.

Risiken und Hindernisse

Obwohl die Aufwandsschätzung ein wesentlicher Bestandteil der Projektplanung und der Auftragsabwicklung ist, genießt die Aufwandsschätzung in der Praxis nicht jene Beachtung, die ihr zukommen sollte. Die Gründe dafür sind etwa Mangel an Wissen über „gute“ Schätzmethoden, unzureichende Unterstützung seitens der Unternehmensleitung oder allgemeine Akzeptanzmängel durch eingeschränkte Ergebnisqualität der Schätzung (etwa wenn die ermittelten Aufwände deutlich von den tatsächlichen Aufwänden abweichen). Dementsprechend ist die Auswahl des richtigen Schätzverfahrens und der korrekte Einsatz im Rahmen der Projektplanung ein erfolgskritischer Faktor bei der Projektplanung.

Anforderungen an Schätzverfahren

Stellt sich die zentrale Frage, welche Anforderungen eine gute und aussagekräftige Aufwandsschätzung erfüllen muss. Gute Schätzverfahren müssen im Wesentlichen folgende Kriterien erfüllen: (a) ausreichende Schätzgenauigkeit, (b) Reproduzierbarkeit, (c) Stabilität der Schätzfunktion, (d) Anwendbarkeit in frühen Phasen des Projekts und (e) Benutzerfreundlichkeit und Transparenz zur Überprüfung des Schätzergebnisses. Wichtig ist auch eine Unterscheidung nach Sichtweise und Detailgrad (etwa ob eine grobe Projektsicht ausreichend ist oder detaillierte Sicht auf die Arbeitspakete benötigt wird) sowie der Grad der Formalisierung des Schätzverfahrens (etwa subjektive Expertenschätzung und Einsatz historischer Daten oder formale Berechnungskriterien).

Top-down und Bottom-up

Je nach Sichtweise und Detailgrad muss man entscheiden, ob eine Top-down-Strategie (beginnend vom Gesamtprojekt bis zu den Arbeitspaketen) oder eine Bottom-up-Strategie (von den einzelnen Arbeitspaketen bis zum Gesamtprojekt) verfolgt werden soll. Die Top-down-Schätzung orientiert sich eher am Gesamtprojekt und an den bisherigen Erfahrungswerten ähnlicher Projekte. Fehlt die Vergleichbarkeit in einzelnen Bereichen, muss ein höherer Detailgrad (etwa Arbeitspakete) verwendet werden. Typische Vertreter dieser Kategorie von Schätzmethoden sind etwa Expertenschätzungen oder die Analogie- bzw. Relationensmethode. Bei fehlenden historischen Daten oder hohem Unsicherheitsfaktor empfiehlt sich ein Bottom-up-Schätzansatz. Dabei wird – ausgehend vom individuellen Arbeitspaket aus der WBS oder der Iceberg-Liste – eine Aufwandsschätzung für jedes Arbeitspaket durchgeführt, die anschließend bis zum Gesamtprojekt zusammengeführt wird. Wichtig dabei ist es aber, Aufwände nicht nur für die jeweiligen konkreten technischen Arbeitspakete zu betrachten, sondern auch Gemeinkosten und -aufwände der Organisation geeignet zu berücksichtigen.

Unsicherheit

Zur Absicherung der Schätzungen, die bei Projekten mit hohem Unsicherheitsfaktor auftreten können, empfiehlt sich der direkte Vergleich der beiden Schätzstrategien, also die Gegenüberstellung von Top-down- und Bottom-up-Schätzergebnissen. Im Idealfall sollten sich die ermittelten Werte „in der Mitte“ treffen. Weichen die Ergebnisse mehr als etwa 30% voneinander ab, sollten die Schätzgrundlagen und eingesetzten Methoden überprüft und die Schätzung wiederholt werden.

Messung des Aufwands

Die Aufwände werden dabei meist in *Personentagen* oder *Personenwochen* geschätzt. Um eine gute, aussagekräftige und nachvollziehbare Schätzung abgeben zu können, sind bereits fundierte Kenntnisse über den Entwurf und die zu verwendende Architektur erforderlich. Vor allem am Projektbeginn steht die genaue Umsetzungsstrategie allerdings meist noch nicht fest. Dadurch ist eine Schätzung des Aufwands nur schwer möglich. Statt einer Schätzung in Personentagen und -wochen kann hier eine nominale Skala verwendet werden, die einen Anhaltspunkt für die Komplexität gibt. Diese sogenannten *„Story Points"* orientieren sich dann an etwa an der WBS bzw. der Iceberg-Liste und geben einen Richtwert für eine spätere genauere Schätzung an.

In der Literatur und Praxis sind zahlreiche unterschiedliche Schätzverfahren zu finden, die sich in einem definierten Projektumfeld bewährt haben. Im Folgenden werden einige gebräuchliche Verfahren kurz angeschnitten.

Expertenschätzung

Ein sehr gebräuchliches Schätzverfahren ist die *Expertenschätzung*. Die Grundidee dabei ist, die Aufwandsschätzung durch einen erfahrenen Spezialisten durchführen zu lassen. Das bedeutet, die Aufwände für die Umsetzung eines Arbeitspaketes basieren auf den Erfahrungswerten des Spezialisten und sind daher subjektiv; eine Erhöhung der Schätzgenauigkeit kann beispielsweise durch das Einholen mehrere unabhängiger Expertenmeinungen erreicht werden. Mehrere unabhängige Expertenmeinungen bilden etwa die Basis der *Delphi-Methode*, die in mehreren Iterationen versucht, aus unterschiedlichen Expertenschätzungen eine gemeinsamen Schätzung zu erreichen. Im Vergleich zu anderen – formaleren – Schätzmethoden gibt es über die konkrete Schätzmethode kaum detaillierte Vorgehensweisen. Trotzdem ist dieser Ansatz in der Praxis recht häufig anzutreffen.

Analogie

Die Erfahrungen aus vergangenen Projekten nutzt etwa auch die *Analogiemethode*. Diese Methode ist etwa mit der Expertenschätzung vergleichbar, nutzt aber historische Daten aus vergleichbaren Projekten. Das bedeutet, dass gleiche oder ähnliche Arbeitspakete (in einem vergangenen Projekt) auch im aktuellen Projekt vergleichbare Aufwände erfordern. Wesentliches Erfolgskriterium für diese Methode ist allerdings, dass die tatsächlichen Aufwände aus dem „Referenzprojekt" auch real verfügbar sind. Auch diese Methode ist in der Praxis sehr häufig anzutreffen, speziell wenn ein Software-Hersteller ähnliche Projekte umsetzt. Durch Methoden aus der Mustererkennung lässt sich diese Methode auch bis zu einem gewissen Grad unterstützen und formalisieren [25]. Bei einem höheren Formalisierungsgrad spricht man auch von der *Relationenmethode*. Weitere formale und mathematisch fundierte Schätzmethoden sind beispielsweise die Faktormethode und parametrische Schätzgleichungen, die auf einer Korrelationsanalyse von historischen Daten in Relation zum aktuellen Projektumfeld aufbauen, um so zu aussagekräftigen Schätzungen zu kommen.

Bei einer Bottom-up-Strategie liegen die geschätzten Aufwände für die konkreten Arbeitspakete vor. Die Frage ist jetzt, wie man von den Aufwänden der Arbeitspakete zum Gesamtaufwand des (Teil-)Projekts kommt.

Multiplikator

Die *Multiplikatormethode* geht von den Aufwänden der einzelnen Arbeitspakete, die konkreten Phasen zugeordnet sind, aus und ermittelt den Gesamtaufwand einer Phase oder des Projekts durch eine Multiplikation unter Berücksichtigung eines entsprechenden Overheads. Sie dient nicht nur der Aufwandsermittlung sondern kann auch für die Beurteilung des Projektfortschritts eingesetzt werden [27].

Prozentsatz

Eine Variante dieses Verfahrens ist die *Prozentsatzmethode*, bei dem vom Aufwand einer Phase auf die übrigen Phasen geschlossen wird. Dieses Verfahren kann auch während des Projektverlaufs angewandt werden, indem von den bereits abgeschlossenen Phasen auf den Gesamtaufwand geschlossen werden kann [27].

Function-Point

Eine wichtige Gruppe von Methoden zur Aufwandsschätzung nehmen – formalere – algorithmische Vorgehensweisen ein, die bereits ein hohes Detailwissen über das zu erstellende Produkt benötigen. Ein wichtiger Vertreter ist etwa die *Function-Point-Methode*, die gekapselte Funktionseinheiten eines Systems als Grundlage für die Aufwandsschätzung nutzt. Funktionseinheiten, wie Datenein- und ausgaben, Anfragen, Schnittstellen zu externen Datenquellen und interne Logik, werden gezählt und entsprechend der Komplexität und definierter Einflussfaktoren bewertet [72].

Cocomo II

Ein ebenfalls weit verbreitetes Verfahren ist auch *Cocomo II (Constructive Cost Model)*, das auf Barry Boehm zurückgeht. Cocomo II basiert auf der Schätzung der Produktgröße in *Lines of Code*. Daraus werden mit definierten Kostenfaktoren sowohl Aufwand als auch Durchlaufzeiten berechnet [14].

Die vorgestellten Schätzverfahren sollen nur einen groben Überblick zur Orientierung geben. Eine detaillierte Betrachtung dieser Schätzverfahren würde den Rahmen dieses Buches sprengen, daher wird auf die jeweiligen Literaturquellen verwiesen.

4.3.3 Technische und wirtschaftliche Planung

Nach der Festlegung der Projektstruktur (siehe Abschnitt 4.3.1) und der Ermittlung der benötigen Aufwände für das Projekt bzw. die relevanten Arbeitspakete (siehe Abschnitt 4.3.2) erfolgt die konkrete initiale Planung für die Projektdurchführung. Anhand eines Beispiels wird exemplarisch gezeigt, wie die Projektplanung schrittweise durchgeführt werden kann.

Beispiel und Rahmenbedingungen

Das zu erstellende Produkt, das in einem Projektvorschlag bzw. -auftrag definiert wurde, soll Anmeldungen für eine Veranstaltung, etwa für eine Lehrveranstaltung, verwalten können. Für die Umsetzung stehen 5-6 Personen mit zugeordneten Rollen (siehe Abschnitt 4.2) zur Verfügung. Das Projekt soll innerhalb von vier Monaten abgeschlossen werden. Der grobe Projektrahmen (Meilensteine) wird in Anlehnung an den Rational Unified Process definiert. Die einzelnen Entwicklungsschritte orientieren sich an einem inkrementellen agilen Prozess, in diesem Fall SCRUM. Das komplet-

Tabelle 4.3 *PSP auf Phasenebene für das BPSE-SAT-Projekt.*

Nr.	Arbeitspakete	Anfang	Ende	Personen-tage
MS.1	Kick-off	01.10.2009	02.10.2009	1
1.1	Anforderungs-Analyse	02.10.2009	04.11.2009	23
MS.2	*Projektfreigabe*	*15.10.2009*	*16.10.2009*	*1*
2.1	Entwurf und Design	19.10.2009	04.11.2009	13
MS.3	*Anforderungs Review*	*04.11.2009*	*5.11.2009*	*1*
1.2	Anf.-Analyse fertigstellen	05.11.2009	11.11.2009	5
2.2	Entwurf und Design fertigstellen	05.11.2009	04.12.2009	21
3.1	Implementierung Sprint 1	06.11.2009	10.12.2009	24
MS.4	*Design Review, Version 1*	*10.12.2009*	*11.12.2009*	*1*
3.2	Implementierung Sprint 2	11.12.2009	12.01.2010	22
MS.5	*Prototyp, Version 2*	*12.01.2009*	*13.01.2010*	*1*
3.3	Implementierung Sprint 3	13.01.2010	28.01.2010	11
MS.6	*Projektabnahme, Version 3*	*28.01.2010*	*29.01.2010*	*1*

te Beispiel ist auf der begleitenden „Best-Practice Software-Engineering" Projekt-Webseite[3] zu finden.

Projektauftrag

Ausgehend vom Projektauftrag und den definierten Zielen sowie des gewählten Entwicklungsvorgehens werden die jeweiligen Arbeitspakte im Rahmen eines Projektstrukturplans erstellt. Daraus resultieren konkrete Arbeitspakete, etwa als *Iceberg-Liste*, die sowohl technische Arbeitspakete als auch Maßnahmen der Qualitätssicherung beinhalten.

PSP auf Phasenebene

Meilenstein

Da das Projekt innerhalb einer definierten Zeitspanne abgeschlossen werden soll (in diesem Beispiel 4 Monate) wird der Projektstrukturplan grob in entsprechende Teilprojekte (in diesem Beispiel die Projektphasen) und Meilensteine aufgeteilt. Meilensteine sind dabei besondere „Aktivitäten", die zusammengehörende Teilbereiche – etwa Phasen oder Teilprojekte – trennen. Der Projektbeginn und das Projektende sind beispielsweise immer Meilensteine. Die einzelnen Phasen definieren in weiterer Folge zusammengehörende Arbeitspakete die für das Erreichen eines Meilensteins erforderlich sind. Die Tabelle 4.3 zeigt etwa die groben Meilensteine und Teilprojekte mit den entsprechend geplanten Anfangs- und Enddaten sowie dem erforderlichen Aufwand. Der erforderliche Aufwand ergibt sich über Methoden der Aufwandsschätzung, die im Rahmen der erweiterten Iceberg-Liste erstellt werden. Der PSP wird in weiterer Folge unter Berücksichtigung der Iceberg-Liste erweitert. Die Tabelle 4.4 zeigt den PSP dieses Projekts in einem höheren Detailgrad.

PSP auf Arbeitspaketebene

Aufwandsschätzung

Im Rahmen der Aufwandsschätzung werden die benötigten Ressourcen, die zur Umsetzung der Arbeitspakete erforderlich sind, festgelegt. Daraus ergibt sich der geschätzte Aufwand (in Personentagen, -wochen, oder -monaten) oder ein Komplexitätsgrad, der auf den geschätzten Aufwand ab-

[3] `http://best-practice-software-engineering.ifs.tuwien.ac.at`

Tabelle 4.4 *PSP auf Arbeitspaketebene für das BPSE-SAT-Projekt.*

Nr.	Arbeitspakete	Anfang	Ende	Personen-tage	Zuständiger
MS.1	Kick-off	01.10.2009	02.10.2009	1	
1.1	Anforderungs-Analyse	02.10.2009	04.11.2009	23	
1.1.1	Projektvorschlag	2.10.09	15.10.09	9	
1.1.2	Projektauftrag	16.10.09	04.11.09	13	
MS.2	*Projektfreigabe*	*15.10.2009*	*16.10.2009*	*1*	
2.1	Entwurf und Design	19.10.2009	04.11.2009	13	
2.1.1	Komponenten-Diagramm	19.10.09	24.10.09	5	AS
2.1.2	User-Interface-Skizzen	19.10.09	29.10.09	8	SB
2.1.3	SCM-Konfiguration	19.10.09	06.11.09	14	MD
2.1.4	Dokumentationsrichtlinien	22.10.09	29.10.09	5	DW
2.1.5	DB-Schema & ERD	26.10.09	03.11.09	6	EGF
2.1.6	Verteilungsdiagramm	26.10.09	28.10.09	2	TOe
2.1.7	...	...	...	...	
MS.3	*Anforderungs Review*	*04.11.2009*	*5.11.2009*	*1*	
1.2	Anf.-Analyse fertigstellen	05.11.2009	11.11.2009	5	
1.2.1	Abnahmetests	05.11.09	11.11.09	4	AS
1.2.2	Risikoanalyse	05.11.09	11.11.09	4	SB
1.2.3	...	...	...	...	
2.2	Entwurf und Design fertigstellen	05.11.2009	04.12.2009	21	
2.2.1	Framework-Evaluation	06.11.09	20.11.09	10	AS, MD
2.2.2	Testplan, Testfälle	11.11.09	25.11.09	10	DW, EGF
2.2.3	Testberichte	20.11.09	24.11.09	2	EGF
2.2.4	Klassendiagramme	23.11.09	04.12.09	9	TOe
2.2.5	...	...	...	...	
3.1	Implementierung Sprint 1	06.11.2009	10.12.2009	24	
MS.4	*Design Review, Version 1*	*10.12.2009*	*11.12.2009*	*1*	
3.2	Implementierung Sprint 2	11.12.2009	12.01.2010	22	
MS.5	*Prototyp, Version 2*	*12.01.2009*	*13.01.2010*	*1*	
3.3	Implementierung Sprint 3	13.01.2010	28.01.2010	11	
MS.6	*Projektabnahme, Version 3*	*28.01.2010*	*29.01.2010*	*1*	

bildbar ist. In diesem Beispiel wird der Komplexitätsgrad als Maß für den erforderlichen Aufwand verwendet.

Releases

Vor dem Hintergrund der Projektrahmenbedingungen (definierte Meilensteine) und dem Konzept der komponentenorientierten und inkrementellen Entwicklung ist es erforderlich, die jeweiligen Features einer konkreten Version (Release) zuzuordnen. Es müssen also diejenigen Features ausgewählt werden, die bis zu diesem Zeitpunkt realistisch umgesetzt werden können. Bezogen auf einen Sprint in SCRUM bedeutet das etwa, dass diejenigen Features aus dem Produkt-Backlog (etwa Projektauftrag) in den Sprint-Backlog übernommen werden, die realistischerweise durch das Team auch umgesetzt werden können. Die Auswahl dieser Features kann aufgrund definierter Prioritäten (aus Kundensicht) erfolgen, wie sie beispielsweise durch die priorisierte Featureliste im Produkt-Backlog vorliegt. Die Iceberg-Liste wird um die Zuordnung der einzelnen Features zu den jeweiligen Releases ergänzt.

Erweiterte Iceberg-Liste

Die Tabelle 4.5 zeigt einen Auszug aus einer Iceberg-Liste dieses Beispiels. In dieser erweiterten Iceberg-Liste [19] sind folgende Einträge vorhanden: (a) eine laufende Nummer zur eindeutigen Identifikation der einzelnen Arbeitspakete und Features, (b) die eigentlich Features (entsprechend dem PSP), (c) den zugeordneten Anwendungsfall, (d) eine einfache Priorisierung (hoch (H), mittel (M) bzw. niedrig (N)) aus Kundensicht (abgeleitet

Tabelle 4.5 *Angefangene Iceberg-Liste für das BPSE-SAT-Projekt.*

#	Feature, Aktor	Anwendungsfälle	Kunden-Priorität	Komplexität	Version	Zuständiger
1.1	Registrierung, Student	Registrieren, Account anlegen, Accountdaten einsehen/ändern, neues Passwort anfordern, Passwort ändern.	H	8	1	AS
1.2	Login/Verwaltung, Administrator	Login, Accountdaten einsehen/ändern, Passwort ändern, Anmeldungen einsehen	H	8	1	EGF
2.1	LVA Verwaltung, Administrator	LVA erstellen/bearbeiten/löschen, LVA Daten übernehmen	H	8	1	SB
3.1	Terminverwaltung, Student	zu Termin anmelden, von Termin abmelden	M	4	1	DM
3.2	Terminverwaltung, Administrator	Termin erstellen	M	2	1	TOe
...	..., ...	..., ..., ...	...	...	1	...
1.3	Registrierung & Login, Administrator	Account suchen/löschen	H	8	2	MD
2.2	..., ...	..., ..., ...	...	...	2	...
3.3	..., ...	..., ..., ...	...	...	3	...

aus dem Projektauftrag), (e) den geschätzten Aufwand als Story-Points (ermittelt durch eine Expertenschätzung), (f) die Zuordnung zu einer definierten Version (Release) sowie (g) das dafür verantwortliche Teammitglied.

Horizontale Rollen und vertikale Arbeitspakete

Wie bereits in Abschnitt 4.2.3 kurz beschrieben, definieren die strategischen Rollen, die im Projektauftrag festgelegt sind, Verantwortlichkeiten horizontal über den Projektverlauf *(horizontale Rollen)*; es gibt etwa einen Testverantwortlichen, der für die Definition und Durchführung der Tests verantwortlich ist. Explizite Zuständigkeiten, wer konkrete Tests für eine Komponente umsetzt, wird über „vertikale" Zuständigkeiten für die jeweiligen Arbeitspakete definiert *(vertikale Arbeitspakete)*. Die Zuständigkeiten für die vertikalen Arbeitspakete werden in der erweiterten Iceberg-Liste definiert. Die Unterscheidung zwischen horizontalen Rollen und vertikalen Arbeitspaketen wird bei der Rolle „Tester" unter Verwendung des Test-Driven-Development-Paradigmas (TDD) gut ersichtlich (siehe auch Kapitel 5). Die Grundidee von TDD ist etwa, Testfälle vor bzw. spätestens während der Implementierung zu erstellen. Die horizontale Rolle (Testverantwortlicher) ist dabei dafür verantwortlich, dass die Unit-Tests erstellt werden; die eigentliche Umsetzung der Unit-Tests ist über vertikale Arbeitspakete festgelegt, die durch alle Teammitglieder während der Entwicklung umgesetzt werden müssen. Jedes Teammitglied muss also einen guten Überblick über das eingesetzte Testframework haben. Wenn nun jeder Entwickler seine eigenen Tests schreibt, was bleibt dann für die Tester übrig? Eine horizontale Verantwortlichkeit des Testverantwortlichen umfasst etwa die technische Mitarbeit im Rahmen von Analyse und Design, um die Testbarkeit des Software-Produkts auf Architekturebene (etwa im Rahmen

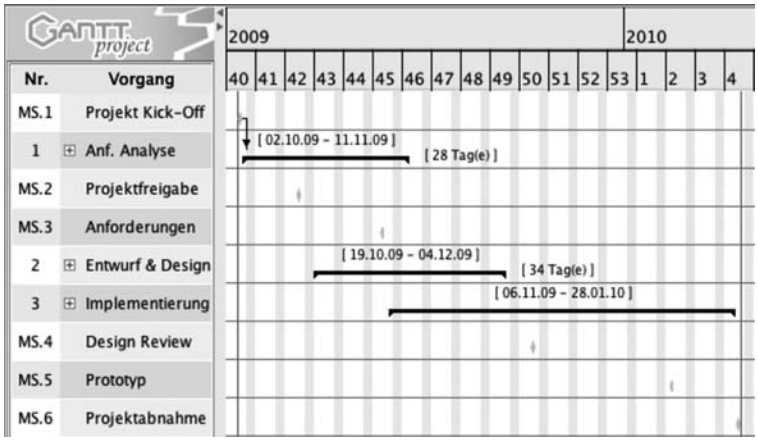


Abbildung 4.4
Balkendiagramm auf Meilensteinebene.

von Integrationstests) durch geeignete Maßnahmen der Qualitätssicherung sicherzustellen.

Wirtschaftlicher Plan

Bei der Projektplanung müssen neben den technischen Aspekten (etwa einzelne Arbeitspakete) auch wirtschaftliche Aspekte (Termin und Kosten) berücksichtigt werden. Kernergebnis der wirtschaftlichen Planung ist also ein Arbeits-, Aufwands- und Kostenplan über den gesamten Projektverlauf. Wie lange dauert das Projekt, wie groß ist der Aufwand an Personal und Ressourcen, welche Risiken sind je nach Projektumfang in Kauf zu nehmen? Für die Abschätzung des Aufwand einzelner und aggregierter Arbeitspakete gibt es eine Reihe von Schätzmethoden, wie sie bereits in Abschnitt 4.3.2 kurz beschrieben wurden.

Planungshilfsmittel

Da bereits in kleinen Projekten eine hohe Anzahl an unterschiedlichen und abhängigen Aktivitäten erforderlich ist, sind geeignete Hilfsmittel zur Projektplanung und Projektverfolgung erforderlich, um die Planung zu erleichtern und den Überblick über das Projekt zu behalten. Die wirtschaftliche Planung baut auf den Ergebnissen der technischen Planung auf, um die Arbeitsaufwände unter Berücksichtigung der verfügbaren Ressourcen im Hinblick auf realistische Termine abzuschätzen.

Diagramm-Methoden

Zur Planung von Ressourcen und Terminen müssen auch die jeweiligen Abhängigkeiten zwischen den Arbeitspaketen bzw. den notwendigen Abläufen erfasst werden. Zur Erfassung dieser Abhängigkeiten kann beispielsweise der Netzplan (PERT-Diagramm) eingesetzt werden, der in Abschnitt 4.3.4 beschrieben wird. Eine gute grafische Aufbereitung des zeitlichen Projektablaufs ermöglicht etwa ein Balkendiagramm (GANTT-Diagramm), das Projektphasen, Meilensteine und Aufgaben visualisiert. Abbildung 4.4 zeigt ein einfaches GANTT-Diagramm für das aktuelle Beispiel (siehe auch Abschnitt 4.3.5).

Projektplan

Der Projektplan umfasst als Ergebnis der Planungsphase alle wesentlichen Elemente der technischen und wirtschaftlichen Planung eines Projekts. Er dient als Grundlage für den Projektleiter einerseits für die Beurteilung des

aktuellen Projektzustands und andererseits für eine spätere Projektverfolgung. Im Wesentlichen unterstützt der Projektplan den Projektleiter bei der Beantwortung von grundlegenden Fragen:

Typische Fragestellungen

- Wie lange wird das Projekt voraussichtlich dauern?
- Wann beginnt und endet eine Aktivität frühestens?
- Wann beginnt und endet eine Aktivität spätestens?
- Welche Aktivitäten sind kritisch, da ihre Verzögerung sich direkt auf den Endtermin des Projekts auswirkt?
- Wie lange darf eine Aktivität länger dauern als geplant, ohne den Endtermin des Projekts zu verzögern?
- Wie sieht die zeitliche Verteilung des Ressourceneinsatzes (Geld, Personal, Maschinen) aus?
- Zu welchem Zeitpunkt wird eine bestimmte Ressource (Personal oder Maschine) eingesetzt?

Bei Abweichungen oder unvorhergesehenen Ereignissen (Risiken) kann bzw. muss der Projektleiter geeignet reagieren und gegebenenfalls eine Anpassung des initialen Projektplans durchführen (Projektverfolgung).

Werkzeugunterstützung

Zur Unterstützung der technischen und wirtschaftlichen Projektplanung existieren zahlreiche Methoden und Werkzeuge. Wichtige und in der Praxis häufig anzutreffende Vertreter von Werkzeugen, die sich typischerweise in mittelgroßen – also MIDI – Projekten finden, werden in den nächsten Abschnitten vorgestellt. Für kleinere Projekte können diese entsprechend vereinfacht angewandt werden. Größere Projekte erfordern meist zusätzliche Maßnahmen, wie etwa geeignete Organisationsmodelle, und eine entsprechende Werkzeugunterstützung.

4.3.4 Netzplan (PERT-Diagramm)

Abhängigkeiten zwischen Arbeitspaketen

Ein *Netzplan* oder *PERT-Diagramm (Program Evaluation and Review Technique)* ist ein grafisches Planungsinstrument zur Beschreibung von Abhängigkeiten und der notwendige Bearbeitungsreihenfolge der jeweiligen Arbeitspakete. Beispielsweise können Akzeptanztests erst nach Vorliegen eines implementierten und integrierten Systems durchgeführt werden. Grundlage dafür ist der Projektstrukturplan, der zwar alle wesentlichen Arbeitspakete beinhaltet, aber keine zeitliche Abfolge bzw. Abhängigkeiten definiert, wie sie für die Projektplanung erforderlich sind. Beispielsweise wird aus einem PSP auf Phasenebene (siehe Beispiel in Tabelle 4.3) ein PSP auf Arbeitspaketebene (siehe Beispiel in Tabelle 4.4). Bei diesem

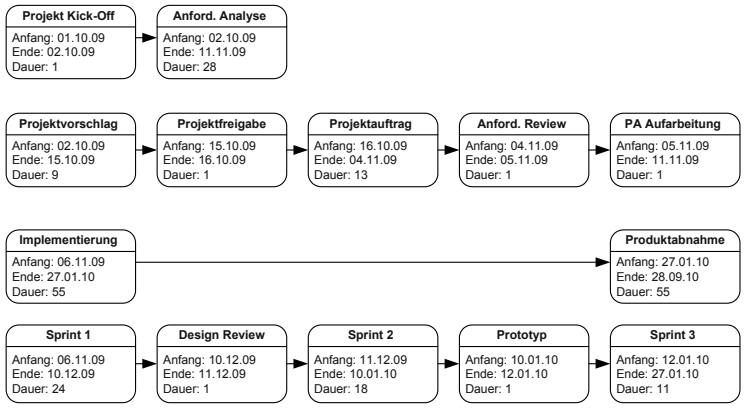


Abbildung 4.5
Schematischer Netzplan für das BPSE-SAT-Projekt.

Verfeinerungsschritt werden die Arbeitspakete speziell im Hinblick auf bekannte Start- und Endzeitpunkte und die benötigten Ressourcen genauer definiert.

Beispiel

Die Abbildung 4.5 zeigt ein Beispiel eines Netzplans auf Meilensteinebene – entsprechend dem PSP in Tabelle 4.3 – mit den relevanten Phasen des Projekts (Aktivitäten) und entsprechenden Abhängigkeiten zwischen diesen Aktivitäten. Jede Aktivität sollte durch eine (zeitlich) vorangegangene Aktivität ausgelöst werden und für eine folgende Aktivität erforderlich sein. Dieses Verfahren hat den Vorteil, Vorher-Nachher Abhängigkeiten gut abzubilden.

Der Vorgänger für alle Aktivitäten ist der Projektbeginn, der Nachfolger ist das Projektende. Übergeordnete Aktivitäten können etwas als Meilensteine definiert werden, die etwa im Rahmen der Projektverfolgung eingesetzt werden. Durch die Definition des jeweiligen Anfangs- und Enddatums sowie der Dauer ist die Ermittlung des Gesamtaufwands aus dem Netzplan einfach ablesbar. Typischerweise finden sich im PERT-Diagramm auch Angaben zum *frühestmöglichen Anfang (FA)*, das *frühestmögliche Ende (FE)* bzw. Daten für den *spätestmöglichen Anfang (SA)* und das *spätestmögliche Ende (SE)*. Aus Gründen der Übersichtlichkeit sind diese in der Abbildung nicht dargestellt. Diese Daten werden für die Projektverfolgung eingesetzt, um etwa auf Unvorhergesehenes geeignet reagieren zu können.

Kritischer Pfad

Aus den zeitlichen Rahmenbedingungen ergibt sich auch der sogenannte *kritische Pfad* durch das Projekt, der die Mindestprojektdauer festlegt. Dadurch sind sowohl minimale Projektdauer als auch maximale Projektdauer ableitbar. Die minimale Projektdauer ergibt sich aus der Differenz des frühesmöglichen Endzeitpunkt des Endknotens (FE, meist das Projektende) und des spätestemöglichen Startzeitpunks des Anfangsknotens (FA, meist der Projektbeginn). Auf der anderen Seite lässt sich die maximale Projektdauer als die Differenz des spätestmöglichen Endzeitpunkts des Endknotens (SE) und des frühestmöglichen Startzeitpunkts des Anfangsknotens

Abschätzung der Projektdauer

(SA) ermitteln. In der Praxis sind FE und SE meist gleich definiert, d.h. es gibt nur ein definiertes Projektende.

Pufferzeit

Aus diesen „Toleranzzeiten“ ergibt sich auch eine mögliche Pufferzeit, die sich aus der Differenz von SA und FA einer Aktivität berechnet. Diese Pufferzeit kann zur Projektsteuerung verwendet werden, falls eine Aktivität länger dauert oder erst später beginnt, ohne eine Projektverzögerung zu verursachen. Ist die Nutzung einer Pufferzeit notwendig, bewirkt das zwar keine direkte Projektverzögerung, kann sich aber auf folgende und abhängige Pufferzeiten auswirken, da sich ja die Anfangszeitpunkte entsprechend verschieben. Im Rahmen der Projektverfolgung bzw. -steuerung sind das wichtige Informationen für die Priorisierung von Ressourcen und Anpassung der Planung. Aktivitäten ohne mögliche Pufferzeiten werden als *kritische Aktivitäten* bezeichnet und liegen auch auf dem kritischen Pfad. Treten hier Verzögerungen auf (es gibt ja keine Pufferzeit), hat das einen direkten Einfluss auf die Projektdauer und führt zu Projektverzögerungen.

Projektkalender

Es ist auch empfehlenswert, einen Projektkalender für jedes Projekt bzw. jeden Mitarbeiter aus dem Netzplan abzuleiten. Dadurch ergibt sich die Möglichkeit die Verfügbarkeit bzw. auch die Nicht-Verfügbarkeit von „Ressourcen“ entsprechend einzuplanen und zu dokumentieren. Über den Projektkalender werden die Zeitpunkt des Netzplanes in konkrete Kalenderdaten transformiert.

Nach der initialen Erstellung des Netzplanes wird er durch den Projektleiter überprüft und gegebenenfalls optimiert und angepasst. Dabei sind folgende ausgewählte Punkte als mögliche „Checkliste“ zu beachten:

Review des PERT-Diagramms

- Die Intensität der Ressourcenverwendung darf das Maß der real verfügbaren Ressourcen nicht überschreiten, insbesondere ist das für parallel ablaufende Arbeitspakete zu berücksichtigen.
- Alle Aktivitäten, die als unabhängig voneinander geplant wurden, müssen auch tatsächlich parallel ausgeführt werden können.
- Neben expliziten Abhängigkeiten gibt es auch implizite Abhängigkeiten, etwa über die Verwendung der gleichen Ressourcen (Projektkalender).
- Im Netzplan ist die Unsicherheit der Aufwandsschätzung über ausreichende Pufferzeiten zu berücksichtigen.
- Verkürzte Pufferzeiten können erhöhte Risiken verursachen.
- Arbeitspakete, die im kritischen Pfad liegen, sollten mit besonders erfahrenen Mitarbeitern geplant werden, um das Risiko von Verzögerungen zu verringern.

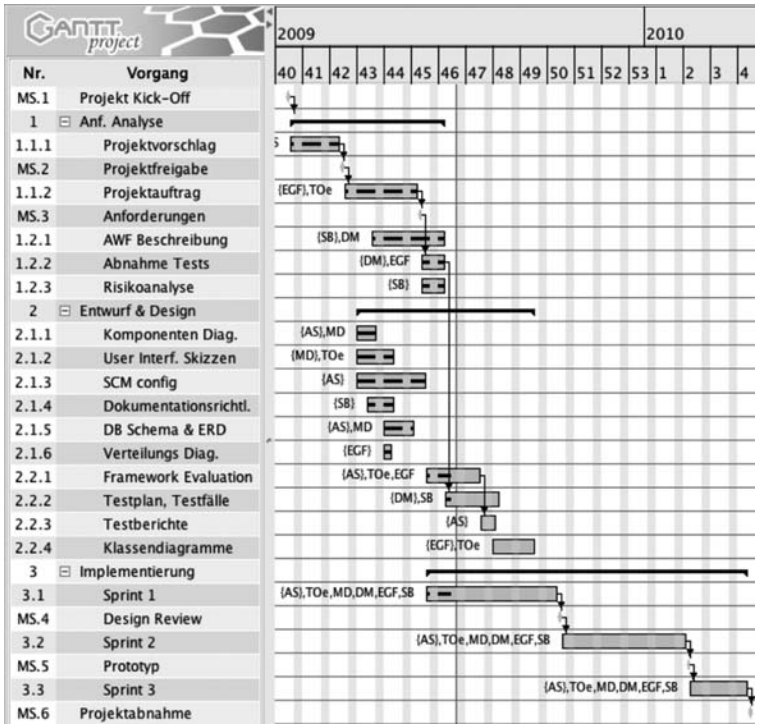


Abbildung 4.6
GANTT-Diagramm auf Arbeitspaketebene.

PSP und Netzplan modellieren den Ressourcen- und Terminplan als Abfolge von Aktivitäten. Zur Visualisierung des zeitlichen Abfolge von Aktivitäten hat sich in der Praxis ein weiteres Hilfsmittel – ein Balkendiagramm – durchgesetzt.

4.3.5 Balken- oder GANTT-Diagramm

Zeitliche Abfolge

Um den zeitlichen Ablauf des Projekts übersichtlich darstellen zu können, hat sich ein Balkendiagramm (GANTT-Diagramm) in der Praxis bewährt. Damit können die Informationen aus dem PSP und dem Netzplan aufbereitet werden, um die Vor- und Nachteile konkreter Projektpläne im Zeitraster zu visualisieren und zu analysieren. Zusätzlich können Balkendiagramme aus der vorhandenen Planung (zu Projektbeginn und während des Projektverlaufs) automatisch generiert werden [33]. Während der Projektlaufzeit liefert das Balkendiagramm Informationen über den aktuellen Projektstatus und ermöglicht den direkten Vergleich zwischen Ist-Stand und Planung.

Vorgehensweise und Beispiel

Nach der Ermittlung der einzelnen Arbeitspakete (im Rahmen der Erstellung des PSP) erfolgt die Planung der für die Umsetzung notwendigen Ressourcen, etwa Personalbedarf und technische Hilfsmittel. Aufbauend auf den zeitlichen Abhängigkeiten, z.B. Startzeitpunkt und Dauer, werden die Arbeitspakte entsprechend eingeplant und gemäß Projektstruktur aggregiert. Ein wesentliches Ergebnis der Planungsphase ist auch die Auslastung

der Ressourcen, die in weiterer Folge auch für die Projektverfolgung einen guten Überblick über den aktuellen Projektzustand liefern kann. Typischerweise werden diese Informationen aus dem PERT-Diagramm und dem PSP generiert. Ein einfaches GANTT-Diagramm [33] wurde bereits in Abbildung 4.4 auf Meilensteinebene vorgestellt. Abbildung 4.6 baut darauf auf und liefert eine genauere Sicht auf das Projekt und die enthaltenden Arbeitspakete, in diesem Fall die Implementierungsphase. Jeder „Balken" bezeichnet dabei eine konkrete Aktivität, wie etwa die Umsetzung eines Arbeitspaketes. Die Länge des entsprechenden Balkens stellt den zeitlichen Aufwand, also die Dauer für die Umsetzung, dar. Diese Balken werden horizontal über eine Zeitachse, die den Projektverlauf visualisiert, angeordnet. Gängige Praxis ist es, den frühestmöglichen Anfangszeitpunkt (FA) und den frühestmöglichen Endzeitpunkt (FE) einzuzeichnen. Analog zum PERT-Diagramm können am Ende jeder Aktivität auch entsprechende Pufferzeiten berücksichtigt werden. Eine vertikale Betrachtung ermöglicht die Sicht auf aktuell aktive Arbeitspakete zu einem bestimmten Zeitpunkt und erlaubt auch die Ermittlung der entsprechenden Ressourcenauslastung.

Erweiterungen

In der Praxis ist es auch empfehlenswert, die jeweilige Dauer und das verantwortliche Teammitglied bei den jeweiligen Arbeitspaketen und Aktivitäten zu vermerken.

4.3.6 Burn-down-Charts

Planung „agiler" Arbeitspakete

PERT-Diagramm und GANTT-Diagramm stammen aus dem klassischen Projektmanagement, das primär auf traditionellen Prozessen, wie dem V-Modell oder dem Rational Unified Process, aufgebaut ist. Wie eingangs bereits erwähnt, empfiehlt sich in der Praxis eine grobe Projektplanung mithilfe eines dieser traditionellen Ansätze. In den konkreten Umsetzungsphasen finden sich immer häufiger flexible und agile Ansätze, die ebenfalls über entsprechende Planungsinstrumente verfügen.

Burn-down-Chart

Im Beispielprojekt BPSE SAT wurden über das GANTT-Diagramm (siehe Abbildung 4.6) beispielsweise in Anlehnung an SCRUM drei Sprints eingeplant. Ein derartiger Sprint könnte beispielsweise – dem „klassischen" Ansatz folgend – weiter auf entsprechende Arbeitspakete heruntergebrochen werden. Eine bessere und flexiblere Planung wird aber durch die Verwendung von *Burn-down-Charts* möglich.

Aufbau und Verwendung

Die Grundlage für die Planung eines agilen Projekts oder Arbeitspaketes nach SCRUM ist das Produkt- bzw. das Sprint-Backlog, das in Form einer Iceberg-Liste realisiert ist (siehe auch Abschnitt 3.8). Das Burn-down-Chart visualisiert dabei den Projektfortschritt unter Berücksichtigung der geplanten, fertiggestellten und offenen Arbeitspakete. Die Abbildung 4.7 zeigt ein Beispiel für ein Burn-down-Chart des ersten Sprints des BPSE SAT Projekts. Auf der x-Achse findet sich der zeitliche Verlauf des Sprints bis zum Ende des Sprints, auf der y-Achse wird der Aufwand für die ver-

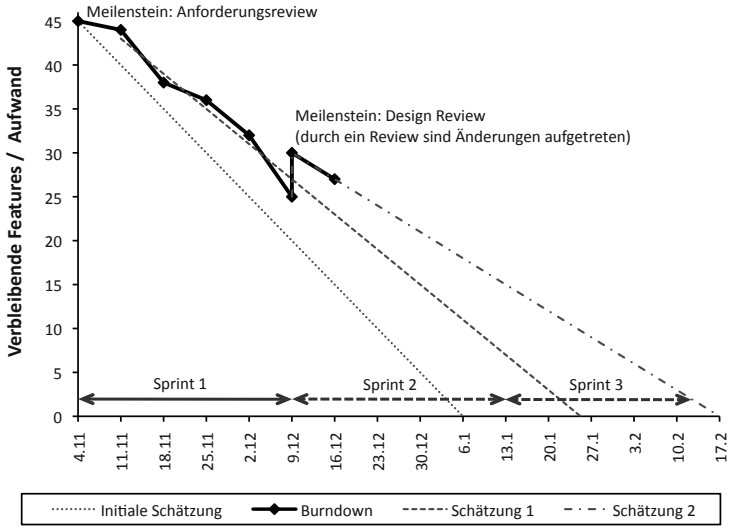


Abbildung 4.7
Burn-down-Chart für das BPSE-SAT-Projekt (Sprint 1).

bleibenden (offenen) Arbeitspakete und Features skizziert. Der Aufwand wird dabei – analog zu den Schätzverfahren – entweder in Personentagen oder -wochen oder als „Komplexitätsgrad" auf einer Skala von 1 bis 10 geschätzt.

Plan und Realität

Bei der initialen Planung bzw. im Idealfall weist das Burn-down-Chart einen linearer Projektverlauf auf, der in der Praxis praktisch nie erreicht werden kann, da die Arbeitspakete kaum mit einem linearen Aufwand bearbeitbar sind („beste Schätzung"). Während des Projektverlaufs wird das Burn-down-Chart täglich („Daily Scrum") überprüft und an die aktuellen Gegebenheiten angepasst, d.h. abgearbeitete Arbeitspakete aber auch Projektverzögerungen werden eingetragen. Meist hat das auch zur Folge, dass sich die Schätzungen bis zur Fertigstellung entsprechend ändern (siehe „Schätzung 1" und „Schätzung 2"). Je weiter der Sprint fortgeschritten ist, desto geringer sollte der verbleibende Aufwand sein, wie in Abbildung 4.7 dargestellt ist – daher auch die Bezeichnung „Burn-down". Treten etwa unerwartete Verzögerungen auf, kann der verbleibende Aufwand natürlich größer werden. Im Beispielprojekt wurden etwa im Rahmen eines Design-Reviews notwendige Änderungen gefunden, die eine Projektverzögerung verursachen; das Burn-down-Chart muss entsprechend angepasst werden, d.h. der verbleibende Aufwand steigt wieder an.

Projektverfolgung

Durch diese laufende Überprüfung (Daily Scrum) und eine gegebenenfalls erforderliche Anpassung wird eine unmittelbare und schnelle Reaktion auf geänderte Projektparameter ermöglicht. Da sich dadurch auch die Schätzungen, die einen direkten Einfluss auf die Fertigstellung des Sprints haben können, ändern können, kann das Burn-down-Chart nicht nur zur initialen Planung sondern auch zur Projektverfolgung eingesetzt werden.

Unabhängig von den verwendeten Planungsmethoden kann die Projektplanung als abgeschlossen betrachtet werden, wenn der Projektplan einen ausreichend hohen Detailgrad aufweist und auf die geschätzten Aufwände der jeweiligen Arbeitspakete abgestimmt ist. Er dient dann als Basis und Hilfsmittel für die Projektverfolgung.

4.4 Projektverfolgung

Von der Planung zur Umsetzung

Nach der Fertigstellung der initialen Projektplanung beginnt die operative Umsetzung des Projekts. Ab diesem Zeitpunkt beginnt die Projektverfolgung, die primär durch den Projektleiter wahrgenommen wird. Der Projektleiter muss dafür sorgen, dass die Aufgaben entsprechend der Planung umgesetzt werden, und die notwendigen Ressourcen, wie Personal, Zeit und Budget, zur Verfügung stehen [56]. Die Projektverfolgung umfasst spezielle Aufgaben des Projektmanagements zur Erhebung des aktuellen Projektstatus und den Vergleich mit der Projektplanung. Bei Abweichungen muss der Projektleiter geeignete Maßnahmen einsetzen, um das Projekt wieder „auf Kurs“ zu bringen. Abweichungen können dabei Zeit und Kosten aber auch Qualitätsproblem und eingetretene Risiken umfassen. Da korrigierende Maßnahmen meist einen zusätzlichen Aufwand erfordern, muss der Projektplan auch entsprechend angepasst werden.

Statusfeststellung

Die Feststellung des aktuellen Projektstatus stellt – speziell bei großen und komplexen Projekten – meist eine besondere Herausforderung dar. Ein erfolgreicher Projektleiter behält trotz der vielfältigen Projektaktivitäten den Überblick über die einzelnen Arbeitspakete auf den unterschiedlichen Projektebenen (gemäß Projektstruktur) und in weiterer Folge über das Gesamtprojekt. Die in der Projektplanungsphase eingeführten Methoden und Werkzeuge können den Projektleiter bei dieser Aufgabe unterstützen, sofern die Planungsunterlagen immer aktuell gehalten werden. Eine ständige Beobachtung des Projektverlaufs ermöglicht es auch, auf geänderte Rahmenbedingungen und auf neue Herausforderungen zeitnah und geeignet reagieren zu können. Daher empfiehlt sich in der Praxis, die Planungsdokumente immer auf dem aktuellsten Stand zu halten.

Projektstatusbericht

In definierten Zeitintervallen, meist bei Meilensteinen, ist die Erstellung eines Projektstatusberichts empfehlenswert, um den Zustand des Projekts geeignet zu dokumentieren. In diesem Bericht wird beschrieben, welche Aktivitäten durchgeführt wurden und welche Arbeitspakete für die nächste Projektiteration bzw. den nächsten Scrum-Sprint geplant sind. Integraler Bestandteil ist auch ein Soll/Ist-Vergleich der Arbeitspakete, um auf Abweichungen reagieren zu können. Der Projektstatusbericht soll zusammengefasst zeigen, ob das Projekt innerhalb der festgelegten Projektparameter (Zeit, Kosten und Qualität) abgewickelt wird und – bei Abweichungen – gegebenenfalls um einen Problembericht ergänzt werden. Wichtig für die Nachvollziehbarkeit ist es auch, aufgetretene Verzögerungen, eingetretene

Risiken oder (temporäre) Ausfälle von Teammitgliedern zu dokumentieren. Gegebenenfalls ist es auch sinnvoll, eingetretene Risiken zu begründen, um daraus lernen zu können.

Berichtswesen

Speziell in größeren Organisationen, die eine Vielzahl an unterschiedlichen Projekten abwickeln, ist auch eine organisationsweite Berichterstattung erforderlich, um den Status aller Projekte kompakt darstellen zu können. Diese Berichte dienen dann auch zur Synchronisierung aller Projekte und können dabei helfen, Risiken wie etwa Ressourcenengpässe geeignet zu kompensieren. Voraussetzung ist dabei allerdings, dass geeignete Unterlagen, wie etwa aktuelle Projektfortschrittsberichte, verfügbar sind.

Fortschrittskontrolle

Die Fortschrittskontrolle dient primär dem Ziel, Planabweichungen frühzeitig zu erkennen und geeignete steuernde Maßnahmen rechtzeitig und vor allem zeitnahe einzuleiten. In diese Kontrolle werden sowohl quantifizierbaren Größen (etwa Termin, Aufwand und Kosten) sowie der Sachfortschritt miteinbezogen. In der Praxis können diese Fortschrittskontrollen durch geeignete Werkzeuge, wie beispielsweise Apache Maven (siehe Abbildung 10.8 in Abschnitt 10.7) unterstützt werden. Dabei werden die definierten Arbeitspakete, die typischerweise in einer zentralen Produktdokumentation eingebunden sind, einem Soll/Ist-Vergleich unterzogen und entsprechende Berichte automatisch erstellt.

Risikomanagement

Risiken spielen in der Projektverfolgung eine wichtige Rolle, da sie – falls sie eintreten – zu erheblichen Projektverzögerungen und im Extremfall bis zum Projektabbruch führen können. Im Rahmen des Projektauftrags und der initialen Projektplanung werden Risiken vorab identifiziert und bereits geeignete Gegenmaßnahmen eingeplant. Beispielsweise dienen die Pufferzeiten bei der Projektplanung bereits dem Ziel, das Risiko der Projektverzögerung durch Terminüberschreitungen einzelner Arbeitspakete zu adressieren. Bei der Projektverfolgung müssen diese initial erkannten Risiken periodisch überprüft und gegebenenfalls aktualisiert werden. In vielen Fällen werden Risiken erst im Lauf des Projekts erkannt und müssen entsprechend berücksichtigt werden. Für kritische Projektrisiken (meist Top 5) ist es empfehlenswert, in einer periodischen Analyse (a) die Schadenswahrscheinlichkeit, (b) die erwartete Schadenshöhe sowie (c) geeignete Maßnahmen zur Minimierung des erwarteten Schadens zu erheben. Unter einem „Risiko“ wird dabei ein Ereignis verstanden, das mit abschätzbarer Wahrscheinlichkeit auftreten kann und dem Projekterfolg erheblichen und quantifizierbaren Schaden zufügen kann. Typische Risiken sind dabei beispielsweise der Ausfall von wichtigen Teammitgliedern, der Einsatz unbekannter Technologien und unrealistische Zeitpläne. Da sich diese Risiken im Lauf des Projekts ändern können, ist es empfehlenswert, diese laufend zu überprüfen, zu aktualisieren oder zu ergänzen.

Risiko

Eine in jedem Projekt auftretende Aufgabe des Projektleiters ist die Projektverfolgung im Hinblick auf die erwarteten (geplanten) und tatsächlichen Aufwände. Daher werden in den nächsten Abschnitten einige Techniken,

Tabelle 4.6 *Auszug aus einer Stundenliste.*

Datum	Von	Bis	Stunden	Kategorie	Tätigkeit
20.10	16:30	17:30	1	Besprechung	Erstellen des Testplans.
20.10	18:00	20:00	2	Implementation	Terminverwaltung. Test erstellen, Methoden für speichern, aktualisieren, löschen und suchen getestet.
	Summe		**3**		

die bereits im Rahmen der Planung verwendet wurden, vorgestellt, um die Aufwände geeignet zu erfassen.

4.4.1 Aufwandserfassung mit dem PSP

Aufwände

Da die Aufwände einen direkten Einfluss auf den Projektfortschritt und auch auf die Kosten haben, ist es notwendig, diese möglichst detailliert zu erfassen. Beim PSP wurden bereits initiale Aufwände ermittelt und eingeplant. Im Rahmen der Projektverfolgung eignet sich diese Methode aber nur eingeschränkt, um eine effiziente Aufwandserfassung umzusetzen. In der Praxis ist es empfehlenswert, und auch üblich, eine Werkzeugunterstützung für die Aufwandserfassung einzusetzen. Beispiele dafür werden im Kapitel 10 beschrieben.

Stundenliste

Stehen derartige Werkzeuge nicht zur Verfügung (etwa mangels Infrastruktur oder bei kleinen Projekten und Organisationen), muss auf eine manuelle Stundenliste zurückgegriffen werden. Diese Stundenlisten beinhalten neben den investierten Aufwänden auch die jeweiligen Tätigkeiten und Arbeitspakete. Daher ist der Projektleiter darauf angewiesen, die aktuellen Stundenlisten von allen Teammitgliedern zu erhalten, um den Projektstatus erheben zu können und eine Projektverfolgung überhaupt durchführen zu können. Die Tabelle 4.6 illustriert den Auszug einer exemplarischen Stundenliste aus dem BPSE SAT Projekt. Wesentliche Komponenten sind neben den persönlichen Daten natürlich Datum und Uhrzeiten (Start- und Endzeit), Dauer, aber auch Tätigkeit und – zur einfacheren Weiterverarbeitung – eine Kategorisierung der Tätigkeiten. In vielen Fällen sind die Stundenlisten standardisiert, d.h. sie verfügen über denselben Aufbau und dasselbe Layout. Jedes Teammitglied erfasst sowohl die individuellen Aufwände für die zugeteilten Aktivitäten und Arbeitspakete als auch den Gesamtaufwand.

Richtigkeit der Stundenliste

Um die Richtigkeit der Stundenliste sicherstellen zu können, sollten sie zumindest wöchentlich im Team besprochen und abgestimmt werden. Diese Durchsprache dient primär nicht der Kontrolle der Teammitglieder, sondern vielmehr der Identifikation von aufwändigen Arbeitspaketen und zur Aufdeckung von fehlerhaften Schätzungen. Diese Durchsprachen können quasi als „Frühwarnsystem“ für potentielle Risiken eingesetzt werden. Werden Stundenlisten nicht gepflegt, gibt es auch die Möglichkeit, Aufwandsaufzeichnungen anhand von konkreten Arbeitsergebnissen, z. B. Check-in,

Durchführung von Testläufen, Tickets, Dokumenten, automatisch zu gewinnen. Dadurch kann auch der Mehraufwand für das Projektmanagement reduziert werden, da eine exakte Zuordnung zu den Arbeitspaketen möglich ist.

Nutzen von Stundenlisten

Aus vollständige Stundenlisten können die jeweiligen Aufwände für die Arbeitspakete bestimmt und für die Aktualisierung der Projektplanung eingesetzt werden. Meist geschieht das im Rahmen der Erstellung des Projektstatusbericht. Dabei sollen folgende Kernfragen beantwortet werden können:

- *Wie lange wurde an dem Arbeitspaket gearbeitet?* Diese Information dient in erster Linie dem Zweck, den tatsächlichen Aufwand abzuschätzen und die ursprüngliche Schätzmethode (etwa Analogiemethode) zu verbessern.
- *Welcher Aufwand ist für die Fertigstellung noch notwendig?* Falls ein Arbeitspaket noch nicht abgeschlossen wurde, wird dadurch die weitere Projektplanung erleichtert.

In jedem Fall sollten die jeweiligen Planungsdokument, speziell das GANTT-Diagramm basierend auf den aktualisierten Daten, neu erstellt werden. Besonders wichtig ist diese Aktualisierung, falls kritische Arbeitspakete, d.h. Arbeitspakete, die im kritischen Pfad liegen, betroffen sind.

4.4.2 Meilenstein-Trendanalyse

Verfolgung der Meilensteine

Die initiale Planung baut auf einer groben Strukturierung des Projekts in Meilensteinen, also auf definierten Zeitpunkten innerhalb des Projekts, auf (siehe Tabelle 4.3 als PSP und Abbildung 4.4 als GANTT-Diagramm). Die Meilenstein-Trendanalyse (MTA) ermöglicht die Betrachtung dieser Zeitpunkte im Hinblick auf die noch nicht erreichten Meilensteine und eine Anpassung der zukünftigen Meilensteine an aktuelle Projektgegebenheiten. Die Abbildung 4.8 zeigt exemplarisch einen Auszug einer derartigen Meilenstein-Trendanalyse.

Aufbau und Verwendung

Zur übersichtlichen Darstellung werden zwei Zeitachsen verwendet, wobei etwa auf der x-Achse die Betrachtungszeitpunkte (z. B. im Zusammenhang mit einem Meilenstein-Review oder der Erstellung eines Projektstatusberichts) und auf y-Achse die (geschätzten) Termine für die Meilensteine eingetragen werden. Die Skalierung der Zeitachsen ergibt sich aus der Projektlaufzeit und kann in Monaten, Kalenderwochen oder auch Tagen erfolgen. In der Projektplanung werden die geplanten Zeitpunkte für die Erreichung der Meilensteine definiert und in der MTA eingetragen. Bei einem Meilenstein-Review wird der aktuelle Status des Projekts erhoben und die Schätzung für das Erreichen der weiteren Meilensteine angepasst und in die MTA im zeitlichen Raster eingetragen. Dadurch sieht man einerseits

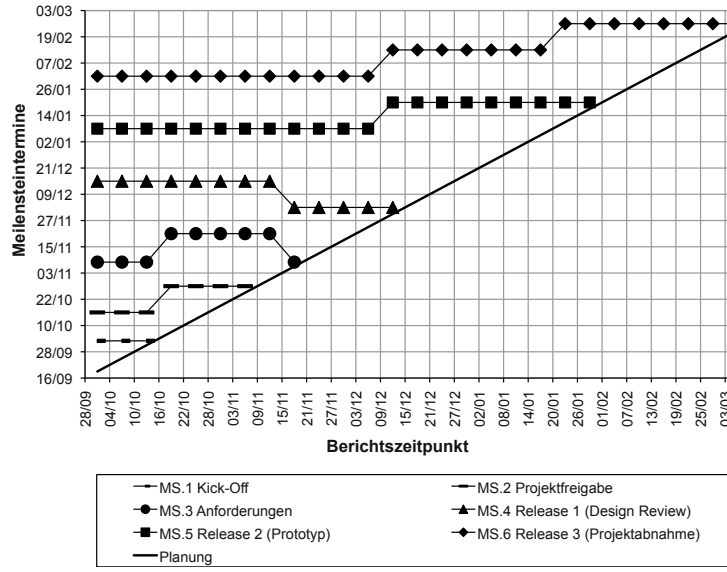


Abbildung 4.8
Meilenstein-Trendanalyse.

den aktuellen Projektstatus und anderseits Abweichungen gegenüber früheren Schätzungen.

Interpretation

Im Idealfall werden die Meilensteine zeitgerecht erreicht. Dadurch ändert sich natürlich auch die Schätzung nicht. Die „Schätzungen" für bereits abgeschlossene Meilensteine bleiben auch ebenfalls konstant. Im MTA ist das durch eine senkrechte Linie gekennzeichnet. In der Regel sind jedoch Anpassungen erforderlich. Typischerweise sind in der MTA folgende charakteristische Schätzverläufe zu finden:

Schätzverlauf

- *Verzögerungen* sind dadurch gekennzeichnet, dass sich die Meilensteine in Richtung Projektende verschieben, falls der geplante Termin nicht eingehalten werden kann.
- Gelegentlich werden Meilensteine *frühzeitig* erreicht, falls zu große Pufferzeiten eingeplant wurden oder ein Arbeitspaket schneller abgeschlossen werden kann. In der MTA ergibt sich eine Verschiebung „nach vorne".
- *Unsicherheiten in der Schätzung* sieht man beispielsweise an einem „Zick-Zack-Kurs"der Schätzwerte, d. h,. die Termine werden bei jedem Review nach hinten oder aber auch nach vorne verschoben. Eine andere Interpretation wäre etwa, dass der Meilenstein durch die Erkennung eines konkreten Problems verschoben werden musste (Verzögerung) und durch die Lösung des Problems wieder auf „Kurs" gebracht werden konnte.

Alternativen

Die Interpretation und die Einführung geeigneter Maßnahmen ist eine zentrale Aufgabe des Projektleiters im Rahmen der Projektverfolgung und Be-

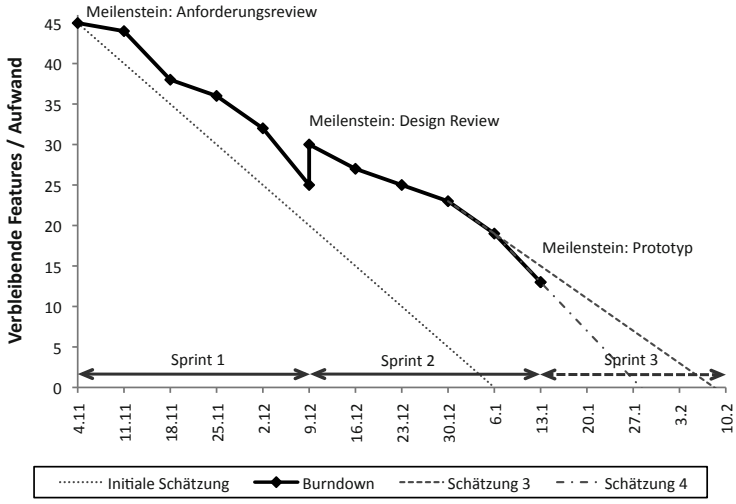


Abbildung 4.9
Burn-down-Chart für das BPSE-SAT-Projekt (Sprint 2).

richterstattung. Typischerweise findet sich die MTA in jedem Projektstatusbericht, da sie gute Informationen über die terminliche Situation im aktuellen Projekt liefern kann. Betrachtet man ein agiles Projekt scheint eine MTA nur eingeschränkt sinnvoll zu sein, da sie etwa bei jedem Daily Scrum aktualisiert werden müsste. Bei einer agilen Vorgehensweise empfiehlt sich eine Terminverfolgung im Burn-down-Chart, das bereits im Rahmen der Planung vorgestellt wurde.

4.4.3 Aufwandserfassung mit Burn-down-Charts

Wie bereits in Abschnitt 4.3.6 in der Projektplanungsphase beschrieben, wird das Burn-down-Chart sowohl zur initialen Planung als auch zur laufenden Projektverfolgung im Rahmen der Daily Scrums verwendet. Die Abbildung 4.9 zeigt etwa ein Burn-down-Chart des zweiten Sprints des BPSE SAT Projekts. Betrachtet man beispielsweise den Projektverlauf, ist eine Anpassung durch Änderungen, die im Rahmen des Design-Reviews notwendig wurden, zu erkennen. Entsprechend dem Projektfortschritt ändert sich der zu erwartende Restaufwand, der sich in einer tendenziell sinkenden Kurve zeigt. Wie bereits im Rahmen der Planung vorgestellt, erfolgt bei jedem Daily Scrum auch eine Schätzung des verbleibenden Restaufwands (illustriert durch die strichlierte Linien). Es ist auch deutlich erkennbar, dass die Schätzgenauigkeit mit steigendem Projektfortschritt immer genauer wird (die Differenz zwischen einzelnen Schätzungen ändert sich nur noch geringfügig), da immer mehr Detailwissen über das Projekt bzw. über die offenen Arbeitspakete verfügbar ist.

Nach Erreichen aller Meilensteine und Abschluss aller Sprints nähert sich das Entwicklungsprojekt dem Projektende bzw. Projektabschluss. Der Projektabschluss betrifft sowohl das Ende eines erfolgreichen Projekts als auch

den Abbruch eines Projekts. In jedem Fall sind beim Projektabschluss einige wichtige Aspekte zu betrachten, die in folgendem Abschnitt erläutert werden.

4.5 Projektabschluss

Jedes Projekt hat ein Ende

Jedes Projekt – unabhängig davon, ob es erfolgreich abgeschlossen wird oder abgebrochen werden muss – soll kontrolliert zum Ende gebracht werden. Inhaltlich sind dabei zwei unterschiedliche Aspekte zu betrachten: (a) der Abschluss der fachlichen Projektarbeit, d. h., der Abschluss aller technischen Projektinhalte und (b) ein formaler und administrativer Abschluss einschließlich der formalen Abschlusserklärung durch den Auftraggeber. In der Praxis sind dabei zwei wesentliche Schritte, ein *Count-down* und ein *Close-down* einschließlich einer eventuell notwendigen Nachbetreuung empfehlenswert [56].

Count-down

Der *Count-down* umfasst eine provisorische Projektabnahme, die Abarbeitung etwaiger offener Arbeitspakete gemäß Projektplan und Iceberg-Liste, die Erstellung der Abschlussdokumentation und die Organisation einer Projektabschluss-Veranstaltung. Typischerweise wird dieser Count-down durch die Erreichung aller Projektziele oder durch eine Entscheidung zum Projektabbruch eingeleitet. Ziel der provisorischen Projektabnahme ist die Feststellung (meist aus Anwendersicht), ob die im Projektauftrag definierten Anforderungen auch geeignet umgesetzt wurden. Noch fehlende Arbeitspakete werden entsprechend notiert. Für den Count-down sollte ein entsprechendes Zeitbudget eingeplant werden, um noch etwaige Nacharbeiten bzw. Ergänzungen durchführen zu können.

Close-down

Mit dem letzten Schritt – dem *Close-down* – wird das Projekt endgültig abgeschlossen. Der Abschluss umfasst die Abnahme des Projektergebnisses durch den Auftraggeber und die Übergabe der gesamten Abschlussdokumentation. In vielen Fällen werden auch spezielle Vereinbarungen für Nacharbeiten getroffen oder Folgeprojekte angestoßen. Folgeprojekte können dabei entweder auf noch offenen Punkte der Iceberg-Liste oder – in den meisten Fällen – aus den definierten Nicht-Zielen des konkreten Projekts aufgebaut sein. Die Nicht-Ziele grenzen, wie im Rahmen des Projektauftrags definiert wurde, das Projekt klar ab und legen fest, was das Projekt nicht beinhalten sollte. Dementsprechend wird bereits im Projektauftrag des jetzt abgeschlossenen Projekts der erste Schritt für ein etwaiges Folgeprojekt gelegt.

Projektabnahme

Bei der Abnahme des Projektergebnisses wird zwischen Managementebene und Benutzerebene unterschieden. In der Regel wird das Management des Auftraggebers ein Projekt nur abnehmen, wenn die Benutzer „grünes Licht“ gegeben haben. Mit dieser formellen Abnahme wird das Entwicklerteam und Projektleiter seitens des Auftraggebers von der Projektverantwortung entlastet. Die Abnahme erfolgt meist im Rahmen einer einfachen

Abschlussbesprechung mit Abschlusspräsentation, die je nach Projektgröße und Projekterfolg mit oder ohne Organisation eines sozialen Events verbunden wird. Wichtig ist, dass vor dieser Abschlussbesprechung eine umfassende Phasenabschluss-Vorbereitung (Count-down) gemacht wurde. Dadurch wird vermieden, dass in dieser Besprechung unnötige Diskussionen (etwa über das Erreichen oder Nicht-Erreichen vereinbarter Ziele) stattfinden.

Projektendbericht

Die Abschlussdokumentation eines Projekts ist eine wesentliche Komponente des Projektabschlusses. Sie soll in jedem Fall einen Projektendbericht enthalten, der an die Projektstatusberichte angelehnt ist. Im Projektendbericht sollte kurz zusammengefasst werden, wie das Projekt generell verlaufen ist. Man betrachtet hier das gesamte Projekt, vom Projektstart bis zum Projektende, um unter folgenden Gesichtspunkten ein Resümee über das Projekt ziehen zu können: (a) Wurden die geschätzten Aufwände über- bzw. unterschritten? (b) Warum wurden die geschätzten Aufwände über- bzw. unterschritten? (c) Wo traten Probleme auf? (d) Mit welchen Maßnahmen konnte man diese Probleme beheben? und (e) Wie gut hat das Team zusammengearbeitet bzw. welche Stärken/Schwächen konnte man erkennen?

Lessons Learned

Dieser Projektendbericht ist im Wesentlichen eine „Zusammenfassung“ des beendeten Projekts. Ein sehr wichtiger Aspekt sind aber die dokumentierten und gemachten Erfahrungen des abgelaufenen Projekts die als „Lessons Learned“, die in anderen Projekten wiederverwendet werden können und auch sollen.

4.6 Zusammenfassung

Projektmanagement

Projektmanagement definiert die Rahmenbedingungen für ein erfolgreiches Software-Entwicklungsprojekt und beginnt mit einer realistischen Projektdefinition und einer geeignete initialen Projektplanung begleitet das Projekt durch Techniken der Projektverfolgung und endet mit einem geordneten Projektabschluss.

Projektdefinition

Die Projektdefinition ist die Grundlage für eine realistische und realisierbare Projektplanung sowie für eine erfolgreiche Projektdurchführung. Ziel der Projektdefinition ist – aufbauend auf einer freigegebenen Projektidee (Projektvorschlag) – einen Projektauftrag zu erstellen. Diese Festlegungen sind die Basis für die Durchführung des Projekts. Der Projektauftrag beinhaltet unter anderem Ziele und Nicht-Ziele, Rollendefinitionen, Arbeitsstrukturen und Arbeitspakete.

Projektplanung

Im Rahmen der initialen Planung werden die Rahmenbedingungen für den weiteren Projektverlauf unter Berücksichtigung von Projektstrukturen, Ressourcen, Aufwänden und Budget festgelegt. Wichtig dabei ist es auch, die Arbeitspakete und etwaige Projektrisiken zu betrachten. Methoden und

Werkzeuge, wie PERT- und GANTT-Diagramme sowie Burn-down-Charts unterstützen den Projektleiter bei der Erstellung der initialen Planung.

Projektverfolgung

Um auf geänderte Situationen im Projekt geeignet reagieren zu können, ist eine laufende Verfolgung aller Aktivitäten im Projektverlauf notwendig. Im Rahmen der Projektverfolgung werden Daten zu Projektfortschritt und Projektaufwänden bzw. -kosten gesammelt. Damit wird auch festgestellt, ob der aktuelle Kurs zum gewünschten Projektziel führen kann oder ob Korrekturen erforderlich sind. Typische Korrekturen und Anpassungen können etwa bei Aufwands- und Terminschätzung, in der Priorisierung von Arbeitspaketen oder der Ressourcenzuteilung zu Arbeitspaketen auftreten.

Projektabschluss

Mit dem Projektabschluss wird das Projekt kontrolliert beendet. Ein kontrollierter Projektabschluss ist nicht nur bei erfolgreich beendeten Projekten sinnvoll, sondern sollte auch einem Projektabbruch stattfinden. In beiden Fällen ist das Ziel des Projektabschlusses die geordnete Übergabe der Projektergebnisse an den Auftraggeber, die Entlastung des Projektteams, die Abrechnung der Projektaufwände und die Erstellung eines Projektendberichts. Dieser Projektendbericht wird nicht nur zur Dokumentation des Projekts (als Zusammenfassung) sondern auch im Sinn von „Lessons Learned“ für weitere Projekte eingesetzt.

5 | Qualitätssicherung und Test-Driven Development

Die Überprüfbarkeit von Anforderungen ist ein wesentliches Kriterium in der modernen Software-Entwicklung. Man muss sich bei der Erhebung der Kundenanforderungen, der Definition der Architektur und der Festlegung der Realisierung der Software-Lösung bereits darüber Gedanken machen, wie Komponenten, Subsysteme und Systeme überprüft werden können. Dieses Kapitel widmet sich grundlegenden Konzepten der Qualitätssicherung in der modernen Software-Herstellung, wie Software Reviews, Inspektionen, Architekturreviews und Konzepten des Software-Testens. Ergänzend zur Testbarkeit von Anforderungen ist auch die Nachvollziehbarkeit von qualitätssichernden Maßnahmen und deren Ergebnissen relevant.

Neben den analytischen Maßnahmen, also der Feststellung, ob Komponenten der Spezifikation (*Verifikation*) oder dem Kundenwunsch (*Validierung*) entsprechen, sind konstruktive Maßnahmen sinnvoll, um das Qualitätsniveau der erstellten Lösung bereits während der Erstellung zu gewährleisten. *Test-Driven Development* ist ein bewährtes Konzept im Rahmen der komponentenorientierten Software-Herstellung um qualitativ hochwertige Produkte herstellen zu können. Eine geeignete Werkzeugunterstützung ist erforderlich, um Verifikations- und Validierungsschritte häufig durchführen zu können. Dadurch werden Konzepte, wie *Continuous Integration* und *Daily Builds* erst möglich und tragen zur Verbesserung der zu erstellenden Software-Lösung bei.

Übersicht

5.1 Der Qualitätsbegriff

Qualitätssicherung und Test-Driven Development

Die Herstellung qualitativ hochwertiger Software-Produkte umfasst einerseits eine *strukturierte Vorgehensweise*, wie sie beispielsweise durch ein Vorgehensmodell definiert werden kann (siehe Kapitel 3), *konstruktive Methoden* zur Herstellung der Produkte sowie *analytische Methoden* zur Überprüfung dieser Produkte. Die Qualitätssicherung umfasst dabei ein Bündel von Maßnahmen zur Sicherstellung der Produkt- oder Dienstleistungsqualität auf einem definiertem Niveau zu einem bestimmten Zeitpunkt im Entwicklungsprozess. In vielen Fällen wird ein derartiger Zeitpunkt im Projekt als Meilenstein bezeichnet, der unterschiedliche Phasen und Iterationen trennt.

Sichten auf Qualitätsbegriffe

Der Begriff *Qualität* hat eine lange Tradition und ist im heutigen Informationszeitalter – je nach Sichtweise – ein Schlagwort mit einer nicht klar abgegrenzten Definition. Beispielsweise kann Qualität im Rahmen der Software-Entwicklung aus unterschiedlichen Perspektiven betrachtet werden:

- Die *Zufriedenheit der Auftraggeber und Anwender* lässt Rückschlüsse auf die Software-Qualität, beispielsweise durch ein fehlerfrei lauffähiges Software-System, zu.
- *Erfüllung von Anforderungen, Attributen und Systemfunktionen* von funktionalen und nicht funktionalen Anforderungen und Produkteigenschaften.
- *Erfüllung von Vorgaben, Richtlinien, Standards und gesetzlichen Regelungen* für Produkte, Prozesse und Dienstleistungen.

Formale Definition

In der ISO 9000 ist eine formale Definition von Qualität zu finden [49]: *„Qualität ist die Gesamtheit von Merkmalen einer Einheit bezüglich ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen.“*

Kundenzufriedenheit

Die Kundenzufriedenheit ist ein zentrales Kriterium für die Definition von Qualität. Ist der Kunde mit der Lösung zufrieden, d. h., die Lösung entspricht seinen Vorstellungen und erfüllt die gestellten Erwartungen, wird dieser Erfüllungsgrad in der Regel mit einem „qualitativ hochwertigen“ Produkt in Verbindung gebracht. Diese Eigenschaften lassen sich mit Anforderungen und Qualitätsattributen (siehe Abschnitt 2.3) beschreiben und werden im Rahmen der Anforderungsanalyse gesammelt, analysiert und entsprechend in weiteren Phasen der Software-Entwicklung umgesetzt.

Überprüfbarkeit von Qualitätsattributen

Wesentlich dabei ist, dass diese Anforderungen nicht nur definiert und umgesetzt, sondern auch überprüft werden müssen. Methoden der Qualitätssicherung dienen der Überprüfung von Anforderungen.

Je nach Projekt und konkretem Vorgehen gibt es unterschiedliche Vorgehensweisen (siehe Kapitel 3) und unterschiedliche Ansätze für die Qualitätssicherung. Bei traditionellen und phasenorientierten Vorgehensweisen,

wie beispielsweise dem V-Modell, müssen Maßnahmen der Qualitätssicherung frühzeitig Fehler erkennen, da Fehler hohe Aufwände und Kosten verursachen können, je später sie erkannt werden. Beispiele für diese Maßnahmen sind Reviews, Inspektionen und traditionelle Testverfahren. Flexible und agile Prozesse unterstützen häufige Änderungen und beinhalten implizite Methoden der Qualitätssicherung, wie beispielsweise kontinuierliche Reviews oder Test-Driven-Development-Ansätze. Kritische Produkte, wie beispielsweise Architektur und Design, müssen allerdings auch bei agilen Ansätzen möglichst frühzeitig stabil bleiben, also empfiehlt sich auch hier der Einsatz von Reviews und Inspektionen.

Frühzeitige Fehlererkennung

Demnach ist beiden Prozessmodellfamilien gemeinsam, dass Fehler möglichst frühzeitig, idealerweise bei deren Entstehung, gefunden werden müssen. Fehler können sich negativ auf Entwicklungszeit und -kosten auswirken und bis zum Projektabbruch und zur Neuerstellung des Produkts führen, falls diese Fehler Architektur, Entwurf oder Design betreffen. Diese Auswirkungen spielen speziell bei traditionellen und phasenorientierten Prozessmodellen eine große Rolle, wie Untersuchungen in der industriellen Praxis zeigen [17]. Um derartige Abweichungen und Fehler in unterschiedlichen Entwicklungsstadien zu erkennen, gibt es verschiedene Methoden der Qualitätssicherung. Drei wesentliche Gruppen von Methoden und deren wichtigste Vertreter werden in diesem Kapitel vorgestellt.

Maßnahmen der Qualitätssicherung

- *Maßnahmen zur Erkennung von Fehlern in frühen Phasen der Entwicklung*. Da in frühen Phasen meist nur Textdokumente und Modelle vorliegen, sind geeignete Methoden zur Fehlersuche notwendig. Diese Methoden sind beispielsweise Reviews und Inspektionen (siehe Abschnitte 5.3 bis 5.5).
- *Maßnahmen zur Fehlerfindung in existierenden Teillösungen*. Wesentlich ist natürlich auch, Fehler in implementierten Komponenten, Subsystemen oder Gesamtsystemen zu finden und zu korrigieren. Methoden des traditionellen Software-Testens adressieren diese Maßnahmen (siehe Abschnitt 5.6).
- *Test-Driven Development*. Dieser konstruktive Ansatz, der im Rahmen der agilen Software-Entwicklung erhebliche Bedeutung erlangt hat, stellt die Definition und die Ausführung von Tests vor und parallel zur Implementierung in den Vordergrund und ermöglicht eine unmittelbare Rückkopplung während der Umsetzung (siehe Abschnitt 5.7).

5.2 Verifikation und Validierung

Verifikation und Validierung

Die Überprüfung von Anforderungen und Qualitätsmerkmalen ist für die Einschätzung der Produktqualität erforderlich. Diese Ergebnisse dienen zudem auch als Entscheidungsgrundlage für das Projektmanagement. Diese

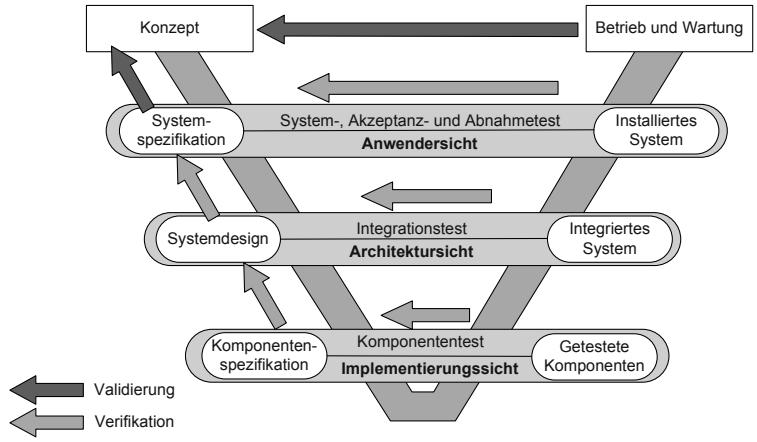


Abbildung 5.1
Verifikation und Validierung im V-Modell.

Entscheidungen können beispielsweise die Durchführung weiterer Iterationen oder die Durchführung weiterer Maßnahmen der Qualitätssicherung betreffen. Die Begriffe *Verifikation* und *Validierung* werden aber häufig synonym verwendet, obwohl sie gänzlich andere Zielsetzungen verfolgen.

Verifikation

Wie in der Analysephase des Lebenszyklusses in Abschnitt 2.3 beschrieben, werden in einem ersten Schritt die Anforderungen des Kunden erfasst und in einem zweiten Schritt in eine technische Spezifikation umgesetzt, die die Basis für die Umsetzung darstellt. Unter *Verifikation* versteht man die Prüfung eines Produkts im Hinblick auf dessen *Spezifikation*. Es wird also überprüft, ob das Ergebnis den spezifizierten und technischen Vorgaben entspricht. Die zentrale Fragestellung dabei ist, ob *das Produkt richtig entwickelt* wurde. Beispielsweise wird eine Komponente in einer Komponentenspezifikation festgelegt. Ein Komponententest stellt fest, ob die implementierte Lösung dieser Komponentenspezifikation entspricht. Die Verifikation umfasst also die Prüfung gegen die technische Spezifikation.

Validierung

Die Umsetzung gemäß Spezifikation bedeutet aber nicht immer, dass die Spezifikation auch dem Kundenwunsch entspricht. Beispielsweise können Fehler, ungenaue Aufgabenstellungen oder Missverständnisse dazu führen, dass Anforderungen falsch in die Spezifikation übernommen werden. Hier setzt die *Validierung* an. Sie dient primär der Überprüfung der erstellten Spezifikation oder in weiterer Folge der umgesetzten Lösung im Hinblick auf die Kundenanforderungen. Im Rahmen der Validierung wird die Frage beantwortet, ob *das richtige Produkt entwickelt* wurde, also ob die Lösung die erwarteten Anforderungen des Kunden erfüllt, und der Kunde das gewünschte Produkt erhält. Typischerweise finden sich Validierungsaktivitäten bei Akzeptanz- und Abnahmetests (Prüfung gegen die Anforderungen).

Speziell in flexiblen (agilen) Prozessen sind Verifikation und Validierung durch einen engen Kundenkontakt eng miteinander verbunden. Bei traditionellen Prozessen, die beispielsweise im öffentlichen Bereich nach dem V-Modell abgewickelt werden, ist diese Trennung durchaus sinnvoll. Das

V-Modell, das bereits in Abschnitt 3.3 vorgestellt wurde, bietet eine gute Grundlage, um die Einordnung von Verifikation und Validierung zu visualisieren (siehe Abbildung 5.1). Validierungsaktivitäten finden zwischen der fertigen Software-Lösung und den ursprünglichen Kundenanforderungen statt. Besonders wichtig ist ein Validierungsschritt zwischen den Anforderungen und der Spezifikation, da diese Dokumente die Grundlage für die weitere Entwicklung darstellen. Verifikationsschritte finden sich beispielsweise auf jeder Ebene des V-Modells zwischen der Spezifikationsphase und der Realisierungsphase, etwa aus Anwendersicht zwischen „Installiertem System“ und „Systemspezifikation“. Wichtige Verifikationsschritte sind auch im analytischen Zweig des V-Modells zwischen den einzelnen Ebenen zu finden, um die jeweils erstellten Produkte zu überprüfen. An dieser Stelle werden typischerweise Reviews und Inspektionen eingesetzt.

V-Modell

5.3 Software-Reviews

Software-Reviews

Die frühzeitige Fehlererkennung im Entwicklungsprozess ist die Grundlage für den Erfolg eines Software-Projekts. Software-Reviews sind qualitative Methoden und werden zur Verbesserung der Qualität sowohl von Prozessen als auch von Software-Produkten eingesetzt. Im Mittelpunkt steht dabei die Überprüfung eines Software-Produkts, eines Dokuments oder eines Programms gegenüber Vorgaben, Richtlinien oder Spezifikationen. Die Zielsetzung eines Reviews ist es, Fehler in einem konkreten Produkt zu finden. Die IEEE 610 definiert ein Review als *„ein formelles Treffen, bei dem ein Produkt oder Dokument dem Benutzer, Kunden oder anderen interessierten Personen vorgelegt wird, um es zu kommentieren und abzusegnen“* [46].

Universelle Einsetzbarkeit

Ein wesentlicher Vorteil von Reviews ist deren universelle Anwendbarkeit. Die Durchführung von Reviews ist nicht auf spezielle Artefakte oder Phasen beschränkt, sondern für jedes Artefakt in jeder Phase und in jeder Iteration möglich und auch empfehlenswert. Dementsprechend existieren zahlreiche unterschiedliche *Reviewtypen* mit unterschiedlichen Zielsetzungen (siehe Abschnitt 5.3.1). Reviews werden in der Regel in Teams mit definierten *Rollen* (siehe Abschnitt 5.3.2) durchgeführt und folgen einem *definierten Prozess* (siehe Abschnitt 5.3.3). Diese unterschiedlichen Aspekte werden in den folgenden Kapiteln behandelt.

5.3.1 Reviewtypen

Arten von Reviews

Je nach Anwendungsbereich existieren zahlreiche verschiedene Ansätze von Reviews, die im Entwicklungsprojekt zum Einsatz kommen können. Abbildung 5.2 gibt einen Überblick über wichtige Ausprägungen von Reviews [89]. Wesentliche Unterscheidungskriterien umfassen einerseits die *Wahl eines definierten Vorgehensmodells* und die Integration eines geeigneten Reviewtyps und andererseits den *Grad der Einbeziehung des Kunden*.

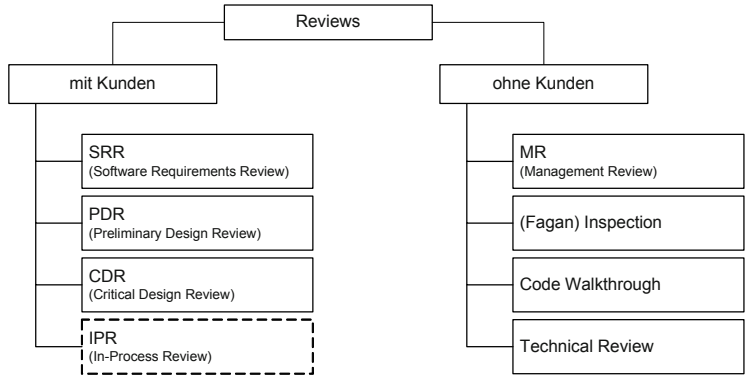


Abbildung 5.2 *Arten von Reviews [89].*

Bei einer phasenorientierten Vorgehensweise finden Reviews typischerweise am Ende der jeweiligen Phase mit oder ohne Kunden statt, bei flexiblen und agilen Prozessen haben sich *kontinuierliche* Reviews mit dem Kunden bewährt. Unabhängig davon verfolgen die unterschiedlichen Reviewarten auch unterschiedliche Ziele, die sich im Folgenden auf eine phasenorientierte Vorgehensweise beziehen, aber auch – von der Grundidee – für agile Prozesse anwendbar sind.

Software Requirements Review

Ein *Software Requirements Review (SRR)* verfolgt das Ziel, nach der Definition der Anforderungen und vor dem Entwurf ein Review durchzuführen, um alle Unklarheiten sowohl seitens des Kunden als auch seitens des Entwicklerteams auszuräumen. Dementsprechend soll der Kunde an diesem Review teilnehmen. Nach dem SRR müssen die Ziele des Projekts und die Anforderungen allen beteiligten Personen bekannt sein. Im Anschluss daran startet meist die Entwurfs- und Designphase.

Preliminary Design Review

Im Rahmen der Entwurfsphase finden *Preliminary Design Reviews (PDR)* statt, um den Entwurf der Software-Architektur und die Grobstruktur des Produkts zu überprüfen, bevor die Entwürfe im Detail ausgearbeitet werden. Dadurch wird sichergestellt, dass alle wesentlichen Anforderungen berücksichtigt und umgesetzt werden können.

Critical Design Review

Bei besonders kritischen Komponenten kann ein *Critical Design Review (CDR)* sinnvoll sein, um den Entwurf und das Design dieser Komponenten vor der Implementierung erneut gründlich zu analysieren. Speziell in phasenorientierten Prozessmodellen mit einer strikt sequenziellen Vorgehensweise ist das in der Regel der letztmögliche Zeitpunkt, an dem der Kunde das Projekt noch stoppen kann. Bei agilen Prozessen kann das Projekt nach jeder Iteration beendet werden. In der Praxis werden PDR und CDR meist in einem Reviewzyklus zusammengefasst.

In-Process Review

Sehr große und komplexe Projekte, deren Implementierung sich über einen besonders langen Zeitraum erstreckt, können sogenannte *In-Process Reviews (IPR)* erforderlich machen, um dem Kunden Fortschritte in der Implementierung, Prototypen oder auch Testfälle zur Begutachtung vorzule-

gen. Die kontinuierlichen Reviews agiler Vorgehensweisen entsprechen im Wesentlichen diesem Reviewtyp.

In der Praxis sind auch Reviewtypen zu finden, die für interne Bewertungen der Produkt- und Projektqualität eingesetzt werden und in vielen Fällen ohne Kundenbeteiligung stattfinden können. Dazu gehören etwa *Management-Reviews*, *technische Reviews* oder *Inspektionen* und *Code-Walkthrough*.

Management-Reviews

Management-Reviews dienen der formellen Bewertung des Projekts und der Erhebung des aktuellen Projektstatus. Werden Abweichungen festgestellt, müssen geeignete steuernde Maßnahmen eingeleitet werden (siehe auch Projektverfolgung in Abschnitt 4.4). Meist werden Management-Reviews im Projektplan zu definierten Zeitpunkten eingeplant und orientieren sich an Phasen und Iterationen, können aber auch bei außerplanmäßigen Ereignissen, beispielsweise bei geänderten Projektzielen, angesetzt werden.

Technisches Review

Innerhalb des Projekts dienen *technische Reviews* dazu, einen konkreten Teil der Software zu überprüfen und zu bewerten. Weitere Aufgaben von technischen Reviews beinhalten etwa die Überprüfung der Realisierung im Hinblick auf Spezifikationen oder Standards. Auch die Fehlersuche in bestehenden Software-Produkten kann im Fokus eines technischen Reviews sein. Diese Reviews können je nach Bedarf eingeplant werden oder als integraler Bestandteil des Entwicklungsprozesses vorgesehen sein.

Integrierter Review-Prozess

Beispielsweise ist die Verwendung von Sourcecode-Managementsystemen (SCM) zur Unterstützung der kollaborativen Software-Entwicklung eine gängige Praxis in der modernen Software-Entwicklung (siehe Abschnitt 10.2). Eine wichtige Frage ist, wie geänderte Codeelemente oder Artefakte in das Repository übernommen werden sollen. Beispielsweise kann ein integrierter Reviewprozess vorgesehen werden, der die Übernahme in das Haupt-Repository erst zulässt, wenn der Sourcecode und die Artefakte zuvor, beispielsweise durch den Hauptentwickler oder Reviewer, überprüft wurden. Dazu wird durch den Entwickler ein Klon des Haupt-Repositories angelegt, der dann die entsprechenden Änderungen oder Erweiterungen vornimmt. Nach Abschluss dieser Änderungen wird dem Hauptentwickler oder Reviewer ein sogenannter „Pull Request“ – eine Anforderung zur Überprüfung der Änderungen – geschickt. Nach dem Review und der Freigabe werden der geänderte Sourcecode bzw. die Artefakte in das Haupt-Repository übernommen. Details zu diesem Ansatz werden in Abschnitt 10.2 erläutert.

Inspection und Code Walkthrough

Jetzt stellt sich noch die Frage, wie ein Review konkret durchgeführt werden kann. Eine Möglichkeit zur Beurteilung von Artefakten oder Sourcecode sind beispielsweise *Inspection* und *Code Walkthrough*. Diese Methoden sind spezielle und stark formalisierte Reviewtypen mit der Zielsetzung, Abweichungen und Fehler in einem Produkt zu finden. Aufgrund der systematischen Vorgehensweise können sie auch zur formellen Nachweisführung oder für Schulungszwecke eingesetzt werden. Aufgrund der Bedeutung von

Software-Inspektionen wird dieser Themenbereich in Abschnitt 5.4 im Detail vorgestellt.

5.3.2 Rollen in einem Review

Rollen in einem Review

Bei den Reviewtypen wurden bisher nur die Rollen *Kunden* und sinngemäß *Entwickler* unterschieden. Es ist jedoch wichtig, dass innerhalb einer Reviewsitzung spezifische Rollen definiert werden, um einen effizienten Ablauf des Reviews gewährleisten zu können. In Reviewteams sind beispielsweise folgende Rollen zu finden, die individuelle Aufgaben wahrnehmen. Je nach Review kann eine Person auch mehrere Rollen wahrnehmen:

Rollenverteilung in einem Review

- *Moderator (keeper of process)*. Der Moderator ist – als Leiter des Reviews – für den Ablauf und die Organisation zuständig. Nach einem abgeschlossenem Review ist er auch für Auswertungen und später für die richtig durchgeführte Korrektur der gefundenen Fehler verantwortlich.

- *Leser (keeper of focus and pace)*. Der Leser ist verantwortlich für die Aufbereitung des Reviewobjekts während der eigentlichen Reviewsitzung. Durch seine Präsentation des Reviewobjekts kann er den Ablauf eines Reviews steuern und den Fokus auf wesentliche Aspekte legen. Diese Steuerung umfasst auch die angemessene Geschwindigkeit des Reviews. Ein zu schnelles Review birgt die Gefahr, kritische Fehler zu übersehen, ein zu langsames Review erzeugt unnötigen Mehraufwand ohne zusätzliche Nutzen.

- *Schreiber (preserver of knowledge)*. Der Schreiber fungiert als Berichterstatter über den Verlauf des Reviews. Er muss alle Ergebnisse und gefundenen Fehler mitprotokollieren und erstellt daraus ein Reviewprotokoll, das als Basis für die Nachbearbeitung des Review (*Rework*) dient.

- *Gutachter (Reviewer)*. Der Gutachter hat die Aufgabe, sich anhand der vom Moderator erhaltenen Unterlagen auf das Review vorzubereiten (*Preparation*) und sich während des Reviews aktiv an dem Diskussionsprozess zu beteiligen. Seine zentrale Aufgabe ist es, das Reviewobjekt kritisch zu hinterfragen und Fehler zu identifizieren, die dann vom Schreiber protokolliert werden können. Die Hauptzielsetzung ist es, Fehler zu erkennen und nicht, Fehler zu beseitigen oder alternative Lösungen zu finden.

- *Autor (author)*. Der Autor ist der Urheber des Reviewobjekts oder ein Repräsentant des Erstellerteams. Nimmt der Autor am Review teil, hat er die Aufgabe, auftretende Fragen zu klären und Hintergrundinformationen bereitzustellen. Die aktive Teilnahme an dem Review oder

Rechtfertigungen der Lösung sind keine Aufgaben des Autors im Rahmen einer Reviewsitzung. In der Praxis wirkt der Autor daher nur selten an der Reviewsitzung mit.

Größe von Reviewteams

Neben der Rollenverteilung ist auch die richtige Anzahl der Teilnehmer für ein erfolgreiches und effizientes Review von großer Bedeutung. Ein zu großes Reviewteam erhöht zwar den Aufwand, bringt aber nur ein Minimum an zusätzlichem Nutzen bzw. an zusätzlichen gefundenen Fehlern. Als Richtgröße schlägt die IEEE eine Größe des Reviewteams von drei bis sechs Personen vor [44]. In der betrieblichen Praxis sind meistens Reviewteams mit vier bis fünf Personen zu finden.

5.3.3 Ablauf eines Reviews

Ablauf eines Reviews

Der Reviewprozess läuft in kontrolliertem Umfeld unter der Leitung eines Moderators ab und gliedert sich in definierte Phasen, die typischerweise sequenziell durchlaufen werden. Je nach Projekt werden Reviews fix im Projektplan eingeplant, können aber auch – sofern es im Projektverlauf notwendig ist – anlassbezogen durchgeführt werden.

Planung eines Reviews

Bei der Planung (*planning phase*) eines Reviews müssen sogenannte Arbeitspakete durch den Autor oder das Autorenteam angefertigt werden. Diese Arbeitspakete beinhalten, je nach Objekttyp, den Prüfling (*Reviewobjekt*), notwendige Referenzunterlagen bzw. Richtlinien und Vorgaben sowie Checklisten für die Durchführung des Reviews. Die Aufgabe des Moderators ist es, für die Bereitstellung der Arbeitspakete zu sorgen und sowohl das Reviewteam als auch den zeitlichen und örtlichen Rahmen zu bestimmen.

Initialisierung

Für jedes Review muss es klar definierte Eingangskriterien geben, die ein Software-Produkt erfüllen muss, bevor mit der Durchführung des Reviews begonnen werden darf. Solche Eingangskriterien können beispielsweise fordern, dass der Softwarecode kompilierbar ist oder ein Textdokument Korrektur gelesen wurde. In der Initialisierungsphase (*initialization phase*) sorgt der Moderator dafür, dass der Prüfling – als Bestandteil des Arbeitspakets – die notwendigen Eingangskriterien erfüllt. Weiters muss dabei beachtet werden, dass ein Aspekt jeweils von mindestens zwei Reviewern (*Gutachtern*) betrachtet wird. Je nach Komplexität des Prüflings kann es auch notwendig sein, eine Einführungssitzung durchzuführen, bei der sowohl der Prüfling als auch die Unterlagen präsentiert werden. Das empfiehlt sich speziell bei neuartigen oder komplexen Prüflingen.

Vorbereitung

Nach der Festlegung und Bekanntgabe der Daten für die Reviewsitzung (Arbeitspakete, Aufgaben- und Rollenverteilung, Ort und Termin) müssen sich alle Reviewteilnehmer auf das Review vorbereiten. In dieser Vorbereitungsphase (*preparation phase*) untersuchen die Reviewer den Prüfling (das Prüfobjekt) entsprechend den ihnen zugeteilten Rollen und Aspekten und markieren gefundene Mängel. Dadurch wird die Effizienz

des Reviews bei der eigentlichen Reviewsitzung gesteigert. Beispiele für Mängel in Dokumenten (z. B. in Anforderungsdokumenten) können Abweichungen von Richtlinien und Vorgabedokumenten, Rechtschreibfehler, falsche oder fehlende Begriffsdefinitionen, widersprüchliche, falsche und fehlende Informationen und inhaltliche Mängel sein. Mängel in Codedokumenten sind beispielsweise Verletzungen der Namenskonventionen, die Nicht-Einhaltung von Kommentarrichtlinien, fehlgeschlagene Kompilierung sowie strukturelle oder inhaltliche Mängel. Die einzelnen Reviewer sollten also vorbereitet zur Reviewsitzung kommen, um diese möglichst effizient durchführen zu können.

Reviewsitzung

Der Moderator leitet die *Reviewsitzung* und übt die Rolle des Diskussionsleiters aus. Während der Reviewsitzung präsentiert der Leser das Software-Dokument (den Prüfling) und die entsprechenden Punkte auf der Checkliste. Jeder Teilnehmer bringt die in der Vorbereitungsphase gefundenen Mängel ein, die vom Schreiber protokolliert werden. Am Ende der Reviewsitzung beschließen die Reviewer gemeinsam, ob und welche Nacharbeiten am Prüfling notwendig sind. Sind spezielle Auflagen zu erfüllen oder zu viele Änderungen durchzuführen, kann ein erneutes Review *(Folgereview)* notwendig sein. Bei geringfügigen Änderungen entscheidet der Moderator nach der Überarbeitung über die Freigabe. Eine derartige Reviewsitzung soll üblicherweise nicht länger als zwei Stunden dauern, da dann die Aufmerksamkeit der Teammitglieder stark abnimmt. Bei Bedarf (z. B. komplexe, umfangreiche oder risikobehaftete Software-Produkte) können mehrere Reviewsitzungen durchgeführt werden.

„Die dritte Stunde“

Das Ziel eines Reviews ist die Fehlerfindung und nicht die Korrektur oder die Diskussion von Lösungsmöglichkeiten. Um die Möglichkeit der Diskussion zu schaffen, wird gegebenenfalls nach Beendigung der Reviewsitzung noch eine *dritte Stunde* angehängt. Dieser Zeitraum ist für Diskussionen, Lösungsvorschläge und Feedback zum Reviewprozess gedacht. Dieses Meeting ist allerdings kein zwingender Bestandteil eines Reviewprozesses.

Reviewbericht

Nach der Reviewsitzung erstellt der Schreiber den *Reviewbericht* (*reporting*) in Abstimmung mit den Reviewteilnehmern. Ziel dieses Schritts ist einerseits die Protokollierung des Reviews und andererseits die Nachweisführung im Hinblick auf die Nachvollziehbarkeit der getroffenen Entscheidungen.

Nacharbeit und Abschluss

Der Autor des Dokuments nimmt die beschlossenen Änderungen oder sonstigen Nacharbeiten (*rework*) vor und informiert die Reviewteilnehmer über den Abschluss der Änderungen. Der Moderator überprüft die korrekte Durchführung der Nacharbeiten. Wurde das Dokument im Zuge der Nacharbeiten signifikant verändert oder im Rahmen des Reviews ein Folgereview beschlossen, wird eine weitere Reviewsitzung durch den Moderator geplant und einberufen. Abschließend werden sämtliche im Rahmen des Reviews erstellten Dokumente als Reviewpaket zusammengefasst und archiviert.

5.4 Software-Inspektionen

Software-Inspektion

Inspektionen sind formale, effiziente und wirtschaftliche Methoden, um Fehler im Design und Code zu finden [44]. Inspektionen sind grundsätzlich dafür geeignet, Fehler in allen Software-Produkten, beispielsweise in Textdokumenten, Modellen, aber auch in Sourcecode, zu finden. Inspektionen können etwa mit dem *Blick durch ein Mikroskop* auf das Software-Produkt verglichen werden [34].

Ähnlichkeit zu Reviews

Software-Inspektionen sind, als spezielle Ausprägung von Reviews, ebenfalls gut dazu geeignet, um Fehler frühzeitig im Entwicklungsprozess zu finden. Wesentliche Unterschiede zu Reviews sind beispielsweise (a) ein höherer Formalisierungsgrad, (b) eine genau definierte Vorgehensweise sowie (c) eine gezielte Unterstützung der Reviewer durch sogenannte Lesetechniken. Durch eine formalere Vorgehensweise können Ergebnisse aus der Inspektion gut zur Nachweisführung über die „Qualität" eines Produkts im Hinblick auf Normen und Standards bzw. die Spezifikation eingesetzt werden, wie es beispielsweise für sicherheitskritische Anwendungen erforderlich ist. Diese Vorgehensweise beinhaltet aber auch Richtlinien, wie ein Software-Dokument *gelesen* werden soll, um etwa bestimmte Fehlerklassen zu finden. Diese Richtlinien werden als *Lesetechniken* bezeichnet. Diese Festlegungen können – sofern sie im Projekt erforderlich sind – auch für Reviews verwendet, angepasst und eingesetzt werden.

5.4.1 Inspektionstypen und Ablauf

Inspektionsvarianten

Die *Fagan Inspection* [26] ist die ursprüngliche Form von Inspektionen und bildet die Basis für die meisten anderen Inspektionstechniken. Generell ist sowohl der Ablauf als auch die Rollenverteilung ähnlich wie bei den Reviews (siehe dazu Abschnitt 5.3), wobei die Vorgehensweise formaler geregelt ist. Auch die Größe eines Inspektionsteams orientiert sich mit empfohlenen drei bis sechs Personen am Reviewprozess. Aufgrund unterschiedlicher Anforderungen haben sich zahlreiche Varianten herausgebildet. Eine *Two-Fold Inspection* [11] umfasst beispielsweise zwei Personen und wird etwa in kleinen Projekten eingesetzt, um mit geringem Ressourcenaufwand die Qualität der Produkte zu verbessern. Häufig wird diese Art der Inspektion als Vorstufe zu umfangreicheren Inspektionsmethoden eingesetzt. Eine *N-fold Inspektion* [67] geht von der Annahme aus, dass mehrere kleine parallele Inspektionen eine höhere Anzahl und andere Arten von Fehlern finden (also effektiver sind). Obwohl der Aufwand für parallele Teams auch entsprechend höher ist, kann diese Methode bei sicherheitskritischen Anwendungen sinnvoll sein. Je nach Anwendungsbereich sind andere Varianten von Inspektionen denkbar. Unabhängig von der Größe eines Inspektionsteams folgen Inspektionen aber demselben formalen Prozess [64].

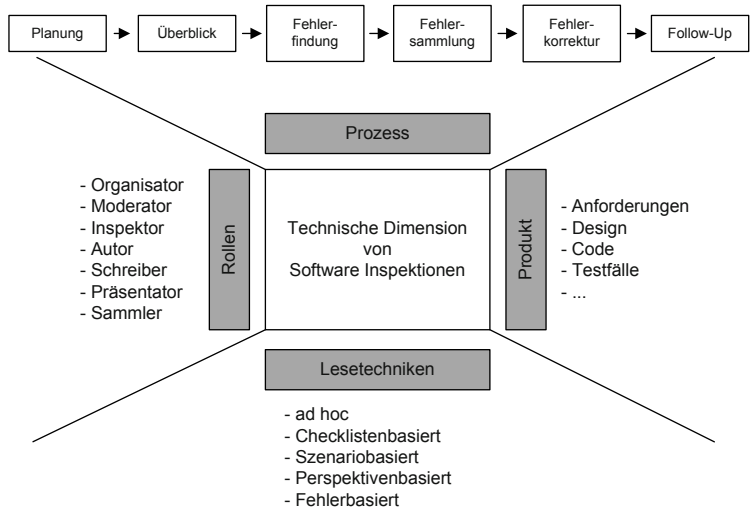


Abbildung 5.3
Technische Dimension von Inspektionen [64].

Die Abbildung 5.3 illustriert den Inspektionsprozess, definiert die wesentlichen Rollen und die Anwendbarkeit der Inspektionstechniken. Weiterhin sind in der Abbildung ausgewählte Lesetechniken dargestellt, die in Abschnitt 5.4.2 genauer beschrieben werden.

Inspektionsablauf

Die *Planungsphase* zielt darauf ab, den eigentlichen Ablauf der Inspektion festzulegen, und ist, wie bereits beim Reviewprozess erwähnt, die Aufgabe des Moderators. Nachdem die Eingangsbedingungen erfüllt sind, muss das Inspektionsteam gebildet werden. Nach Fertigstellung der Inspektionsplanung werden die Arbeitspakete an die Gruppenmitglieder verteilt. Beim ersten Treffen *(kick-off-meeting)* gibt der Autor einen *Überblick* über die zu untersuchenden Software-Dokumente [34]. Diese Phase ist dann sinnvoll, wenn (a) das zu untersuchende Produkt sehr komplex und nur sehr schwer zu verstehen ist oder (b) es ein Teil eines großen Software-Systems ist. In beiden Fällen werden die Zusammenhänge durch den Autor vorgestellt [63].

Fehlerfindung und -sammlung

Das Ziel der Fehlerfindung ist der eigentliche Kernpunkt der Software Inspektion. Unter *Fehlerfindung* versteht man die Tätigkeit, das zu inspizierende Software-Dokument zu untersuchen und im Hinblick auf Fehler zu bearbeiten, das bedeutet, gefundene Fehler zu dokumentieren. Der folgende Schritt umfasst die *Fehlersammlung*, die je nach Inspektionsansatz individuell oder als Teammeeting durchgeführt werden kann. Durch die Fehlersammlung sollen nur *tatsächliche* Fehler berücksichtigt und vermeintliche Fehler eliminiert werden. Die Fehlersammlung erfolgt in vielen Fällen im Rahmen eines Teammeetings, in dem eine sogenannte *Teamfehlerliste*, also eine Liste mit Fehlern aus der Sicht des gesamten Inspektionsteams, erstellt wird. Ein alternativer Ansatzpunkt sind sogenannte *nominelle Teamfehlerlisten*, die alle individuellen Fehlerlisten zu einer gesamten Teamfehlerliste zusammenfassen, ohne ein Teammeeting durchzuführen. Den Vorteilen, dass kein individuell gefundener Fehler verloren geht und kein zusätzli-

cher Aufwand für die Durchführung eines Teammeetings erforderlich ist, stehen die Nachteile gegenüber, dass (a) auch keine zusätzlichen Fehler gefunden werden können (beispielsweise durch die Diskussion innerhalb des Teams) und (b) fälschlicherweise notierte Fehler nicht eliminiert werden können [8].

Fehlerkorrektur und Follow-up

Im Rahmen der *Fehlerkorrektur* passt der Autor sein Software-Dokument aufgrund der erhaltenen Teamfehlerliste an und legt es dem Moderator zur Überprüfung und Freigabe vor. Bei sehr vielen oder komplexen Änderungen oder bei einer sehr hohen Anzahl an gefundenen Fehlern kann ein *Follow-up* des Inspektionsprozesses beschlossen werden. Das ist eine optionale Möglichkeit, um die Veränderungen des Autors noch einmal von dem Inspektionsteam überprüfen zu lassen. Im Rahmen des Follow-up wird entschieden, ob das Software-Produkt freigegeben werden kann, oder ob ein zweiter informeller Inspektionsprozess – eine *Re-Inspektion* – angestoßen werden muss.

5.4.2 Lesetechniken

Lesetechniken unterstützen den Review- und Inspektionsprozess

Auch die Fehlersuche wird in kontrollierter Weise durchgeführt, um beispielsweise systematisch alle Fehlerarten abdecken zu können. Lesetechniken *(Reading Techniques)* unterstützen die Inspektoren bei der gezielten Suche nach Fehlern in den Software-Produkten. Eine Lesetechnik ist dabei als eine Reihe von Schritten bzw. als systematische Vorgehensweise zu verstehen, die dem Inspektor als Anleitung für die Bearbeitung des Software-Produkts dient. Durch die Anwendung einer Lesetechnik erhält der Inspektor einen besseren Zugang und ein tieferes Verständnis des zu betrachtenden Software-Produkts und ist somit in der Lage, Fehler effizienter und effektiver zu finden [64] [79]. Betrachtet man unterschiedliche Lesetechniken, die in weiterer Folge kurz beschrieben werden, lassen sie sich durch folgende Charakteristika beschreiben:

Charakteristika von Lesetechniken

- *Anwendungsspektrum*. Eine Lesetechnik muss für die jeweilige Anwendung passen und auf das zu untersuchende Dokument abgestimmt sein. Abhängig vom Aufbau und der Struktur werden spezifische Fehler in Dokumenttypen bevorzugt gefunden, beispielsweise in Modellen oder Designdokumenten.
- *Anleitung (Guidelines)* definieren eine konkrete Vorgehensweise beim „systematischen“ Lesen und müssen ebenfalls auf den Anwendungsbereich abgestimmt sein. Beispielsweise sind bei verteilten Systemen andere Schwerpunkte zu setzen als bei Echtzeitsystemen.
- Die *Anpassbarkeit* bezeichnet die Fähigkeit, dieselbe Lesetechnik auf verschiedene Produkte anzuwenden. Typische Vertreter dafür sind generische Checklisten, die unabhängig von Software-Produkt und Anwendungsbereich eingesetzt werden können. Beispielsweise kann in

einer generischen Checkliste gefordert sein, Inkonsistenzen in der Interfacedefinition zu finden ("Werden Attribute in Interfacedefinitionen konsistent, in der richtigen Reihenfolge und mit demselben (richtigen) Datentyp verwendet?").

- Die *Wiederholbarkeit und Nachvollziehbarkeit* sind wesentliche Kriterien bei der Anwendung einer Lesetechnik. Beispielsweise sollte die Anwendung einer bestimmten Lesetechnik – unabhängig von den einzelnen Inspektoren – vergleichbare Fehler aufdecken.

- *Abdeckung und Überdeckung*. Lesetechniken (oder eine Kombination unterschiedlicher Lesetechniken) sollen möglichst alle Fehlertypen abdecken und finden können. Bei der Auswahl der einzusetzenden Lesetechniken ist es notwendig, dass verschiedene Fehler gefunden werden und möglichst wenige Überlappungen auftreten.

Strategie zur Fehlerfindung

Lesetechniken definieren eine konkrete Vorgehensweise, wie die Inspektoren ein Dokument bearbeiten. Sie legen also eine Strategie zur Fehlerfindung fest. Diese Vorgehensweise ist typischerweise in *Checklisten* und *Guidelines* festgelegt. Im Rahmen der Planung einer Inspektion ist es die Aufgabe des Projektleiters und Moderators, geeignete Lesetechniken für den jeweiligen Anwendungsfall auszuwählen und entsprechend anzupassen. In diesem Abschnitt werden die grundlegenden Merkmale ausgewählter Lesetechniken definiert. Illustrierende Beispiele sind auf der begleitenden „Best-Practice Software-Engineering" Projekt Webseite [1] zu finden.

Ad-hoc

Die einfachste „Lesetechnik" ist die *Ad-hoc-Methode*, bei der ohne explizite Leseanleitung unsystematisch nach Fehlern gesucht wird. Diese Technik ist etwa vergleichbar mit dem Lesen eines Buches: beispielsweise können Leser ein Buch vom Anfang bis zum Ende sequenziell lesen, einzelne Kapitel oder Abschnitte herausgreifen oder punktuell nach Informationen suchen. Die Ad-hoc-Methode ist zwar auf alle Software-Produkte anwendbar, allerdings leidet die Wiederholbarkeit und die Qualität der Ergebnisse durch den unsystematischen Zugang der Inspektoren. Die Ergebnisse hängen auch stark von individuellen Erfahrungen ab.

Checklisten

Eine Verbesserung lässt sich durch den Einsatz von Checklisten erzielen, wie sie durch die *checklistenbasierte Lesetechnik (Checklist-Based Reading, CBR)* realisiert ist. Eine Checkliste beinhaltet dabei eine Reihe von Aspekten (etwa Vollständigkeit und Konsistenz), die für das zu inspizierende Objekt (den Prüfling) relevant sind. Beispielsweise beinhaltet eine Checkliste für die Inspektion eines Projektauftrags alle relevanten Themenbereiche, die im Projektauftrag zu finden sein müssen. Der Inspektor nimmt sich diese Checkliste und den Prüfling (also den Projektauftrag) und untersucht, ob der Prüfauftrag tatsächlich alle Elemente gemäß Checkliste bein-

[1] `http://bpse.ifs.tuwien.ac.at/`

haltet. Abweichungen werden dokumentiert. Je nach Anwendungsbereich (beispielsweise Projektauftrag, Anforderungsdokument, Spezifikation oder Softwarecode) muss die Checkliste auf den zu untersuchenden Prüfling abgestimmt werden.

Die generelle Vorgehensweise bei CBR-Inspektionen besteht im Wesentlichen aus den folgenden Schrittfolgen:

1. Gesamtüberblick über das Inspektionspaket (wie Checklisten, Prüfling, unterstützendes Material).
2. Auswahl des ersten noch nicht bearbeiteten Checklistenpunktes.
3. Untersuchung der Prüfung auf Abweichungen im Hinblick auf den ausgewählten Aspekt der Checkliste und Dokumentation von Abweichungen als „Fehler“.
4. Fortsetzung bei Schritt 2, solange es noch offene Checklistenpunkte gibt. Sind alle Punkte abgearbeitet, Fortsetzung bei 5.
5. Nach Abarbeitung aller Checklistenpunkte ist die Inspektion abgeschlossen, und die Fehlerliste liegt vor.

Durch die Fokussierung auf einen Checklistenpunkt wird erreicht, dass sich der Inspektor immer auf einen eingeschränkten Fehlerbereich oder Bereich des Dokumentes konzentrieren kann. Im Gegensatz zum Ad-hoc-Lesen weiß der Reviewer anhand des Checklistenpunkts genau, wo der aktuelle Schwerpunkt liegt.

Allgemein gilt aber, dass Checklisten zum Inspektionsobjekts passen müssen, um eine effiziente Fehlersuche zu ermöglichen. Das heißt aber auch, dass sie für jede neue Objektart, neue Fehlertypen bzw. für andere Anwendungsdomänen entweder neu erstellt oder anhand von Checklisten aus vergangenen Projekten entwickelt werden müssen. In jedem Fall muss man vor dem Einsatz einer Checkliste überprüfen, ob sie für die geplante Inspektion passend ist, sonst läuft man Gefahr, kritische Fehler nicht zu erkennen. Checklistenbasierte Lesetechniken eignen sich daher besonders gut für ähnliche Inspektionsobjekte.

Eine zentrale Frage ist auch die Länge einer Checkliste. Da eine formelle Inspektion zwei Stunden nicht überschreiten soll, muss das Inspektionspaket und damit auch die Checkliste in diesem Zeitrahmen bearbeitbar sein. Sowohl zu lange Checklisten als auch zu umfangreiche Prüflinge können nicht vollständig bearbeitet werden. In der Praxis finden sich daher Checklisten mit maximal einer A4 Seite [34].

Fehlerbasiertes Lesen

Lesetechniken unterstützen den Inspektor bei der Fehlersuche. Demnach ist es naheliegend, speziell unterschiedliche Fehlertypen, etwa inkonsistente, fehlende oder falsche Informationen, Notationsfehler, Abweichungen

von Namenskonventionen und Dokumentationsrichtlinien, aber auch fehlerhafte Benutzerabläufe oder logische Fehler zu adressieren. Bei der *fehlerbasierten Lesetechnik* werden die Checklisten und Guidelines so aufgebaut, dass diese unterschiedlichen Fehlerkategorien beispielsweise in Anforderungsdokumenten einfacher gefunden werden können. Für jede dieser Fehlergruppen werden spezielle Fragestellungen und – im Unterschied zu checklistenbasiertem Lesen – konkrete Vorgehensweisen und Szenarien zur Unterstützung der Fehlersuche entwickelt. Die Inspektoren beantworten diese Fragestellungen, indem sie der definierten Vorgehensweise bzw. dem Szenario folgen. Ein Szenario kann beispielsweise einen konkreten Benutzerablauf beschreiben.

Perspektiven und Szenarien

Unterschiedliche Sichtweisen spielen auch bei den auf *perspektiven- und szenariobasierten Lesetechniken* eine zentrale Rolle. Anders als bei der fehlerbasierten Lesetechnik stehen hier die individuellen Interessen und Sichtweisen der beteiligten Stakeholder (beispielsweise durch Kunden, Designer oder Tester) im Vordergrund [3]. Hintergrund dabei ist die Annahme, dass individuelle Sichtweisen unterschiedliche Fehlerklassen aufdecken. Beispielsweise fokussiert ein Tester andere Schwerpunkte, etwa die Testbarkeit eines Inspektionsobjekts, während ein Kunde eher die Anwendbarkeit betrachtet. Durch diese unterschiedlichen Sichtweisen sind keine großen Überlappungen bei den gefundenen Fehlern zu erwarten, d. h. nur wenige Fehler (die in den betrachteten Perspektiven eine Rolle spielen) werden mehrfach gefunden. Durch die sorgfältige Auswahl der relevanten Perspektiven kann verhindert werden, dass einzelne Fehlertypen nicht gefunden werden, also von keiner Perspektive adressiert werden.

Usage-Based Reading

Die bisher besprochenen systematischen Lesetechniken zielen auf Fehler aus unterschiedlichen Perspektiven ab, ohne deren Einfluss und Wichtigkeit auf die Geschäftsfälle zu betrachten. Bei der Anwendung von *verwendungsbasiertem Lesen (Usage-Based Reading, UBR)* konzentriert sich der Inspektor auf Fehler, die die Qualität des Endprodukts aus Benutzersicht am stärksten (negativ) beeinflussen. Beispielsweise haben Fehler in wichtigen Anwendungsfällen eine deutlich höhere Auswirkung als Fehler in weniger wichtigen Anwendungsfällen. Um diese Fehler zu adressieren, bestimmt man konkrete Anwendungsfälle (*Use Cases*), vergibt entsprechende Prioritäten und sortiert sie nach dieser Priorität. Ziel ist es, vor Beginn der eigentlichen Inspektion eine priorisierte Liste mit Use Cases zur Verfügung zu haben. Diese priorisierten Anwendungsfälle stehen etwa in einem SCRUM Produkt-Backlog (siehe Abschnitt 3.8) oder einer Iceberg-Liste (siehe Abschnitt 4.3) zur Verfügung. Der Inspektor beginnt also mit dem wichtigsten Anwendungsfall und sucht nach vordefinierten Kriterien (*Guidelines*) gezielt nach Fehlern. Nach Abschluss des wichtigsten Anwendungsfalls wird die Liste entsprechend den priorisierten Anforderungen systematisch weiter abgearbeitet. Falls aus Zeitmangel oder anderen Gründen nicht alle Anwendungsfälle bearbeitet werden können, sind zumindest die erfolgskritischen Anforderungen abgedeckt. Die verwendungsbasierte Lesetechnik (UBR) weist eine gewisse Ähnlichkeit mit der Kun-

denperspektive der perspektivenbasierten Lesetechnik (PBR) auf, da diese im Normalfall auch mit Anwendungsfällen arbeitet. Während das Vorliegen der Anwendungsfälle eine Grundvoraussetzung für die Anwendung von UBR ist, müssen sie bei PBR-Techniken meist erst erstellt werden. In der Praxis der betrieblichen Software-Entwicklung von administrativen Systemen hat sich gezeigt, dass sich Inspektionen mit UBR-Lesetechniken sehr gut für die Fehlersuche in Anforderungs- und Designdokumenten eignen [8] [92].

Eine spezielle Form von Reviews adressiert die Architektur und die Evaluierung möglicher unterschiedlicher Architekturvarianten im Hinblick auf nicht funktionale Anforderungen wie Erweiterbarkeit und Änderbarkeit.

5.5 Architekturevaluierung

Reviews und Inspektionen zielen – wie bereits beschrieben – auf das Finden von Abweichungen und Fehlern in frühen Phasen der Software-Entwicklung ab. Fehler, die nicht erkannt werden, können hohe Kosten und Aufwände für deren Korrektur verursachen. Aber nicht nur Fehler, sondern auch spätere Änderungen und Erweiterungen (etwa in der Wartungsphase) sind Kostentreiber, falls Änderungen speziell in der Architektur und im Design durchgeführt werden müssen. Typischerweise betreffen diese Änderungen Qualitätsmerkmale (nicht funktionale Anforderungen), wie Erweiterbarkeit oder Änderbarkeit. Daher ist es sinnvoll, sich möglichst früh darüber Gedanken zu machen, welche möglichen Änderungen künftig gefordert sein könnten und die Architektur und das Design bereits während der Entwicklung entsprechend darauf auszurichten. Diese Überlegungen können eine spätere Wartung deutlich erleichtern. Häufig tritt dabei auch die Situation auf, dass unterschiedliche Architekturvarianten zur Diskussion stehen, und es gilt, die *bestmögliche* Architekturvariante auszuwählen und einzusetzen.

ATAM

Die *Architecture Tradeoff Analysis Method (ATAM)* ermöglicht – als Maßnahme der Qualitätssicherung – eine systematische Evaluierung von unterschiedlichen Architekturvarianten als Entscheidungsbasis für das Entwicklungsteam [57]. ATAM ist eine szenariobasierte und modellbasierte Analysetechnik, um unterschiedliche Software-Architekturvarianten im Hinblick auf unterschiedliche Qualitätsattribute zu analysieren. ATAM verfolgt dabei folgende Zielsetzungen:

Ziele von ATAM

1. Besseres Verständnis der Anforderungen der zu erstellenden Software-Lösung unter Berücksichtigung der jeweiligen Wertbeiträge für die beteiligten Stakeholder.
2. Ermittlung möglicher Architekturvarianten sowie Analyse von Vor- und Nachteilen jeder möglichen Architekturvariante im Hinblick auf definierte Qualitätsmerkmale.

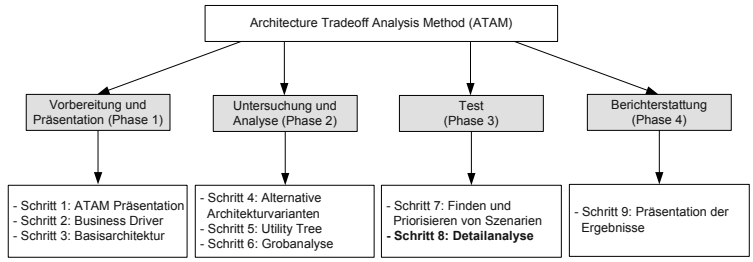


Abbildung 5.4 *Architekturevaluierung mit ATAM [1].*

3. Identifikation von Risiken und Nicht-Risiken in Bezug auf die Entwicklung und die Betriebs- und Wartungsphase.

Beispielsweise kann der Auftraggeber planen, die Software-Lösung nach einigen Betriebsmonaten oder -jahren zu erweitern (initiiert durch geänderte Geschäftsfälle). Diese Anforderung kann während der Entwicklung bereits berücksichtigt werden. Somit können die nicht funktionalen Anforderungen *Erweiterbarkeit* und *Änderbarkeit* der Software-Lösung als Basis für die Architekturevaluierung dienen. Für die Erhebung solcher Anforderungen haben sich in der Praxis *Zieldefinitionen* und *Szenarien*, die durch die unterschiedlichen Stakeholder ermittelt werden, bewährt. Die systematische Analyse vorliegender Architekturvarianten und Szenarien durch die Anwendung von ATAM ermöglicht eine optimale Auswahl der geeigneten Architekturvariante. Abbildung 5.4 zeigt die wesentlichen Phasen und Komponenten des ATAM-Evaluierungsprozesses [1] [58].

Der ATAM-Prozess umfasst dabei neun Schritte, die in vier grundlegende Phasen unterteilt sind [1]: (1) Vorbereitungs- und Präsentationsphase), (2) Untersuchungs- und Analysephase, (3) Testphase und (4) Berichterstattungsphase. Zur *Vorbereitungs- und Präsentationsphase (Phase 1)* zählen:

Vorbereitung und Präsentation

- *Schritt 1: ATAM-Präsentation*. In einem ersten Schritt werden sowohl ATAM-Methode als auch die verwendeten Techniken allen beteiligten Stakeholdern vorgestellt, um ein gemeinsames Verständnis herzustellen und die Zielsetzungen für die Architekturevaluierung zu definieren.
- *Schritt 2: Präsentation der Business Driver*. Um eine gemeinsame Sicht auf die zu erstellende Software-Lösung herzustellen (auch in Bezug auf die Geschäftsziele), werden die wichtigsten funktionalen Anforderungen, Zielsetzungen, beteiligte Stakeholder, Rahmenbedingungen und zu adressierende Qualitätsmerkmale durch den Projektleiter vorgestellt.
- *Schritt 3: Präsentation der zu evaluierenden Basisarchitektur*. Dieser Schritt beinhaltet die Vorstellung der grundlegenden Architektur des geplanten Systems. Hier kommen beispielsweise Schichten- oder Komponentendiagramme zum Einsatz (siehe Kapitel 6). Da die Architektur die Basis für die spätere Untersuchungs- und Analysephase ist,

müssen hier auch noch offene Punkte besprochen und geklärt werden. Dabei sollen sowohl die technischen Rahmenbedingungen als auch die Interaktion mit anderen Systemen als geplante Architekturelemente in Bezug auf die umzusetzenden Qualitätsmerkmale diskutiert werden. Bei der Präsentation der Architektur ist es wesentlich, dass sich alle beteiligten Stakeholder (nicht nur die Architekten) ein umfassendes Bild vom geplanten System machen können. Dementsprechend muss die Vorstellung der Architektur für alle Stakeholder auch gut verständlich sein.

Zur eigentlichen *Untersuchungs- und Analysephase (Phase 2)* der unterschiedlichen Architekturvarianten zählen:

Untersuchung und Analyse

- *Schritt 4: Identifikation alternativer Architekturvarianten*. Dieser Schritt beinhaltet das Finden von unterschiedlichen Architekturvarianten, die die geforderten Qualitätsmerkmale adressieren können. Dieser Schritt wird typischerweise durch das Architekturteam umgesetzt.

- *Schritt 5: Erstellung des Utility Trees*. In diesem Schritt werden die wichtigsten nicht funktionalen Qualitätsanforderungen und -merkmale durch das Architekturteam, die Manager und die Kundenvertreter identifiziert und priorisiert. Das Ergebnis des *Utility Trees* sind priorisierte Anforderung (Qualitätsmerkmale), die in Form von *Szenarien* vorliegen. Diese Szenarien bilden die Basis für die weitere Analyse. Dieser Utility Tree kann sich an einem Qualitätsmodell, wie es bereits in Abschnitt 2.2.2 beschrieben wurde, orientieren. Wichtig dabei ist, dass alle relevanten Merkmale auf konkrete – und auf das Projekt abgestimmte – Anforderungen heruntergebrochen werden. Für das Merkmal *Änderbarkeit* kann beispielsweise ein maximaler Aufwand für Änderungen an der Benutzerschnittstelle definiert werden. Ein anschaulicheres Beispiel wäre beispielsweise eine Definition der maximal zulässigen Antwortzeiten für einen Webservice.

- *Schritt 6: Grobanalyse möglicher Architekturvarianten*. Basierend auf den priorisierten Anforderungen und Qualitätsmerkmalen (aus Schritt 5) werden die Architekturvarianten (aus Schritt 4) untersucht und priorisiert. Wichtig dabei ist, ob die unterschiedlichen Architekturvarianten auch in der Lage sind, die priorisierten Anforderungen ausreichend zu erfüllen; beispielsweise, ob eine spezielle Architekturvariante so ausgelegt ist, dass Änderungen der Benutzerschnittstelle einfach möglich sind, oder ob die zulässigen Antwortzeiten eines Webservice erfüllt werden können. In diesem Schritt werden auch *Risiken*, *Sensitivity Points* und *Trade-Offs* der jeweiligen Architekturvarianten identifiziert. Das bedeutet, dass an der Stelle bekannt sein muss, wo eine Architektur besonders „empfindlich“ reagiert, und an welcher Stelle Kompromisse eingegangen werden können.

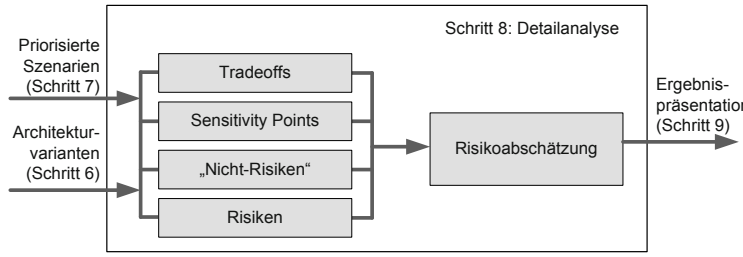


Abbildung 5.5 *Detailanalyse von Architekturvarianten [57].*

Ergebnis der Phase *Test (Phase 3)* ist es, anhand konkreter Zieldefinitionen und Szenarien die bestmögliche Architekturvariante zu ermitteln. In dieser *Testphase* werden folgende Schritte umgesetzt:

- *Schritt 7: Finden und priorisieren von Szenarien*. Basierend auf den bereits verfügbaren Szenarien (aus Schritt 5) werden durch die beteiligten Stakeholder weitere Szenarien gesammelt und priorisiert (beispielsweise auch mit Werkzeugen wie etwa Easy-Win-Win [35]).
- *Schritt 8: Detailanalyse und Test der Architekturvarianten*. In diesem Schritt werden die Architekturvarianten (aus Schritt 6) mithilfe der priorisierten Szenarien (aus Schritt 7) analysiert. Im Prinzip ist dieser Schritt eine Wiederholung von Schritt 6 – die priorisierten Szenarien werden dabei als Testfälle für die Architekturvarianten betrachtet. Daraus können sich weitere Architekturvarianten, Risiken, Sensitivity Points und Trade-Off-Abschätzungen ergeben. Abbildung 5.5 illustriert die Vorgehensweise beim Test der Architektur.

Nach Abschluss der Analyse- und Testphase werden die Ergebnisse entsprechend zusammengefasst und die *Best-Practice Architekturvariante* herausgearbeitet *(Phase 4)*. Wichtig dabei ist, dass Designentscheidungen entsprechend protokolliert werden, um die Nachvollziehbarkeit der Entscheidungen zu ermöglichen.

Abschluss und Berichterstattung

- *Schritt 9: Präsentation der Ergebnisse*. Dieser abschließende Schritt fasst die während der Untersuchung und Analyse der Architekturvarianten und Qualitätsmerkmale gewonnenen Erkenntnisse zusammen. Die empfohlene Architekturvariante wird mit entsprechenden Detailinformationen allen beteiligten Stakeholdern als Ergebnis von ATAM vorgestellt.

Weitere Methoden zur Evaluierung

Neben ATAM existieren auch andere Methoden zur Evaluierung der Software Architektur. Beispiele dafür sind SAAM (*Scenario-based Architecture Analysis Method*), CBAM (*Cost Benefit Analysis Method*) oder ALMA (*Architecture-Level Modifiability Analysis*) [75].

5.6 Software-Testen

In den bisherigen Abschnitten wurden analytische Methoden der Qualitätssicherung zur Identifikation von Fehlern in frühen Phasen der Software-Entwicklung (Reviews und Inspektionen) sowie Methoden zur Evaluierung von Architekturvarianten (ATAM) beschrieben. Dieser Abschnitt beschäftigt sich mit Software-Testen zur Überprüfung der Produktqualität bei existierendem Softwarecode.

Definition und Abgrenzung

Unter *Software-Testen* versteht man den Prozess des Planens, der Vorbereitung und der Messung von Produkteigenschaften, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen [73]. Myers' Definition bringt es auf den Punkt: *„Das Ziel des Testens ist es, Fehler zu finden“* [70]. Etwas polemisch formuliert könnte man sagen, dass ein Test erfolgreich ist, wenn tatsächlich ein Fehler gefunden wird: *„A test that reveals a problem is a success. A test that did not reveal a problem was a waste of time“* [55], d. h., das Ziel des Testens ist es, jenes Stück Code, das getestet wird, zu einem Fehlverhalten zu bringen.

Fehlerfreie Software durch Testen?

Der Umkehrschluss ist allerdings keinesfalls gültig: Wird durch einen Test *kein* Fehler gefunden, so bedeutet das nicht, dass die getestete Software fehlerfrei ist; eventuell wurden ungeeignete Datensätze für den Test verwendet oder der Testfall selbst war zur Erkennung der noch vorhandenen Fehler nicht geeignet. Durch Testen kann also *keine Fehlerfreiheit* nachgewiesen werden. Dazu wäre eine formale Verifikation notwendig. Software-Tests dienen auch nicht primär der Lokalisierung von Fehlern oder deren Beseitigung – das ist die Aufgabe von Debugging-Aktivitäten.

Fehlerdefinition

Wie wird nun ein *Fehler* definiert? Ein Fehler ist eine Abweichung einer Software-Lösung von ihrer Spezifikation [55]. Dadurch wird der Aspekt der *Verifikation* in den Vordergrund gerückt. Eine andere Sichtweise auf einen Fehler befasst sich mit der *Validierung*, in der ein Software-Fehler dann vorliegt, wenn ein Programm nicht die Aufgaben erfüllt, die ein Endbenutzer vom System erwartet [70].

Error, Fault und Failure

In der englischen Sprache wird ein *Fehler* deutlich differenzierter betrachtet, indem man zwischen einem *Error*, einem *Fault* oder einem *Failure* unterscheidet. Unter einem *Error* wird typischerweise eine menschliche Aktivität verstanden, die inkorrekte Ergebnisse produziert, also beispielsweise inkorrekten Softwarecode durch Tippfehler. Ein Error zielt daher auf die Prozessqualität ab. Ein Error wird zum *Fault*, wenn dadurch ein System dazu gebracht wird, eine erforderliche Funktion, beispielsweise aufgrund von inkorrektem Softwarecode, nicht mehr oder fehlerhaft auszuführen. Ein Fault lässt sich also direkt mit der Produktqualität assoziieren. Als weitere Folge ist ein *Failure* eine Abweichung des tatsächlichen Ergebnisses vom erwarteten Ergebnis in der Betriebsphase, quasi dessen Auswirkung. Ein Failure beeinflusst die Qualität in der Verwendung. Diese Begriffe werden in Abbildung 5.6 im Zusammenhang dargestellt.

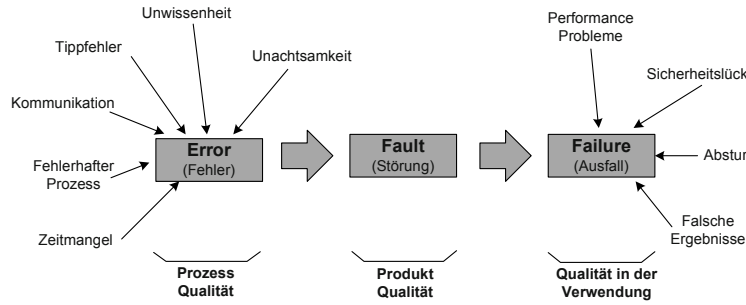


Abbildung 5.6 *Error – Fault – Failure.*

Testen als destruktiver Prozess

Ein allgemeines Ziel des Testens ist es also, in einer Komponente oder einem System durch geeignete Maßnahmen einen Error (Fehler) zu finden, um Faults (Störungen) und Failures (Ausfälle) zu vermeiden. Dementsprechend ist Testen typischerweise destruktiv, da versucht werden muss, das System zu einem Ausfall zu bringen.

5.6.1 Rollen im Software-Testen

Unterschiedliche Rollen und Aufgaben

Je nach Projektgröße können Tests von einzelnen Personen, aber auch – bei entsprechender Größe des Projekts – von Testteams geplant und durchgeführt werden. In jedem Fall sind im Testprozess spezifische Rollen definiert, die unterschiedliche Aufgaben wahrnehmen:

Testmanager

Der *Testmanager* ist für den funktionierenden Ablauf des Testprozesses verantwortlich. Er führt in Zusammenarbeit mit der Projektleitung die *Testplanung* durch, teilt den Testern und Testentwicklern Arbeitspakete zu, kontrolliert den Testfortschritt, erstellt Testberichte und stellt die Schnittstelle zwischen Testteam und Projektleitung dar.

Testingenieur

Der *Testingenieur* hat die Aufgabe, die vom Testmanager entwickelte Teststrategie durch Testfälle umzusetzen. Eine *Teststrategie* beschreibt dabei die Vorgehensweise für möglichst effiziente und ökonomische Tests. Die Teststrategie wird typischerweise im Testplan festgelegt und definiert die einzusetzenden Testtypen, deren Reihenfolge und die Testintensität für ein Qualitätsmerkmal, eine Komponente oder ein System.

Testentwickler

Der *Testentwickler* realisiert die vom Testingenieur entworfenen Testfälle.

Tester

Der *Tester* ist diejenige Person, die einen Test tatsächlich durchführt. Als Grundlage für seine Arbeit dienen die vom Testentwickler erzeugten Testfälle in einer vorgegebenen Infrastruktur, sowie die Vorgaben des Testmanagers. Seine Testprotokolle dienen als Basis für Korrekturmaßnahmen durch das Entwicklerteam aber auch für die Fortschrittskontrolle und das Berichtswesen des Testmanagers.

Rollenaufteilung

In großen Entwicklungsteams finden sich typischerweise alle Rollen, die auf unterschiedliche Personen oder Teams aufgeteilt sind. Speziell bei kleinen Projekten ist es eine gängige Praxis, dass eine Person mehrere Rollen

einnimmt, beispielsweise werden meist Testingenieur und Testentwickler von einer Person abgedeckt.

5.6.2 Der traditionelle Testprozess

Traditionelle Testprozesse

Der Testprozess ist in einen Entwicklungsprozess eingebettet, wie es am Beispiel des V-Modells in Abbildung 5.1 bereits gezeigt wurde. Speziell das V-Modell eignet sich gut zur Illustration von unterschiedlichen *Testebenen*. Beispielsweise wird durch das V-Modell der Kontext von Spezifikation, Umsetzung und Test auf unterschiedlichen Ebenen – Implementierungssicht (Komponententests), Architektursicht (Integrationstests) und Anwendersicht (Akzeptanz- und Abnahmetests) – hervorgehoben (siehe Abschnitt 3.3).

Frühes Testen auf unterschiedlichen Ebenen

In einem *guten* Softwareprojekt beginnt die Testplanung bereits recht früh, nämlich bei den Kundenanforderungen. Idealerweise stellt man bereits bei vorliegenden Kundenanforderungen während der Spezifikationsphase Überlegungen darüber an, wie das System getestet werden kann. Im Rahmen der Entwurfs- und Designphase sind Überlegungen anzustellen, wie die geplante Architektur getestet werden kann. Auch auf Komponentenebene stehen Überlegungen, wie einzelne Komponenten getestet werden können, im Vordergrund. Diese frühen Überlegungen haben den Vorteil, dass Testfälle frühzeitig systematisch erstellt und daher auch unklare Vorgaben früh identifiziert werden können. Falls eine Vorgabe (etwa eine konkrete Anforderung) nicht getestet werden kann, stellt sich die Frage, ob sie auch klar genug definiert wurde und technisch überhaupt sinnvoll umgesetzt werden kann. In den meisten Fällen bewirkt die frühzeitige Planung von Tests auch eine Verbesserung der Umsetzung und der Architektur, da schlecht oder nicht testbare Umsetzungen vermieden werden. Die frühe Definition von Testfällen lehnt sich an das Konzept des *Test-First (oder Test-Driven) Development* an. Details zu Test-Driven Development werden in Abschnitt 5.7 beschrieben.

Planung und Steuerung

Systematisches und traditionelles Software-Testen folgt einem definierten Prozess, wie er beispielsweise in Abbildung 5.7 dargestellt wird [85]. Die *Planung und Steuerung* des Testprozesses startet am Beginn des Entwicklungsprozesses und begleitet ihn bis zum fertigen Produkt. Wie bei jeder Planungsaktivität müssen Aufgaben und Zielsetzungen – unter Berücksichtigung der verfügbaren Ressourcen – definiert und überwacht werden. Eine wichtige Aufgabe während der Planungsphase ist die Definition der *Teststrategie* sowie die Festlegung der Testrahmenbedingungen und die Vorbereitung der Infrastruktur. Das Ziel einer Teststrategie umfasst die Definition der wichtigsten Testaufgaben, der Testkriterien sowie die Festlegung der zu verwendenden Testmethoden. Im Rahmen der Festlegung der Teststrategie erfolgt ebenfalls die Priorisierung der zu testenden Systemteile, basierend auf priorisierten Anforderungen oder Risikoabschätzungen. So können bei fehlenden Ressourcen und mangelnder Zeit für Testaktivitäten zumindest

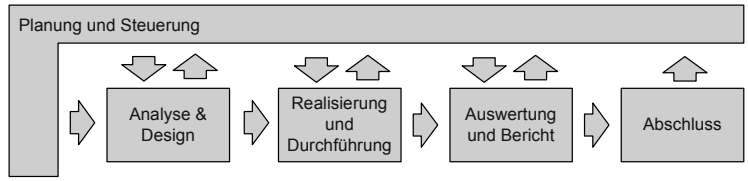


Abbildung 5.7
Schematische Darstellung eines Testprozesses [85].

wichtige und erfolgskritische Anforderungen mit dem entsprechenden Risiko getestet werden, ohne das Augenmerk auf Anforderungen mit niedriger Priorität zu lenken. Da ein Software-System nicht vollständig getestet werden kann, sind auch Kriterien für die erfolgreiche Beendigung des Testprozesses erforderlich (*Testkriterien*). Dazu werden beispielsweise *Überdeckungsmaße (Code Coverage)* verwendet (siehe dazu Abschnitt 5.6.6).

Analyse und Design

In der *Analyse- und Designphase* wird bestimmt, ob die Vorgaben (z. B. Anforderungen) in ausreichender Qualität vorliegen, um konkrete Testfälle daraus ableiten zu können. Aus der Teststrategie und den verfügbaren Testmethoden können dann konkrete Testfälle abgeleitet werden (siehe Testfalldokumentation in Abschnitt 5.6.5). Wichtig bei der Testfallerstellung ist, dass jeweils konkrete und reproduzierbare Zustände und Daten verwendet werden. Beispielsweise muss immer ein definierter Startzustand (*Vorbedingungen*) des Systems vorliegen, auf den der Testfall angewandt wird. Vor dem nächsten Durchlauf muss das System wieder in denselben Ausgangszustand versetzt werden, da es sonst zu Abhängigkeiten von Testfällen und zu nicht reproduzierbaren Testabläufen kommen kann. Ein Testfall einer Datenbankanwendung könnte zum Beispiel das Einfügen eines neuen Datensatzes abdecken. Wird der Testfall erfolgreich durchgeführt, ist dieser Eintrag nach der Ausführung in der Datenbank vorhanden. Die erneute Ausführung desselben Testfalls (Einfügen eines neuen Datensatzes mit denselben Testdaten) würde fehlschlagen, da dieser Datensatz schon existiert. Das Einfügen eines identischen Datensatzes wäre demnach ein anderer Testfall, der natürlich auch entsprechend geprüft werden muss.

Bei der Erstellung von Testfällen ist es auch notwendig, nicht nur das konkrete gewollte Systemverhalten (*Normalfälle*) zu überprüfen, sondern speziell auch Randbereiche (*Sonderfälle*) und *Fehlerfälle* zu betrachten. Ziel ist es, dass sich das System in jedem Fall vorhersehbar verhält, also die gewollte Aufgabe erfüllt oder auf unerwartete Zustände geeignet mit entsprechenden Fehlermeldungen reagiert.

Realisierung und Durchführung

Die *Realisierungs- und Durchführungsphase* beschäftigt sich mit der detaillierten Definition der konkreten Testfälle (sofern sie nicht schon während der Analyse- und Designphase erstellt wurden) sowie der Priorisierung und Gruppierung von gleichartigen Testfällen zu Testszenarien. Ein *Testszenario* deckt dabei beispielsweise einen konkreten Ablauf aus Anwendersicht (z. B. basierend auf einem konkreten Anwendungsfall) ab. Vor der konkreten Ausführung der Testszenarien und Testfälle ist es auch notwendig, diese auf Richtigkeit und Vollständigkeit zu überprüfen. Dazu wer-

den typischerweise Reviews und Inspektionen eingesetzt. Sind diese Vorbereitungen (d. h. Testinfrastruktur, Testrahmen und Testfälle) abgeschlossen, können die Tests manuell oder automatisiert durchgeführt werden (siehe dazu auch Abschnitt 10.3).

Kein Testlauf ohne Protokoll!

Die Dokumentation der Testdurchführung ist unbedingt erforderlich, um die Nachvollziehbarkeit der Testausführung auch im Hinblick auf die Fehlerkorrektur (falls Abweichungen gefunden werden) zu ermöglichen. Weiterhin kann durch die Protokollierung auch die Durchführung der Tests (z. B. für den Kunden) nachgewiesen werden. Sofern keine automatische Unterstützung der Testdurchführung eingesetzt wird, empfiehlt sich die Testfalldokumentation auch für die Protokollierung der Testdurchführung (siehe Abschnitt 5.6.5).

Tatsächliches und erwartetes Ergebnis

Bei einer gefundenen Abweichung (das tatsächliche Ergebnis entspricht nicht dem erwarteten Ergebnis) muss entschieden werden, ob es sich tatsächlich um einen gefundenen Fehler handelt, die Vorgabedokumente zu ungenau sind, die Vorgaben falsch verstanden wurden oder der Testfall fehlerhaft erstellt wurde. In jedem Fall bringt jeder Testlauf wertvolle Erkenntnisse über die Qualität des getesteten Objekts und des Systems. Daher ist es auch für Projekt- und Qualitätsleiter hilfreich, eine zusammenfassende Darstellung der Testläufe zu erhalten. Diese Informationen sollten nicht nur die Anzahl der durchgeführten erfolgreichen und nicht erfolgreichen Tests, sondern auch den Überdeckungsgrad – wie viele Teile des Systems bereits durch Tests abgedeckt werden – enthalten.

Auswertung und Bericht

Die Phase *Auswertung und Bericht* beantwortet die grundlegende Frage, ob die Testkriterien in *ausreichendem* Maß erfüllt wurden und das Testen der betroffenen Komponente oder des Systems beendet werden kann. Eine Voraussetzung für das erfolgreiche Beenden der Testläufe ist die Erfüllung aller wesentlichen Testkriterien, das kann – je nach definierter Teststrategie – beispielsweise aus der Testfallüberdeckung oder der Fehlerfindungsrate abgeleitet werden. Abhängig vom Ergebnis der Auswertungen können auch weitere Tests geplant werden, d. h. die Planungsphase würde an dieser Stelle wieder aufgenommen. Die Berichterstattung kann als formlose Mitteilung, z. B. dass alle Testkriterien für die Komponente erfüllt sind, aber auch als formeller Bericht, der speziell bei Integrations- oder Akzeptanztests erforderlich sein kann, ausgeführt werden.

Abschluss

In der *Abschlussphase* werden neben produktspezifischen Information über die Qualität des Produkts auch Rückschlüsse auf die Prozessqualität (also den Testprozess) gezogen. Informationen fließen im Sinn der ständigen Verbesserung von Produkten und Prozessen in folgende Planungsphasen (auch anderer Projekte) ein. Besondere Wichtigkeit in dieser Phase erlangt auch die Archivierungsfunktion der durchgeführten Tests einschließlich aller relevanten Informationen wie Testumgebung, Testfälle und Infrastruktur, um eine Nachvollziehbarkeit und Reproduzierbarkeit der Tests gewährleisten zu können.

Neben dem „traditionellen Testen“ steigt zunehmend auch die Bedeutung der Beobachtung des Laufzeitverhaltens von Systemen im Sinn des Monitoring zur Laufzeit. Wichtige Informationen über den Systemzustand beinhalten beispielsweise Performancebeobachtungen, Ressourcenverbrauch, Benutzerverhalten, aber auch Systembeobachtungen im Hinblick auf Sicherheitslücken oder eine mögliche missbräuchliche Verwendung von Systemen (siehe auch Abschnitt 10.5).

5.6.3 Testebenen

Betrachtet man das V-Modell (siehe Abschnitt 5.1), sind die Testebenen (a) Komponententests, (b) Integrationstests und (c) System-, Akzeptanz- und Abnahmetests relativ einfach abzuleiten. Jede Ebene adressiert somit spezielle Aspekte der Software-Lösung. Diese Ebenen finden sich grundsätzlich in nahezu allen Software-Projekten. Unterschiede gibt es bei Produkten „für den breiten Markt“ (siehe auch Abschnitt 2.7.1), bei denen keine individuelle Abnahmeprüfung erfolgen kann. Weiterhin ist bei sogenannten Inhouse-Lösungen meist auch keine formelle Abnahme vorgesehen. In diesem Abschnitt wird von Individualsoftware ausgegangen, die alle Schritte des Entwicklungsprozesses umfasst.

Komponententest

Die Tests auf *Komponentenebene* stehen in direktem Näheverhältnis zur Implementierung der einzelnen Komponenten und zielen darauf ab, Fehler im Hinblick auf die Komponentenspezifikationen zu finden. Da diese Tests meist von den Entwicklern selbst durchgeführt werden, bezeichnet man diese Testebene auch als *Entwicklertest*. Je nach Programmiersprache werden diese auch als Komponententests, Unit-Tests oder Klassentests bezeichnet. In jedem Fall wird dabei ein Software-Baustein – etwa eine Komponente in der komponentenorienterten Software-Entwicklung – isoliert von anderen Komponenten getestet. Im Vordergrund steht dabei die Überprüfung der Anforderungen an die einzelne Komponente. Daher können gefundene Fehler auch eindeutig zugeordnet werden.

Mock-Objekt

Abhängigkeiten zu anderen (auch externen) Komponenten erfordern allerdings einen Mechanismus um die zu testende Komponente auch isoliert testen zu können. Steht also die benötigte Funktionalität zum Zeitpunkt der Tests nicht zur Verfügung, kann eine Simulation dieser (fehlenden) Systemteile erforderlich sein (siehe Abschnitt 5.6.4). Das trifft bei Komponententests – speziell aber bei Integrationstests – meist dann zu, wenn andere oder externe Komponenten benötigt werden.

Integrationstest

Nachdem die einzelnen Komponenten in ausreichendem Maß getestet wurden, ist der zweite Schritt, diese Komponenten zu einem Subsystem oder System zusammenzubauen – zu integrieren (siehe Integrationsstrategien in Abschnitt 2.6.3) und zu testen. Ziel des *Integrationstests* ist es, das Zusammenspiel der Komponenten über die verfügbaren Schnittstellen zu testen. Integrationstests decken aber nicht nur die Schnittstellen zu den selbstent-

wickelten Komponenten ab, sondern müssen auch Schnittstellen zur Systemumgebung und zu externen Systemen umfassen, die gegebenenfalls simuliert werden müssen. Auch bei Integrationstests müssen Test-Treiber eingesetzt werden, um abhängige – noch nicht verfügbare – Komponenten zu simulieren, wobei die Test-Treiber, die bereits im Rahmen des Komponententests zum Einsatz gekommen sind, wiederverwendet werden können.

In der Praxis empfiehlt sich auch der Einsatz von automatisierten Tests, wie sie beispielsweise von *Continuous-Integration-Konzepten* (siehe Abschnitt 10.3.6) beschrieben werden. Durch Integrationstests werden also Fehler im Zusammenspiel unterschiedlicher Komponenten aufgedeckt, beispielsweise inkompatible Schnittstellenformate, fehlerhafte oder unvollständige Spezifikationen (und dadurch eine falsche Kommunikation zwischen den Komponenten), Leistungsprobleme (z. B. Durchsatz, Last oder Zeitverhalten) oder ungewollte Seiteneffekte von Komponenten.

System-, Akzeptanz- und Abnahmetests

System-, Akzeptanz- und Abnahmetests bilden die dritte Testebene. Alle Komponenten sind verfügbar und getestet, die Komponenten wurden integriert und das Zusammenspiel der unterschiedlichen Komponenten wurde getestet. Abschließend muss festgestellt werden, ob das Gesamtsystem den Erwartungen des Kunden entspricht (Verifikation gegen die Anforderungsdokumente bzw. Validierung gegen die Kundenanforderungen). Test-Treiber sind an dieser Stelle kaum mehr notwendig, da alle Komponenten umgesetzt sein sollten. Eine Ausnahme sind Schnittstellen zu anderen Systemen, die für den Systemtest nicht zur Verfügung stehen. Wichtig dabei ist, dass das Testsystem so genau wie möglich der realen Umgebung beim Kunden entspricht.

Ein Systemtest sollte jedenfalls möglichst nicht in der Produktivumgebung des Kunden ausgeführt werden, da einerseits Fehler des zu testenden Systems auch Fehlerwirkungen auf andere Systeme des Produktivsystems haben können und die Kontrolle über das Produktivsystem nicht oder nur eingeschränkt möglich ist. Allgemein ist der Systemtest die letzte vollständige Überprüfung in der Verantwortung des Herstellers und ist – da gegen die Kundenanforderungen getestet wird – mit der Validierung zu vergleichen.

Abnahmetest

Nach der Inbetriebnahme obliegt es dem Auftraggeber, einen *Abnahmetest* durchzuführen. Typischerweise umfasst der Abnahmetest einen Teil des Systemtests und wird in der Produktivumgebung des Kunden durchgeführt. Je nach Projekt und Projektauftrag kann der Abnahmetest unterschiedlich intensiv ausfallen. Das kann beispielsweise das Durchspielen einzelner Benutzerszenarien, das Testen der Benutzerakzeptanz oder auch ausgedehnte Feldtests (im Realbetrieb) umfassen.

Regressionstest

Einen besonderen Stellenwert innerhalb der Testebenen nehmen *Regressionstests* ein. Diese Tests werden eingesetzt, wenn Änderungen bzw. Erweiterungen am System vorgenommen wurden. Durch Änderungen oder Erweiterungen kann sich das Systemverhalten ändern, und es ist unter Umständen nicht mehr sichergestellt, dass das bisherige Verhalten unverändert

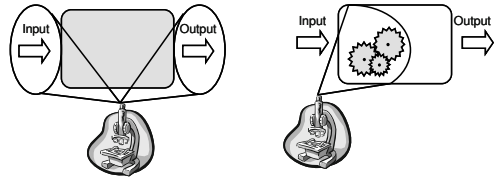


Abbildung 5.8 *Schematischer Vergleich von Black-Box- und White-Box-Testverfahren [10].*

geblieben ist. Daher ist es erforderlich, das System bzw. die Komponente erneut zu testen – die bisher durchgeführten Tests also zu wiederholen. Regressionstests sind typischerweise gute Kandidaten für automatisiertes Testen. Stellt sich noch die Frage, welche grundlegenden Möglichkeiten es gibt, Tests auf den unterschiedlichen Ebenen effizient durchzuführen.

5.6.4 Testmethoden

Statische und dynamische Verfahren

Generell wird zwischen statischen und dynamischen Testverfahren unterschieden. Bei den statischen Testverfahren, wie Reviews (siehe Abschnitt 5.3) und Inspektionen (siehe Abschnitt 5.4), werden keine Daten und kein ausführbarer Code benötigt – die Testobjekte werden einer (statischen) Analyse unterzogen. Natürlich können statische Testverfahren auch auf Software-Code angewandt werden, der Code wird allerdings nicht ausgeführt. Im Gegensatz dazu umfassen dynamische Verfahren die Überprüfung von existierenden Komponenten und Systemen durch Software-Tests. Der folgende Abschnitt stellt einige wichtige und grundlegende Techniken, wie das *Black- und White-Box-Verfahren* als spezifische Testmethoden vor. Weiterhin ist es wichtig, geeignete Eingangsparameter zur effizienten Testfallerstellung und Testausführung zu bestimmen. Techniken der *Äquivalenzklassenzerlegung* und der *Grenzwertanalyse* helfen bei der Festlegung geeigneter Eingangsparameter.

Black-Box-Test

Die Grundlage für das *Black-Box Testverfahren* sind Anforderungen und Spezifikationen, für die der innere Aufbau der Komponenten oder des Systems (also die konkrete Umsetzung und Implementierung) zum Zeitpunkt des Tests nicht bekannt sein muss. Die Testobjekte werden daher unabhängig von ihrer Realisierung getestet. Auf der linken Seite in Abbildung 5.8 wird die grundlegende Funktionsweise und Idee des Black-Box-Verfahrens dargestellt. Aufgrund der Betrachtung des zu testenden Objekts als *Black-Box* werden die Testfälle von Daten getrieben (*Data-Driven*) und beziehen sich auf Anforderungen und das spezifizierte Verhalten der Testobjekte. Das Testobjekt wird mit definierten Eingangsparametern aufgerufen. Die Ergebnisse – nach Abarbeitung durch das Testobjekt – werden mit den erwarteten Ergebnissen verglichen. Stimmen die tatsächlichen Ergebnisse nicht mit den erwarteten Ergebnissen überein, liegt ein Fehler vor. Ziel ist es dabei, eine möglichst hohe *Anforderungsüberdeckung* zu erreichen, also möglichst alle Anforderungen zu testen. Um die Anzahl der Tests bei gleichbleibender Testintensität zu reduzieren, bedient man sich Tech-

niken, wie *Äquivalenzklassenzerlegung* und *Grenzwertanalyse*. Dabei versucht man, die Menge an Eingabe- und Ausgabedaten zu beschränken, um mit minimalem Aufwand möglichst alle Testfälle abzudecken.

White-Box-Test

Im Gegensatz zum Black-Box-Verfahren berücksichtigt das *White-Box-Verfahren* die innere Struktur und die konkrete Implementierung. Die rechte Seite der Abbildung 5.8 illustriert diese grundlegende Idee. Der implementierte und vorliegende Softwarecode ist dabei die Grundlage für die Testfallerstellung und Testdurchführung. Daher ist die detaillierte Kenntnis der internen Struktur notwendig (*Logic-Driven*). Ziel ist es, alle Codesequenzen zu testen. Dementsprechend gehen die Maße für die Testfallqualität und Abdeckung weit über die Anforderungsebene hinaus und beziehen die konkreten Abläufe (z. B. Anweisungen, Bedingungen und Verzweigungen) mit ein. Auch hier ist es notwendig, Äquivalenzklassen – diesmal unter Berücksichtigung der inneren Logik – zu finden, um die Komplexität und den Aufwand des Testens zu reduzieren. Durch das White-Box-Verfahren ist zusätzlich zur Fehlererkennung (primäres Ziel des Testens) auch die Lokalisierung des Fehlers durch den Testfall möglich, was beim Black-Box-Verfahren aufgrund der unbekannten inneren Struktur nicht möglich ist.

Lokalisieren von Fehlern

Äquivalenzklasse

Das vollständige Testen einer Komponente oder eines Software-Systems ist in der Regel aufgrund der Vielzahl an unterschiedenen Werten, die sowohl Eingangsparameter als auch Ausgangsparameter annehmen können, kaum realisierbar und auch meist nicht sinnvoll. Um die Anzahl der Testdaten auf ein vernünftiges Minimum zu reduzieren, bedient man sich der *Äquivalenzklassenzerlegung*.

Möchte man beispielsweise eine Bedingung mit nur einem ganzzahligen Parameter testen, der >18 sein soll, gibt es unendlich viele unterschiedliche Eingabemöglichkeiten (z. B. 19, 20, 100 oder 1000 bzw. 18, 17, 2 oder –10 als ungültige Werte), die bei vollständigem Testen alle geprüft werden müssten. Die Äquivalenzklassenzerlegung hilft dabei, Bereiche von Eingabewerten zu identifizieren, die jeweils dieselben Ergebnisse liefern. Aus diesen *Klassen* von Eingabewerte wählt man jeweils einen Vertreter (*Repräsentanten*) aus, der dann für den konkreten Testfall verwendet wird. Beispielsweise möchte man eine Anforderung, die eine Altersüberprüfung durchführt, testen: `18 < Alter <= 65`. Aus dieser Anforderung ergeben sich drei grundlegende Klassen von Eingabewerten (A1, A2 und A3):

- A1: Alter $<= 18$ (ungültige Eingabe).
- A2: Das Alter liegt zwischen 18 und 65 (gültige Eingabe).
- A3: Alter >65 (ungültige Eingabe).

Alle Werte aus einer Äquivalenzklasse liefern dieselben Ergebnisse. Daher ist es ausreichend, jeweils einen Vertreter pro Klasse zu wählen. Im obigen Beispiel könnten das 15 als Repräsentant der Klasse A1, 30 für A2 und 70

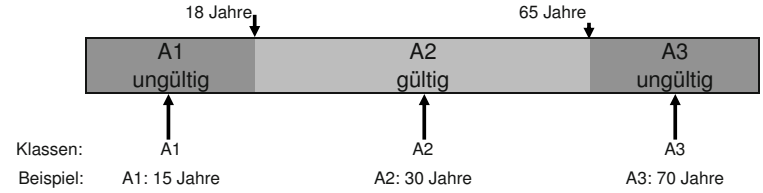


Abbildung 5.9 *Altersüberprüfung mit drei Äquivalenzklassen.*

für A3 sein. Die Abbildung 5.9 zeigt dieses Beispiel in grafisch aufbereiteter Form.

Komplizierter wird die Äquivalenzklassenzerlegung, falls Bedingungen mit mehreren Parametern zu testen sind, da sämtliche Kombinationen von Parameterklassen getestet werden müssen. Allerdings gilt auch hier wieder, dass pro Äquivalenzklassenkombination nur ein Repräsentant ausgewählt werden muss und die Anzahl der Testfälle auf diese Weise beschränkt werden kann. Das folgende Beispiel zeigt eine Unterscheidung nach Alter und dem Body-Maß-Index (BMI) mit jeweils zwei unterschiedlichen Klassen, die als Bedingungen für eine Ernährungsempfehlung verwendet werden können: Aus dieser Anforderung ergeben sich zwei Altersklassen (A1 und A2) und für jede Altersklasse zwei BMI-Klassen (B1 und B2). Durch alle Kombination ergeben sich insgesamt vier Äquivalenzklassen, wie Abbildung 5.10 verdeutlicht. Mit diesen vier Testfällen können also alle Kombination getestet werden.

Grenzwertanalyse

Einen besonderen Stellenwert nehmen Bereichsgrenzen von Äquivalenzklassen ein, da diese häufige Ursachen für Fehler sind, die beispielsweise durch Tippfehler verursacht werden. Statt eines „<=“ wird ein „<“ geschrieben, was speziell an den Systemgrenzen zu einem Fehlverhalten des Systems führt. Diese Fehlerquelle kann beispielsweise mit der *Grenzwertanalyse* erkannt werden. Die Kernidee bei der Grenzwertanalyse ist, (a) Grenzbereiche zu identifizieren und (b) Testdaten aus dem nahen Umfeld dieser Bereichsgrenzen auszuwählen. Dabei empfiehlt es sich, jeweils einen Wert aus dem Grenzbereich der einen und der anderen Klasse auszuwählen. Optional kann natürlich auch der exakte Grenzwert in einem Testfall spezifiziert werden, der aber – bei genauer Analyse – ohnehin einer der beiden Klassen zugeordnet werden kann. Durch diese Vorgangsweise, wie sie in Abbildung 5.11 schematisch darstellt ist, werden Fehler in diesen Grenzbereichen gefunden.

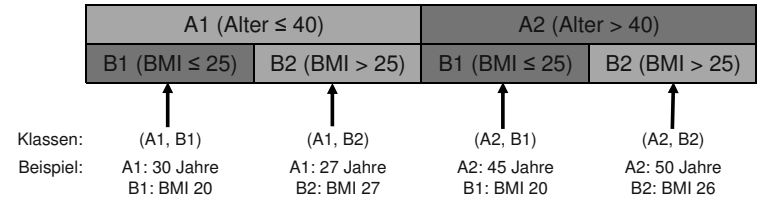


Abbildung 5.10 *Beispiel mit vier Äquivalenzklassen.*

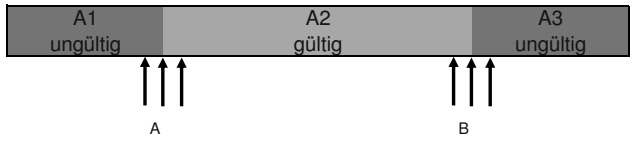


Abbildung 5.11 *Beispiel für eine Grenzwertanalyse.*

Simulation

Wie bereits erwähnt, ist für Testzwecke auf allen Testebenen häufig die Simulation von bis zu diesem Zeitpunkt fehlenden oder nicht implementierten Komponenten oder Systemteilen erforderlich, um beispielsweise Schnittstellen oder Kommunikationsmechanismen zu testen (etwa Datenaustausch mit anderen Komponenten oder externen Systemen).

Dummy, Fake-Objekte

Diese Simulation kann etwa durch Dummy- oder Fake-Objekte umgesetzt werden. *Dummy-Objekte* stellen eine Schnittstelle zur Verfügung, verfügen aber über keine Implementierung. Sie werden meist eingesetzt, falls das zu testende Objekt keinen Rückgabewert erfordert (etwa Testen zum Befüllen von Parameterlisten). *Fake-Objekte* beinhalten Implementierungen und „simulieren“ Funktionalitäten, in denen vordefinierte Werte zurückgegeben werden. Ein typischer Anwendungsfall ist eine Datenbankabfrage, die vordefinierte Werte zurückliefert, ohne jedoch eine reale Datenbankabfrage durchzuführen. Im Echtbetrieb wird dieses Fake-Objekt durch eine Komponente, die den Datenbankaufruf realisiert, ersetzt.

Mock-Objekte

Eine besondere Bedeutung nehmen die Mock-Objekte auf höheren Testebenen (etwa Integrations- oder Systemtests) ein, indem eine Komponente getestet werden kann, die über Schnittstellen mit diesen gemockten Objekten kommuniziert. Dabei werden über Test-Treiber die Umgebung bzw. benötigte Informationen anderer Komponenten simuliert. Wichtig dabei ist, dass die Schnittstellen klar definiert sind und bekannt ist, welche Informationen (etwa Variablen und Datenformate) verwendet werden können.

Mocking-Beispiel

Beispielsweise könnte eine Komponente Funktionalität von einem externen Service (etwa einem ERP-System) benötigen, um korrekt zu funktionieren. Für Komponententests ist es aber in der Regel nicht vernünftig, ein komplexes externes System tatsächlich zu installieren, um eine Komponente zu testen. In diesem Fall könnte man eben ein Mock-Objekt erstellen, das das externe System für den Test „vertritt“. Dabei simuliert es aber nicht etwa die Funktion des externen Systems, sondern gibt nur immer bestimmte vordefinierte Antworten auf Anfragen der Komponente zurück. Damit ist der Zustand des gemockten Systems auch deterministisch und führt nicht zu unerwartetem Verhalten. So kann das Testobjekt beispielsweise mit vordefinierten und statischen Informationen (Wiederholbarkeit und Nachvollziehbarkeit) versorgt und entsprechend getestet werden.

Mock-Objekte finden beispielsweise auch bei den Entwurfsmustern (siehe Kapitel 8) Anwendung.

5.6.5 Testfalldokumentation

Die Dokumentation von Testfällen und die Protokollierung der Ergebnisse ist eine wichtige Tätigkeit, um ein systematisches Testvorgehen zu ermöglichen und die Durchführung der Tests transparent und nachvollziehbar zu gestalten. Weiterhin hilft die Testfalldokumentation den Entwicklern bei der Fehlersuche, da Testfälle auch als *Kommunikationsmittel* eingesetzt werden können. Schlägt ein Testfall fehl, kann der Entwickler mit den Informationen aus der Testfallbeschreibung gezielt auf die Fehlersuche gehen. Dazu ist es allerdings notwendig, die Dokumentation der Testfälle systematisch und vollständig zu erstellen.

Ebenen der Tests

Die systematische Testfalldokumentation umfasst dabei alle Ebenen, beginnt also – aus Kundensicht – bei den System-, Akzeptanz- und Abnahmetests über Integrationstests, bis hin zu den einzelnen Komponententests. Je nach Testebene (siehe Abschnitt 5.6.3) adressieren die unterschiedlichen Tests definierte Teile des Gesamtsystems. Speziell bei Entwicklertests empfiehlt sich der Einsatz von Unit-Tests, da neben den Vorteilen für Automatisierung auch die Dokumentation und das Reporting (z. B. mit Code-Coverage-Methoden) mit abgedeckt werden kann. Je höher die Testebene ist, desto umfangreicher und komplexer werden die Unit-Tests. Daher kann es sinnvoll sein, Szenarien, die konkrete Benutzerabläufe abdecken, für die Gestaltung der Testfälle zu verwenden. Folgende Informationen sind daher typischerweise in der Testfalldokumentation zu finden:

Elemente einer Testfalldokumentation

- Eine *laufende Nummer* ist zur eindeutigen Identifikation des Testfalls notwendig.
- *Typ des Testfalls*. Das System muss nicht nur im Normalbetrieb die erwarteten Anforderungen erfüllen, sondern auch auf Sonderfälle und Fehlerfälle deterministisch reagieren können, um ein stabiles Systemverhalten zu garantieren. Dementsprechend ist es auch hilfreich, die Testfälle entsprechend in *Normalfälle (NF)* (für den Normalbetrieb), *Sonderfälle (SF)* und *Fehlerfälle (FF)* zu klassifizieren.
- Einen besonderen Stellenwert in der Testfalldokumentation nehmen die *Vorbedingungen* ein. Sie definieren die Rahmenbedingungen, unter denen der Testfall ausgeführt wird. Wichtig ist, dass für einen konkreten Testfall immer dieselben Vorbedingungen erfüllt sind, um die Nachvollziehbarkeit und Wiederholbarkeit der Tests zu gewährleisten.
- *Eingabewerte* umfassen konkrete Werte, die eine Komponente oder ein System verarbeiten sollen.
- Die *Testfallbeschreibung* erläutert, welche Komponente im Hinblick auf welche Eigenschaft überhaupt getestet werden soll.
- Eine besondere Bedeutung nehmen auch die *erwarteten Ergebnisse* ein. Sie definieren, was nach Ausführung der Komponente vorliegen

soll. Die zu erwartenden Testergebnisse werden auch als *Testorakel* bezeichnet. Meist wird hier die konkrete Spezifikation des zu testenden Objekts verwendet, um die jeweiligen zu erwartenden Ergebnisse bzw. das zu erwartende Systemverhalten abzuleiten.

- Je nach Bedarf können weitere Informationen, beispielsweise der Bezug zu Äquivalenzklassen, zu den Komponenten, dem Design oder dem Use Case hergestellt werden. Sinnvoll sind auch Informationen über den Autor des Testfalls als Ansprechpartner für den Entwickler.

Frühe Definition der Testfälle!

An diesem Punkt sind die Testfälle spezifiziert (nicht jedoch notwendigerweise ausgeführt). Anzumerken ist, dass diese Testfälle bereits sehr früh, beispielsweise während der Anforderungsanalyse bzw. dem Entwurf des Systems, definiert werden können (*Test-First Development*). Das erleichtert das Verständnis des zu entwickelnden Systems und verbessert die Qualität der *frühen Produkte* durch qualitätssichernde Maßnahmen. Beispielsweise können durch die frühe Testfalldefinition falsche, fehlende oder widersprüchliche Informationen recht schnell aufgedeckt werden.

Dokumentation der Ergebnisse

Diese definierten Testfälle können dann im Rahmen der Implementierung für das Testen der Komponente oder des Systems eingesetzt werden. Die Ergebnisse werden entsprechend in der Testdokumentation ergänzt:

- Das *tatsächliches Ergebnis* dokumentiert das erhaltene Ergebnis, basierend auf definierten Vorbedingungen und unter Verwendung der festgelegten Eingabewerte.
- Im Feld *Entscheidung* wird dokumentiert, ob das tatsächliche Ergebnis mit dem erwarteten Ergebnis übereinstimmt, ob also ein Fehler vorliegt oder nicht.
- Es kann im Einzelfall sinnvoll sein, zusätzliche Bemerkungen oder eine Fehlernummer als Basis für die Nachverfolgung des Fehlers (*Issue Tracking*) anzugeben.

Beispiel: Dreiecksberechnung

Das folgende Bespiel illustriert anhand einer einfachen Komponente für eine Dreiecksberechnung, wie die Testfalldokumentation und die Dokumentation der Ergebnisse aussehen kann. Grundlage dafür ist die Spezifikation für eine konkrete Komponente. Abhängig von den Eingabeparametern (Seitenlängen eines Dreiecks) wird durch die Komponente bestimmt, ob es sich dabei um ein gleichseitiges, gleichschenkliges, rechtwinkliges oder überhaupt um ein gültiges Dreieck handelt, oder ob das Dreieck ungültig ist. Als Eingabewerte werden der Komponente drei Werte (a,b und c) übergeben, die die Länge eines Dreiecks repräsentieren. Tabelle 5.1 zeigt Beispiele für mögliche gültige und ungültige Eingabewerte, die zur Berechnung von unterschiedlichen Dreiecken verwendet werden.

Abgeleitet von der Spezifikation werden Testfälle mit konkreten Werten spezifiziert. In diesem Fall werden Black-Box-Tests eingesetzt. Die Abbil-

Tabelle 5.1 *Beispiele für gültige und ungültige Dreiecke.*

Gültige Dreiecke	**Eingabewerte**
Gleichseitiges Dreieck	(3,3,3)
Gleichschenkliges Dreieck	(5,5,3)
Rechtwinkliges Dreieck	(3,4,5)
Sonstiges Dreieck	(3,5,7)
Ungültige Dreiecke	**Eingabewerte**
a + b < c	(3,3,7)
a + b = c	(3,3,6)
negative Seitenlängen	(3,4,-5)
Nullwerte	(0,0,0)
falsche Anzahl von Eingabewerten	(2,4) oder (2,5,5,6)

dung 5.12 zeigt exemplarisch vier Testfälle: Testfall 1 und 2 decken Normalfälle mit korrektem Ergebnis ab. Bei Testfall 4 werden zu viele Parameter übergeben; dieser Fehlerzustand wird – wie erwartet – durch die Komponente erkannt. Testfall 3 testet einen konkreten Fehlerfall, ein ungültiges Dreieck mit $(a + b < c)$, wobei dieser Fehlerfall durch das System nicht richtig erkannt wird.

Fehlersuche und Korrektur sind keine Aufgaben des Testers

Offensichtlich wurde dieser Fehlerfall bei der Implementierung nicht richtig umgesetzt. Die Informationen aus der Testfalldefinition helfen dem Entwickler im Anschluss bei der Fehlersuche: Er kennt den Testfall, die Eingabewerte, Vorbedingungen und die erwarteten und tatsächlichen Ergebnisse. Diese Informationen ermöglichen sowohl die Nachvollziehbarkeit als auch die Wiederholbarkeit des Testfalls und der Ergebnisse und unterstützen den Entwickler bei der Fehlersuche.

5.6.6 Testintensität und Überdeckungsgrade

Testüberdeckung

Neben der Qualität der Testfälle ist die Testintensität ein Kriterium für Software-Testen. Die Testintensität (oder Testüberdeckung) liefert Informationen darüber, welche und wie viel des implementierten Softwarecodes durch Tests abgedeckt sind. Solche Metriken finden typischerweise bei den White-Box-Verfahren Anwendung und adressieren die konkrete Implementierung. Die Testüberdeckung stellt allerdings keinesfalls sicher, dass die

Nr.	Typ	Vor-bedingung	Eingabe-wert	Testfallbeschreibung	Erwartetes Ergebnis	Tatsächliches Ergebnis	Entscheidung
1	NF	System läuft	(3,3,3)	Dreieckü berprüfung wird aufgerufen;	Ausgabe „gleichseitig“	gleichseitig	✓ PASS
2	NF	System läuft	(5,5,3)	Dreieckü berprüfung wird aufgerufen;	Ausgabe: „gleichschenklig“	gleichschenklig	✓ PASS
3	FF	System läuft	(3,3,7)	Dreieckü berprüfung wird aufgerufen;	Ausgabe: „ungültig“	gleichseitig	✗ FAIL
4	FF	System läuft	(3,4,4,5)	Dreieckü berprüfung wird aufgerufen;	Ausgabe: „ungültig“	ungültig	✓ PASS

Abbildung 5.12
Beispiel für eine Testfalldokumentation.

```
...
1  public double calcDiscount (Iperson p) {
2      double discount = 0;
3      if (p.isStudent()) {
4          if (p.getAge() < 26) {
5              discount = 0.5;
6          } else {
7              discount = 0.33;
8          }
9      }
10     return discount;
11 }
...
```

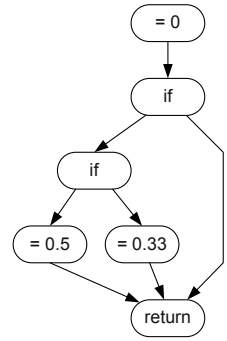


Abbildung 5.13
Beispiel: Rabattberechnung - Sourcecode und Kontrollflussgraph.

Tests auch sinnvoll und brauchbar sind – das muss bereits bei der Erstellung der Testfälle berücksichtigt werden. Beispielsweise täuschen „leere Tests“ zwar eine hohe Testüberdeckung vor, ohne überhaupt einen realen Test durchzuführen. Betrachtet man die Testüberdeckung ohne die eigentlichen Testfälle, kann man eigentlich nur feststellen, ob es Softwareteile gibt, die von keinem Test berührt werden.

Ausgewählte Überdeckungsarten

In diesem Abschnitt werden einige grundlegende Überdeckungsarten, wie *Anweisungsüberdeckung*, *Zweig- oder Kantenüberdeckung*, *Bedingungsüberdeckung* und *Pfadüberdeckung*, vorgestellt. Je nach Anwendungsfall ist die Auswahl einer geeigneten Überdeckungsart im Rahmen der Testplanung notwendig, da die Aufwände und die Anzahl der benötigten Testfälle für die Realisierung und für das Erreichen eines hohen Überdeckungsgrades zum Teil sehr hoch sein kann. Wie bereits erwähnt, ist der vorliegenden Sourcecode bzw. ein vorliegender Kontrollflussgraph die Grundlage für die Ermittlung des Überdeckungsgrades.

Beispiel: Rabattberechnung

Das folgende Beispiel (siehe Abbildung 5.13) zeigt eine konkrete Implementierung zur Berechnung eines Rabatts für Studierende und liefert den jeweiligen Rabattwert zurück. Der Methodenaufruf *p.isStudent()* liefert einen boolschen Wert zurück (Status der Person), und *p.getAge()* liefert die Altersabgabe der Person *p* vom Typ *IPerson* zurück. Der Rabattwert wird in Abhängigkeit des Alters und des Personenstatus (Student) berechnet. Für die Ermittlung der unterschiedlichen Überdeckungsmaße *(Coverage)* kann einerseits der Programmcode oder auch ein Kontrollflussgraph verwendet werden. Ausgehend von diesem Beispiel können die jeweiligen Überdeckungsgrade ermittelt werden.

Programmcode oder Kontrollflussgraph

Statement Coverage

Kernidee der *Anweisungsüberdeckung (Statement Coverage)* oder auch C0-Überdeckung ist es, jede Anweisung zumindest einmal auszuführen. Damit soll ermittelt werden, ob alle Anweisungen überhaupt durch Testfälle erreicht werden können. Die Anweisungsüberdeckung wird als Verhältnis aller durchlaufenden Anweisungen zu allen vorhandenen Anweisungen berechnet. Die Methode enthält sechs Anweisungen, die auch über den Kontrollflussgraphen abgebildet sind. Durch einen konkreten Testfall soll eine Rabattberechnung für einen Studenten mit 27 Jahren durchgeführt

```
    ...
1   public double calcDiscount (Iperson p) {
2 1     double discount = 0;
3 1     if (p.isStudent()) {
4 1         if (p.getAge() < 26) {
5 0             discount = 0.5;
6           } else {
7 1             discount = 0.33;
8           }
9       }
10 1    return discount;
11  }
    ...
```

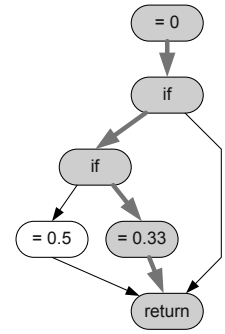


Abbildung 5.14
Beispiel: Anweisungsüberdeckung.

werden. Der Testfall deckt dabei fünf Anweisungen ab. Abbildung 5.14 zeigt die durch diesen Testfall ausgeführten Anweisungen. Daraus ergibt sich eine Anweisungsüberdeckung von *5/6 = 83,3%*. Eine vollständige Anweisungsüberdeckung würde einen weiteren Testfall, z. B. Rabattberechnung für einen Studenten mit 23 Jahren, erfordern.

Branch Coverage

Ein etwas komplexeres Überdeckungsmaß zielt auf die unterschiedlichen Möglichkeiten ab, auf welche Weise der Sourcecode oder der Kontrollflussgraph durchlaufen werden kann. Die *Zweig- oder Kantenüberdeckung (Branch Coverage)* – oder auch C1-Überdeckung – beschreibt die *durchlaufenden Wege* im Kontrollflussgraphen. Im Mittelpunkt stehen dabei Bedingungen, deren Ergebnisse für die Entscheidung, welcher Zweig durchlaufen werden soll, herangezogen wird; man spricht auch von *Entscheidungsüberdeckung (Decision Coverage)*. Wichtig dabei ist, dass alle Verzweigungen des Kontrollflusses durch Testfälle abgedeckt werden müssen, also *true* und *false* bei Bedingungen, alle Variationen bei *case-* oder *switch-statements* und auch Exceptions und Rücksprünge. Werden Bedingungen z. B. durch logische Operatoren zusammengesetzt, wird bei der Zweig- oder Kantenüberdeckung jeweils die gesamte Bedingung als Entscheidungsbasis verwendet.

Abbildung 5.15 illustriert die Zweig- oder Kantenüberdeckung für die Rabattberechnung. Insgesamt existieren zwei Bedingungen mit jeweils zwei Möglichkeiten (*true* oder *false*). Das ergibt insgesamt vier mögliche Zweige durch den Kontrollflussgraphen. Durch den Testfall (Student mit 27 Jah-

```
    ...
1   public double calcDiscount (Iperson p) {
2 1     double discount = 0;
3 1     if (p.isStudent()) {
4 1         if (p.getAge() < 26) {
5 0             discount = 0.5;
6           } else {
7 1             discount = 0.33;
8           }
9       }
10 1    return discount;
11  }
    ...
```

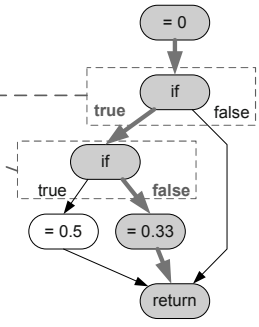


Abbildung 5.15
Beispiel: Zweig- oder Kantenüberdeckung.

ren) wird für die erste Bedingung *true* ermittelt (die Person ist Student) und die zweite Bedingung auf *false* ausgewertet (die Person ist älter als 26). Durch die beiden durchlaufenen Zweige ergibt sich eine Branch Coverage von *2/4 = 50%*. Wollte man sämtliche Zweige durch Testfälle abdecken, wären zwei zusätzliche Testfälle (Student und jünger als 26 bzw. kein Student) erforderlich, wobei einige Kanten mehrfach durchlaufen werden müssen. Die Zweig- oder Kantenüberdeckung gewährleistet auch eine Anweisungsüberdeckung, da zwangsläufig alle Anweisungen durchlaufen werden. Umgekehrt gilt dies jedoch nicht!

Condition Coverage

In vielen Fällen setzen sich Bedingungen aus *atomaren Bedingungen* (siehe Beispiel) und *logischen Operatoren* zusammen. Während bei der beschriebenen Zweig- oder Kantenüberdeckung die Bedingung als Gesamtes ausgewertet wird, ist bei der *Bedingungsüberdeckung (Condition Coverage)* ein detailliertes Testen der zusammengesetzten Bedingungen erforderlich. In der Literatur findet man unterschiedliche Arten von Bedingungsüberdeckungen (*Branch Condition Testing*) [85], die je nach geforderter Testintensität angewandt werden können:

Einfache Bedingungsüberdeckung

Das Ziel der einfachen Bedingungsüberdeckung ist es, dass jede atomare Teilbedingung bei der Durchführung der Testfälle beide Wahrheitswerte (*true* und *false*) annehmen muss. Bei der einfachen Bedingungsüberdeckung werden nur die Teilbedingungen getestet und die zusammengesetzte Bedingung nicht betrachtet. Daher kann es vorkommen, dass nicht alle Zweige getestet werden. Dieses Überdeckungsmaß sollte daher eher vermieden werden.

Mehrfachbedingungsüberdeckung

Dieses Problem soll durch die Mehrfachbedingungsüberdeckung *(Branch Condition Combination Testing)* behoben werden. Dabei werden alle Kombinationen von atomaren Bedingungen sowie deren Zusammensetzung über Testfälle getestet. Um alle Testfälle abzudecken, können beispielsweise Wahrheitstabellen verwendet werden. Allerdings kann es beim realen Testen vorkommen, dass nicht alle Kombinationen möglich sind und durch Testfälle daher nicht abgebildet werden können.

Minimale Mehrfachbedingungsüberdeckung

Die minimale Mehrfachbedingungsüberdeckung (Modified Branch Condition Decision Testing) bedeutet eine Einschränkung der Mehrfachbedingungsüberdeckung, indem nur diejenigen Kombinationen berücksichtigt werden müssen, bei denen die Änderung einer atomaren Teilbedingung auch eine Änderung der Gesamtbedingung verursachen kann.

Path Coverage

Die Zweig- und Kantenüberdeckung betrachtet die Wege durch das Testobjekt isoliert, ohne etwaige Abhängigkeiten zur berücksichtigen. Die *Pfadüberdeckung (Path Coverage)* fordert die Ausführung aller unterschiedlichen Pfade durch das Testobjekt. Unter einem Pfad wird eine eindeutige Abfolge (Sequenz) von Bedingungen und Anweisungen (also von Zweigen) vom Aufruf bis zum Verlassen eines Testobjekts verstanden. Wesentlich dabei ist, dass hier auch Abhängigkeiten betrachtet werden, wie sie beispielsweise bei Schleifen auftreten können, und nicht nur iso-

lierte Zweige (wie bei der Zweig- oder Kantenüberdeckung). Nachteilig wirkt sich aus, dass die Anzahl der möglichen Pfade sehr schnell ansteigt. Beispielsweise würden zehn if-Bedingungen 1024 Testfälle benötigen, bei Schleifen steigt die Anzahl der Pfade ins Grenzenlose. Daher ist während der Testplanung zu berücksichtigen, ob bzw. bei welchen Teilen eines Systems eine Pfadüberdeckung gefordert werden soll.

Welches Überdeckungsmaß?

Jetzt stellt sich abschließend noch die Frage, welches Überdeckungsmaß gewählt werden soll und wie hoch der angestrebte Überdeckungsgrad sein sollte. Häufig werden in der Literatur Überdeckungsgrade von 80 bis 90% empfohlen. Allerdings hängt es in der Praxis meist von Projektparametern wie Risiko, Kosten-Nutzen-Überlegungen und der Projekt- bzw. Qualitätsplanung ab, welcher Überdeckungsgrad anzustreben ist. So können erfolgreiche Projekte auch geringere Überdeckungsgrade aufweisen.

Welche Analysemethode?

Welche Coverage-Analysemethode verwendet werden sollte, hängt von den konkreten Projektparametern ab. Oft wird die Anweisungsüberdeckung ein sinnvoller Startpunkt sein; je nach Umfeld können aufwendigere und komplexere Maße verwendet werden. Bei besonders kritischen Komponenten kann durchaus auch eine Pfadüberdeckung gefordert sein.

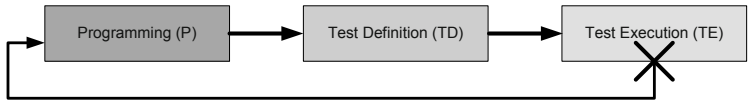
Überdeckungsgrad und Testqualität

Es muss wiederholt werden, dass ein hoher Überdeckungsgrad *nicht* das Vorhandensein von guten Tests bedeutet. Dazu ist eine Analyse der konkreten Testimplementierungen erforderlich. Die Anwendung von Überdeckungsmaßen kann aber durchaus als Anhaltspunkt für die Qualität des Produkts verwendet werden. Zur Feststellung der Überdeckungsgrade ist es in der Praxis empfehlenswert, Werkzeuge zur automatisierten Ermittlung dieser Kennzahlen zu verwenden (siehe dazu auch Werkzeuge in Abschnitt 5.8).

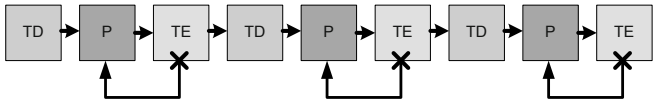
5.7 Test-Driven Development

Traditionelles Testen

Traditionelles Software-Testen verfolgt, als etablierte Methode der Qualitätssicherung, das Ziel, Fehler in bereits existierenden Lösungen zu finden. Der traditioneller Ansatz von Software-Testen baut auf dem Software-Lebenszyklus auf und setzt voraus, dass abgestimmte Anforderungen und Spezifikationen vorliegen und diese auch entsprechend umgesetzt bzw. implementiert wurden. Erst parallel zur Entwicklung oder gar erst nach Abschluss der Implementierung kann man Testfälle geeignet definieren und ausführen. Diese Vorgehensweise hat den Nachteil, dass Testen erst sehr spät im Entwicklungsprozess stattfindet. Fehler in den Anforderungen, dem Entwurf, dem Design und der Implementierung können ebenfalls erst sehr spät – in vielen Fällen zu spät – erkannt werden. Abbildung 5.16a zeigt den schematischen Ablauf dieses Ansatzes.



(a) Traditionelle Einbettung von Software-Testen.



(b) Test-First Development / Test-Driven Development.

Abbildung 5.16 *Schematische Darstellung unterschiedlicher Testansätze.*

Test-Driven Development

Die unter Umständen recht lange Zeit, bis mit dem Testen begonnen werden kann, erschwert die Erstellung von qualitativ hochwertigen Produkten. Eine Verkürzung der Zeitspanne, kurze Iterationen und unmittelbares Feedback der aktuellen Umsetzung im Hinblick auf die zugehörige Spezifikation kann erheblich zur Verbesserung der Produkt- und Prozessqualität beitragen. Die Kernidee der *test-getriebenen Entwicklung (Test-Driven* oder *Test-First Development)* ist es, Testfälle bereits *vor* oder spätestens *parallel* zur Implementierung zu spezifizieren. Ein weiterer wichtiger Punkt ist, dass das System quasi über Tests entworfen wird. Der Entwickler macht sich im Rahmen der Testfallerstellung darüber Gedanken, was die zu schreibende Komponenten leisten soll, wie sie verwendet wird, wie sie mit anderen Komponenten kommuniziert und wie sie getestet wird. Aufbauend darauf werden die Tests erstellt. Abbildung 5.16b zeigt exemplarisch einen schematischen Ablauf dieses test-getriebenen Ansatzes. Am Beginn steht die Testfalldefinition (Test Definition, TD), gefolgt von der Umsetzung und Programmierung (Programming, P). Nach der Umsetzung werden die zuvor gestellten Testfälle ausgeführt (Test Execution, TE).

Etabilierte Agile Praxis

Test-Driven Development (TDD) hat sich im Rahmen der *agilen Software-Entwicklung* als effiziente agile Praxis etabliert und ist fixer Bestandteil zahlreicher Vorgehensmodelle, wie eXtreme Programming und SCRUM. Zunehmend wird TDD auch in Projekten, die traditionellen Vorgehensmodellen folgen, eingesetzt. Durch das frühe Erstellen der Testfälle erhält der Entwickler unmittelbares Feedback zur erstellten Lösung. Vorgaben (z. B. Anforderungen und Spezifikationen) können so auf Plausibilität geprüft werden. Weiters stehen die erstellten Tests im Rahmen der Implementierung zur Verfügung, sodass Probleme und Seiteneffekte frühzeitig und unmittelbar erkannt werden können.

5.7.1 Test-Driven Development im V-Modell Umfeld

Test-First Development im V-Modell

Der *Test-First*-Gedanke auf unterschiedlichen Ebenen kann auch mithilfe des V-Modells gut illustriert werden. Das V-Modell mit den unterschiedlichen Sichten auf Verifikation und Validierung wurde bereits in Abschnitt 5.2 am Beginn dieses Kapitels vorgestellt. Aus dem V-Modell lassen sich somit drei wesentliche Ebenen ableiten, auf denen *Test-First* effizi-

ent zur Verbesserung der Prozess-, Projekt- und Produktqualität eingesetzt werden kann: (a) *System-, Akzeptanz- und Abnahmetests aus Anwendersicht*, (b) *Integrationstests aus Architektursicht* und (c) *Komponententests aus Implementierungssicht*.

Anwendersicht

Aus *Anwendersicht* sind speziell die Anforderungen und die entsprechende Umsetzung in Form eines Gesamtsystems relevant. Im Rahmen der Anforderungs- und Systemspezifikation können Testfälle auf Anwenderebene klar spezifiziert und dann auf das fertiggestellte System angewandt werden. Diese Testfälle finden beispielsweise als *Systemtests*, *Akzeptanztests* oder *Abnahmetests* des Kunden Verwendung. Durch die Testfalldefinition wird recht früh festgelegt, in welcher Form Anforderungen testbar sind bzw. wie sie getestet werden müssen. Kann beispielsweise für eine funktionale oder nicht funktionale Anforderung kein Testfall erstellt werden, muss diese Anforderung hinterfragt werden.

Architektursicht

Aufbauend auf den Anforderungen und der Systemspezifikation werden Architektur und Schnittstellen im Rahmen des Entwurfs und Designs (siehe Abschnitt 2.5) festgelegt. Auch hier gilt es, möglichst frühzeitig die Test- und Nachvollziehbarkeit sicherzustellen. Daher bietet es sich an, sich hier Gedanken über die Integrationstests zu machen und die Testfälle entsprechend abzuleiten. Diese Tests können dann für *System- und Integrationstests* eingesetzt werden. Außerdem dient die Testfalldefinition der Überprüfung der Umsetzung von Anforderungen in ein technisches Design. Durch die Erstellung der Testfälle werden – ebenfalls sehr frühzeitig – mögliche Problemstellen und Fehler identifiziert und können korrigiert werden.

Implementierungssicht

Während höhere Ebenen des V-Modells eher auf Anwendungsfälle und Interaktionsthemen abzielen, adressieren *Komponententests* die Implementierung von einzelnen Komponenten. Hier findet sich auch das derzeitige Hauptanwendungsfeld von Test-Driven-Development-Ansätzen. Die Anforderungen an eine Komponente liegen im Detail vor, die Architekturentscheidungen sind getroffen, und die konkrete Spezifikation einer Komponente ist detailliert verfügbar. Die Testfälle adressieren die spezifizierten Produkteigenschaften und können direkt auf Komponentenebene angewandt werden. Dabei werden erst die Testfälle (beispielsweise mit Unit-Tests) spezifiziert, danach die Komponente implementiert und solange angepasst, bis die spezifizierten Tests fehlerfrei durchlaufen. Die Vorgehensweise von Test-Driven Development wird in Abschnitt 5.7.2 ausführlicher beschrieben.

Vorteile

Zusammengefasst ergeben sich aus der Anwendung des Test-First-Ansatzes eine Reihe von Vorteilen:

- Verbessertes Verständnis des zu entwickelnden Produkts.
- Frühzeitige Abschätzung von Machbarkeit und Testbarkeit.
- Automatisierung (siehe Abschnitt 10.3).
- Verbesserung der Kommunikation durch Testfälle und Testergebnisse.

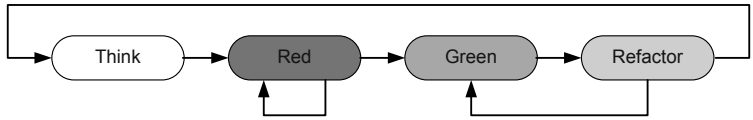


Abbildung 5.17 *Phasen des Test-Driven Development.*

Kommunikation durch Tests

Der letzte Punkt sollte etwas genauer erkärt werden: Oft ist die Kommunikation des Fehlverhaltens eines Systems (zwischen Entwicklern) nicht ganz einfach. Findet ein Entwickler oder Tester ein Problem, so wird anstatt mündlicher oder schriftlicher Erklärungen besser ein Test geschrieben, der den Fehler reproduziert. Damit wird eindeutig kommuniziert, was als Fehlverhalten verstanden wird, der Entwickler kann den Fix gegen den Test prüfen, und die Codebasis ist um einen Test (der offenbar vorher vergessen wurde) reicher.

5.7.2 Ablauf von Test-Driven Development

Test-Driven Development (oder Test-First Development) zielt darauf ab, die Testfälle vor oder spätestens parallel zur Umsetzung (Implementierung) zu definieren. Im Anschluss an die Testfalldefinition werden die Komponenten implementiert bzw. angepasst, bis die definierten Tests fehlerfrei durchlaufen. Konkret läuft Test-Driven Development in vier grundlegenden Schritten ab, wie sie in Abbildung 5.17 skizziert werden:

Schritt 1: „think“

Im *ersten Schritt* erfolgt die Auswahl der Anforderung, die im nächsten Schritt umgesetzt werden soll. Basierend auf dieser Auswahl werden die geeigneten Tests definiert. Dabei ist darauf zu achten, dass die ausgewählte Anforderung durch die Tests auch tatsächlich abgedeckt wird. Auf Implementierungsebene (z. B. für eine Komponente) können beispielsweise Unit-Tests verwendet werden.

Schritt 2: „red“

Im *zweiten Schritt* werden dieses Tests ausgeführt. Nachdem es aber keine Implementierung dazu gibt, müssen diese Testfälle fehlschlagen (Status: red).

Schritt 3: „green“

Ziel ist es, die erstellten Testfälle nach Abschluss der Implementierung erfolgreich zu durchlaufen und in den Status „green“ zu gelangen *(Schritt 3)*. Dazu werden die Anforderungen schrittweise umgesetzt, implementiert und getestet, bis die Tests ohne Fehler durchlaufen. Sind die Testfälle nicht erfolgreich, werden Fehler korrigiert (falls die Anforderungen bereits umgesetzt sein sollten) oder die Funktionalität implementiert (falls die Anforderungen noch nicht umgesetzt sein sollten).

Schritt 4: „refactor“

Wurden die Tests erfolgreich durchlaufen (alle Tests befinden sich im Status „green“), erfolgt im *vierten Schritt* die Optimierung und Anpassung des geschriebenen Softwarecodes (Refactoring). Da dieses Refactoring eine Veränderung des Codes zur Folge hat, müssen die Testfälle jeweils durchgeführt werden (der Status „green“ darf nicht mehr verlassen werden). Nach diesem Schritt ist die Implementierung und das Testen der aus-

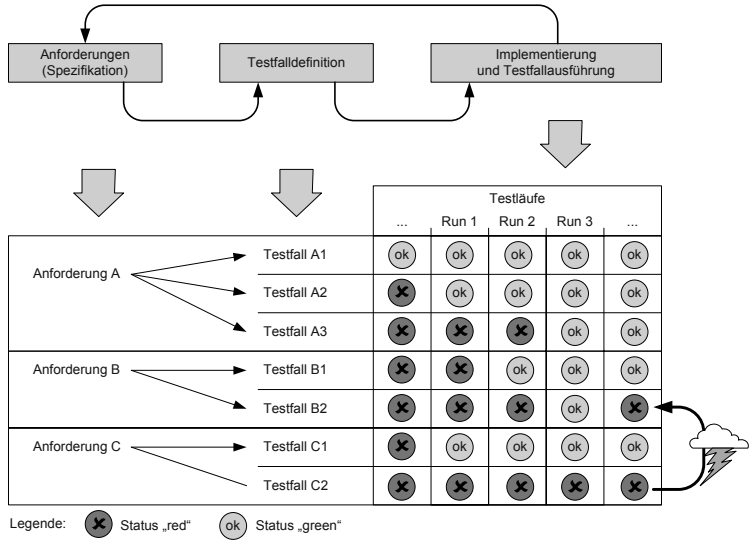


Abbildung 5.18
Beispiel: Test-Driven Development in der Praxis.

gewählten Anforderung abgeschlossen; die nächste Anforderung kann umgesetzt werden (Schritt 1).

Anwendungsbeispiel

Die kontinuierliche Anwendung von Test-Driven Development im Rahmen einer Continuous-Integration-Strategie führt zu einem Szenario, wie es beispielsweise in Abbildung 5.18 dargestellt ist. Aus den Anforderungen werden geeignete Testfälle abgeleitet, um die Anforderung überprüfen zu können, z. B. wird die Anforderung A durch drei Testfälle abgedeckt. Häufige Testläufe (Test Runs, dargestellt auf der x-Achse) ermöglichen unmittelbares Feedback zum Stand der Entwicklung; der Status der Testfälle wechselt von *red* auf *green*, entsprechend den jeweiligen TDD-Schritten. Dadurch bekommt man einen guten Überblick über den aktuellen Projektstatus. Ein besonderer Vorteil ist, dass man durch häufige und automatische Testläufe auch über Auswirkungen auf andere Systemteile informiert wird. In diesem Beispiel bewirkt eine Implementierung, die Testfall C2 adressiert, nicht nur das Fehlschlagen des Testfalls C2 sondern auch das Fehlschlagen des Testfalls B2. Bei traditionellen Testansätzen wäre dieser Seiteneffekt vermutlich erst sehr spät, z. B. während Integrations- oder Systemtests aufgefallen.

5.8 Automatische Codeprüfung

Automatisierung und Werkzeugunterstützung

Aufgaben, die regelmäßig oder zumindest öfter durchgeführt werden sollen, sind Kandidaten für die Automatisierung oder wenigstens für eine effektive und effiziente Werkzeugunterstützung. Die Menge an verfügbaren Werkzeugen ist nahezu unbegrenzt. In der Praxis ist es jedoch sinnvoll, aus der breiten Auswahl an unterschiedlichen Werkzeugen, diejenigen auszuwählen und einzusetzen, die im jeweiligen Projekt- oder Unter-

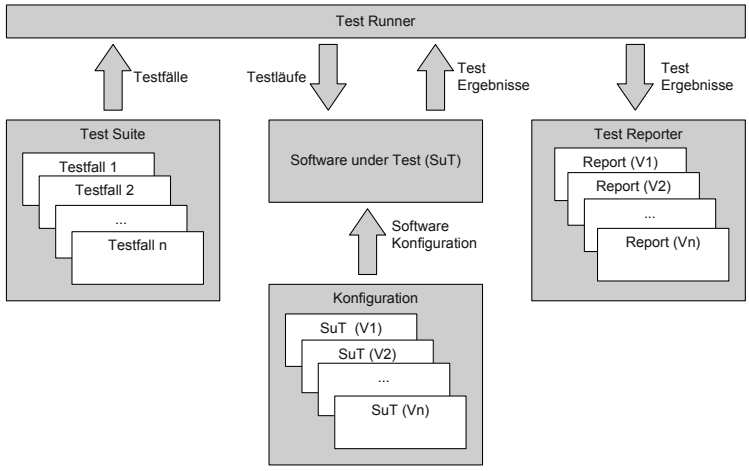


Abbildung 5.19
Test-Framework im schematischen Überblick.

nehmenskontext den größten Nutzen erwarten lassen. Der folgende Abschnitt beschäftigt sich exemplarisch mit drei wichtigen Themenbereichen, die im Rahmen der Qualitätssicherung im Bereich der Software-Entwicklung häufig anzutreffen sind. Weitere Techniken und Werkzeuge, speziell im Hinblick auf die Herstellung qualitativ hochwertiger Software-Produkte, werden in Kapitel 10 detailliert vorgestellt. Ausgewählte und nützliche Werkzeuge sind auch auf der begleitenden „Best-Practice Software-Engineering“ Projekt Webseite [2] zu finden.

5.8.1 Testautomatisierung

Verbesserung der Qualität durch Automatisierung

In jedem Entwicklungsprojekt werden Tests auf unterschiedlichen Ebenen durchgeführt. Im einfachsten Fall sind dies manuelle Tests des Entwicklers, um die grundlegende Funktionalität zu überprüfen. Da diese Test selten dokumentiert werden, sind sie weder nachvollziehbar noch wiederholbar oder gar für die Projektdokumentation verwendbar. Ziel der automatisierten Prüfung von Softwarecode ist eine systematische, wiederholbare, nachvollziehbare und automationsgestützte Durchführung definierter Tests, um Rückschlüsse über die Qualität des Testobjekts bzw. der *Software under Test (SuT)* ziehen zu können. Diese Tests können nach ihrer Erstellung beliebig oft mit keinem bis relativ geringem Mehraufwand durchgeführt werden. Typische Anwendungen finden sich etwa bei *Continuous-Integration-Strategien* (siehe auch Abschnitt 10.3.6) im Rahmen der Entwicklung oder *Regressionstests* bei Änderungen.

[2] `http://bpse.ifs.tuwien.ac.at/`

Test-Framework

Diesen Vorteilen steht aber der Nachteil gegenüber, dass die Erstellung von Testautomatisierungskonzepten durchaus sehr aufwendig und teuer sein kann. Eine Voraussetzung für die Umsetzung von Testautomatisierungsstrategien ist das Vorhandensein von geeigneten Test-Frameworks, also von Infrastrukturen, die die gesamte Testumgebung bereitstellen können. Abbildung 5.19 zeigt einen schematischen Überblick über ein einfaches Test-Framework, das aus folgenden wesentlichen Elementen besteht:

Test-Framework Komponenten

- *Software and System under Test (SuT)* als Testobjekte, die getestet werden sollen.
- Die *Test Suite* umfasst eine Sammlung von konkreten Testfällen. Testfälle können entweder manuell oder mit geeigneten Testgeneratoren erstellt werden. Eine andere Möglichkeit der Gewinnung von Testfällen ist die Aufzeichnung und das „Wiederabspielen“ von Testfällen (Capture/Replay).
- In allen Fällen ist aber ein Mechanismus notwendig, der feststellt, ob die tatsächlichen Ergebnisse des Testobjekts mit den erwarteten Werten übereinstimmen. Dieser Überprüfungsmechanismus wird meist durch das *Test-Framework* bereitgestellt.
- Ein *Test Runner* sorgt dafür, dass die benötigen Vorbedingungen für die Testfallausführung vorhanden sind und führt die Tests bzw. Test Suites nacheinander aus.
- Die Aufgabe des *Test-Reporters* ist es, die Ergebnisse der automatisierten Tests zu sammeln und aufzubereiten. Neben den grundlegenden Auswertungen über erfolgreiche und fehlgeschlagene Tests können Test-Reporter auch entsprechende Coverage-Analysen durchführen.

xUnit

Zur Implementierung und automatisierten Ausführung von Softwaretests werden je nach Testebene (Unit-, Integrations-, Systemebene) verschiedene Test-Frameworks eingesetzt. Der Begriff xUnit bezeichnet Frameworks für verschiedene Programmiersprachen. Das wohl bekannteste Beispiel ist JUnit für Java. Mittlerweile gibt es für fast jede weitverbreitete Programmiersprache ein solches xUnit-Framework – NUnit für .Net, CppUnit für C++ oder RUnit für Ruby. xUnit-Frameworks implementieren meist ihre eigenen *Assert-Methoden*, um aussagekräftige Fehlermeldungen zu liefern, und arbeiten mit verschiedenen automatisierten Buildsystemen wie Ant, Maven, Hudson oder CruiseControl zusammen, um das Ausführen der Tests in den Buildprozess zu integrieren. xUnit-Frameworks integrieren auch mit verschiedenen Reporting-Werkzeugen wie etwa dem BIRT-Projekt (Business Intelligence and Reporting Tools) von Eclipse, um über Berichte über Testläufe zu erstellen. Diese Berichte unterstützen sowohl die Entwickler aber auch das Management durch Informationen über den Projektfortschritt.

xUnit für Unit-, Integration- und Systemtests

Da all diese Frameworks den Begriff „Unit“ im Namen haben, wird oft angenommen, dass sie nur zum Unit testen verwendet werden können, dem ist aber nicht so. Ein xUnit-Framework kann durchaus auch für Integrationstests und mithilfe der richtigen Bibliotheken auch für automatisierte Systemtests eingesetzt werden. Für Integrationstests werden mehrere Komponenten eines Systems, die bereits Unit-Tests unterzogen wurden, mit ihren Interfaces „zusammengehängt“ und auch mittels eines xUnit-Frameworks getestet. Diese Tests sind zwar komplexer und aufwendiger als Unit-Tests, trotzdem reichen die von den meisten xUnit-Frameworks zur Verfügung gestellten Möglichkeiten aus, um Integrationstests ablaufen und auch automatisieren zu können. Für Systemtests sind weitere Werkzeuge notwendig, die aber meistens ein xUnit-Framework erfordern, um ihre Aufgaben zu erfüllen. xUnit ist somit ein Eckpfeiler des automatisierten Software-Testens, unabhängig von der verwendeten Programmiersprache und unabhängig von der Art des durchzuführenden Tests.

SWTBot für Systemtests

Automatisierte Systemtests für grafische Benutzerschnittstellen lassen sich mittels SWTBot realisierten. SWTBot ist Open Source und bietet eine einfache API sowie eigene Assert-Methoden an, um funktionale Tests für auf SWT und Eclipse basierte Applikationen zu implementieren. Mit der richtigen Planung während der Entwicklung eines Software-Produkts ist es sogar möglich, mithilfe eines „Recorders“ Tests anhand der laufenden Applikation „aufzunehmen“ und anschließend wiederzugeben. Während des „recordings“ wird Sourcecode generiert, der die Bibliotheken von SWTBot verwendet, um Elemente im Benutzerinterface anzusteuern. Dieser generierte Code kann auch nachbearbeitet bzw. ganz von Hand geschrieben werden. Der Einsatz eines solches Recorders erleichtert nicht nur die Implementierung der Tests sondern ermöglicht es auch „normalen“ Benutzern ohne technisches Hintergrundwissen, Systemtests „aufzunehmen“, Tests durchzuführen und erkannte Fehler zu dokumentieren.

Unabhängig von der Art des Tests kann eine gute Werkzeug-Unterstützung dabei helfen, Tests festzuhalten und automatisiert wiederzugeben. Im einfachsten Fall ist das eine Überprüfung, ob der Sourcecode das macht was er machen sollte. Anstatt immer über die Benutzerschnittstelle zu testen, können also xUnit-Frameworks eingesetzt werden, um auf tieferen Ebenen Tests festzuhalten. Auf Systemebene sollten Werkzeuge wie SWTBot eingesetzt werden, um auch diese Art von Test effizient durchführen und dokumentieren zu können.

Diese Automatisierungsmechanismen liefern, bei richtiger Anwendung von Test-Frameworks, gute Ergebnisse und können als integraler Bestandteil des Entwicklungsprozesses eingesetzt werden [36]. Automatisch generierte Test-Reports und Analysen können dem Projekt- und Qualitätsleiter entsprechende Informationen über den Projektfortschritt und die Produktqualität liefern und so auch zur Projektsteuerung eingesetzt werden.

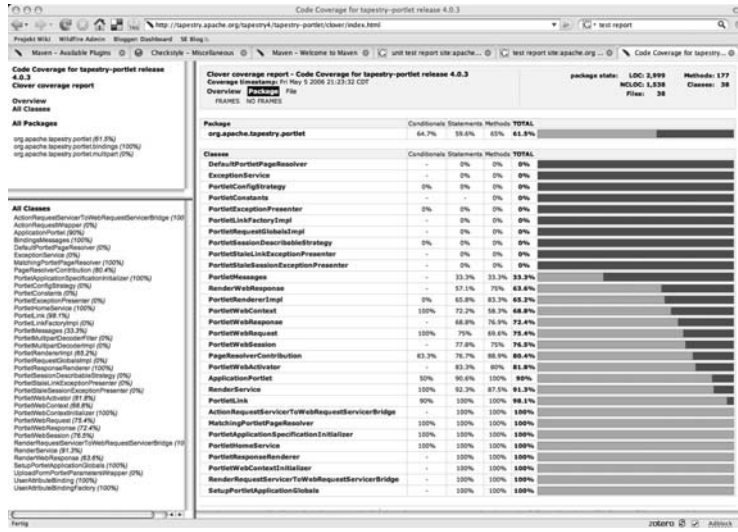


Abbildung 5.20 *Beispiel für eine Code-Coverage-Analyse von JUnit.*

5.8.2 Code Coverage Analyse

Code Coverage Analyse

Neben der Durchführung der Tests ist auch die Testintensität ein mögliches Kriterium für die *Qualität des Testens* (siehe dazu auch Abschnitt 5.6.6). Auch hier gibt es zahlreiche Tools, die Metriken und Messwerte für die Testintensität zur Verfügung stellen. Abbildung 5.20 zeigt etwa die Code Coverage basierend auf Unit-Tests (JUnit), die für ein konkretes Projekt automatisiert erstellt wird. Das Beispiel zeigt etwa die Testüberdeckung für Bedingungs-, Anweisungs- und Methodenüberdeckung für ein Apache Project (Tapestry). Der durch Balken visualisierte Überdeckungsgrad zeigt deutlich, welche Klassen bereits getestet sind, und wo noch Handlungsbedarf besteht.

CodeCover

Werkzeuge wie z. B. CodeCover, beobachten den Testlauf und berichten über dessen Qualität, indem jene Zeilen des Sourcecode markiert werden, welche nicht durch die Tests exekutiert wurden. Bei besonders großen Testbibliotheken, die lange zur Ausführung brauchen, kann CodeCover jene Tests bestimmen, die „neue“ Funktionalität testen und somit höhere Priorität haben. Wichtig ist aber zu verstehen, dass Werkzeuge wie CodeCover nur funktionieren, wenn die Tests auch gut implementiert sind. Es ist immer möglich, Tests zu schreiben, die große Teile des Sourcecodes abdecken, aber eigentlich nichts testen, weil sie etwa keine Assert-Methoden enthalten.

Frühe Einführung von Analysetools empfehlenswert

Derartige Analysetools sind typischerweise auf eine Entwicklungsumgebung und eine Programmiersprache abgestimmt und können in die jeweiligen Entwicklungsumgebung integriert werden. Wichtig bei der Anwendung ist allerdings, dass derartige Tools bereits von Beginn an eingesetzt werden, da ein späterer Einsatz durchaus zu negativen Effekten führen kann. Wurde beispielsweise TDD nur sporadisch verwendet, kann recht

rasch Frustration einsetzen, wenn die Code Coverage trotz fortgeschrittenem Projektstatus als sehr gering ausgewiesen wird. Ist eine späte Einführung von Analysetools erforderlich, sollte diese Einführung schrittweise erfolgen. In der Praxis empfiehlt es sich daher, bereits frühzeitig xUnit-Frameworks, etwa nach [36], in der jeweiligen Zielumgebung einzusetzen.

5.8.3 Code Quality Checks

Qualität des Softwarecodes

Die Qualität des erstellten Softwarecodes ist entscheidend für eine effiziente Software-Entwicklung im Team. Die Einhaltung von Dokumentations- und Programmierrichtlinien erleichtert einerseits die Zusammenarbeit innerhalb des Teams während der Entwicklung und erhöht auch die Wartbarkeit in der späteren Betriebsphase. Zu diesen Richtlinien gehören beispielsweise Namenskonventionen, die Verwendung von Kommentaren, Einrückungen und vieles mehr.

In der Praxis können auch hier Werkzeuge, wie beispielsweise CheckStyle eingesetzt werden, die Unstimmigkeiten zwischen dem konkreten Software-Dokument und der Richtlinie aufdecken können. Auch diese Werkzeuge sind auf die Entwicklungsumgebung bzw. Programmiersprache abgestimmt. Für Java steht beispielsweise das Werkzeug *CheckStyle* zur Verfügung, das für derartige Überprüfungen eingesetzt werden kann. Exemplarisch für die vielfältigen Möglichkeiten, die das Werkzeug zur Verfügung stellt, zeigt die folgende Liste einige mögliche Anwendungsmöglichkeiten:

Beispiel für Code Quality Checks

- Überprüfung von *JavaDoc-Kommentaren*, die dann in der Produktdokumentation verwendet werden können.
- Einhaltung von *Namenskonventionen* zur Verbesserung der Zusammenarbeit und einheitlichen Gestaltung der Klassen-, Methoden- und Variablenbezeichnungen.
- *Klammernsetzung und Einrückungen* zur Verbesserung der Lesbarkeit des Softwarecodes.
- *Duplicate Code*. Diese Funktionalität dient primär dazu, mehrfach verwendeten (identen) Softwarecode zu identifizieren. Duplikate sollten grundsätzlich vermieden werden, da Änderungen an mehreren Stellen durchgeführt werden müssen und daher leicht Fehler entstehen können. In der Praxis taucht dieses Problem jedoch sehr häufig auf.
- *Komplexitätsanalysen* wie beispielsweise Verschachtelungstiefe in Klassen und Methoden. Je größer die Verschachtelungstiefe ist, desto schwieriger ist es in der Regel, die Komponente zu verstehen. Daher ist diese Auswertung sinnvoll, um die Verständlichkeit des Codes im Hinblick auf die Komplexität zu analysieren.

Plattform-abhängigkeit

Derartige Werkzeuge sind stets sehr spezifisch auf die Programmiersprache abgestimmt, da es zu jeder Sprache meist eine Vielzahl an Code-Konventionen, Dokumentationsrichtlinien und Best-Practices gibt. Zudem gibt es meist mehrere Tools für eine Sprache mit typischerweise vergleichbaren Features, aber unterschiedliche Vorgehensweisen (z. B. Sourcecode oder Byte-Code-Analyse in Java). Ein Vorteil dieser Tools ist, dass sie nicht nur in die Entwicklungsumgebung, sondern auch in den Build-Prozess integriert werden können. Dadurch erfolgt die Analyse und Berichterstattung über den Projektstatus im Rahmen des Daily Build automatisch und steht dem Entwicklungsteam unmittelbar nach dem jeweiligen Build zur Verfügung. Weitere Möglichkeiten zur Automatisierung von Entwicklungsschritten werden ausführlich in Kapitel 10 vorgestellt.

5.9 Zusammenfassung

Verifikation und Validierung

Die Qualitätssicherung begleitet das Software-Projekt während der gesamten Laufzeit. Typischerweise werden analytische Methoden der Qualitätssicherung eingesetzt, um Qualitätsmerkmale auf Konformität mit den Vorgaben aus unterschiedlichen Phasen zu verifizieren und im Hinblick auf die Kundenanforderungen zu validieren. Qualitätsmodelle stellen einen organisatorischen Rahmen dar, um Qualitätsmerkmale entsprechend zu klassifizieren. Je früher ein Fehler erkannt und beseitigt wird, umso kostengünstiger ist dies möglich. Werden beispielsweise Architektur- und Designfehler erst in späten Phasen der Entwicklung erkannt (z. B. während der Implementierungs- und Integrationsphase oder erst in der Betriebs- und Wartungsphase) entstehen sehr hohe Aufwände für die Fehlerkorrektur, die bis zum Abbruch des Projekts führen können.

Software-Reviews und Inspektionen

Software-Reviews und Inspektionen sind Maßnahmen der Qualitätssicherung, die für die Analyse und Fehlersuche in Dokumenten und Modellen aber auch Softwarecode bereits in frühen Phasen der Entwicklung eingesetzt werden können. Unterschiedliche Reviewtypen ermöglichen es, unterschiedliche Fehlerbilder zu adressieren, indem sie das Produkt aus verschiedenen Perspektiven betrachten und definierte Zielsetzungen verfolgen. Aufgrund des zum Teil hohen Formalitätsgrades dieser analytischen Methoden sind klare Rollen, Aufgaben und Verantwortlichkeiten definiert. Weiterhin folgt ein Review oder eine Inspektion einem klar festgelegten Prozess, in dem die einzelnen Schritte vorgegeben sind. Speziell bei Software-Inspektionen kommen systematische Lesetechniken zum Einsatz, die *die Art des Lesens* definieren und den Reviewer beispielsweise mit Checklisten aktiv bei der Fehlersuche unterstützen. Reviews und Inspektionen sind aufgrund ihrer Struktur und ihrer Zielsetzung gut geeignet, Fehler in frühen Phasen der Entwicklung (z. B. Anforderungen und Design) zu erkennen.

Architekturevaluierung

Je nach Anwendungsdomäne und Projekt existieren unterschiedlichen Varianten, wie das Software-Produkt aufgebaut sein soll (Entwurf, Design und Architektur). Daher ist es in der Regel sinnvoll, sich unterschiedliche Architekturvarianten (auch im Hinblick auf nicht funktionale Anforderungen, wie Erweiterbarkeit, Skalierbarkeit oder Performance) zu überlegen. Stehen mehrere Architekturvarianten zur Auswahl, ist eine Evaluierung der jeweiligen Architektur sinnvoll, um die passende Variante auszuwählen und einzusetzen.

Statische und dynamische Methoden

Neben den statischen Methoden wie Reviews und Inspektionen nehmen dynamische Methoden der Qualitätssicherung, wie Software-Testen, einen besonderen Stellenwert ein, um implementierte Produkte (existierenden Sourcecode) effizient auf Fehler zu untersuchen. Software-Testen verfolgt das Ziel, gezielt und systematisch Fehler in Software-Dokumenten zu finden. Auch das traditionelle Software-Testen läuft nach einem definiertem Prozess ab, der von der Planung des Testens, über Analyse und Design der Tests, Realisierung und Durchführung bis zur Berichterstattung alle wesentlichen Aspekte beinhaltet.

Testebenen

Je nach Art des Testens unterscheidet man unterschiedliche Testebenen: von der Komponentenebene aus Implementierungssicht über die Architektur- und Designebene aus Integrationssicht bis zur Anforderungsebene aus Kundensicht. Eine erfolgreiche Teststrategie erfordert auch passende Methoden. Black-Box-Techniken fokussieren primär auf Anforderungen und Spezifikationen, in dem die innere Struktur (und Umsetzung) der Komponente ausgeblendet wird. White-Box-Techniken nutzen die Analyse der tatsächlichen Implementierung und adressieren somit Fehler in der inneren Logik bzw. den jeweiligen Algorithmen. Aufgrund der Vielzahl an unterschiedlichen Möglichkeiten, eine Komponente zu testen, ist eine Einschränkung auf eine repräsentative Teilmenge an Eingabewerten erforderlich. Äquivalenzklassenzerlegungen und Grenzwertanalysen helfen bei der Testfallerstellung, besonders kritische Testfälle zu finden.

Black- und White-Box-Techniken

Auswahl von Testdaten

Dokumentation

Um die Tests nachvollziehbar und wiederholbar zu gestalten, ist eine geeignete und vollständige Dokumentation der Testfälle und der Testergebnisse erforderlich. Diese Informationen unterstützen sowohl den Entwickler bei der Fehlersuche also auch die Kommunikation mit dem Tester. Die Qualität der Testfälle wird nicht nur durch den eigentlichen Testfall bestimmt, sondern wird auch über den Überdeckungsgrad der zu testenden Komponente bestimmt. Dabei werden Überdeckungsmaße als Maßstab für die Testintensität verwendet. Überdeckungsmaße zeigen, welche Teile des Produkts in welchem Umfang getestet wurden, und wo noch Nachholbedarf besteht.

Test-Driven Development

Test Driven Development stammt aus der agilen Software-Entwicklung und rückt Software-Tests mit kurzen Iterationen in den Vordergrund. Dabei werden Test- und Implementation fast gleichzeitig geschrieben und Tests automatisiert ausgeführt. Durch die Einbettung in die Entwicklungsumgebung wird der Grundstein für Daily Builds und die Continuous-Integration-Strategie gelegt. Dadurch ist jederzeit der aktuelle Projektstatus ersichtlich und

die Entwickler erhalten eine unmittelbare Rückmeldung über die aktuell umgesetzten Features und mögliche Seiteneffekte zwischen Komponenten.

Automatisierung

Manuelles Testen ist in der Regel zeit- und kostenintensiv und dies führt meist dazu, dass die Testfälle „eher selten“ ausgeführt werden. Daher ist es sinnvoll, automatisierte Mechanismen einzubinden, die diese Aufgabe übernehmen können. Speziell im Zusammenhang mit Test-Driven Development haben sich Testautomatisierungsansätze entwickelt, die ein häufiges Durchführen von automatischen Tests erlauben. Automatische Testläufe können direkt in den Build-Prozess integriert werden, und der aktuelle Projektstatus ist jederzeit aktuell und verfügbar. Auch für die Code-Coverage-Analyse (Analyse der Testfallüberdeckung) stehen Werkzeuge zur Verfügung, die ein Maß für den Grad des Testens liefern und somit einen Indikator für die Qualität des Produkts darstellen.

6 | Notationen, Methoden der Modellierung

Dieses Kapitel gibt einen Überblick zu Notationen und Methoden der Modellierung für die Entwicklung kommerzieller Software-Produkte. Ziel ist die Vorstellung von Notationen, die es sowohl in der Entwicklung als auch bei Besprechungen mit Nicht-Entwicklern erlauben, in einer Gruppendynamik Gedanken zur Software-Entwicklung kohärent und korrekt nach innen und nach außen zu vermitteln und überprüfbar zu machen. Im administrativen Bereich hat sich in den letzten Jahren die Unified Modeling Language (UML) als Notation und Basis für Methoden weitgehend durchgesetzt. Die Unified Modelling Language erlaubt es, ein Software-System durch ein Modell vollständig zu dokumentieren. Dabei sollte ein Modell nicht mit einem Diagramm verwechselt werden. Jedes Software-System hat genau ein Modell, in dem sowohl Anforderungen (Use Cases) als auch die Struktur der Klassen (Klassendiagramm oder Komponentendiagramm) enthalten sind. Diagramme sind ein partielles Abbild des Modells eines Software-Systems. Die Sammlung von Notationen und Methoden in diesem Kapitel wird in diesem Buch durchgehend eingesetzt, um Konzepte und Strukturen zu illustrieren und zu beschreiben.

Übersicht

Abstraktion der Domäne

Die Zielsetzung in der Software-Entwicklung ist es, konkrete Problemstellungen der realen Welt – einer konkreten Domäne – zu lösen. Aus diesen Anforderungen werden Modelle entwickelt, die wesentliche Aspekte des zukünftigen Systems darstellen. Dabei können und sollen nicht alle Aspekte der Realität erfasst werden. Ein Modell ist also eine vereinfachte *(abstrakte)* Abbildung der realen Welt bzw. ein „Realitätsausschnitt" einer bestimmten Domäne. Die Modellierung selbst wird daher oftmals als Reduktion oder Abstraktion der Realität bezeichnet.

Ein Modell, viele Diagramme

Um die Komplexität zu reduzieren oder eine inhaltliche Überschneidung verschiedener Modelle zu vermeiden, hat es sich im Rahmen der Software-Entwicklung bewährt, eine geringe Anzahl von Modellen, im Idealfall ein einziges Modell, zu erstellen. Aufgrund der Größe und Komplexität des Modells sollte eine geeignete Werkzeugunterstützung verfügbar sein. Dieses Modell enthält alle wesentlichen Aspekte der Realität, mit denen sich die zu entwickelnde Software beschäftigt. Von diesem Modell werden anschließend die benötigten Diagramme abstrahiert, die in einer bestimmten Notation einen Aspekt des Modells visualisieren, beispielsweise die Struktur oder das Verhalten des zu erstellenden Systems.

Übersichtsdiagramme

Generell können zwei grundlegende Diagrammarten unterschieden werden, (a) *Übersichtsdiagramme* und (b) *technische Diagramme*. In *Übersichtsdiagrammen* werden komplexe Vorgänge einer Anwendungsdomäne auf die wesentlichen Kernkomponenten reduziert um ein *Big Picture*, also einen Überblick über das gesamte System, zu zeigen. Dadurch ist eine verständliche Darstellung des Systems auch für Personen möglich, die außerhalb des eigentlichen Entwicklungsprojekts stehen (beispielsweise die Entscheidungsträger des Auftraggebers). Beispielsweise finden sich Abwandlungen eines Verteilungsdiagramms (siehe Abschnitt 6.1.7) immer wieder auf Projektpräsentationen, um das Software-System übersichtlich darzustellen.

Technische Diagramme

Technische Diagramme ermöglichen eine effiziente Kommunikation mit den Entwicklern auf technischer Ebene und beinhalten in der Regel einen hohen technischen Detaillierungsgrad.

Aus den erstellten Diagrammen lassen sich die wesentliche Eigenschaften des Systems, beispielsweise bezüglich Vollständigkeit von Ein- und Ausgabedaten, ableiten. Dadurch ergibt sich die Möglichkeit, unklare Anforderungen rasch zu identifizieren und zu korrigieren. Beispielsweise sind aus dem UML-Komponentendiagramm (siehe Abschnitt 6.1.4) relativ leicht Datendefinitionen, Komponentenschnittstellen für die Implementierung und Testfälle ableitbar.

Modelle und deren Diagramme sind also Grundlage für eine klare, verständliche Kommunikation *nach außen* und effektives und effizientes Arbeiten *im Entwicklungsteam*. Voraussetzung dafür ist allerdings eine einheitliche Notation mit konsistenter Bedeutung.

Dieses Kapitel gibt einen Überblick über gebräuchliche Notationen in der modernen Software-Entwicklung, wie beispielsweise der Unified Model-

ling Language (UML), die sich als sehr vielseitig erwiesen hat. Dennoch hat es sich in der Praxis bewährt, auch andere Notationen (beispielsweise für den Datenbankentwurf oder das Projektmanagement) einzusetzen. Diese Notationen werden ab Abschnitt 6.2 vorgestellt.

6.1 UML-Diagrammfamilie

Modell und Diagramm

Ein UML-Modell ist eine Sicht auf die Anwendungsdomäne und besteht aus einer Menge von Modellelementen wie Klassen und Attributen. Ein Diagramm ist eine Darstellung eines ausgewählten Aspekts des Modells mithilfe einer bestimmten Notation. Die einzelnen Elemente in den Diagrammen beziehen sich aber auf ein gemeinsames Modellelement.

UML hat sich als Standard-Notation durchgesetzt und ist daher eine gute Grundlage für die moderne Software-Entwicklung. Allerdings ist der Sprachumfang von UML sehr groß, sodass eine geeignete Auswahl der typischerweise hilfreichen Teile wesentlich ist, um als verständliches Kommunikationsmittel für Entwickler und Domänenexperten einsetzbar zu sein. Die UML-Diagrammfamilie stellt umfangreiche Modellierungsmöglichkeiten im Rahmen der Software-Entwicklung dar. Die Abbildung 6.1 gibt einen kurzen Überblick über die aktuell verfügbaren Diagrammarten. Ausgewählte und passende Diagramme werden in den folgenden Kapiteln dieses Buches vorgestellt und anhand konkreter Anwendungsfälle exemplarisch angewandt.

Struktur und Verhalten

Die UML-Diagrammfamilie unterscheidet zwei grundlegende Arten von Diagrammen: (a) sechs *Strukturdiagramme* zur Beschreibung statischer Zusammenhänge und (b) sieben *Verhaltensdiagramme* zur Darstellung des zeitlichen Verhaltens (*Interaktionsdiagramme*) oder von konkreten Abläufen.

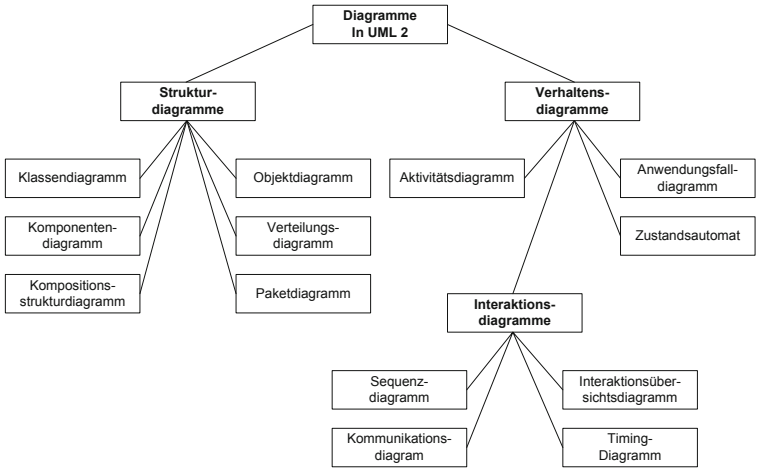


Abbildung 6.1
UML-2-Diagrammfamilie im Überblick [52].

Strukturdiagramme

Strukturdiagramme werden verwendet, um die statische Struktur des Systems zu beschreiben. Sie zeigen Aspekte wie Laufzeitkonfigurationen, physikalische Elemente des Systems und domänenspezifische Entitäten. Zu den Strukturdiagrammen *(structure diagrams)* gehören:

- *Klassendiagramme (class diagrams)* beschreiben die Entitäten eines Systems und welche Beziehungen sie untereinander eingehen können (Struktur der Daten). Neben Paketdiagrammen werden Klassendiagramme bzw. deren Notation wahrscheinlich am häufigsten eingesetzt.
- *Komponentendiagramme (component diagrams)* beschreiben die Subsysteme und Schnittstellen, aus denen das System besteht.
- *Kompositionsstrukturdiagramme (composite structure diagrams)* ermöglichen es, strukturierte Classifier (Klassen und Komponenten) durch ihre interne Struktur darzustellen.
- *Objektdiagramme (object diagrams)* zeigen eine konkrete Ausprägung des Systems zur Ausführungszeit.
- *Verteilungsdiagramme (deployment diagrams)* zeigen die Verteilung von Software-Komponenten und Artefakten im Netzwerk, beispielsweise Server und Client. Außerdem werden Kommunikationsmethoden, etwa HTTP oder RMI, auf den Verbindungen zwischen diesen Software-Komponenten dargestellt.
- *Paketdiagramme (package diagrams)* ermöglichen eine übersichtliche Gliederung des UML-Modells und seiner Artefakte.

Verhaltensdiagramme

Strukturdiagramme werden oft in Verbindung mit Verhaltensdiagrammen benutzt. So kann etwa einer Klasse eine Zustandsmaschine zugeordnet werden, um ihr Verhalten in Abhängigkeit von auftretenden Ereignissen zu beschreiben. Verhaltensdiagramme zeigen die dynamischen Aspekte eines Systems. Die Verhaltensdiagramme *(behavior diagram)* gliedern sich in:

- *Aktivitätsdiagramme (activity diagrams)* zeigen eine bestimmte Sicht auf die Dynamik des Systems (Abläufe). Sie beschreiben das konkrete Verhalten eines Systems. Aktivitätsdiagramme sind meist sehr domänenspezifisch und ohne weiteren Kontext schwer zu verstehen.
- *Anwendungsfalldiagramme (use-case diagrams)* stellen die externe Anwendersicht auf das System dar und beschreiben, welche funktionalen Anforderungen durch das System umgesetzt werden.
- *Zustandsautomaten (state diagrams)* beschreiben die Systemzustände bei definierten Ereignissen. Das bedeutet, ein Zustandsautomat bildet die verschiedenen Zustände ab, die ein Objekt während seiner Lebenszeit durchläuft.

Interaktionsdiagramme

Interaktionsdiagramme sind den Verhaltensdiagrammen zugeordnet und illustrieren primär das zeitliche Verhalten des Systems.

- *Sequenzdiagramme (sequence diagrams)* stellen eine exemplarische Abfolge von Nachrichten zwischen Objekten dar. Der Fokus liegt dabei auf der zeitlichen Ordnung der Nachrichten. Sequenzdiagramme sind meist sehr implementationsspezifisch, das heißt, sie sind ohne näheres Wissen über den Sourcecode des Systems nur schwer nachvollziehbar.
- *Kommunikationsdiagramme (communication diagrams)* zeigen die unterschiedlichen Teile einer (komplexen) Struktur und ihre Zusammenarbeit zur Erfüllung definierter Funktionalitäten. Die Funktionsweise ist mit den Sequenzdiagrammen vergleichbar. Der wesentliche Unterschied liegt in der Darstellung, wie die Objekte verbunden sind und welche Nachrichten sie über diese Verbindungen in einem spezifischen Szenario austauschen.
- *Interaktionsübersichtsdiagramme (interaction overview diagrams)* stellen sowohl die Reihenfolge als auch die Bedingungen für definierte Interaktionen dar (werden hier allerdings nicht behandelt).
- *Timing-Diagramme (timing diagrams)* beschreiben das zeitliche Verhalten der Zustände der unterschiedlichen Interaktionspartner (werden hier nicht behandelt).

In diesem Buch werden die Diagrammtypen – sofern sie benötigt werden – im jeweiligen Abschnitt beschrieben. Weitere detaillierte Informationen zur Anwendung von UML in der Software-Entwicklung sind in den Büchern *The UML User Guide* [78] sowie *Applying UML and Patterns* [65] zu finden. Nähere Informationen zu UML-Notationen und -Diagrammen sind in den Büchern *The UML Reference Manual* [77] und *UML Distilled* [30] zu finden.

6.1.1 UML-Paketdiagramm

Systemüberblick

Paketdiagramme werden eingesetzt, um den Überblick über das gesamte Modell zu behalten und das Modell auf einem hohen Abstraktionsniveau zu strukturieren. Paketdiagramme zeigen Pakete und ihre Abhängigkeiten. Pakete können dabei alle anderen UML-Elemente wie Klassen, Anwendungsfälle und Pakete gruppieren und helfen dabei, ein Modell zu strukturieren.

Notation

Pakete werden grundsätzlich als einfaches Rechteck gezeichnet und müssen mit einem eindeutigen Namen gekennzeichnet werden. Dieser Name steht typischerweise in die Mitte des Rechtecks. Optional können auch die Elemente eines Pakets gezeigt werden. In diesem Fall wird der Name in

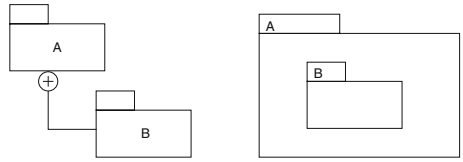


Abbildung 6.2 *Das linke Diagramm zeigt dieselbe Information wie das rechte. Beide Notationen sind UML-konform.*

dem Reiter links oben gezeigt. Alternativ kann die Unterpaketsbeziehung auch durch eine Linie zwischen den Paketen, die am Ende des beinhaltenden Pakets mit einem Kreis, in den ein +-Symbol eingeschrieben ist, dargestellt werden (siehe Abbildung 6.2).

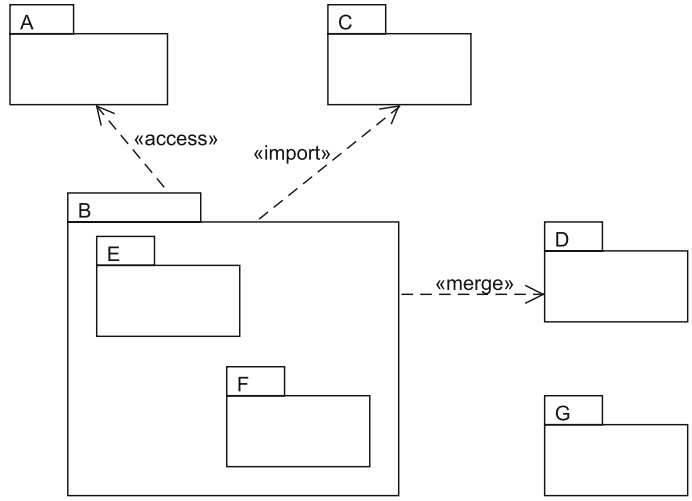


Abbildung 6.3
UML-Paketdiagramm.

Abhängigkeiten

Pakete können untereinander in Abhängigkeit stehen. Die Abbildung 6.3 illustriert ein Beispiel für Paketabhängigkeiten. Diese Abhängigkeiten werden durch einen gestrichelten Pfeil mit offener Pfeilspitze dargestellt. Grundsätzlich kann man die Abhängigkeiten zwischen Paketen in drei Kategorien aufteilen, (a) *access*, (b) *import* und (c) *merge*.

Access

Die Access-Abhängigkeit bedeutet, dass Elemente eines Pakets Elemente eines anderen Pakets nutzen. In Abbildung 6.3 werden definierte Elemente, etwa Klassen, im Paket A vom Paket B verwendet. Das bedeutet, dass jene verwendeten Klassen aus dem Paket A im Paket B mit vollständig qualifizierten (Paket-)Namen angesprochen werden müssen.

Import

Die Import-Abhängigkeit bedeutet, dass die Elemente des importierten Pakets im importierenden Paket ohne vollständig qualifizierten Namen bezeichnet werden können. In Abbildung 6.3 werden also alle Klassen vom Paket C in das Paket B importiert.

Merge

Die Merge-Abhängigkeit definiert eine implizite Generalisierungsbeziehung zwischen den beiden Paketen. Das bedeutet, dass Elemente mit demselben Namen im verschmelzenden Paket zu einem Element zusammengefasst werden. Dieses neue Element trägt denselben Namen und beinhaltet alle Eigenschaften der zu verschmelzenden Elemente. In Abbildung 6.3 „verschmelzen“ alle Elemente in den Paketen B und D, die denselben Namen tragen. Da sich alle Elemente eines Pakets einen Namensraum teilen, müssen sie eindeutige Namen besitzen.

6.1.2 UML-Anwendungsfälle

Anwendungsszenarien

Gemeinsamer Anfangspunkt für die Erstellung eines Software-Produkts ist die Erfassung aller Anforderungen, die im Produkt umgesetzt werden sollen. Wichtig dabei ist, dass bei der Anforderungserhebung alle relevanten Zielgruppen *(Stakeholder)* berücksichtigt werden müssen. Ausgehend von den Zielgruppen werden die Rollen als sogenannte Akteure definiert; die konkreten Anforderungen werden durch Anwendungsszenarien erfasst. Diese Anwendungsszenarien müssen so konkret sein, dass der zukünftige Benutzer daraus ableiten kann, welche Funktionalitäten das Endprodukt aufweist und dennoch so abstrakt definiert sein, dass der Lösungsspielraum nicht frühzeitig eingeschränkt wird.

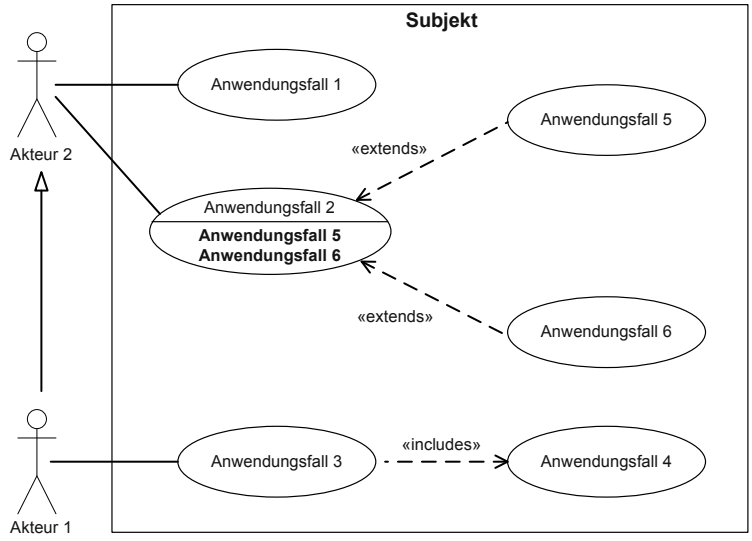


Abbildung 6.4
UML-Anwendungsfalldiagramm.

Diagramm und Beschreibung

Konkrete Anwendungsszenarien können mittels UML-Anwendungsfallbeschreibungen dokumentiert werden. Dabei handelt es sich um eine detaillierte textuelle Beschreibung des Anwendungsfalls. UML-Anwendungsfalldiagramme stellen einen Überblick der Funktionalitäten des Endprodukts dar. Sie beinhalten „nur“ die Elemente Subjekt, Anwendungsfälle,

Akteure und Abhängigkeiten *(Generalisierungen, Assoziationen)* und sind generell viel einfacher zu verstehen als Anwendungsfallbeschreibungen. Durch diese einfache und verständliche Darstellungsmöglichkeit ist auch eine offene und zielgerichtete Kommunikation mit dem Kunden möglich.

Notation

Jeder Anwendungsfall wird als Oval dargestellt und besitzt einen Namen, der ihn von anderen Anwendungsfällen unterscheidet. Der Name eines Anwendungsfalls sollte ein starkes Verb enthalten.

Subjekt

Das Subjekt ist ein System oder ein Subsystem (siehe Abbildung 6.4), das durch eine Menge von Anwendungsfällen beschrieben wird. Die Anwendungsfälle repräsentieren das Verhalten des Systems aus Nutzersicht.

Akteure

Akteure beschreiben die mit dem Subjekt interagierenden Rollen. Akteure können mit Anwendungsfällen nur durch Assoziationen verbunden werden. Eine Assoziation bedeutet dabei, dass der Akteur mit dem Anwendungsfall kommuniziert. Akteure können durch Generalisierungsbeziehungen spezialisiert werden. Sie „erben" dadurch alle Assoziationen des generelleren Akteurs.

Abhängigkeiten

Anwendungsfälle können untereinander mittels *include* bzw. *extend* Beziehungen miteinander verbunden werden. Diese Beziehungen sind Abhängigkeiten und als solche gestrichelt und gerichtet. Include bedeutet, dass bei der Ausführung des inkludierenden Anwendungsfalls der inkludierte Anwendungsfall auch ausgeführt werden *muss*. Extend hingegen bedeutet, dass der erweiternde Anwendungsfall während der Ausführung des erweiterten Anwendungsfalls ausgeführt werden *kann*. Anwendungsfälle können wie Akteure mittels Generalisierungen spezialisiert werden. Der spezialisierte Anwendungsfall erbt alle Assoziationen und Abhängigkeiten.

Extension Points

Anwendungsfälle können Extension Points besitzen. Diese geben an, dass vorgesehen ist, dass der Anwendungsfall mittels Extend-Beziehungen erweitert wird. Grafisch werden Extension Points unterhalb des Namens des Anwendungsfalls angegeben. Die Extension Points werden durch eine horizontale Linie vom Namen getrennt. Extension Points sollten in der Beschreibung des Szenarios erwähnt werden.

Eine häufige Fehleinschätzung ist, dass Beziehungen wie *include* und *extend* zeitliche Abhängigkeiten modellieren. Es gibt keine Möglichkeit, in einem Anwendungsfalldiagramm zeitliche Abhängigkeiten zu modellieren, diese Informationen müssen in den Beschreibungen der Anwendungsfälle definiert werden (siehe Anwendungsfallbeschreibung).

Schrittweise Verfeinerung

Anwendungsfalldiagramme sowie Aktivitätsdiagramme (siehe Abschnitt 6.1.5) werden in der Regel top-down verfeinert. Jede Phase im Entwicklungsprozess erfordert eine andere Abstraktionsebene in den Modellen. Für Test-Driven Development können System- und Integrationstests etwa auch anhand von dieser Top-down-Modelle schrittweise erstellt werden.

Abstraktionsebene

Anwendungsfälle werden in jedem Projekt in verschiedenen Abstraktionsebenen benötigt: In der Analysephase werden in Zusammenarbeit mit dem

Kunden sogenannte *Enterprise Goals* und *User Goals* erstellt, um die Anforderungen festzuhalten. Da die Enterprise-Goal- und User-Goal-Anwendungsfälle die Gesamtfunktionalität des zu entwickelnden Software-Systems auf unterschiedlichen Ebenen beschreiben, müssen diese auch durch alle Teammitglieder verstanden und in den weiteren Phasen des Projekts zu weniger umfangreicheren Anwendungsfällen verfeinert werden.

Enterprise Goals

Ziele der übergeordneten Organisation, die das Software-Produkt betreibt oder betreiben wird, werden *Enterprise Goals* genannt. Die Enterprise Goals für ein Software-Produkt zur Verwaltung einer Universitätsbibliothek könnten beispielsweise den „Datenexport zu bestehenden Verwaltungssystemen“ oder die „Überprüfung der Studienberechtigung“ umfassen. Auf einem höheren Detailgrad beschreiben Aktionen, die ein Benutzer tatsächlich mit der Software ausführen will, die so genannten *User Goals*. User Goals werden fast ausschließlich von der Domäne abstrahiert und sind deshalb wahrscheinlich am schwersten zu erarbeiten. Bei einer Universitätsbibliothek könnte ein User Goal der Studenten „Buch ausleihen“ sein. Anwendungsfälle, die vom Akteur „System“ bzw. „Administrator“ durchgeführt werden, sind unter dem Namen *System Goal* bekannt. Dazu zählen etwa geplante Wartungsabläufe, wie etwa „Index für Schnell-Suche erstellen“ oder Datenbankoperationen auf unterster Ebene, wie etwa Create, Read, Update, und Delete (CRUD).

User Goals

System Goals

Tabelle 6.1 *Struktur einer UML-Anwendungsfallbeschreibung.*

Bezeichnung des Anwendungsfalls	
Name	Titel des Anwendungsfalls
Beschreibung	Kurzbeschreibung des Anwendungsfalls
Priorität	Priorität des Anwendungsfalls
Akteure	Welche Akteure sind beteiligt?
Datum	Erstellungsdaten des Anwendungsfalls (Ersteller, Datum, Version)
Standardabläufe (Flow of Events)	
Hauptszenario	Was soll abgebildet werden?
Alternativszenario	Gibt es Alternativabläufe, wo sind die Abweichungen?
Fehlersituationen	Was passiert im Fehlerfall dem Systemzustand?
Vorbedingung	Voraussetzung für erfolgreiche Ausführung
Nachbedingung	Systemzustand nach erfolgreicher Ausführung
nicht funktionale Anforderungen	Bemerkungen, Angaben über Häufigkeit

Während Diagramme nur eine einfache Übersicht anbieten, etwa um mit einem Kunden über Anwendungsszenarien zu diskutieren, liefern Anwendungsfallbeschreibungen eine ausführliche Dokumentation dieser Szenarien und sind notwendig, um die Anforderungen eines Produkts vollständig zu dokumentieren. Es ist empfehlenswert, die Anwendungsfallbeschreibung strukturiert zu erarbeiten. Tabelle 6.1 zeigt eine mögliche Struktur für die Dokumentation von Anwendungsfällen. Einem Anwendungsfall sollte in der Beschreibung eine Priorität zugewiesen werden. Diese gibt an, wie wichtig der Anwendungsfall für das System bzw. für den Kunden ist (z. B.

1 = hoch, 2 = mittel, 3 = niedrig). So lassen sich die Hauptanwendungsfälle herauskristallisieren. Bei der Implementierung sollte später nach der Priorität entwickelt werden können, um bei Problemen oder Zeitknappheit die wichtigen Anwendungsfälle zuerst umzusetzen.

Alternativ kann anstatt einer textuellen Beschreibung des Szenarios ein Aktivitätsdiagramm (siehe Abschnitt 6.1.5) verwendet werden.

Vom Problem- zum Lösungsraum

Die Anwendungsszenarien beschreiben also den „Problemraum“ und erlauben somit, Produktvorschläge der Entwickler zu evaluieren und zu vergleichen. Die technische Architektur und die verwendeten Klassen (siehe Abschnitt 6.1.3) bzw. Komponenten (siehe Abschnitt 6.1.4) hingegen beschreiben den „Lösungsraum“, in dem eine Menge potentieller Produkte zu finden sind, die unterschiedliche Eigenschaften haben. Ziel der Software-Entwicklung ist daher, zuerst den relevanten Problemraum darzustellen und dann kreativ und effizient im Lösungsraum bestmögliche Produktalternativen zu finden.

6.1.3 UML-Klassendiagramme

Unterschiedliche Abstraktionsstufen und Sichtweisen

UML-Klassendiagramme erfüllen den Zweck, die Daten und das Verhalten des zu erstellenden Systems statisch zu strukturieren. Die Notation von UML-Klassendiagrammen wird verwendet, um verschiedenartige Informationen in unterschiedlichen Abstraktionsstufen und Sichtweisen zu beschreiben. Dies kann oft in der Entwicklung zu Verwirrung führen. Wenn die Diagramme nicht gut beschriftet sind, ist oft unklar, wie ein bestimmtes Diagramm zu interpretieren ist. In diesem Abschnitt werden zunächst die wesentlichen Grundlagen der Notationen für Klassendiagramme beschrieben, anschließend wird eine Übersicht der Verwendung unter verschiedenen Sichtweisen und Abstraktionsstufen gezeigt.

Notation

Die grafische Repräsentation einer Klasse ist ein Rechteck, das mehrere Bereiche aufweisen kann. Der oberste Bereich enthält den Klassennamen, zugewiesene Stereotypen und Properties. Der zweite Bereich enthält die Attribute der Klasse. Attribute repräsentieren die Daten, die eine Klasse bzw. deren Instanzen halten kann. Multiplizitäten drücken aus, wie viele Werte ein Objekt für das jeweilige Attribut einnehmen kann (siehe dazu auch Tabelle 6.2). Attributspezifikationen haben die folgende generelle Form:

Attribute

```
sichtbarkeit attributName : Typ [ multiplizität ]
    = defaultWert { eigenschaften }
```

Konkretes Beispiel:

```
-foo : int = 0 { not null }
```

Methoden

In einem dritten Bereich werden die Methoden eingetragen. Methodenspezifikationen haben die folgende generelle Form:

```
sichtbarkeit methodenName ( parameterName : Typ )
    : rückgabeTyp { eigenschaften }
```

Konkretes Beispiel:

```
+setFoo( foo : int ) : void { reentrant }
```

Multiplizitäten

Multiplizitäten definieren, auf viele Werte sich ein Objekt beziehen kann. Standardmäßig hat etwa eine Instanz der Klasse genau einen Wert für ein Attribut. Sind mehrere Werte möglich, können die jeweiligen Bereichsgrenzen bzw. Schranken angegeben werden (beispielsweise 2 bis 6). Ist die genaue Anzahl nicht festgelegt oder können Attribute beliebig viele Werte annehmen wird das durch einen „*“ dargestellt. Beispielsweise erfordert eine Multiplizität „1..*“ zumindest einen Wert (untere Schranke), kann aber beliebig viele weitere Werte („*“ als obere Schranke) ermöglichen. Tabelle 6.2 gibt einen Überblick über gebräuchliche Multiziplitäten.

Tabelle 6.2 *Multiplizitäten in UML.*

Multiplizität	Erklärung
1	Genau ein Objekt
*	Beliebig viele Objekte (0 oder mehr)
0..*	0 oder mehrere Objekte
1..*	1 oder mehrere Objekte
0..1	kein oder 1 Objekt
2..6	2 bis 6 Objekte

Sichtbarkeiten

Für Attribute und Methoden können Sichtbarkeiten festgelegt werden. Dadurch wird festgelegt, in welchem Kontext das Attribut oder die Methode anwendbar ist. Die von der UML definierten Sichtbarkeiten sind in Tabelle 6.3 dargestellt.

Tabelle 6.3 *Sichtbarkeiten in UML.*

Sichtbarkeit	Kürzel	Beschreibung
Public	+	Sichtbar für jedes Element das die Klasse sieht
Protected	#	Sichtbar für Elemente der Klasse und ihrer Subklassen
Private	-	Sichtbar für Elemente der Klasse
Package	~	Sichtbar für Elemente im selben Paket wie die Klasse

Schnittstelle, Interface

Schnittstellen spielen im Rahmen der komponentenorientierten Software-Entwicklung eine zentrale Rolle. Eine Schnittstelle *(Interface)* definiert dabei eine Liste von Attributen und Methoden mit öffentlicher Sichtbarkeit. Eine Schnittstelle wird grafisch wie eine Klasse repräsentiert und mit dem Stereotyp *interface* markiert. Die Abbildung 6.5 illustriert die Schnittstellendefinition anhand eines Beispiel.

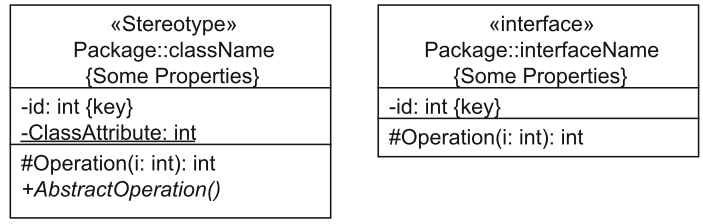


Abbildung 6.5
UML-Klassendiagramm.

Beim Entwurf einer Schnittstelle wird meist davon ausgegangen, dass die eigentliche Implementierung noch unbekannt ist. Schnittstellen erlauben aber die Austauschbarkeit mehrerer unabhängiger Implementierungen, falls sie die Schnittstelle realisieren. In der komponentenorientierten Software-Entwicklung erlangen die Schnittstellen eine besondere Bedeutung. Details und konkrete Anwendungen werden im Kapitel 8 beschrieben.

Angebotene Schnittstelle

Man unterscheidet zwischen angebotenen und benötigten Schnittstellen. Eine angebotene Schnittstelle wird von einem Classifier (beispielsweise einer Klasse) realisiert. Der Classifier sichert damit zu, dass er alle Operationen der Schnittstelle und alle Attribute auf geeignete Weise zu Verfügung stellt. Eine Realisierung wird durch einen gestrichelten Pfeil mit leerer Pfeilspitze dargestellt und zeigt vom realisierenden Classifier zur Schnittstelle.

Benötigte Schnittstellen

Benötigte Schnittstellen sind jene, die ein Classifier für seine Ausführung einer Realisierung der Schnittstelle benötigt. Dargestellt wird diese Abhängigkeit durch den Stereotyp *use* und einem gestrichelten Pfeil, der vom Classifier zum Interface zeigt. Abbildung 6.6 zeigt eine Übersicht der Notation für Abhängigkeiten und Generalisierungen.

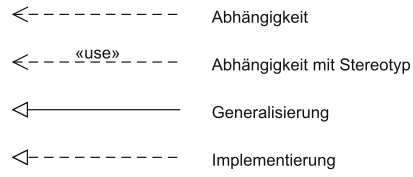


Abbildung 6.6
UML-Abhängigkeiten und Generalisierung.

Assoziationen

Assoziationen stellen Verbindungen zwischen Klassen oder deren Instanzen her. Übliche Assoziationen sind in Abbildung 6.7 dargestellt. Sie werden mittels einer Linie zwischen zwei Classifiern (beispielsweise zwischen Klassen oder Akteuren) dargestellt. Der Name einer Assoziation sollte die Beziehung bezeichnen, in der die beiden verbundenen Classifier zueinander stehen.

Leserichtung und Navigation

Die Leserichtung von Assoziationen kann mit einem Pfeil neben dem Namen angezeigt werden. Die Leserichtung hat nichts mit der Navigierbar-

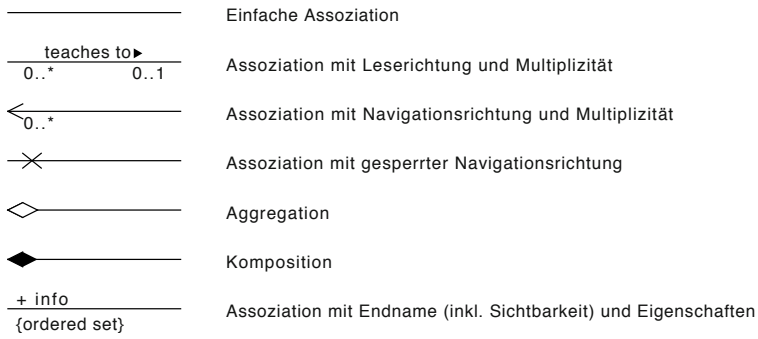


Abbildung 6.7
UML-Assoziationen.

keit zu tun, sie zeigt nur an, wie der Name zu lesen ist. Für die Darstellung der Navigierbarkeit werden Pfeilspitzen an den Assoziationsenden benutzt. Sind keine Pfeilspitzen angebracht, handelt es sich um eine bidirektionale Assoziation, andernfalls ist die Leserichtung definiert. Assoziationsenden können Namen besitzen. Die Namen können beispielsweise das Attribut bezeichnen, durch welches die Instanz an diesem Ende der Assoziation identifiziert werden kann.

Assoziationsklassen

Assoziationen sind selbst Classifier, d. h., sie können wie Klassen oder Anwendungsfälle Attribute und Methoden besitzen (auch wenn darauf meistens verzichtet wird). Diese Attribute können benutzt werden, um einen Link (Instanz einer Assoziation) zu identifizieren. Sinnvoll ist dies z. B. für Tischreservierungen in einem Restaurant, wobei zwischen zwei Objekten (Tisch, Gast) mehrere Verbindungen bestehen können, diese können dann anhand der reservierten Zeit unterschieden werden. Die Attribute und Methoden einer Assoziation werden in einer Klasse dargestellt, die den Namen der Assoziation trägt. Diese Klassen können mit der Assoziation durch eine dünne gestrichelte Linie verbunden sein.

Komposition und Aggregation

Neben der herkömmlichen Assoziation gibt es noch zwei stärkere Arten der Assoziation, nämlich die *Komposition* und die *Aggregation*. Beide bezeichnen eine *Teil-Ganzes-Beziehung*, wobei die Komposition stärker ist und ein Teil immer nur mir einem Ganzen verbunden sein darf. Weiterhin ist das Ganze einer Komposition auch für die Lebensdauer seiner Teile verantwortlich, d. h., sollte das Ganze gelöscht werden, werden auch seine Teile gelöscht. Diese Einschränkungen gelten nicht für die Aggregation, d. h., ein Teil kann von mehreren Ganzen aggregiert werden und auch nach dem Löschen des Ganzen weiterexistieren. Grafisch wird die Komposition mit einer ausgefüllten Raute, die Aggregation mit einer leeren Raute auf der Seite des Ganzen gekennzeichnet.

Sichtweise

Klassendiagramme werden, je nach Entwicklungsphase, aus einer anderen Perspektive bzw. Sichtweise für einen definierten Zweck erstellt. Die verschiedenen Sichtweisen sind ohne ausreichende Dokumentation und Definition der Perspektive nur schwer zu unterscheiden. Die eindeutige Zuordnung zur Sichtweise ist aber für eine korrekte Interpretation des Klassendiagramms unbedingt notwendig.

UML-Klassendiagramme können aus einer *konzeptionellen*, *spezifizierenden* oder *implementierenden* Sichtweise gezeichnet werden [30]. Diese Sichtweisen sind zwar nicht Teil des UML-Standards, jedoch beim Modellieren sehr nützlich, da sie erlauben, die Klassendiagramm-Notation im gesamten Entwicklungsprozess für verschiedene Zwecke zu verwenden, statt nur um die implementierten Klassen zu dokumentieren.

Konzeptionelle Sichtweise

Bei der konzeptionellen Sichtweise repräsentieren die Klassen Konzepte aus der zu untersuchenden Domäne. Diese Klassen werden deshalb zwar mit den Implementationsklassen in Beziehung stehen, jedoch ist ein direktes Mapping nicht möglich. Stattdessen ist die Verwendung von Stereotypen möglich, um komplexere Zusammenhänge zu modellieren. Beispielsweise können alle Klassen, die in weiterer Folge persistiert werden sollen bzw. Entitäten des Software-Systems darstellen, mit dem Stereotyp *Entity* gekennzeichnet werde. Es ist üblich den Namen „Domänenmodell“ beim Bezeichnen von Klassendiagrammen aus der konzeptionellen Sichtweise zu verwenden, um sie von „Klassendiagrammen“ zu unterscheiden, die üblicherweise Klassen aus der Implementierungssicht beschreiben.

Domänenmodell

Umfang des Problemraums

Domänenmodelle werden normalerweise – implementierungsunabhängig – in der Analysephase erstellt. Daher sind Domänenmodelle generell unabhängig von eingesetzten Programmiersprachen und Frameworks. Sie dienen primär dem Zweck, den Umfang des Problemraums klar zu definieren, indem konkrete Objekte und Konzepte der realen Welt beschrieben werden. Die Darstellung in der konzeptionellen Sichtweise der UML-Klassendiagramm-Notation soll der Identifizierung und dem Verständnis aller Objekte der Anwendungsdomäne dienen:

Objekte der Anwendungsdomäne

- Definition von Zuständen eines Prozesses (z. B. Buchung oder Reparatur).
- Beschreibung wichtiger Sachgegenstände für den Prozess (z. B. Vertrag oder Rechnung).
- Objekte des Alltags der Anwendungsdomäne (z. B. Auto bei Autohändlern, Buch bei Büchereien).
- Beschreibung von Infrastrukturen (z. B. Abteilungshierarchie).

Beispiel: Würfelspiel

Abbildung 6.8 zeigt das Domänenmodell eines kleinen Computerspiels: Ein Spieler würfelt mit zwei Würfeln. Wenn die Summe der Würfelaugen genau sieben ergibt, so hat der Spieler gewonnen, andernfalls verloren. Es ist zu beachten, dass die darin enthaltenen Elemente keine Implementation widerspiegeln, sondern an Funktionen des Systems beteiligte oder davon betroffene Objekte und Personen bzw. deren Rolle darstellen. In der spezifizierenden bzw. implementierenden Sichtweise würde die Klasse „Spieler“ nicht eingezeichnet werden, da diese Klasse nicht implementiert wird bzw. „Würfelspiel“ und „Würfel“ werden höchstwahrscheinlich auch anderes implementiert als in diesem Domänenmodell dargestellt. Die Namen

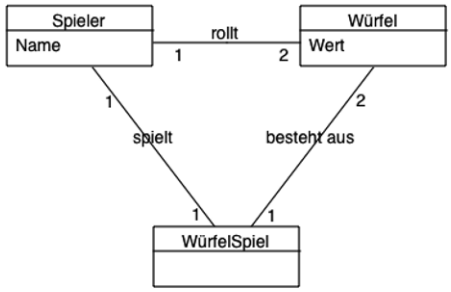


Abbildung 6.8
Domänenmodell des Würfelspiels.

der Objekte und ihre Attribute werden angegeben, jedoch werden die Methoden frei gelassen, da diese in der konzeptionellen Sichtweise nicht relevant bzw. nicht sinnvoll sind.

Spezifizierende Sichtweise

Die spezifizierende Sichtweise als nächste Verfeinerungsstufe von Klassendiagrammen wird durch das Komponentendiagramm abgedeckt. Sie wird in Abschnitt 6.1.4 beschrieben.

Implementierende Sichtweise

Die wahrscheinlich am meisten verwendete Perspektive ist die implementierende Sichtweise. Typischerweise wird ein Diagramm in dieser Sichtweise einfach „Klassendiagramm“ genannt, obwohl diese Notation in allen Sichtweisen eingesetzt wird. Klassen auf dieser Ebene beschreiben tatsächliche Implementationsklassen und es können aus dem UML-Modell direkt Stub-Klassen generiert werden. Eine systematische Generierung von diesen Stub-Klassen wird in der Software-Entwicklung als „Model-Driven-Architecture“ (MDA) bezeichnet. Klassendiagramme der implementierenden Sichtweise werden meist gegen Ende eines Projekts von den technischen Mitarbeitern aus dem Sourcecode als Dokumentation abgeleitet. Diese Vorgehensweise wird in den meisten UML-Werkzeugen als „Sourcecode reverse engineering“ bezeichnet.

Das Verständnis der unterschiedlichen Sichtweisen ist bei der Erstellung von UML-Klassendiagrammen unbedingt notwendig. Da diese Sichten nicht optimal voneinander abgegrenzt werden können und die Entwickler bei der Modellierung nur wenig Rücksicht drauf nehmen, führt das in der Regel zu unübersichtlichen und verwirrenden Klassendiagrammen. Werden die Sichtweisen aber entsprechend beachtet, ist es möglich UML-Klassendiagramme, ausgehend von der Spezifikation (Domänenmodell) einzusetzen, um die Architektur der Software zu planen (Komponentendiagramm) und schließlich ihre Implementationsklassen (Klassendiagramm) zu dokumentieren.

6.1.4 UML-Komponentendiagramm

Spezifizierende Sichtweise

Durch Komponentendiagramme können die Strukturen des Systems zur Laufzeit dargestellt werden. Wie bereits in Abschnitt 6.1.3 beschrieben,

werden Komponentendiagramme der spezifizierenden Sichtweise zugeordnet. In der spezifizierenden Sichtweise werden logische und statisch austauschbare Teile eines Systems auf Komponentenebene betrachtet. Komponenten enthalten prinzipiell Klassen bzw. Klassendiagramme aus der implementierenden Sichtweise, die in Abschnitt 6.1.3 dargestellt wurde. Beim Komponentendiagramm werden konkrete Implementierungen der Komponenten vernachlässigt und das Black-Box-Verhalten in den Vordergrund gerückt. Der Fokus liegt daher auf den Schnittstellen der Komponenten und ihrem externen Verhalten. Das konkrete Verhalten einer Komponente kann mittels Zustandsdiagramm (siehe Abschnitt 6.1.8) oder Aktivitätsdiagramm (siehe Abschnitt 6.1.5) beschrieben werden.

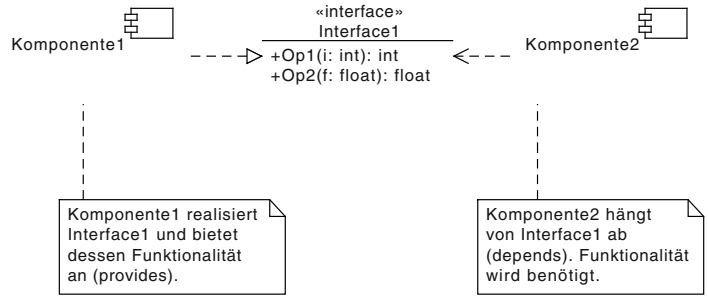


Abbildung 6.9
Komponentendiagramm: Interface-Abhängigkeit zwischen Komponenten.

Notation

In der grafischen Notation werden Komponenten als Rechteck dargestellt, in dessen obere rechten Ecke ein Komponentensymbol eingezeichnet ist. Der Name der Komponente wird in die Mitte des Rechtecks geschrieben (ein Beispiel ist in Abbildung 6.9 dargestellt). Jede Komponente beschreibt die angebotenen Schnittstellen und das zur Ausführung benötigte Verhalten. In Abbildung 6.9 wird „Interface1“ durch eine Realisierungsbeziehung mit Komponente1 verbunden, da diese Schnittstelle von der Komponente „angeboten“ wird. Das bedeutet, dass eine mögliche Implementation dieser Komponente auch eine Implementation von „Interface1“ beinhalten muss. Das Interface ist auch mittels einer Abhängigkeitsbeziehung mit Komponente2 verbunden, da jene im Interface enthaltenen Operationen von der Komponente benötigt (required) werden, um ihre Funktionen auszuführen. Komponenten können im Wesentlichen all jene Beziehungen eingehen, die auch Klassen eingehen können. Die am häufigsten gebrauchten sind allerdings Realisierungsbeziehungen und Abhängigkeiten zwischen einer Schnittstelle und einer Komponente sowie Assoziationen zwischen Komponenten. Weiterhin können Komponenten wie alle anderen UML-Elemente zu Paketen zusammengefasst werden.

Lollipop-Notation

Schnittstellen können neben der bereits beschriebenen Methode auch in kompakter Form als sogenannte „Lollipop-Notation“ dargestellt werden. Dabei wird eine angebotene Schnittstelle als Kreis und eine benötigte Schnittstelle als Halbkreis dargestellt. Um den Zusammenhang zu zeigen,

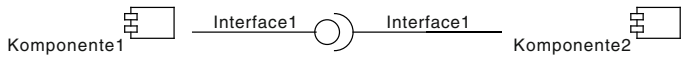


Abbildung 6.10
Interface-Abhängigkeit zwischen Komponenten (Lollipop-Notation).

wird eine Abhängigkeitsbeziehung von der benötigten zur angebotenen Schnittstelle gezeichnet. Diese kann weggelassen werden, wenn der Kreis der angebotenen Schnittstelle in den Halbkreis der benötigte Schnittstelle gezeichnet wird (siehe Abbildung 6.10).

Ersetzbarkeit

Eine wichtige Eigenschaft von Komponenten ist die Ersetzbarkeit der Realisierung, wie sie beispielsweise bei Entwurfsmustern benötigt wird (siehe Kapitel 8). Ersetzbarkeit ist durch das Austauschen einer Implementation durch eine andere mit gleichen Schnittstellen gewährleistet. Dadurch ist es möglich, bereits bestehende Komponente um zusätzliche Funktionalität zu erweitern, ohne dass andere Komponenten davon betroffen sind. Die Implementierung einer Komponente kann durch eine andere Implementierung ausgetauscht werden, wenn die beiden Komponenten dieselben Schnittstellen zur Verfügung stellen und alle von der neuen Komponente benötigten Schnittstellen im System vorhanden sind. Weiterhin kann man neue Komponenten erweitern, ohne das ganze System neu entwerfen zu müssen.

Komponentendiagramme werden in der Entwurfsphase von den technischen Architekten angefertigt, um das System in seine groben Bestandteile aufzugliedern. Das Projektmanagement verwendet Komponentendiagramme, um zu ermitteln, welche Teile des Systems zugekauft werden können. Jene Komponenten, die nicht zugekauft werden können, müssen natürlich selbst programmiert werden. Dazu leiten die Entwickler die benötigten Schnittstellendefinitionen aus dem Komponentendiagramm ab.

6.1.5 UML-Aktivitätsdiagramme

Modellierung des Verhaltens

Mit Aktivitätsdiagrammen lässt sich das Verhalten eines Systems, etwa die Abarbeitung eines Anwendungsfalls, Szenarios oder einer Operation visualisieren. Aktivitätsdiagramme zeigen grundsätzlich den Fluss von einer Aktivität zur nächsten. Die Reihenfolge der Aktivitäten wird durch gerichtete Pfeile mit offenen Spitzen bestimmt. Aktivitäten stellen Gruppierungen von Aktionen und verschachtelten Aktivitäten dar und haben ein sichtbare innere Struktur. Man kann sich eine Aktivität als eine „Zusammenfassung“ vorstellen, deren Kontrollfluss aus Aktionen und anderen Aktivitäten besteht. Zoomt man in eine Aktivität hinein, findet man ein anderes Aktivitätsdiagramm, das einen detaillierteren Kontrollfluss der jeweiligen Aktivität beschreibt. Eine Aktivität ist also eine andauernde *nicht atomare* Ausführung. Die Ausführung einer Aktivität führt zur Ausführung von anderen Aktivitäten oder individuellen Aktionen. Aktionen werden im Ge-

Aktivität und Aktion

gensatz zu Aktivitäten als *atomar* betrachtet. Eine Aktion kann den Zustand des Systems ändern oder Nachrichten verschicken.

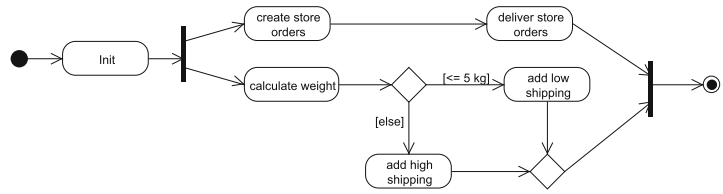


Abbildung 6.11
Aktivitätsdiagramm.

Notation

Aktivitätsdiagramme enthalten im Allgemeinen Aktivitäten, Aktionen, Flüsse und Objekte. Wie alle Diagramme können auch Aktivitätsdiagramme Notizen und Einschränkungen enthalten. Aktionen werden, wie in Abbildung 6.11 illustriert, durch ein Rechteck mit abgerundeten Ecken dargestellt. Grafisch besteht kein Unterschied zwischen einer Aktivität und einer Aktion, das bedeutet, sie werden ebenfalls als Rechtecke mit abgerundeten Ecken dargestellt. Der Anfang eines Kontrollflusses – des gesamten Flusses, nicht eines einzigen Pfeils – wird durch einen ausgefüllten Kreis dargestellt, sein Ende bzw. seine Enden durch einen ausgefüllten Kreis in einem leerem Kreis. Eine Aktivität kann Vor- und Nachbedingungen aufweisen, was vor allem bei der Modellierung von Anwendungsfällen eine große Hilfe darstellt, aber auch beim Modellieren einzelner Methoden hilfreich sein kann. Weiterhin definiert jede Aktivität ihren eigenen Namensraum, was wichtig ist, weil zwei verschiedene Unteraktivitäten in zwei oder mehreren Klassen denselben Namen erhalten können. Sobald eine Aktion oder Aktivität ihre Ausführung beendet hat, wird der Kontrollfluss unmittelbar an die nächste Aktion oder Aktivität weitergereicht, wobei von einer Aktivität oder Aktion immer nur ein Kontrollfluss ausgeht.

Kontrollfluss

Entscheidungen

Um Entscheidungen auszudrücken, werden *Entscheidungsknoten* verwendet. Entscheidungsknoten weisen einen eingehenden und mehrere ausgehende Kontrollflüsse auf. Die Bedingungen, unter denen einem Kontrollfluss gefolgt wird, werden in eckigen Klammern an den Pfeil geschrieben. Ein Pfeil ohne Bedingung wird gewählt, falls keine der Bedingungen der anderen Pfeile erfüllt ist. Um mehrere alternative Flüsse wieder zusammenzuführen werden *Vereinigungsknoten* benutzt. Diese haben dieselbe Ausprägung wie Entscheidungsknoten, allerdings mehrere eingehende Flüsse und nur einen ausgehenden Fluss. Um nebenläufige Flüsse zu modellieren, bedient man sich der Fork- bzw. Join-Knoten. Diese werden durch einen schwarzen Balken dargestellt, wie sie in Abbildung 6.11 zu finden sind. Fork-Knoten besitzen einen eingehenden Kontrollfluss und mehrere ausgehende Kontrollflüsse. Join-Knoten dementsprechend mehrere eingehende Kontrollflüsse und einen ausgehenden Kontrollfluss.

Nebenläufige Flüsse

Schwimmbahnen

Um die Verteilung von Aktivitäten auf ausführende Entitäten (Klassen, Komponenten und Akteure) anzuzeigen, kann man sogenannte Schwimmbahnen benutzen. Diese verlaufen horizontal oder vertikal über das gesam-

te Diagramm. Am linken bzw. oberen Rand wird der Name bzw. der Typ der ausführenden Entität eingezeichnet. Schwimmbahnen dienen also der Strukturierung komplexer Aktivitätsdiagramme.

6.1.6 UML-Sequenzdiagramme und Kommunikationsdiagramme

Daten- und Informationsaustausch

Sequenzdiagramme helfen dabei, den Informationsaustausch zwischen beliebigen Kommunikationspartnern innerhalb des Systems zu beschreiben. Kommunikationsdiagramme zeigen, welche Teile eines Systems wie zusammenarbeiten, um eine bestimmte Funktion zu erfüllen. Mit Sequenzdiagrammen werden normalerweise die Logik von Services und von Methoden sowie von Anwendungsszenarien modelliert. Sie bilden die Interaktion zwischen mehreren Classifiern (Klassen, Anwendungsfällen oder Akteuren) oder Instanzen ab. Die Classifier und Instanzen werden in der für sie üblichen UML-Notation dargestellt. Abbildung 6.12 zeigt ein einfaches Beispiel unterschiedlicher Nachrichten in der Notation eines Sequenzdiagramms.

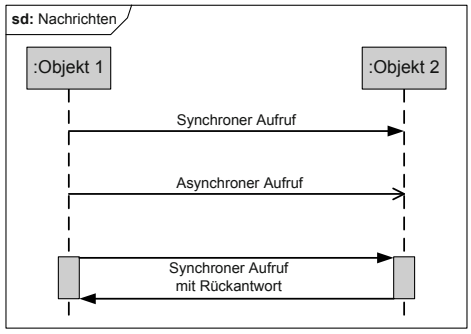


Abbildung 6.12
Sequenzdiagramm.

Notation

Die von den Classifiern und Instanzen nach unten hängenden gestrichelten Linien, *Lebenslinien* genannt, repräsentieren die Lebensdauer eines Objekts während des gezeigten Szenarios. Die länglichen leeren Rechtecke auf den Lebenslinien (Aktivierungsbalken) zeigen das aktive Ausführen einer Nachricht an. Nachrichten werden durch Pfeile zwischen den Lebenslinien der Objekte dargestellt. Es gibt drei Arten von Nachrichten, (a) *asynchrone Nachrichten*, (b) *synchrone Nachrichten* und (c) *Antworten*.

Synchrone und asynchrone Nachricht

Bei der synchronen Nachricht wartet der Sender, bis er vom Empfänger eine Antwort enthält. Synchrone Nachrichten können den Sender blockieren, falls eine Antwort ausbleibt. Asynchrone Nachrichten werden abgeschickt, ohne auf eine Antwort zu warten; der Sender wird dabei nicht blockiert. Allerdings ist auch nicht sichergestellt, dass die Nachricht auch tatsächlich ankommt.

Fragmente

Sequenzdiagramme stellen auch Fragmente zur Verfügung, um komplexere Abläufe zu modellieren [52]. Ein Fragment besteht aus einer oder mehrerer Nachrichten, die in einem Rahmen eingeschlossen sind und unter bestimmten genannten Umständen ausgeführt werden. Dadurch ist eine klare Strukturierung möglich. Die wichtigsten Vertreter verfügbarer Fragmente sind:

- *alt* : Alternative Fragmente zur Modellierung von Konstrukten wie „if... then... else".
- *opt* : Option-Fragment, modelliert switch Konstrukte.
- *loop* : Loop-Fragment, für alle Arten von Schleifen.

alt

In einem Alternativ-Fragment wird der Körper des Fragments mit einer gestrichelten Linie geteilt. Der obere der beiden Teile enthält eine boolsche Bedingung in eckigen Klammern. Der unter Teil das Wort *else* in eckigen Klammern. Der obere Teil wird ausgeführt, falls die boolsche Bedingung zu wahr evaluiert, sonst wird der untere Teil ausgeführt.

opt

Option-Fragmente enthalten ebenfalls einen boolschen Ausdruck in eckigen Klammern. Wenn der boolsche Ausdruck wahr ist, wird das Fragment ausgeführt, sonst nicht.

loop

Loop-Fragmente stellen einfache Schleifen dar. Die Schleifenbedingung wird wie schon bei den vorherigen Fragmente in einer eckigen Klammer als boolscher Ausdruck angegeben.

ref

Weiterhin gibt es noch die Interaktionsvorkommen, die den Fragmenten sehr ähnlich sind und Referenzen zu einem anderen Sequenzdiagramm darstellen, das an dieser Stelle ausgeführt werden muss. Interaktionsvorkommen werden mit dem Wort *ref* in der linken oberen Ecke markiert, während im Rahmeninneren der Name des referenzierten Sequenzdiagramms angegeben ist.

Kommunikationsdiagramm

Den Sequenzdiagrammen sehr ähnlich sind die Kommunikationsdiagramme, die ebenfalls das Senden von Nachrichten zwischen beteiligten Objekten zeigen. Im Unterschied zum Sequenzdiagramm liegt der Fokus aber eher auf der Struktur der beteiligten Objekte als auf dem zeitlichen Ablauf.

Kommunikationsdiagramme zeigen die beteiligten Klassen und Instanzen, sowie die Links (Instanzen von Assoziationen), mit denen die Objekte verbunden sind. Nachrichten werden über diese Links geschickt. Die Nachrichten werden nummeriert, damit die zeitliche Abfolge der Nachrichten erhalten bleibt. Neben der Nachricht wird ein kleiner Pfeil gezeichnet, um die Richtung der Nachricht zu bestimmen. Details sowie Beispiele sind beispielsweise in Jeckle *et al.* anschaulich beschrieben [52].

6.1.7 UML-Verteilungsdiagramm

UML-Verteilungsdiagramme werden benutzt, um Komponenten bzw. deren implementierende Artefakte auf Verarbeitungsknoten zu verteilen. Sie enthalten normalerweise folgende Elemente: Verarbeitungsknoten, Artefakte bzw. Komponenten und Assoziationen.

Verarbeitungsknoten

Verarbeitungsknoten repräsentieren Ressourcen, welche Speicher sowie Rechenkapazitäten zur Verfügung stellen. Sie können mit Stereotypen ausgezeichnet werden, um ihre spezielle Aufgabe besser zu beschreiben. Ein Verarbeitungsknoten wird grafisch als stilisierter Würfel dargestellt, dessen Name im oberen Bereich der Front des Würfels eingezeichnet wird (siehe Abbildung 6.13).

Kommunikationspfad

Assoziationen zwischen Verarbeitungsknoten werden Kommunikationspfade genannt. Sie werden mit Stereotypen ausgezeichnet, um die Art der Kommunikation näher zu spezifizieren. Ein Kommunikationspfad zwischen einem Webserver und einem Webbrowser kann beispielsweise mit dem Stereotyp «HTTP» ausgezeichnet werden.

Artefakt

Artefakte stellen konkrete Entitäten, wie zum Beispiel Dateien dar. Die Verteilung von Artefakten auf Verarbeitungsknoten wird gezeigt, indem das Artefakt in den Verarbeitungsknoten eingezeichnet wird. Grafisch wird ein Artefakt wie eine Klasse, die mit dem Stereotyp «artifact» ausgezeichnet ist, dargestellt. Spezielle Artefakte, wie zum Beispiel Datenbanken oder Webseiten, können durch Spezialisierungen des Stereotyps «artifact» ausgezeichnet werden (z. B. «database» oder «web service»).

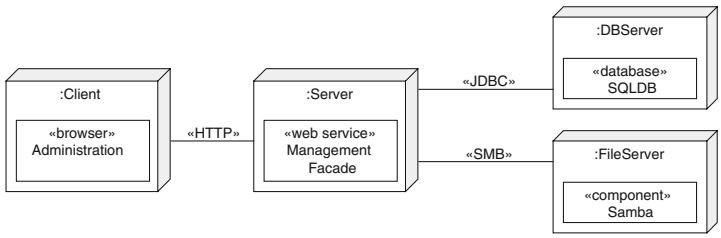


Abbildung 6.13
UML-Verteilungsdiagramm.

Technische Architekten fertigen Verteilungsdiagramme während der Designphase an, um einen Überblick über die Systemarchitektur zu erhalten (speziell bei verteilten Projekten). Von einem Verteilungsdiagramm lässt sich sehr leicht ein Übersichtsdiagramm abstrahieren, welches zur Erklärung der Architektur, z. B. für Präsentationen mittels Foliensatz, verwendet werden kann. In diesen Übersichtsdiagrammen werden meist visuelle Hilfen verwendet, um die Bedeutung von gewissen Verarbeitungsknoten zu verdeutlichen – so wird beispielsweise das Internet als Wolke oder eine Datenbank als Zylinder dargestellt.

6.1.8 UML-Zustandsdiagramme

Zustände

Ein Zustandsdiagramm *(State Machine Diagram)* beschreibt die möglichen Folgen von Zuständen eines Modellelements, im Allgemeinen eines Objekts einer bestimmten Klasse

- während seines Lebenslaufs (Erzeugung bis Destruktion)
- während der Ausführung einer Operation oder Interaktion.

Zustandsdiagramme enthalten normalerweise folgende Modellelemente: die Zustände, in denen sich das Objekt befinden kann; die möglichen Zustandsübergänge (Transitionen) von einem Zustand zu einem anderen; die Ereignisse, die Transitionen auslösen; Aktivitäten, die in Zuständen bzw. im Zuge von Transitionen ausgeführt werden.

Zustand

Zustände können in drei Kategorien aufgeteilt werden: einfache Zustände, Pseudozustände und komplexe Zustände. Zu den einfachen Zuständen gehören alle Zustände, die keine Subzustände enthalten und der Endzustand. Pseudozustände, oder transiente Zustände, sind all jene Zustände, in denen ein System nicht verweilt, z. B. der Initialzustand oder Verzweigungen. Komplexe Zustände sind Zustände, die Subzustände enthalten.

Aktivitäten, die ausgeführt werden während das Objekt in einem Zustand verharrt, können in den Zustand eingezeichnet werden. Wenn sie vorhanden sind, wird der Zustand mit einer durchgehenden Linie geteilt und in die untere Hälfte die Aktivitäten eingezeichnet. Man unterscheidet vier Arten von Aktivitäten aufgrund des Zeitpunkts an dem sie ausgeführt werden (Tabelle 6.4). Endzustände werden als umrandeter schwarzer Kreis dargestellt. Eine Zustandsmachine kann mehrere Endzustände enthalten; sobald einer erreicht wird, endet die Abarbeitung der gesamten Zustandsmaschine.

Tabelle 6.4 *Aktivitäten von UML-Zustandsdiagrammen.*

Aktivität	Ausführung innerhalb eines Zustands
entry / aktivität	Aktivität wird beim Eingang in den Zustand ausgeführt.
exit / aktivität	Aktivität wird beim Verlassen des Zustands ausgeführt.
do / aktivität	Aktivität wird ausgeführt, Parameter sind erlaubt.
event / aktivität	Aktivität behandelt Ereignis innerhalb des Zustands.

Zustandsübergang

Ein Zustandsübergang *(state transition)* erfolgt, wenn das Ereignis eintritt und die Bedingung *(guard)* erfüllt ist. Eine eventuell noch andauernde Aktivität im Vorzustand wird unterbrochen! Bei Nicht-Erfüllung der Bedingung geht das nicht konsumierte Ereignis verloren. Durch entsprechende Bedingungen können Entscheidungsbäume modelliert werden. Ein fehlendes Ereignis entspricht dem Ereignis „Aktivität ist abgeschlossen“, eine fehlende Bedingung entspricht der Bedingung [true]. Aktionen, z. B. das Senden einer Nachricht an ein anderes Objekt, sind auf Zustandsübergän-

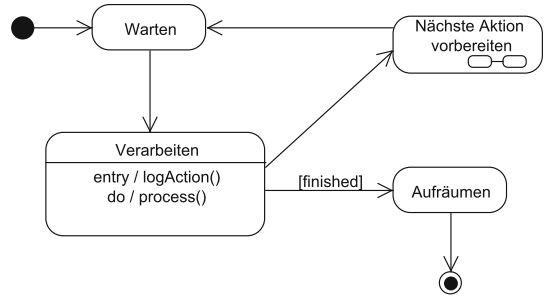


Abbildung 6.14
UML-Zustandsdiagramm.

gen möglich. Übergänge werden als Pfeile mit offenen Spitzen dargestellt, ihre Beschriftung folgt folgender Syntax:

```
Ereignis [ Bedingung ] / Aktion
```

Pseudozustand

Pseudozustände, wie der Initialzustand oder Verzweigungen, sind transient, d. h., ein Objekt verweilt nicht in diesem Zustand, sondern wird sofort einem ausgehenden Zustandsübergang folgen. Jede Zustandsmaschine enthält einen Initialzustand, der als ausgefüllter Kreis dargestellt wird und einen nicht beschrifteten ausgehenden Zustandsübergang besitzt.

Komplexer Zustand

Zustände können zu komplexen Zuständen verfeinert werden. Die Verfeinerung eines Zustands dient dazu, Subzustände in einen Zustand einzuführen. Man unterscheidet hierbei zwischen UND- und ODER-Verfeinerung.

ODER-Verfeinerung

Die ODER-Verfeinerung erlaubt es, disjunkte Sub-Zustände einzuführen, d. h., genau ein Subzustand ist aktiv, wenn der komplexe Zustand aktiv ist. Man kann sich die ODER-Verfeinerung so vorstellen, als ob der Zustand eine eigene Zustandsmaschine beschreibt.

UND-Verfeinerung

Die UND-Verfeinerung erlaubt es nebenläufige Sub-Zustände zu modellieren. Dazu wird ein Zustand mit einer gestrichelten Linie in Bereiche aufgeteilt. Jeder dieser Bereiche, kann als eigene Zustandsmaschine betrachtet werden, die parallel (also nebenläufig) ausgeführt wird. Die Bereiche können Namen erhalten, die in eckigen Klammern in den Bereich gezeichnet werden.

Verfeinerungen eines Zustands können ausgeblendet werden. In diesem Fall wird in die untere rechte Ecke des Zustands ein Icon eingezeichnet und der verfeinerte Zustand in einem anderen Diagramm dargestellt.

6.2 Modellierung von Daten und Systemschichten

In einem typischen Software-Projekt reicht die UML oft nicht aus. So hat sich zum Beispiel für die Datenbankmodellierung das Entity-Relationship-Modell als Standard etabliert. Außerdem werden andere Notationen für nicht technische Artefakte, wie zum Beispiel für das Projektmanagement, benötigt.

6.2.1 Entity-Relationship-Modell

Konzeptionelles Datenschema

Das Entity-Relationship-Modell aus der Datenmodellierung ist eine abstrakte Methode, um ein konzeptionelles Datenschema zu erzeugen, das im Software-Engineering oft verwendet wird, um die Struktur von relationalen Datenbanken zu entwerfen. Dabei werden die Daten, die in den Anforderungen vorkommen (*Entitäten, Attribute, Schlüssel* und *Relationen/Operationen*), samt *Kardinalitäten/Multiplizitäten* dargestellt. Es wird in der Designphase eines datenintensiven Projekts von den technischen Architekten entworfen und dient den Entwicklern bei der Umsetzung der Datenbank.

Entitäten

Entitäten repräsentieren eine Klassifizierung von eigenständigen Objekten, die die gleichen Attribute aufweisen. Beispiele für Entitäten eines Handelsunternehmens sind etwa *Stadt, Abteilung, Mitarbeiter* oder *Verkauf*. Die Instanz einer Entität (Datensatz) beschreibt ein Objekt dieser Klassifizierung. *Wien* oder *New York* könnten also Instanzen der Entität Stadt sein.

Relationen

Relationen sind logische Verbindungen zwischen zwei oder mehreren Entitäten. *Wohnort* ist ein Beispiel für eine Relation, die zwischen den Entitäten *Stadt* und *Mitarbeiter* existieren könnte – wenn das Software-System für jene Domäne dies vorsieht. Die Instanz einer Relation ist ein n-Tupel, das jeweils eine Instanz für jede der n-Entitäten enthält, die mit der Relation in Verbindung stehen, beispielsweise *(Max, Wien)*.

Attribute

Attribute repräsentieren generell elementare Eigenschaften von Entitäten oder Relationen, wobei Erweiterungen der ER-Notation auch Zusammengesetzte Attribute (Composites) erlauben. *Vorname* und *Nachname* sind mögliche Attribute der Entität *Mitarbeiter*, während *Adresse* und *Dauer* Beispiele für Attribute der Relation *Wohnort* sind. Ein Attribut verbindet die Instanz jeder Entität oder Relation mit einem konkreten, zulässigen Wert aus der Domäne.

Schlüssel

Ein oder mehrere Attribute können eine Instanz einer Entität eindeutig referenzieren, diese Attribute werden auch *Schlüsselattribute* oder nur *Schlüssel* genannt. Eine Entität kann einen primären und mehrere Fremdschlüssel enthalten, wobei der primäre Schlüssel sich wiederum aus mehreren Attributen zusammensetzen kann. Primäre Schlüssel identifizieren Instanzen selber, während Fremdschlüssel bei Abfragen dienen, bei denen mehrere Entitäten über ihre Relationen durch sogenannte Join-Operationen zusam-

Join

mengefügt werden. Die primären Schlüssel der Entitäten werden in Entity-Relationship-Diagrammen (ERDs) unterstrichen, um die Eigenschaften der Attribute vollständig zu beschreiben ist jedoch eine weitere Analyse bzw. eine erweiterte Notation (etwa UML) notwendig.

ER-Diagramm (ERD)

Die Diagramme, die während der Entity-Relationship-Modellierung entstehen, werden Entity-Relationship-Diagramme genannt. Die Referenz für Entity-Relationship-Modellierung und die hier vorgestellte ER-Diagramm-Notation ist die Arbeit von Peter Chen, „The entity-relationship model – toward a unified view of data“ [18].

Die Notation von Chen ist einfach zu verstehen und sehr weitverbreitet, hat aber einige Nachteile. Während der Analysephase ist die Einfachheit und Eleganz sehr nützlich, sobald aber technisches Design stattfindet, sind die Möglichkeiten, die ein einfaches ERD bietet, zu limitiert.

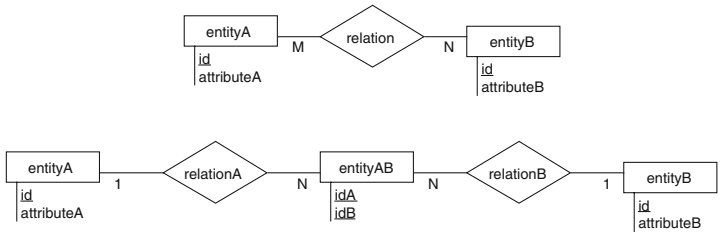


Abbildung 6.15
Relation zwischen Entität A und B mit Kardinalität M:N.

Abbildung 6.15 zeigt eine M:N-Beziehung zwischen den Entitäten A und B, die zu zwei 1:N-Beziehungen aufgelöst wird. Dabei entsteht ein sogenannter Link-Table-entityAB, der keinen eigenen Primärschlüssel besitzt, sondern eine Kombination der zwei Fremdschlüssel verwendet, um jede Instanz (Datensatz) eindeutig zu identifizieren. M:N-Relationen sind spätestens im technischen Design & Entwurf in zwei 1:N-Relationen mittels top-down-Analyse zu zerlegen, da wir im relationalen Schema modellieren und die dritte Normalform (3NF) anstreben. Die dritte Normalform ist erreicht wenn:

Kardinalität M:N

Dritte Normalform (3NF)

- Jedes Attribut der Relationen einen atomaren Wertebereich aufweist. Das heißt, zusammengesetzte oder verschachtelte Wertebereiche sind nicht erlaubt.
- Jedes Nicht-Schlüsselattribut von jedem Schlüsselkandidaten voll funktional abhängig ist. Das heißt, jedes Attribut, das nicht Teil des Schlüssels ist, ist jeweils vom gesamten Schlüssel (allen Schlüsselkandidaten) abhängig.
- Jedes Nicht-Schlüsselattribut von keinem Schlüsselkandidaten transitiv abhängt. Das heißt, jedes Attribut muss über des gesamten primären Schlüssel eindeutig auffindbar sein.

Erweiterungen (EER)

Seit der Einführung von ERDs 1976, wurde die Notation auf verschiedene Arten erweitert. Diese Erweiterungen sind allgemein unter Extended Entity Relationship (EER) bekannt und erlauben Modellierung mit mehr Semantik durch Konstrukte wie Generalisierungen oder Aggregationen. Im Weiteren werden verschiedene Erweiterungen der ER-Notation vorgestellt, die je nach Präferenz einzusetzen sind, um ein ERD zu zeichnen.

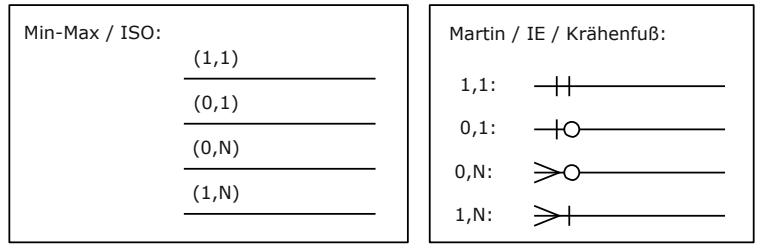


Abbildung 6.16
Erweiterungen bzw. Weiterentwicklungen von Relationen.

Relationen, Extended

Abbildung 6.16 zeigt verschiedene Weiterentwicklungen bei Relationen. Die Kardinalität 0,1 (Null-oder-Eines) ist neu sowie eine Verfeinerung der :N Kardinalität mit 0,N (Null-oder-mehrere) bzw. 1,N (Eines-oder-mehrere).

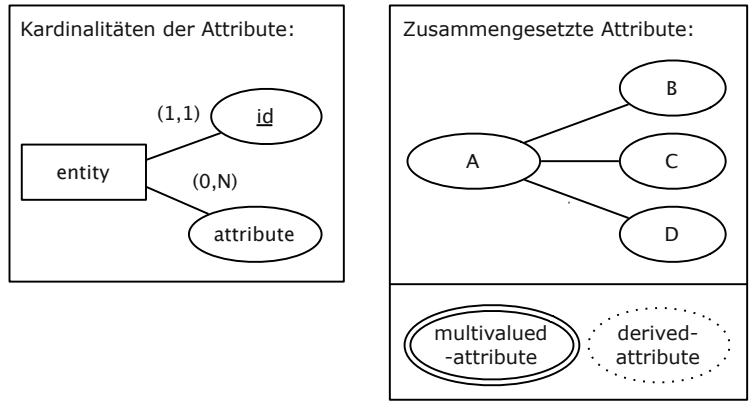


Abbildung 6.17
Erweiterungen bzw. Weiterentwicklungen von Attributen.

Attribute, Extended

Abbildung 6.17 zeigt verschiedene Weiterentwicklungen bei Attributen. Für Attribute können auch Kardinalitäten angegeben werden, zudem können Attribute verschachtelt und von anderen Attributen abgeleitet sein (*derived*). Für Attribute mit mehreren Werten sind auch eigene Notationselemente vorgesehen.

Trotz der Verbesserungen durch EER ist es nicht möglich, Konsistenz- bzw. Integritätsbedingungen direkt im Diagramm zu modellieren. Eine genaue Beschreibung der Attribute jeder Entity in Tabellenform ist notwendig, um das Schema vollständig zu beschreiben, egal welche Version der ER- oder EER-Notation verwendet wird. Um die Attribute gemeinsam mit den Entitäten und Relationen zu behandeln, ist eine andere Notation notwendig,

die dies erlaubt. Dabei ist die UML-Klassendiagramm-Notation eine vielversprechende Alternative, da UML-Klassen direkt mit Entitäten der relationalen Modellierung vergleichbar sind.

Klassendiagramm

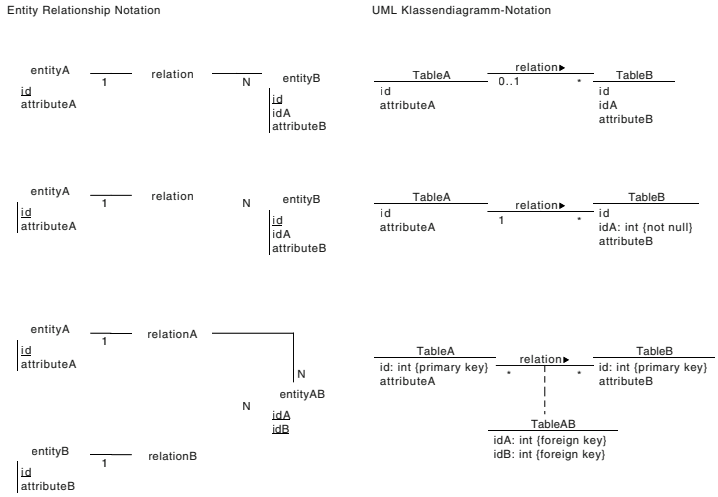


Abbildung 6.18
Entity-Relationship-Modellierung mit UML.

Erweiterungen (EER)

Abbildung 6.18 zeigt, wie die UML-Klassendiagramm-Notation verwendet werden kann, um die Entitäten, Attribute und Relationen zu modellieren. Hier ist jedoch zu beachten, dass richtig modelliert wird – die Entitäten (persistente Daten) sind nicht mit den Domänenklassen bzw. Transfer-Objects des Software-Systems gleichzusetzen! Bei der ER-Modellierung wird mit Relationen gearbeitet, während bei Domänenklassen im objektorientierten Schema gearbeitet wird.

Relational & objektorientiert

Die Entitäten, die im ER-Modell dargestellt werden, sind auch in einem Domänenmodell, das die UML-Klassendiagramm-Notation verwendet, aufzufinden – in beiden Fällen werden wichtige semantische Informationen der realen Welt abgebildet. Da beim ER-Modell aber ein Datenbankschema modelliert wird, handelt es sich explizit um Daten, die persistiert werden, also die Laufzeit des Software-Systems überdauern müssen.

Erweiterungen (EER)

Persistente Daten

Datenmodelle wie ERDs finden im weiteren Projektverlauf eine Anwendung bei Verifikation und Validierung der Anforderungen. Ein Entity-Relationship-Diagramm kann in das relationale Schema übergeführt werden, so können schon frühzeitig im Projekt datengetriebene, Black-Box-Datenbanktests bzw. Unit-Tests für die Datenbank erstellt werden, um beispielsweise die Datenintegrität für eine nicht funktionale Anforderung zu gewährleisten.

Verifikation & Validierung

Tests

Die Analyse könnte ergeben, dass der Datenbestand einer Entity besonders hoch ist. Mittels eines definierten Datenbankschemas können frühzei-

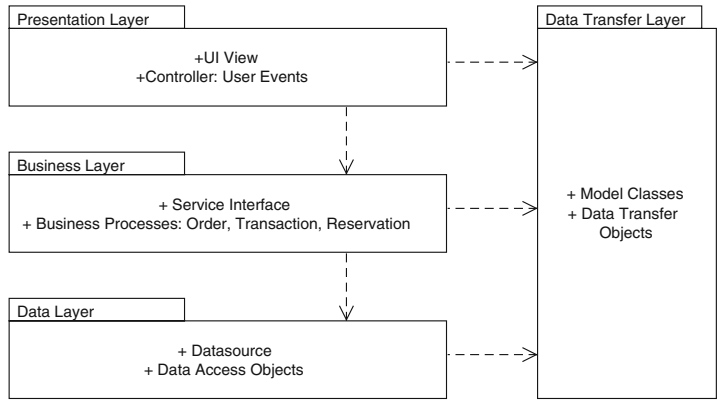


Abbildung 6.19
Schichtendiagramm mittels UML-Packages und Dependencies.

tig Performance-Tests formuliert und aufgezeichnet werden, um auch diese Anforderungen im weiteren Projektverlauf zu verifizieren.

6.2.2 Schichtendiagramme

Überblick

Schichtendiagramme sind Übersichtsdiagramme mit sehr wenigen semantischen Informationen. Grundsätzlich sollen nur folgende Informationen ausgedrückt werden:

- Gruppierung von technischen Artefakten in Pakete bzw. Schichten. Diese Pakete haben nichts mit Paketen im Sourcecode zu tun, obwohl durchaus Klassen und Komponenten eingezeichnet werden können. Externe Bibliotheken, Programme oder Subsysteme können ebenso verwendet werden, um Schichten zu beschreiben.
- Abhängigkeiten der Schichten zueinander mit Dependency-Pfeilen. Grundsätzlich werden Schichtendiagramme so gezeichnet, dass immer eine Abhängigkeit von der oberen Schicht zu der jeweils angrenzenden unteren Schicht besteht.
- Durch Abhängigkeiten wird impliziert, dass nicht angrenzende Schichten, auch nicht miteinander kommunizieren können. Dies ist auch generell der Fall, außer es werden zusätzliche Abhängigkeiten eingezeichnet.

Schichtenmodell

In Abbildung 6.19 sind insgesamt vier Schichten vorhanden, obwohl es sich um eine dreistufige Architektur handelt, hier wurde die vierte Schicht „Data Transfer Layer“ eingeführt, um den Zusammenhang zum architekturellen MVC Pattern herzustellen. Schichtendiagramme modellieren keinesfalls architektonische Muster, sondern dienen rein als Überblick über eine bestimmte Architektur.

Schichtendiagramme werden während der Analysephase von den technischen Architekten erstellt, um eine konzeptionelle Übersicht über das Projekt zu erhalten. Es dient dem Verständnis der Systemarchitektur für die Mitarbeiter des Projekts sowie externen Personen.

6.3 Projektmanagement-Artefakte

Wie bereits in Abschnitt 2.2.1 und in Kapitel 4 ausführlich beschrieben, sind für die Projektplanung und Projektsteuerung effektive und effiziente Methoden und Werkzeuge erforderlich. Folgende Notationen und Methoden des Projektmanagements werden dazu verwendet, die Arbeit im Rahmen eines Projekts auf die eine oder andere Weise zu planen und zu steuern. Dazu ist es wichtig den Begriff „Arbeitspaket“ zuerst zu definieren. Ein Arbeitspaket setzt sich zusammen aus:

- Name,
- Start- und Endzeitpunkt bzw. Dauer,
- Vorbedingungen / Abhängigkeiten,
- Prozentsatz der Fertigstellung,
- Priorität und
- Ressourcen (Arbeitskräfte).

6.3.1 Projektstrukturplan

Der Projektstrukturplan (PSP), auch Work Breakdown Structure (WBS) genannt, besteht aus einer Aufgliederung der Arbeit in Arbeitspakete und Attribute, die diese Arbeitspakete näher zu beschreiben. Durch eine Top-down-Analyse des Arbeitsaufwands eines Projekts wird ein abstraktes Arbeitspaket, das den Gesamtaufwand repräsentiert, in kleinere Unterpakete aufgeteilt (*breakdown*). Dadurch ergibt sich eine hierarchisch geordnete Struktur, in der jedem Arbeitspaket eine eindeutige geordnete Nummer (z. B. 1, 2, 3) und ein Prozentsatz des Aufwands zugeteilt wird. Der PSP ist vor allem für das Projektmanagement relevant (siehe Abschnitt 4.3.5).

Top-down-Analyse

Hierarchie

Auf oberster Ebene sowie in jeder weiteren Verfeinerungsstufe muss der Aufwand aller Pakete insgesamt den Gesamtaufwand (100% der Arbeit) des Projekts ergeben. Durch diese sogenannte 100%-Regel beschreibt jede Ebene den gesamten Umfang der Arbeit. Jede weitere Ebene verfeinert die Aufteilung der Pakete und lässt eine genauere Schätzung des Aufwands für jedes einzelne Arbeitspaket zu. Verfeinerungsstufen und die 100%-Regel sind in Abbildung 6.20 grob veranschaulicht.

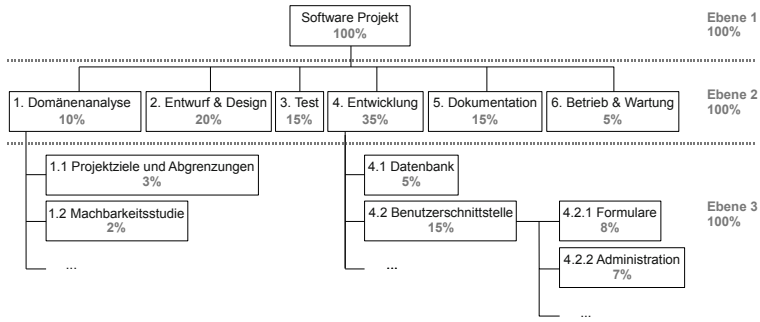


Abbildung 6.20 *Auszug eines PSP für ein typisches mittelgroßes Software-Projekt.*

Arbeitspakete strukturiert erfassen

Diese Notation ist zwar zum strukturierten Erfassen der Arbeitspakete hilfreich, wird jedoch sehr schnell unübersichtlich. Zur Dokumentation des PSP kann eine einfache Tabelle (siehe Tabelle 6.5) verwendet werden. Die Tabelle erlaubt eine hierarchische Darstellung der Pakete sowie die Zuteilung von weiteren Attributen zu jedem Paket durch Hinzufügen von weiteren Spalten, wie etwa die Spalten „Anfang“, „Ende“ und „Personentage“ in Tabelle 6.5.

Tabelle 6.5 *PSP auf Phasenebene.*

Nr.	Arbeitspakete	Anfang	Ende	Personentage
1	Domänenanalyse	02.10.09	11.11.09	28
1.1	Projektziele und Abgrenzungen	02.10.09	15.10.09	9
1.2	Machbarkeitsstudie	16.10.09	04.11.09	13
1.3	...	...	...	...
2	Entwurf und Design	19.10.09	04.12.09	34
3	Test	06.11.09	28.02.10	29
4	Entwicklung	06.11.09	28.01.10	57
4.1	Datenbank	06.11.09	21.11.09	15
4.2	Benutzerschnittstelle	09.11.09	14.12.09	26
4.2.1	Formulare	09.11.09	04.11.09	11
4.2.2	Administration	19.11.09	14.12.09	9
4.2.3	...	...	...	...
4.3	...	...	...	...
5	Dokumentation	02.10.09	28.03.10	30
6	Betrieb & Wartung	11.01.10	28.03.10	20

Aktivitäten statt Methoden

Ein häufiger Fehler ist, Methoden statt Aktionen als Pakete zu definieren. Methoden ändern sich häufig, und PSPs müssen mit dem Projektfortschritt aktualisiert werden. Beispielsweise könnte das Erstellen eines Prototyps gewisse Technologien ausschließen. Alle Arbeitspakete, die auf den Methoden dieser Technologien basieren, müssten aktualisiert werden. Es soll also vermieden werden im Namen der Pakete die dafür vorgesehenen Methoden zu verankern.

> Bad Practice: JSP Formulare X, Y und Z der Benutzerschnittstelle.

> Good Practice: Entwicklung der Benutzer-Views X Y Z.

Der PSP wird vom Projektmanager während den Analyse- und Designphasen erstellt und im weiteren Projektverlauf ständig aktualisiert und von al-

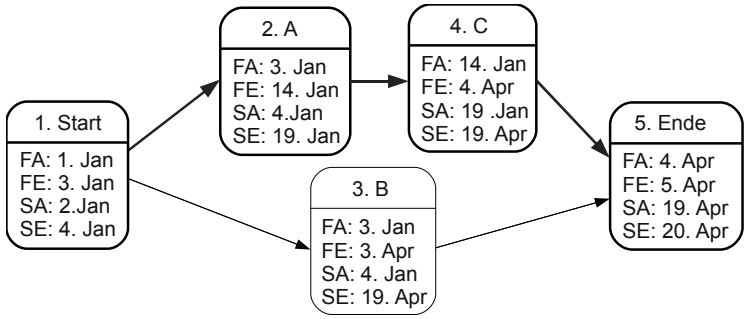


Abbildung 6.21
Netzplan eines Software-Projekts als PERT-Diagramm.

len Teammitgliedern verwendet. Der PSP beschreibt den gesamte Aufwand eines Projekts und ist das grundlegende Projektmanagementartefakt, aus dem Projektpläne (siehe Abschnitt 6.3.3) erstellt werden.

6.3.2 Netzplan, PERT

PERT

Die *Program Evaluation and Review Technique* (PERT) ist eine ereignisorientierte Netzplantechnik. Jede Aktivität eines Projekts wird als Knoten eines Graphen dargestellt. Eine gerichtete Kante von einer Aktivität A zu einer Aktivität B bedeutet: Aktivität B kann frühestens dann beginnen, wenn Aktivität A beendet ist. A heißt Vorgänger von B und B heißt Nachfolger von A.

Notation

Jedes Projekt hat einen Anfangsknoten, eine Aktivität, die den Projektbeginn darstellt, und einen Endknoten, das Projektende. Alle anderen Knoten besitzen zumindest einen Vorgänger und einen Nachfolger. Da es keine zyklischen Abhängigkeiten geben kann – das Projekt wäre dann undurchführbar – enthält der Netzplan auch keine geschlossenen Kantenzüge oder Zyklen. Graphentheoretisch gesprochen ist ein Netzplan ein azyklischer gerichteter Graph mit genau einem Anfangsknoten und genau einem Endknoten. Der sogenannte „kritische Pfad“ eines Projekts ist die Verkettung jener Knoten im Netzplan, bei denen eine Änderung des Endtermins eine Verzögerung des gesamten Projekts zur Folge hätte. In einem Netzplan wird der kritische Pfad durch die Kette von Knoten definiert, die in Summe die längste Dauer aufweist. Eine Visualisierung des kritischen Pfades ist empfehlenswert, in Abbildung 6.21 ist der kritische Pfad rot markiert. Aufgrund der Aktivitätsdauern, des gegebenen Projektanfangs, der fixen Termine der Meilensteine und des Kalenders können für jede Aktivität folgende Zeitpunkte berechnet werden:

- *FA* ... frühestmöglicher Anfangszeitpunkt,
- *FE* ... frühestmöglicher Endzeitpunkt,
- *SA* ... spätestmöglicher Anfangszeitpunkt,
- *SE* ... spätestmöglicher Endzeitpunkt.

Der FA einer Aktivität wird graphentheoretisch als Länge des längsten Weges vom Anfangsknoten zur Aktivität berechnet. Der SA ist der spätestmögliche Anfangszeitpunkt, sodas gerade noch der SE des Endknotens (Projektende) erreicht werden kann. Für jede Aktivität gilt:

- FE = FA + Dauer,
- SE = SA + Dauer.

Netzpläne dienen dazu Abhängigkeiten zwischen Aktivitäten bzw. Arbeitspaketen für das Projektmanagement aufzuzeigen. Sie werden normalerweise von Projektmanagern in der Analysephase erstellt, jedoch ist insbesondere bei technischen Arbeitspaketen die Mitarbeit von Entwicklern notwendig. Sie sind zwar als Vorstufe zum Projektplan (siehe Abschnitt 6.3.3) nützlich, aber für den weiteren Projektverlauf nicht wesentlich, da auch im Projektplan die Abhängigkeiten zwischen den Arbeitspaketen erfasst werden können.

6.3.3 Balkendiagramm (GANTT-Diagramm)

Projektfortschritt

Balkendiagramme, die in der heutigen Software-Entwicklung verwendet werden, tragen meist den Namen „Gantt“ (nach Henry Gantt). GANTT-Diagramme werden im Projektmanagement eingesetzt, um den Projektfortschritt zu illustrieren. Dabei kann die bisher erledigte sowie die noch bevorstehende Arbeit übersichtlich dargestellt werden. Die hierarchisch gegliederten Arbeitspakete mit ihren Start- und Endzeitpunkten werden untereinander als Liste dargestellt. Arbeitspakete fallen in 3 Kategorien:

1. Tasks: Arbeitspaket mit definierter Dauer und Zuständigkeiten.
2. Aktivitäten: Übergeordnete Tasks umfassen mehrere Tasks.
3. Meilensteine: Tasks, die einen bestimmten Zeitpunkt spezifizieren.

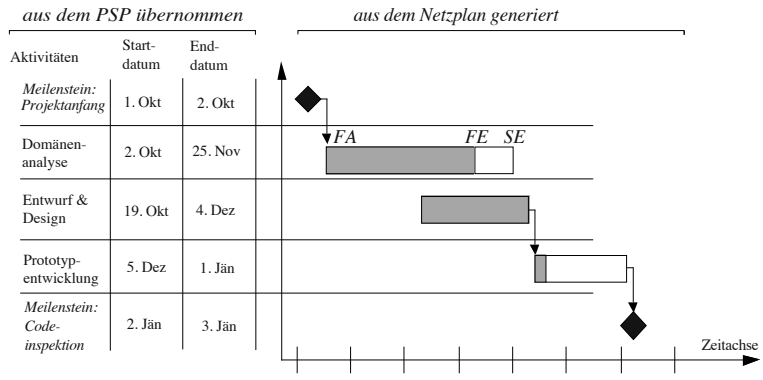


Abbildung 6.22 *Aufbau des GANTT-Diagramms.*

Abbildung 6.22 illustriert, aus welchen Informationen ein Balkendiagramm zusammengesetzt wird. Die hierarchische Struktur von Aktivitäten, Tasks und Meilensteinen werden aus der WBS (siehe Abschnitt 6.3.1) übernommen und im linken Teil des Diagramms untereinander aufgelistet.

Zeitbalken

Der rechte Teil des Diagramms wird aus dem Netzplan (siehe Abschnitt 6.3.2) generiert und visualisiert die Tasks von der linken Seite auf einer Zeitachse als sogenannte „Zeitbalken“ . Die frühestmöglichen Anfangszeitpunkte (FA) und spätestmöglichen Endzeitpunkte (SE) der Tasks definieren die Länge der Zeitbalken. Um den Grad der Fertigstellung auch grafisch darzustellen, wird der frühestmögliche Endzeitpunkt (FE) für begonnene Tasks eingezeichnet. Meilensteine werden als Route visualisiert, und die Zeitachse selber wird je nach Detailgrad der Ausarbeitung in Monate, Kalenderwochen oder Tage eingeteilt.

Projektplan

Kritischer Pfad

GANTT-Diagramme werden von den Projektmanagern für das Erstellen von Projektplänen eingesetzt (siehe Abschnitt 4.3.5). Populär ist auch die Darstellung von Abhängigkeiten der einzelnen Tasks und Meilensteine untereinander, diese Abhängigkeiten ergeben einen „kritischen Pfad“ durch das Projekt (der kritische Pfad ist auch aus dem Netzplan abzulesen). Abhängige Arbeitspakete unterliegen auch zeitlichen Restriktionen, beispielsweise können zwei Tasks, die von einander abhängig sind, nicht gleichzeitig ablaufen und müssen hintereinander ausgeführt werden.

Task Ressourcen

Die meisten Tools zum Erstellen von GANTT-Diagrammen erlauben zudem die Darstellung von zusätzlichen Informationen wie Dauer, zugewiesene Personen (Ressourcen) oder Identifikationsnummer jedes einzelnen Tasks.

Kritik

Während GANTT-Diagramme leicht zu verstehen sind, können nur verhältnismäßig wenig Information übersichtlich dargestellt werden, dies ist einer der Hauptkritikpunkte dieser Darstellungsmethode für Projektpläne. Die gesamte Projektlaufzeit eines mittelgroßen Projekts ist bereits schwierig darzustellen, auch wenn nur Aktivitäten und Meilensteine ausgearbeitet werden und auf detailliertere Tasks vorerst verzichtet wird. Außerdem ist das Ausmaß an Arbeit, also die Gewichtung der Arbeitspakete zueinander im Diagramm nicht enthalten, so kann es oft dazu kommen, dass zwei Aktivitäten oder Tasks, die eine gleiche Dauer haben, als gleich „schwierig“ eingestuft werden, obwohl sie um einige Größenordnungen unterschiedlich sind.

6.3.4 Burn-Charts

Repräsentation des Arbeitsfortschritts

Burn-Charts sind ein Eckpfeiler von agilen Software-Prozessen wie z. B. SCRUM, (siehe Kapitel 3 für eine Beschreibung des Vorgehensmodells und Kapitel 4 für die Verwendung von Burn-Charts im Projektmanagement). Burn-Charts zeigen eine visuelle Repräsentation des aktuellen Status und des Arbeitsfortschritts, insbesondere (a) wie viel und wie schnell Arbeit

in der Vergangenheit verrichtet wurde, (b) wie viel Arbeit noch zu erledigen ist und (c) wann die Arbeiten insgesamt fertig werden, also wann ein Projekt bzw. die Iteration eines Projekts voraussichtlich fertig sein wird. Ein Team von Entwicklern arbeitet beispielsweise an einem Projekt, der Arbeitsaufwand wird auf der Y-Achse eines Burn-Charts aufgetragen, und das Team „brennt" sozusagen durch die Arbeit, bis alle Arbeitspakete abgeschlossen werden und eine Iteration bzw. das ganze Projekt fertig ist.

Burnup, Burndown

Je nachdem, ob der Arbeitsaufwand im Diagramm nach oben oder nach unten aufgetragen wird, heißen die Diagramme *Burndown* oder *Burnup* respektive. Abbildung 6.23 zeigt ein typisches Burndown-Chart, das die gesamte Projektlaufzeit betrachtet. Hier wurde für eine Iteration bzw. einen SCRUM-Sprint, eine Woche gewählt und am Ende jeder Iteration die erbrachten Leistungen aufgetragen.

Einheiten der Achsen

Für Burn-Charts werden die Elemente eines einfachen Liniendiagramms verwendet, optional können auch Markierungen der Datensätze auf den Linien eingezeichnet werden. Auf der X-Achse wird die Zeit in Tagen, Wochen oder auch abstrakter in Iterationen aufgetragen. Auf der Y-Achse wird der Arbeitsaufwand aufgetragen. Der verwendeten Einheit für die Y-Achse soll je nach Projekt besondere Aufmerksamkeit geschenkt werden, denn in der Software-Entwicklung ist es oft schwierig, eine Einheit zu finden, die nicht expandiert. Unter besten Voraussetzungen könnte es sein, dass eine relativ stabile Anzahl von zu implementierenden Anwendungsfällen oder Komponenten in einem Projekt vorliegt. Es gibt hier jedoch keine definitive Antwort – in jeder neuen Iteration könnten weitere Arbeitsaufwände, z. B. in Form von Anforderungen der Kunden oder Bugs, hinzukommen.

Y-Achse: Arbeitsaufwand

In der agilen Entwicklung werden für die Y-Achse fast immer User Stories oder Anwendungsfälle aufgetragen, wobei hier aber der Aufwand aufgetragen wird und nicht einfach die Anzahl. Die User Stories bzw. Anwendungsfälle werden mit Aufwand in Personentagen oder abstrakten Punkten bewertet, die Summe des Aufwands aller Anwendungsfälle bildet das Maximum bzw. Minimum auf der Y-Achse, je nach Burndown oder Burnup. Für abstrakte Punkte wird etwa eine Skala von eins bis zehn gewählt, wobei ein Anwendungsfall mit vier Punkten doppelt so schwierig zu implementieren sein soll wie einer mit zwei Punkten. Es könnten aber auch Sourcecode-Zeilen (LoC) oder zu implementierende Klassen als Einheit der Y-Achse verwendet werden.

Burndown-Linie & Initiale Schätzung

In Abbildung 6.23 repräsentiert die Burndown-Linie die tatsächlich verrichtete Arbeit. Jeder Punkt auf der Linie ist Anfang bzw. Ende einer Iteration oder eines Sprints, d. h., zu jedem dieser Zeitpunkte wurde in einem Meeting festgehalten, welche Features seit dem letzten Meeting fertiggestellt wurden und welche Features für den nächsten Sprint vorgesehen sind. Aufgrund der verfügbaren Ressourcen im Projekt, wurde vorher festgelegt, dass der erwartete Arbeitsfortschritt fünf Produkt-Features pro Woche beträgt, diese Werte sind als „Initiale Schätzung" in Abbildung 6.23 aufgetragen.

Werden weniger Features implementiert als geplant, so biegt sich die Burndown-Linie nach oben und entfernt sich immer mehr von der Initialen Schätzung. Werden mehr Features als geplant fertiggestellt, nähert sich die Burndown-Linie wieder der Initialen Schätzung. Zu beliebigen Zeitpunkten können anhand des momentanen Trends der Burndown-Linie (Steigung), Schätzungen über den weiteren Projektverlauf gemacht werden, etwa um ein voraussichtliches Projektende festzustellen. In Abbildung 6.23 wurden zwei solche Schätzungen vorgenommen.

Trends & Schätzungen

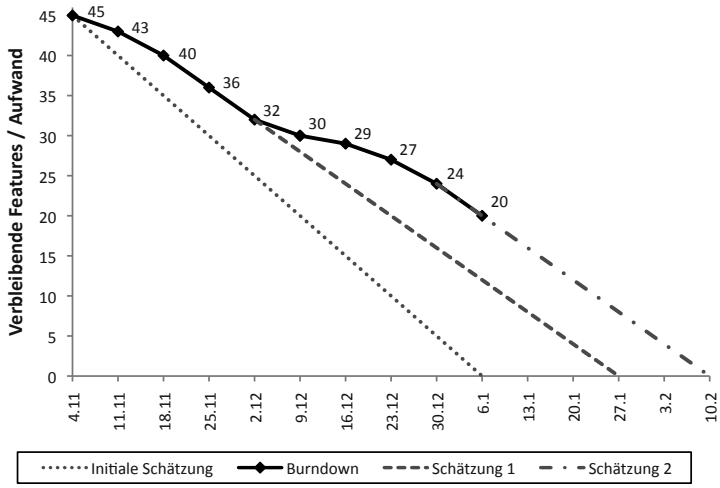


Abbildung 6.23
Burndown Chart: Jeder Punkt auf der Burndown-Linie repräsentiert das Ende eines SCRUM-Sprints.

Detailgrad

Burn-Charts sind eine sehr einfache aber erstaunlich mächtige Darstellungsmethode des Projektfortschritts. Je nach Detailgrad der Charts, kann (a) der Fortschritt der einzelnen Sprints, (b) der Fortschritt der nächsten Version (Release) und (c) der Fortschritt des Produkts anhand des Trends der Burndown-Linie dargestellt werden.

Burn-Charts enthüllen die verfolgte Strategie, zeigen den erbrachten Fortschritt gegenüber Schätzungen und öffnen die Tür zu Diskussionen über die weitere Vorgehensweise und schwierige Entscheidungen, wie z. B. den Arbeitsaufwand für kommende Iterationen anzupassen oder Deadlines zu verlängern.

6.4 Zusammenfassung

Notationen und Methoden der Modellierung

Notationen und Methoden der Modellierung beschreiben praktisch besonders relevante Beschreibungsformen für die Entwicklung kommerzieller Software-Produkte. Wesentliche Ziele sind das Erfassen von Beiträgen zu Anforderungen an das geplante Software-Produkt, Vorschläge für Lösungsansätze sowie die Überprüfung der Modellsichten auf Brauchbarkeit so-

wohl durch technische Experten als auch durch Anwender und Auftraggeber.

Auswahl an Diagrammen

Neben einer Auswahl an Diagrammen aus der Familie der Unified Modeling Language, die besonders geeignet für die Beschreibung von Software-Artefakten ist, werden für kommerzielle Software-Produkte Modellsichten zur Beschreibung persistenter Daten, konzeptioneller Systemschichten und zur Projektplanung gebraucht.

UML-Diagramme

UML-Anwendungsfälle dienen zur groben Erfassung der Kontexte von Anforderungen in der Anwendungsdomäne. Zur Strukturierung des technischen Lösungsansatzes werden Paketdiagramme, Klassendiagramme und Komponentendiagramme eingesetzt. Das geplante Verhalten des Systems bzw. von Systemteilen beschreiben Aktivitätsdiagramme, Sequenzdiagramme und Zustandsdiagramme. Die physische Verteilung der Software-Komponenten kann mit einem Verteilungsdiagramm beschrieben werden.

Datenstrukturen und Architekturkonzepte

Für die Darstellung zu persistierender Datenstrukturen hat sich das Entity-Relationship-Diagramm seit Langem bewährt. Software-Architekten verwenden Schichtendiagramme zur Darstellung und Diskussion konzeptioneller Architekturentwürfe.

Projektmanagement

Im Projektmanagement gibt es als gängige Modellsichten den Projektstrukturplan zur Beschreibung der Hierarchie von Arbeitspaketen, den Netzplan zur Erfassung zeitlicher und logischer Abhängigkeiten zwischen Arbeitspaketen und Balkendiagramme, die den Zusammenhang zwischen Arbeitspaketen, Ressourcen und Terminen im Projektkalender darstellen.

Die Verwendung ausreichend ausdrucksstarker und einheitlicher Notationen und Modellsichten im Projektteam trägt grundlegend zu einem effektiven und effizienten Arbeiten im Projekt bei.

7 | Software-Architektur

Dieses Kapitel beschäftigt sich mit der Strukturierung von Software-Systemen und stellt einen Überblick verschiedener Architekturstile vor. Mit der zunehmenden Komplexität moderner Software-Systeme gewinnt die Software-Architektur und die Rolle eines Software-Architekten immer mehr an Bedeutung. Die Architektur bringt klare Grenzen und Struktur in die Anwendung, befasst sich aber nicht nur mit der statischen Systemstruktur, sondern legt auch besonderes Augenmerk auf die nicht funktionalen Anforderungen, wie Skalierbarkeit, Performanz oder Verfügbarkeit.

Übersicht

7.1 Was ist eine Software-Architektur

Komplexität und Struktur

In der heutigen Zeit werden ein Großteil der Kerngeschäfte in einem Unternehmen ausschließlich über die EDV abgewickelt. Die Komplexität der notwendigen Systeme nimmt rasant zu. Daher ist auch eine Struktur in Software-Systemen von großer Bedeutung. Software-Architektur ist ein Spezialgebiet des Software-Engineering, spielt dabei aber eine wesentliche Rolle. Durch die Strukturierung werden klare Grenzen in eine Anwendung gebracht.

Sehr oft wird dieser Bereich mit der Architektur der Baubranche verglichen. Ganz kann man diesem Vergleich jedoch nicht zustimmen, da Gebäude viel konkreter sind als Software-Systeme. Moderne Software-Systeme sind während der Einsatzzeit meist auch kontinuierlichen Änderungen unterworfen. Die Software-Architektur beschreibt daher nicht nur die statische Struktur einer Anwendung, sondern es fließen auch nicht funktionale Anforderungen, wie Skalierbarkeit, Performanz oder Verfügbarkeit in die Architektur ein. Eine gute Architektur erleichtert zudem auch zukünftige Änderungen und Weiterentwicklungen der Software.

Definition

Das *Software-Engineering-Institut* (SEI)[1] hat vor einiger Zeit begonnen, eine Definition zur Software-Architektur zu suchen. Das Ergebnis sind mittlerweile mehr als 50 Definitionen. Eine mögliche Definition ist jene von Len Bass:

> „Structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them“ *Len Bass [4]*

Software-Architekt

Eine Software-Architektur identifiziert also Komponenten und regelt deren Zusammenspiel so, dass sowohl die funktionalen als auch die nicht funktionalen Anforderungen erfüllt sind. Die Bedeutung der Software-Architektur ist sogar so stark geworden, dass eine neue Ingenieurdisziplin, jene des Software-Architekten, entstanden ist. Zu diesem Thema gibt es eine Reihe guter Fachbücher, die sich ausschließlich den Tätigkeiten eines Software-Architekten widmen. Eine ausführliche Beschreibung über die Aufgaben und Tätigkeiten eines Software-Architekten können in [7] nachgelesen werden. Zu den wichtigsten Aufgaben des Software-Architekten zählen:

- Er ist die zentrale Drehscheibe zwischen Kunden, Projektmanager, Designer und Entwicklern. Diese Aufgabe verlangt vom Software-Architekten, dass er sich in unterschiedliche Rollen versetzen kann.
- Die Anforderungen (funktionale und nicht funktionale) in eine Software-Architektur abbilden. Für die Abbildung der Software-Architek-

[1] `http://www.sei.cmu.edu/`

tur wird sehr häufig UML verwendet, da UML für die unterschiedlichen Aspekte der Architektur die passenden Modelle anbietet (siehe auch Abschnitt 6.1).

- Der Software-Architekt argumentiert Architektur- und Technologie-Entscheidungen gegenüber dem Kunden und dem Management.
- Die Dokumentation von Software-Architekturen ist eine wichtige Aufgabe des Architekten, denn die Dokumentation hilft allen Projektbeteiligten, ein Verständnis von der Architektur zu bekommen. In der Dokumentation sollten auch Begründungen von Architektur-Entscheidungen aufgelistet sein, damit man später weiß, warum man sich für diesen oder jenen Stil oder eine bestimmte Technologie entschieden hat.
- Der Software-Architekt arbeitet in enger Beziehung mit dem Projektmanagement zusammen. Er wird daher auch sehr stark in die Projektplanung der Iterationen, der Arbeitspakete und Meilensteine einbezogen.
- Während der Implementierung muss der Software-Architekt sicherstellen, dass die Implementierung nicht von der vorgegebenen Architektur abweicht. Für die Überwachung der Architektur werden in den meisten Fällen Werkzeuge eingesetzt, wie z. B. SonarJ[2].

Zentrale Drehscheibe

In heutigen Software-Projekten nimmt die Software-Architektur eine zentrale Rolle ein und steht mit vielen anderen Projektbeteiligten in enger Verbindung. Die Software-Architektur als zentrale Drehscheibe zwischen allen Projektbeteiligten ist sehr deutlich aus Abbildung 7.1 ersichtlich. Im Folgenden wird nun kurz auf die Wechselbeziehungen zwischen der Architektur und den Projektbeteiligten eingegangen.

- Die Anforderungsanalyse gibt sowohl die funktionalen als auch die nicht funktionalen Anforderungen vor. Natürlich gibt es Wunschvorstellungen, die in einer Architektur nicht umgesetzt werden können und die mit den Analysten abgestimmt werden müssen. Gerade in der Analysephase ist die Architektur einer ständigen Änderung unterzogen (siehe auch Abschnitt 2.3).
- Die Software-Architektur bildet die Basis für die Umsetzung. In ihr wird sowohl die statische Struktur als auch das dynamische Verhalten abgebildet.

[2] `http://www.hello2morrow.com/products/sonarj`

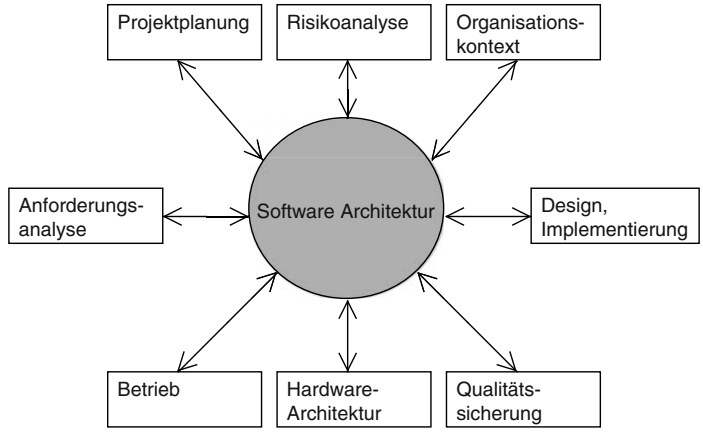


Abbildung 7.1 *Die Software-Architektur als zentrale Drehscheibe [86].*

- Wesentlichen Einfluss hat die Architektur auch auf die gesamte Projektplanung. Aus den definierten Komponenten und Schnittstellen lassen sich sehr gut Arbeitspakete für das Projektteam ableiten. Man spricht dabei auch oft von einer Work Break Down Structure (siehe auch Abschnitt 4.3).
- Qualitätssicherung sollte parallel zur Software-Architektur betrieben werden. Dies umfasst vor allem die Erstellung von Testfällen. Aber auch Architektur-Reviews tragen maßgeblich zur Qualität von Software-Architekturen bei (siehe auch Kapitel 5).
- Die Risiko-Analyse erarbeitet Lösungsansätze und -maßnahmen bei kritischen Komponenten.
- Viele Organisationen setzen Vorgehensmodelle (RUP, V-Modell, agile Methoden ...) für die Software-Entwicklung ein. Der Einsatz eines Vorgehensmodells hat natürlich auch einen Einfluss auf das Vorgehen bei der Umsetzung einer Architektur (siehe auch Kapitel 3).
- Die Software-Architektur beschreibt nicht nur die statische Struktur des Systems, sondern auch die Anforderungen an die Laufzeitumgebung, wie etwa Datenbanken, Speicher usw.
- Gerade bei komplexen Systemen muss die Software mit der Hardware optimal abgestimmt werden. Dies betrifft vor allem Systeme mit hoher Performanz, Verfügbarkeit oder Ausfallsicherheit.

7.2 Wie entstehen Architekturen

Anforderungen

Software-Architekturen entstehen bereits in frühen Phasen der Software-Entwicklung. Streng genommen werden die ersten Steine für die Archi-

tektur bereits in der Anforderungsphase (siehe Abschnitt 2.3) gelegt. Die Anforderungen an ein Software-System geben schon einen großen Teil der Richtlinien vor und haben somit direkten Einfluss auf die Architektur. Die klassische Software-Architektur beginnt jedoch mit dem Entwurf und dem Design (siehe Abschnitt 2.5). In diesem Abschnitt soll ein Vorgehensmodell vorgestellt werden, wie Software-Architekturen entstehen.

Technologie ist zweitrangig

Leider verbindet man Software-Architektur zu oft mit der einzusetzenden Technologie. Entwickler sind oft sehr fokussiert auf bestimmte Technologien. Leider folgt daraus, dass bei vielen Software-Projekten bereits zu früh eine Entscheidung für eine bevorzugte Technologie fällt. Dies kann unterschiedliche Gründe haben:

- Firmenpolititk: Das gesamte Unternehmen setzt eine Datenbank eines bestimmten Herstellers ein. Alle Folgeprojekte müssen zwingend auch mit dieser Datenbank arbeiten, obwohl die nicht funktionalen Anforderungen gegen diese sprechen.
- Technologie-Hype, daher fällt die Entscheidung für die gerade moderne Technologie.
- Die Entscheidungsträger sind bereits in eine Technologie eingearbeitet und interessieren sich nicht für andere. Bei nicht bekannten Technologien fehlt die Erfahrung, erneute Einarbeitungszeit fällt an. Diese Punkte tragen dazu bei, eine andere Technologie zu meiden.
- Industrie-Standards, wie EBJ, JSF oder andere.

Technologie-getrieben

Daher ist zu beobachten, dass heutige Software-Architekturen häufig zu sehr technologiegetrieben sind. Aussagen wie: „Wir haben eine Web-Service-Architektur“ oder „unsere Architektur ist Hibernate, Spring, JSF“, sind keine Seltenheit. Den Fokus bereits in der Analyse- und Designphase auf eine bestimmte Technologie zu legen, kann fatale Folgen für die Architektur, Flexibilität und Qualität eines Systems haben. Das Ergebnis kann eine starre Struktur und die zu starke Auslegung auf die Konzepte einer bestimmten Technologie sein. Oft werden dabei die tatsächlichen Anforderungen an ein System nicht hinreichend berücksichtigt, da sie beispielsweise mit der ausgewählten Technologie nicht oder nur umständlich umgesetzt werden können.

Iterative Entwicklung

Auch bei der Erstellung einer Software-Architektur haben sich *Best-Practices* herauskristallisiert. Ein generelles Vorgehensmodell gibt es jedoch noch nicht. Eine Architektur entsteht vielmehr in iterativen Schritten. Die Entwürfe werden immer wieder überarbeitet und mit jeder Iteration gegen die Anforderungen geprüft. Bei der Ausprägung der einzelnen Schritte macht es einen großen Unterschied, ob das zu entwickelnde System eine Individuallösung oder eine Produktentwicklung darstellt. Es soll hier ein möglicher Ansatz [82] vorgestellt werden, wie Architekturen entworfen werden können.

Phasen der Architekturentwicklung

Das nun vorgestellte Vorgehensmodell für den Entwurf von Software-Architekturen gliedert sich in drei Phasen. Die *erste Phase* beschäftigt sich mit der *Evaluierung* der Anforderungen und deren Abbildung in eine Architektur. In dieser Phase werden die erstellten Artefakte aus der Anforderungsanalyse benötigt, wie der Projektauftrag, Anwendungsfälle, funktionale und nicht funktionale Anforderungen, Designbedingungen und weitere Artefakte. Sie liefern notwendige Informationen, um die Modelle für die Software-Architektur zu erstellen.

Die Evaluierungsphase soll einen ersten Entwurf der Software-Architektur liefern. Das Ergebnis dieser Phase sind mehrere Modelle, die die jeweiligen Aspekte der Architektur darstellen: (a) technologieneutrales Modell, (b) Programmiermodell, (c) Technologie-Mapping, (d) Mock-Plattform und der (e) vertikale Prototyp.

In der *zweiten Phase* werden die Modelle der ersten Phase in *iterativen* Schritten überarbeitet und an die Anforderungen angepasst. Durch den iterativen Ansatz werden die Modelle immer mehr an die endgültige Architektur des Systems angepasst und verbessert.

Die *dritte Phase* ist nur für große Projekte von Interesse und konzentriert sich um die *Automatisierung* von Artefakten. Dabei entstehen Artefakte wie das (a) Architektur-Metamodel, ein (b) domänenspezifisches Programmiermodell, ein (c) Codegenerator sowie ein (d) Modell für modellgetriebene Architekturen. Diese Phase geht davon aus, dass die Anforderungen in ausreichender Form vorliegen, speziell in Beziehung auf Design-Einschränkungen.

Betrachten wir nun die einzelnen Phasen etwas detaillierter:

Technologieneutrales Modell (P1)

Die Anforderungen müssen zunächst in ein technologieneutrales Modell abgebildet werden. Vorwiegend werden dabei die Systemkomponenten, deren Interaktion und Strukturierung definiert. Häufig bedient man sich dabei der UML, um die Strukturen und das Verhalten des Systems darzustellen. Es stellt sich jedoch immer wieder die Frage, wie viel Technik in diesem Modell abgebildet werden kann. Hier ist der Architekt gefragt, in welchem Detailgrad dieses Modell ausgearbeitet werden soll.

Programmiermodell (P1)

Nachdem das technologieneutrale Modell vorhanden ist, muss dieses für die Entwickler aufbereitet werden. Das Programmiermodell ist das Modell aus der Sicht des Entwicklers und stellt die API der Architektur dar. In diesem Modell werden Muster (siehe Kapitel 8) verwendet, die Vorgaben für die Entwickler enthalten, z. B. soll gegen Interfaces (siehe Abschnitt 8.2.1) programmiert werden oder bestimmte Objekte nicht direkt instanziert, sondern eine Factory verwendet werden (siehe Abschnitt 8.3.2).

Technologie-Mapping (P1)

Mithilfe des technologieneutralen und des Programmiermodells können diese beiden Modelle auf konkrete Technologien abgebildet werden. Die Entscheidung erst in diesem Schritt zu treffen hat mehrere Vorteile:

- Man kann sich zu Beginn auf die eigentliche Problemstellung konzentrieren und ist automatisch viel offener bei der Strukturierung und Gestaltung der Architektur.
- Die Auswahl einer Technologie kann besser getroffen werden, da die Anforderungen an eine solche sehr konkret sind.
- Die beiden Modelle bleiben technologieneutral und können somit von mehreren Rollen verwendet werden.

Mock-Plattform (P1)

Das technologieneutrale und das Programmiermodell geben bereits eine klare Strukturierung des zu entwickelnden Systems vor. Basierend auf diesen Informationen werden auch die Entwicklerteams zusammengestellt, die für die Umsetzung von Subsystemen zuständig sind. Zwischen den einzelnen Komponenten in einem Subsystem und den Subsystemen untereinander bestehen Abhängigkeiten. Die Mock-Plattform wird dazu verwendet, Teile der Architektur als Mock oder Stub zur Verfügung zu stellen und ist für das Unit Testing essenziell. Dadurch wird die Abhängigkeit von der Implementierung Komponenten minimiert, da diese bereits im Vorfeld getestet werden können. Voraussetzung sind die klar definierten Schnittstellen. Details zu Mock-Objekten finden Sie in Abschnitt 5.6.3.

Vertikaler Prototyp (P1)

Durch die Ausführung von Unit Tests werden zwar die funktionalen Anforderungen getestet, die nicht funktionalen Aspekte werden jedoch größtenteils ignoriert. Ob die Architektur mit der definierten Technologie nun die vorgegebenen nicht funktionalen Anforderungen erfüllt, kann nur mit einem Prototypen getestet werden. In dieser Phase kommen vor allem die Performance- und Last-Tests zum Einsatz.

Architektur-Metamodell (P3)

Wenn in einem Unternehmen mehrere Projekte umgesetzt werden, kristallisieren sich nach einiger Zeit Gemeinsamkeiten zwischen den einzelnen Projekten heraus. Bei ähnlichen Problemlösungen der gleichen Domäne und Anforderungen lassen sich ähnliche Architekturen ableiten. Bei der Abbildung des Programmiermodells werden vermehrt die gleichen Schritte durchgeführt, und nicht selten werden dabei Teile des Sourcecodes einfach von einem Projekt in ein anderes kopiert. Architektur-Metamodelle fassen die Gemeinsamkeiten von unterschiedlichen Architekturen zusammen. Diese Metamodelle machen nur bei größeren Projekten Sinn, da es einen erheblichen Mehraufwand mit sich bringt, eine formale Architektur zu definieren.

Codegenerierung (P3)

Mit dem technologieneutralen Architektur-Modell und dem Technologie-Mapping wird genau definiert, welche Teile mit welcher Technologie umgesetzt werden. Müssen im Laufe der Iterationen Teile des Modells geändert werden, so muss dies bis auf Codeebene nachgezogen werden, was eine Fehlerquelle mit sich bringt. Durch Spezifikationen beim Technologie-Mapping können Teile des Codes generiert werden. Gute Beispiele für Codegenerierungen sind Java-Klassen aus XML-Schemadateien oder Konfiguriationsdateien für Frameworks.

Domänenspezifisches Programmiermodell (P3)

Software-Systeme sind oft für eine spezielle Domäne ausgelegt. Folglich wird auch viel Fachvokabular verwendet, das nicht eins zu eins in das Programmiermodell übernommen werden kann. Da das Programmiermodell sehr stark auf den Entwickler ausgelegt ist, kann dieses kaum von einem Domänenexperten verwendet werden. Durch die Einführung einer domänenspezifischen Sprache (DSL) können die Strukturen und das Verhalten in dieser beschrieben werden. Ausgehend von diesem Modell kann der darunterliegende Code generiert werden.

Modellgetriebene Architekturen (P3)

Architekturänderungen haben meistens zur Folge, dass viele Einzelteile betroffen sind, und auch die Entwickler direkt in deren Implementierung Änderungen vornehmen müssen. Der Weg vom Architekturmodell bis zur Umsetzung kann dabei sehr lang sein, und nicht selten gehen dabei Informationen verloren. Architekuränderungen sollten auf der Modellbasis gemacht werden und sich bis in die Implementierung durchziehen. Für diesen Zweck werden Regeln und Bedingungen beim Architektur-Metamodell eingeführt, die Auswirkungen auf den generierten Code haben.

Dieses Vorgehensmodell stellt einen möglichen Ansatz dar, wie Software-Architekturen entstehen können. Es wird Ansätze geben, bei denen nur ein Teil dieser Modelle verwendet wird. Wichtig ist, dass Architekturen in iterativen Schritten entstehen und dass die verschiedenen Modelle für die unterschiedlichen Aspekte der Architektur verwendet werden können. Neben den Modellen gibt es auch unterschiedliche Sichten auf eine Software-Architektur, die im nächsten Abschnitt beschrieben werden.

7.3 Sichten auf eine Software-Architektur

Beim Entwurf von Software-Architekturen müssen sehr viele Komponenten und Einflussfaktoren berücksichtigt werden. Viele sehen in einer Software-Architektur nur die Strukturierung des Systems und vergessen dabei völlig, auf das Verhalten der Komponenten zur Laufzeit zu achten, oder darauf, in welchem Kontext das System steht. Die Architektur dient in vielen Projekten als zentrale Drehscheibe. Folglich sind an der Software-Architektur auch viele unterschiedliche Personen beteiligt.

Unterschiedliche Sichten auf die Software-Lösung

Unterschiedliche Rollen, die im Rahmen der Entwicklung eines Software-Projekts involviert sind, haben unterschiedliche Sichtweisen auf die konkrete Umsetzung, wie es in Abbildung 7.2 dargestellt ist. Die Architektur wird dabei als Kommunikationsmedium zwischen den Rollen verwendet und dient in vielen Fällen als Basis für weitere Entscheidungen. Dabei ist es leicht vorstellbar, dass jede Rolle eine andere Sicht auf die Architektur hat und sich die Interpretationen zwischen den Beteiligten auch differenzieren. Um alle Faktoren einer Software-Architektur zu beschreiben, können mehrere Sichten verwendet werden.

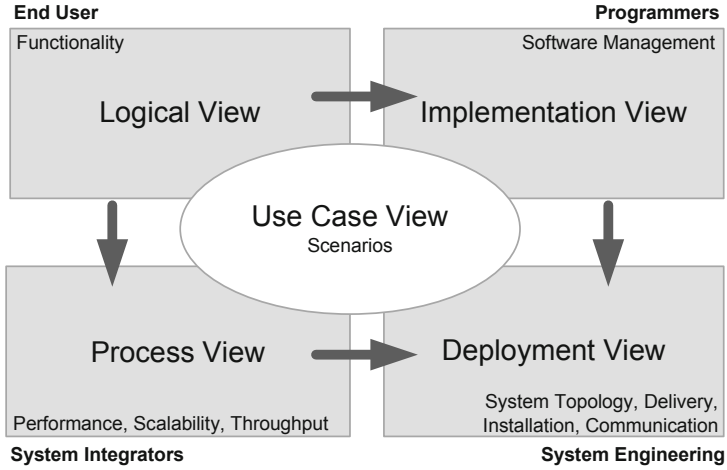


Abbildung 7.2
„4+1"-Model View of Architecture.

4+1 View

Eine Sicht (*view*) beschreibt einen Teilaspekt der Architektur und des Designs und die spezifischen Eigenschaften eines Systems. Die Elemente des *4+1 View Model of Architecture* lassen sich mit Diagrammen aus der UML-Diagramm-Familie übersichtlich darstellen [61]. Aufbau, Funktionsweise und Nutzen der verwendeten Diagramme in der *4+1 View Model of Architecture* aus der UML-2-Familie werden in Kapitel 6 vorgestellt.

Funktionalität

Die logische Sicht (*Logical View*) beschreibt funktionale Anforderungen der Endbenutzer (*end user*) unter Verwendung von Klassendiagrammen, Zustandsautomaten (*State-Machines*) oder Package Diagrams. Diese Sicht wird auch sehr oft als Kontextsicht bezeichnet.

Implementierung

Die Implementierungssicht (*Implementation View*) beschreibt den statischen Zusammenhang der Module unter Verwendung von Komponentendiagrammen und Klassendigrammen aus der Sicht des Programmierers. Die statische Struktur (Konfiguration) der Software-Komponenten, wie Sourcecode, Daten, Binaries und Dokumentation – kurz *Software-Management* – steht im Vordergrund. Gebräuchliche Artefakte in dieser Sicht sind unter anderem Komponenten, Packages, Subsysteme und Schnittstellen. Diese Sicht ist jene, die dem Sourcecode am nächsten kommt, und es lassen sich auch verwendete Architekturstile und Design-Patterns ablesen, da diese bereits strikte Vorgaben der Aufteilung vorgeben.

Laufzeit

Die Prozesssicht (*Process View*) legt den Fokus auf das Laufzeitverhalten im Rahmen der Systemintegration, wie beispielsweise eine Lastverteilung bei verteilten Anwendungen, Fehlertoleranz, Performance und Erweiterbarkeit. Typischerweise werden diese Eigenschaften eher den nicht funktionalen Anforderungen zugeordnet. In der Regel werden Komponenten aus der Implementierungssicht verwendet und es wird beschreiben wie diese Komponenten kommunizieren bzw. deren Zustände in bestimmten Arbeitsabläufen sind. Vor allem bei Systemen mit einem hohen Entkoppelungsgrad sind Laufzeitsichten von großer Bedeutung. Für die Modellierung in

Tabelle 7.1 *Übersicht der Sichten.*

Sicht	Was wird beschrieben	UML-Diagramme	Rollen
Anwendungsfall	Beschreibung der Funktionalität	Anwendungsfalldiagramm	Kunde, Architekt
Logische Sicht	Beschreibung der Funktionalität in Form von Objekten	Klassendiagramm, Zustandsautomaten, Paketdiagramme	Kunde, Architekt
Implementierung	Statisches Zusammenspiel der Komponenten	Komponentendiagramm	Architekt, Designer, Entwickler
Prozess	Zusammenspiel der Komponenten zur Laufzeit	Sequenzdiagramm, Kollaborationsdiagramm, Aktivitätsdiagramm	Architekt, Designer, Entwickler
Verteilung	Darstellung der Infrastruktur, in der das System laufen wird	Verteilungsdiagramm	Kunde, Architekt, Designer, Entwickler

UML können beispielsweise Sequenz-, Aktivitäts-, oder Kommunikationsdiagramme verwendet werden.

Deployment

Die Verteilungssicht (*Deployment View*) beschreibt die ausführbaren Applikationen unter Berücksichtigung der jeweiligen Plattformen und der für die Anwendung notwendigen Ressourcen. Bei komplexen Software-Systemen werden immer wieder andere Systeme (z. B. Warenwirtschaftssysteme, Lohnverrechnungssysteme etc.) integriert. Diese Systeme müssen nicht zwingend in der gleichen Umgebung laufen, wie die zu entwickelnde Software. In der Verteilungssicht wird auch angeführt, welche Protokolle für die Kommunikation der einzelnen Knoten verwendet werden, wie etwa HTTP, JMS, TCP oder ähnliche. Beispiele aus der UML-2-Familie sind die Deployment-Diagramme. Typische Kandidaten, die in dieser Sicht angeführt werden, sind Datenbanken, externe Systeme, Repositories sowie Middleware.

Zusammenführung durch Use-Cases

Die Anwendungsfallsicht *Use-Case View* dient als Ergänzung zu den bisherigen Sichten. Sie ist der gemeinsame Nenner, in dem die Anwendungsfälle und Aktivitäten beispielsweise als Szenarien abgebildet werden (Schnittstellenfunktionalität zwischen den anderen Sichten). Somit kann die Anwendungsfallsicht zu Verifikations- und Validierungszwecken verwendet werden. In UML bilden die Anwendungsfalldiagramme und entsprechende Beschreibungen sowie die Aktivitätsdiagramme die zentralen Darstellungsmöglichkeiten dieser Sicht.

Die *4+1 View Model of Architecture* hat sich bereits in vielen Software-Projekten bewährt und stellt sicherlich eine *Best-Practice* beim Entwurf von Software-Architekturen dar. Durch die unterschiedlichen Sichtweisen auf die Architektur können mit diesem Ansatz mehrere Rollen angesprochen werden. Der Einsatz von UML-Diagrammen ermöglicht eine einheitliche Modellierung der Architektur, und die einzelnen Modelle können elegant

miteinander verknüpft werden. Tabelle 7.1 fasst nochmals die Sichten und ihre Bedeutung zusammen.

7.4 Separation of Concerns

Edgser Dijkstra

Separation of Concerns (SoC) ist eine der elementaren Ideen, die vielen modernen architektonischen Konzepten der Software-Entwicklung zugrunde liegt. Der Begriff selbst wurde schon im Jahr 1974 von Edgser Dijkstra eingeführt [23]. Dijkstra bezieht sich in seinem Artikel zunächst auf eine Methodik, die der Wissenschaft eigen ist: ein komplexes System wird in seine Einzelheiten zerlegt und dann der Versuch gemacht, jedes Detail möglichst unabhängig von anderen Details zu verstehen und handzuhaben. Dijkstra schreibt:

> „*A scientific discipline separates a fraction of human knowledge from the rest*: we have to do so, because, compared with what could be known, we have very, very small heads.“[3]

Die einzelnen Disziplinen müssen also möglichst vom Rest des Wissens der Menschheit abgekoppelt werden, weil „unser Kopf zu klein ist“, um mit allem gleichzeitig umgehen zu können. Dijkstra beschreibt also ein methodisches Vorgehen, um auch sehr komplexe Systeme verstehen und manipulieren zu können, indem man dieses sinnvoll in kleineren Teilen beschreibt. Da ein Software-System aus vielen Komponenten besteht, macht man es sich selbst leichter, wenn man es in kleine, überschaubare Gruppen unterteilt. Dies wird nach dem nächsten Zitat aus demselben Artikel klarer:

> „But of course, any odd collection of scraps of knowledge and an arbitrary bunch of abilities, both of the proper amount, do not constitute a scientific discipline: *for the separation to be meaningful, we have also an internal and an external requirement*. The internal requirement is one of *coherence*: the knowledge must support the abilities and the abilities must enable us to improve the knowledge. The external requirement is one of what I usually call *a thin interface*; the more self-supporting such an intellectual sub-universe, the less detailed the knowledge that its practitioners need about other areas of human endeavour, the greater its viability.“

Was ist eine Disziplin?

Dijkstra verweist in diesem Zitat auf einige wesentliche Rahmenbedingungen, die für eine wissenschaftliche Disziplin notwendig sind:

[3] Hervorhebungen nicht im Original

- Eine beliebige Ansammlung von Fakten oder Wissen ist nicht ausreichend, um eine Disziplin zu formen.
- Es gibt *innere* und *äußere* Bedingungen, die erfüllt sein müssen.
- Zu den inneren Bedingung zählt *Kohärenz*, d. h. das System muss aus zusammengehörigem Wissen bestehen, das die Fähigkeiten des Systems beschreibt und die Möglichkeit bietet, Verbesserungen am System vorzunehmen.
- Zu den äußeren Bedingungen zählt eine *schlanke Schnittstelle* zu anderen Systemen.
- Das System soll möglichst abgeschlossen sein (d. h. möglichst wenig Abhängigkeiten nach außen haben).
- Je weniger Wissen ein Ingenieur über *andere* Systeme haben muss, um seinen Teil zu beherrschen, desto besser.

Software-Engineering

Mit diesen Bedingungen ist man natürlich schon mitten im Software-Engineering angelangt. Die Kohärenz-Bedingung ebenso wie die Forderung nach einem schlanken Interface kann aus heutiger Sicht auf Objekte und Software-Komponenten bezogen werden. Etwas technischer ausgedrückt versteht man unter SoC den Entwurf einer Software in Module oder Komponenten, die sich funktional möglichst wenig überlappen und über klare und schlanke Schnittstellen kommunizieren. Diesem Ziel versucht man auf verschiedenen Wegen näherzukommen, unter anderem durch:

- Objektorientierte Programmierung mit seinen Prinzipien von Kapselung, bewusstem Verstecken von Daten und definierten (public) Interfaces.
- Aspektorientierte Programmierung, d. h., Aspekte die in verschiedenen Objekten zum Einsatz kommen, werden ausgelagert (z. B. Logging, Security, Transaktionsmanagement). Benötigt ein Objekt einen oder mehrere derartiger Aspekte, so können diese dem Objekt flexibel hinzugefügt werden (siehe Abschnitt 9.5).
- Komponentenorientierte- sowie serviceorientierte Architekturen, d. h., Kapselung größerer Funktionsblöcke mit gegebenenfalls plattformunabhängigen Schnittstellen (siehe Kapitel 9).
- Schichtenarchitektur, d. h., das Trennen einer Anwendung in mehrere Schichten, die spezifische Funktionalität haben und die jeweils nur mit der nächst höheren oder niedrigeren kommunizieren.[4]

[4] `http://www.cs.jhu.edu/~scott/oose/lectures/design.shtml`

- Lockere Kopplung durch architektonische Maßnahmen, aber auch durch den Einsatz von Middleware (siehe Abschnitt 9.7).

Erwartete Vorteile

Welche Vorteile kann man sich nun aus einem SoC-Ansatz erwarten?[5] Die einzelnen Teile des Systems können leichter gewartet und weiterentwickelt werden: Jedes Modul beschränkt sich idealerweise auf eine bestimmte Funktionalität; diese kann komplex sein, aber Spezialisten können sich isoliert mit dieser konkreten Problematik auseinandersetzen. Auch können diese Teile besser getrennt voneinander entwickelt und getestet werden, und die Verwendbarkeit eines Moduls in Kombination mit verschiedenen anderen ist meist leichter möglich.

Beispiele

Die Idee der SoC ist zum Beispiel die Basis für Schichtenarchitektur und aspektorientierte Programmierung. Beide Konzepte werden in diesem Kapitel noch genauer diskutiert.

7.5 Schichtenarchitektur

Die Schichtenarchitektur zählt zu den beliebtesten Architektur-Stilen in der Software-Entwicklung. Jede Schicht (*Layer*) nimmt dabei eine klar definierte Rolle ein und bietet darüberliegenden Schichten eine Menge an Diensten an. Wie bei den Architektur-Stilen erwähnt, werden durch den Stil auch Regeln beschrieben, die eingehalten werden müssen. Bei der Schichtenarchitektur sind folgende Punkte zu beachten:

Regeln

- Eine Schicht verbirgt sowohl darunterliegende Schichten als auch die interne Komplexität. Untere Schichten konzentrieren sich meistens auf die Technik, während obere Schichten eher den Fokus auf die Benutzerschnittstelle legen.
- *Top-down-Kommunikation*, d. h., Komponenten der höheren Schicht verwenden Dienste der unteren Schicht und nicht umgekehrt.
- Komponenten innerhalb einer Schicht sind von ähnlichem Abstraktionsgrad (z. B. Komponenten in der Daten-Schicht konzentrieren sich um Persistenz).
- Das Design einer Schicht soll eine lose Kopplung zu anderen Schichten ermöglichen.
- Die Kommunikation zwischen den Schichten erfolgt über klar definierte Schnittstellen und Protokolle.

[5] Im Prinzip wird eine ähnliche Diskussion noch einmal in Abschnitt 8.6 geführt, wo Systemintegration und lockere Kopplung thematisiert werden.

Einfach und verständlich

Der Grund für den häufigen Einsatz der Schichtenarchitektur liegt in der Einfachheit und der klaren Verständlichkeit des Ansatzes. Eine einzelne Schicht kann als eigenständige Programmkomponente gesehen werden, ohne notwendigerweise die Gesamtarchitektur zu kennen. Kommunizieren die einzelnen Schichten über klar definierte Schnittstellen und dies im Top-down-Mechanismus, so wird nicht nur eine minimale Abhängigkeit zwischen den Schichten erzielt, sondern einzelne Schichten können im Bedarfsfall gegen eine andere Implementierung ausgetauscht werden, ohne dass das restliche System negativ beeinflusst wird.

Top-down

Bevor auf die unterschiedlichen Arten der Schichtenarchitektur eingegangen wird, soll zuerst ein exemplarischer Aufbau einer einzelnen Schicht betrachtet werden. Eine Schicht wird oft als Subsystem entworfen, das sich wiederum aus einer Menge von Teilsystemen (z. B. Services, Geschäftsprozesse etc.) zusammensetzt, wie in Abbildung 6.19 dargestellt ist. Um der darüberliegenden Schicht einen einfachen und komfortablen Zugriff auf die Services zu ermöglichen, würde sich zum Beispiel das Fassade-Pattern (siehe Abschnitt 8.4.1) eignen.

Horizontale Schichtenbildung

Die Schichtenarchitektur ist in ihrer Verwendung und Gestaltung sehr flexibel. Man kann unterschiedliche Formen und Strategien der Schichtenbildung unterscheiden. Die einfachste Form ist die horizontale Schichtenbildung. Dabei werden die Schichten horizontal gestapelt und jede horizontale Schicht hat eine Aufgabe, wie z. B. den Datenzugriff. Bei komplexeren Systemen kann eine Schicht in weitere Teile geteilt (partitioniert) werden, wobei jede Partition einen bestimmten Business-Fokus hat. Diese Partitionen werden oft in Form von Subsystemen umgesetzt.

Vertikale Schichten

Querschnitts-Funktionalitäten, die von allen Schichten in einem System benötigt werden, können in vertikalen Schichten abgebildet werden. Dabei hat eine vertikale Schicht vollen Zugriff auf alle angrenzenden horizontalen und umgekehrt.

Vorteile

Zusammengefasst bietet die Schichtenarchitektur folgende Vorteile:

- Einfaches, effizientes und verständliches Architekturmuster.
- Es besteht eine saubere Trennung der einzelnen Schichten. Dadurch kann auch ein verteiltes Arbeiten in Team durchgeführt werden.
- Schichten sind austauschbar und können auch auf verschiedenen Infrastrukturen betrieben werden. Beispielsweise können die Datenbank, die Services und die Komponenten für den Client auf unterschiedlichen Rechnern betrieben werden.
- Minimale Abhängigkeit zwischen den Schichten. Die Kommunikationswege der Schichten sind durch Schnittstellen klar definiert.
- Die Wartung und Erweiterbarkeit ist durch die klare Trennung der Schichten einfach.

Es sollten auch einige wenige Nachteile der Schichtenarchitektur bedacht werden:

Nachteile

- Wird die Top-down-Kommunikation bei einer Schichtenarchitektur streng eingehalten, kann eine triviale Operationen in einer Schicht oft nur aus einem „Durchreichen“ in die nächste Schicht bestehen. Das kann dazu führen, dass aus relativ einfachen Operationen komplexe Funktionen werden.[6]
- Mehr Schichten bedeutet mehr Komplexität und Implementierungsaufwand. Weniger Schichten bedeutet stärkere Kopplung und weniger Flexibilität. Hier muss abgewogen werden, wie viele Schichten wirklich notwendig sind, um die Anforderungen des Systems zu erfüllen.
- Änderungen an den Schnittstellen der unteren Schichten können Änderungen in den oberen Schichten und damit einen erhöhten Aufwand bei der Anpassung zur Folge haben.

In den folgenden Abschnitten werden verschiedene Ausprägungen (Stile) von Schichtenarchitekturen etwas detaillierter vorgestellt, konkret die 2-, 3- und 5-Schichtenarchitektur. Für welche Ausprägung man sich letztendlich entscheidet, wird von den konkreten Anforderungen abhängig sein. Die Schichtenarchitektur wird hier im Speziellen an verteilten Anwendungen (wie beispielsweise Webanwendungen) gezeigt, weil dort die Vorteile am klarsten hervortreten. Grundsätzlich kann Schichtenarchitektur aber auch bei nicht verteilten Anwendungen (z. B. einem Grafikprogramm) Sinn machen, weil dadurch die Wartbarkeit und die Erweiterbarkeit verbessert werden können.

Schichten nur in verteilten Systemen?

7.5.1 2-Schichtenarchitektur

Die 2-Schichtenarchitektur ist bei verteilten Systemen auch als Client/Server-Architektur bekannt. Der Client ist dann in der Regel ein Programm, das sowohl die GUI als auch die gesamte Applikationslogik beinhaltet. Der Server besteht meist aus einer relationalen Datenbank. Für eine Stammdatenverwaltung oder ähnliche Systeme war diese Architektur unter Umständen ausreichend und auch effizient. Durch die wachsende Komplexität in den einzelnen Domänen und die Anforderungen an verteilte Systeme ist es jedoch meist notwendig und auch sinnvoll, Teile des Applikationscodes auf mehrere Schichten aufzuteilen. Einzelne Schichten können auch auf einen Server ausgelagert werden.

Client/Server

[6] Beim Durchreichen von Informationen kommt dabei sehr häufig das *Delegation Pattern* (siehe Abschnitt 8.2.2) zum Einsatz.

7.5.2 3-Schichtenarchitektur

Logik vom Client lösen

Die 3-Schichtenarchitektur ist der bekannteste und beliebteste Stil, um ein System zu strukturieren. Der Wunsch den Code der Geschäftslogik vom Client zu lösen, führt dazu, dass eine neue Schicht (Logik- oder Business-Schicht) eingeführt wird, die diese Logik beinhaltet. In der Praxis gibt es mehrere Variationen, wie die Logik-Schicht auf den Client und den Server aufgeteilt ist.

Daten-Schicht

Auf der untersten Ebene der 3-Schichtenarchitektur befindet sich die *Daten-Schicht*, die eine Menge an Datenquellen zur Verfügung stellt. Typische Objekte in dieser Schicht sind relationale oder objektorientierte Datenbanken. In diesen Objekten werden die Daten von den Applikationen persistiert. Näheres zum Thema Persistenz kann in Abschnitt 9.4 nachgelesen werden.

Logik-Schicht

Aufbauend auf der Daten-Schicht befindet sich nun die neue *Logik-Schicht*, welche die Kernfunktionalität des Systems beinhaltet. Hier erfolgt die tatsächliche Abbildung der Problemdomäne in Form von Logik- und Serviceobjekten. Typische Funktionen, die in der Business-Schicht angeboten werden, sind die Verarbeitung und die Aufbereitung von Daten für die Präsentationsschicht. Die Logik-Schicht wird bei komplexen Systemen oft noch in eine *Datenzugriffs-Schicht (Data Access Layer)* und eine *Service-Schicht* unterteilt. Die Datenzugriffs-Schicht besteht aus einer Menge von Datenzugriffsobjekten (siehe Abschnitt 8.4.5), sogenannten *Data Access Objects (DAOs)*, die den Zugriff auf die darunterliegende Daten-Schicht abstrahieren. Diese DAOs werden von Servicekomponenten der Service-Schicht in Anspruch genommen, um Daten zu laden und zu speichern.

Präsentationsschicht

Jegliche Form der Interaktion mit dem Benutzer erfolgt in der *Präsentationsschicht*. Komponenten in dieser Schicht beinhalten auch eine gewisse Logik für die Verarbeitung von Ereignissen, etwa die Logik, die auf das Betätigen eines Buttons oder eine Tastenkombination reagiert. Die Logik sollte jedoch hauptsächlich auf GUI-Komponenten bezogen sein und nicht darauf, Geschäftslogik zu implementieren. In Anschnitt 9.6.3 werden Methoden zur Kommunikation zwischen der Präsentationsschicht und der Logik-Schicht vorgestellt. In dieser Schicht findet sich beispielsweise die Logik, die notwendig ist, einen Web-Client (z. B. mit JSF, Wicket, Javascript) oder einen Fat-Client (z. B. mit Java Swing) darzustellen.

7.5.3 5-Schichtenarchitektur

Data Mapper

Durch die Trennung der Logik-Schicht in die Datenzugriffs-Schicht und Service-Schicht wurde eine weitere Entkoppelung von Services zu den Daten erzielt. Jedoch sind die Implementierungen der DAOs sehr stark mit der Struktur der zu verwendeten Datenquelle verknüpft. Eine Änderung der Struktur in der Datenquelle (etwa das Hinzufügen einer neuen Spalte in einer Tabelle) bewirkt auch eine Änderung der DAO-Implementierung. Um

Domänenobjekte mit externen Datenquellen in Verbindung zu setzen, die anschließend über DAOs geladen und gespeichert werden können, wird ein Framework, das *Datenmapping* durchführt, verwendet. Dieses Datenmapping löst Struktur-Information weitgehend aus der DAO-Implementierung. In Abschnitt 9.4.8 wird beschrieben, wie ein objektrelationales Mapping (O/R Mapping) funktioniert.

Prozess-Schicht

Die nächste neue Schicht ist die sogenannte *Prozess-Schicht*. Diese liegt zwischen der Logik-Schicht und der Präsentationsschicht. Ziel ist es, die Services so atomar wie möglich und damit wiederverwendbar zu implementieren. Die Prozess-Schicht kümmert sich dann um Abläufe (Prozesse) höherer Granularität und wird daher häufig unter Verwendung einer Prozess-Engine[7] umgesetzt. Die atomaren Services aus der Logik-Schicht werden (häufig mithilfe grafischer Tools) zu einem Prozess kombiniert, der dann in einer eigenen Prozess-Sprache (z. B. BPEL) vorliegt. Da sich die Ablauflogik nicht mehr im Code befindet, können Prozesse leichter verändert und an neue Bedürfnisse angepasst werden.

7.6 Serviceorientierte Architekturen

7.6.1 Ein neues Architekturparadigma

Management und IT

In einem Software-Projekt sind unterschiedlichen Rollen involviert, wobei jede dieser Rollen eine andere Betrachtungsweise auf das Software-System hat. Sehr oft kommt es dabei zu Unklarheiten (beispielsweise zwischen Management und IT-Verantwortlichen). Für die IT sind Technik, Architektur und wie Problemlösungen implementiert werden entscheidend. Das Management hingegen möchte eine möglichst schnelle und kosteneffiziente Lösung und ist an technischen Details weniger interessiert. Serviceorientierte Architekturen (SOA) sollen helfen, diese beiden Sichtweisen miteinander zu verbinden.

SOA als Architekturparadigma

Serviceorientierte Architekturen sind eine Zusammenfassung von Design-Patterns und Architekturansätzen, die sich in den letzten 20 Jahren entwickelt haben. Eine SOA ist keine *Technologie* oder ein *Produkt* sondern ein *Architekturparadigma*, das auf mehrere Arten und in unterschiedlichen Technologien umgesetzt werden kann. Die bekannteste und weitverbreitetste Art eine SOA zu realisieren, ist jene mit Web Services, siehe Abschnitt 9.7.

[7] Prozess-Engines interpretieren Prozesse, die mit grafischen Tools modelliert werden können.

Service als zentraler Baustein

Zentraler Bestandteil in einer SOA ist der *Service*, wie auch in Abschnitt 9.1 beschrieben. Wesentliche Aspekte von Services sollen hier an einem einfachen Beispiel eingeführt werden:

Beispiel

Eine Person begibt sich in ein Möbelhaus und sucht die Wohnzimmer-Abteilung auf, um ein Sofa zu kaufen. Drei Modelle kommen in die engere Wahl. Ein Kundenberater wird über den Informationsschalter gesucht und erklärt die wesentlichen Unterschiede, Merkmale sowie Vor- und Nachteile der ausgesuchten Modelle. Mithilfe dieser Informationen trifft der Kunde die Kaufentscheidung. Zusammen mit dem Mitarbeiter wird die Bestellung des Sofas vorgenommen. Es werden noch weitere Randbedingungen wie Lieferung und Aufstellung geklärt. Das Szenario soll an dieser Stelle beendet werden, um die *Elemente einer serviceorientierten Architektur* zu identifizieren:

- Der Kunde nimmt einen *Service* (Beratung) des Möbelhauses (*Anbieter*) in Anspruch.
- Der Kunde hat klare *Anforderungen*, nämlich den Kauf eines geeigneten Sofas.
- Zur Beratung sucht er einen Kundenberater über den Informationsschalter (*Service-Registry*).
- Das Möbelhaus hat den Kunden ausführlich beraten und somit den Service *durchgeführt*.
- Das *Kommunikationsmodell* zwischen dem Kunden und dem Möbelhaus war klar definiert (über den Kundenberater, mündlich, in deutscher Sprache).
- Details zu Berechnung der Preise, Auslieferung, Lagerhaltung etc. werden vor dem Kunden „versteckt“ und nur die für den Kunden wesentlichen Informationen werden kommuniziert.
- Das *Ergebnis* für den Kunden ist der Auftrag und die Bestellung.

Es kann also ein *Service Consumer* (Kunde) und ein *Service Provider* (Möbelhaus) aus obigem Anwendungsfall abgeleitet werden, siehe Abbildung 7.3. Ein Service Consumer möchte von einem bestimmten Service Provider Dienste in Anspruch nehmen. Dieser Service benötigt gegebenenfalls weitere Informationen (im obigen Fall waren es die Anforderungen sowie die Rahmenbedingungen für die Lieferung). Nach der Ausführung liefert der Service Provider dem Service Consumer das Ergebnis.

Zur Suche eines bestimmten Services wird außerdem häufig eine Registry in Anspruch genommen. Der Service (in Form von Kundenberatern) steht mehrfach zur Verfügung, und ein geeigneter wird von der Registry ausgewählt (beispielsweise ein Kundenberater, der Experte im gewünschten

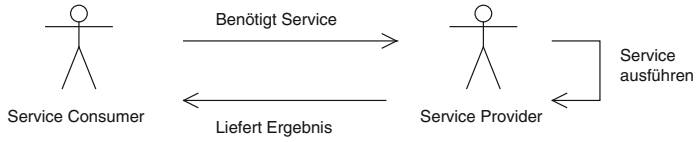


Abbildung 7.3 *Abstraktes Zusammenspiel von Service Provider und Service Consumer.*

Bereich ist und keinen anderen Kunden hat). Dies führt zum weiter unten genauer ausgeführten Grundmodell der SOA und ist in Abbildung 7.4 illustriert.

Technische Wurzeln der SOA

Die technischen Wurzeln der serviceorientierten Architekturen liegen in den Entwicklungen der Programmiersprachen und -plattformen, den verteilten Systemen und im Business Computing. Unter *Business Computing* werden Software-Systeme aus dem Bereich *Enterprise Resource Planing (ERP)* oder *Customer Relation Management (CRM)* zusammengefasst, die von Unternehmen für die Abwicklung ihrer Geschäftsprozesse eingesetzt werden.

Gerade im Business Computing haben viele Unternehmen für ihre unterschiedlichen Geschäftsbereiche Software-Lösungen verschiedenster Hersteller im Einsatz. Dazu kommt eine Vielzahl von Eigenentwicklungen, die zum Teil ebenfalls in unterschiedlichen Plattformen implementiert wurden, was oft zu einer Ansammlung verschiedener *Insellösungen*[8] führt. Durch die mangelnde Kommunikation zwischen Systemen kommt es vor, dass gleichartige Information (z. B. Kundendaten) mehrfach in unterschiedlichen Systemen vorliegen und auch gepflegt werden müssen. Dies ist natürlich kein optimaler Zustand. Dazu kommt, dass heute auch immer häufiger Kooperation zwischen Firmen mit sehr heterogenen IT-Systemen gefordert wird. Daher bietet sich gerade im Bereich des Business Computing der Einsatz von SOA an [60]:

> „Because of the closeness of services to concrete business functionality, service-orientation has the potential to become the first paradigm that truly brings technology and business together on a level where people from both sides can equally understand and talk about the underlying concepts“ *Dirk Krafzig et. al [60]*

Daraus folgt die Anforderung, Daten sowie Services über Systemgrenzen hinweg nutzen zu können.

SOA-Grundmodell

In Abbildung 7.4 ist der SOA-Grundgedanke dargestellt. Es gibt einen *Service Provider*, der einen *Service* anbietet. Jeder Service verfügt über eine neutrale *Beschreibung*, die in einem *Service-Repository* abgelegt wird.

[8] Eine Insellösung ist eine Software, die sich auf einen speziellen Bereich fokussiert und relativ stark von den restlichen Systemen im Unternehmen abgekapselt ist.

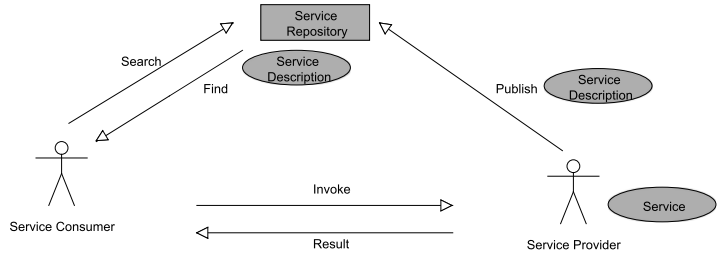


Abbildung 7.4 *Der SOA-Grundgedanke.*

Ein *Service Consumer* sucht in einem Service-Repository nach einem Service, der seinen Bedürfnissen entspricht. Das Service-Repository liefert dem Service Consumer die Service-Beschreibung des ausgewählten Service, damit dieser den Service bei sich einbinden kann. Zur Laufzeit wird der Service des Service Providers in Anspruch genommen.

Plattform-unabhängigkeit

Der Service selbst sowie die Service-Beschreibung und die Transportprotokolle sind in einer plattformunabhängigen (neutralen) Weise gehalten, sodass die Verwendung des Services keine bestimmte IT-Infrastruktur oder Software-Plattform voraussetzt. Es werden hier beispielsweise häufig Standards auf XML-Basis wie SOAP für Webservices, WSDL als Service-Beschreibung und http als Transportprotokoll eingesetzt. Service Provider sowie die Service-Nutzer können dann in einer beliebigen Technologie implementiert werden.

7.6.2 SOA und Schichten

In Abschnitt 7.5 wurde erläutert, dass die Schichtenarchitektur ein weitverbreiterter Ansatz für die Strukturierung von Software Systemen ist. Eine Schichtenbildung findet sich auch bei serviceorientierten Architekturen wieder, wobei die Bezeichnung und Aufgaben der Schichten anders verteilt sind, als bei der herkömmlichen Schichtenarchitektur, (siehe Abbildung 7.5). In diesem Abschnitt sollen die Schichten, die bei einer SOA zum Einsatz kommen, etwas näher beschrieben werden. Die einzelnen Schichten in einer SOA sind in vielen Fällen verteilt auf viele unterschiedliche Rechner und sogar unterschiedliche Plattformen.

Horizontal und Vertikal

In Abschnitt 7.4 wurden die Ideen von *Separation of Concern* erläutert, die Trennung von Zuständigkeiten. Security, Logging und andere Dienste werden in allen Schichten der SOA in unterschiedlicher Ausprägung benötigt. Um mehrfache Implementation ähnlicher Funktionalität zu vermeiden, lagert man solche Funktionen auch in einer SOA in vertikale Schichten aus. In der SOA werden daher sowohl horizontale Schichten als auch vertikale Schichten verwendet. Zum einen können die horizontalen Schichten die Dienste der vertikalen in Anspruch nehmen, zum anderen können aber auch die vertikalen Schichten die Dienste der horizontalen Schichten

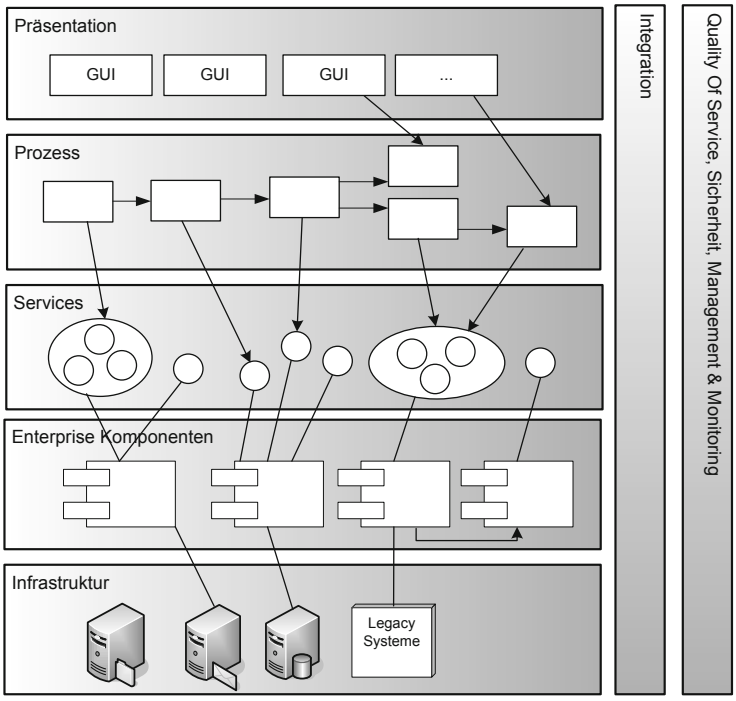


Abbildung 7.5
Schichten einer SOA.

verwenden. Die Kommunikationsrichtung muss für jeden Anwendungsfall definiert werden, und wird beim Entwurf der SOA genau festgelegt.

Infrastruktur-System-Schicht

In der *Infrastruktur-Schicht* werden alle bestehenden Systeme eines Unternehmens zusammengefasst. Solche Systeme werden oft auch als *Legacy-Systeme* bezeichnet. Als Legacy-System wird ein historisch gewachsenes System bezeichnet, das sich im Laufe der Zeit im Unternehmen etabliert hat. Es besteht aus unterschiedlichen Anwendungen verschiedener Hersteller und auch aus Eigenentwicklungen. Auf bereits etablierte und teure Software-Systeme soll meist auch in Zukunft nicht verzichtetet werden.

Enterprise-Komponenten-Schicht

Enterprise-Komponenten sind zum einen die adaptierten Schnittstellen der Legacy-Systeme, die es anderen Komponenten ermöglichen auf die Legacy-Systeme zuzugreifen. Aber auch neu entwickelte Systeme, die *komponentenorientiert* entwickelt werden, befinden sich in dieser Schicht. Enterprise-Komponenten implementieren die Geschäftslogik und stellen sicher, dass *Quality of Service* und vereinbarte *Service Level Agreements* eingehalten werden. In dieser Schicht werden auch häufig Komponenten-Container und/oder Applikations-Server eingesetzt. Themen wie Skalierbarkeit, Performance, Load Balancing oder Verfügbarkeit sind Schwerpunkte dieser Schicht.

Service-Schicht

Die *Service-Schicht* ist aus Sicht der SOA die Kernschicht. In dieser befinden sich die Services. Diese Services können Basisfunktionalitäten (ato-

mare Services) anbieten oder bereits komplexere Funktionen, die sich aus mehreren (atomaren) Services zusammensetzen. Man spricht dann von *Composite Services*. In dieser Schicht befindet sich auch die *Service-Registry*, ein Verzeichnis, in dem alle Services registriert sind. Sowohl atomare als auch Composite Services können von anderen Services als auch von darüberliegenden Geschäftsprozessen oder anderen Anwendungen verwendet werden. Daher ist der Aufbau und die Wiederverwendbarkeit der Services in dieser Schicht von großer Bedeutung.

Geschäftsprozesse

In der *Prozess Schicht* werden die Geschäftsprozesse abgebildet und durch einen geeigneten Prozess-Engine ausgeführt. Innerhalb des Prozesses können Services aus der Service-Schicht aufgerufen werden, aber auch Interaktionen mit dem Benutzer erfolgen. Der Benutzer interagiert über die zur Verfügung gestellten GUIs mit dem Prozess.

Präsentationsschicht

In der *Präsentationsschicht* befinden sich die GUI-Komponenten, die jegliche Art der Interaktion zwischen dem Benutzer und dem Backendsystem ermöglichen.

Integrationsschicht

Die *Integrationsschicht* wird sehr oft als *Enterprise Service Bus (ESB)* beschrieben, der die Integration von unterschiedlichen Systemen ermöglicht und darüber hinaus eine Menge von Diensten, wie intelligentes Routing, Protokoll-Transformierung und vieles mehr anbietet. In Abschnitt 9.7.2 werden ein Überblick über wesentliche Middleware-Komponenten gegeben und die Konzepte eines ESB beschrieben.

Quality-of-Service-Schicht

In der *Quality-of-Service-Schicht* sind Komponenten angesiedelt, die Dienste für Security, Performance und Verfügbarkeit anbieten. Weiterhin befinden sich auch nützliche Komponenten für die gesamte Überwachung und Steuerung der SOA-Plattform in dieser Schicht.

7.6.3 SOA und Geschäftsprozesse

Definition

Ein *Geschäftsprozess* ist ein strukturierter Ablauf von Aktivitäten mit definiertem Start und Ende. Der Prozess hat einen Input und einen Output. Typische Beispiele für Geschäftsprozesse sind etwa die Auftragsabwicklung, das Einstellen eines Mitarbeiters, die Beschaffung von Waren, Freigabeprozesse oder Urlaubsanträge.

Bei einem Urlaubsantrag muss zuerst der Antrag vom Mitarbeiter ausgefüllt und anschließend dem Vorgesetzten weitergeleitet werden. Der Vorgesetzte prüft den Antrag und genehmigt oder verweigert den Urlaubsantritt. Zum strukturierten Ablauf zählen etwa auch die möglichen Verzweigungen, die ein Prozess einnehmen kann. Weiterhin benötigt jede Aktivität einen Input und produziert einen Output. Im Falle des Urlaubsantrages wird ein Antrag (Input) vom Mitarbeiter ausgefüllt und der fertige Antrag (Output) wird weitergeleitet. Gerade Geschäftsprozesse sind eine wesentliche

Komponente, die im Unternehmen ständig optimiert und angepasst werden muss.

Agile IT-Infrastruktur und Geschäftsprozesse

Eines der Hauptziele einer SOA ist die Verbindung von Fachbereich und IT. Ein häufig anzutreffendes Thema in heutigen Software-Projekten ist, dass die vorhandene IT-Infrastruktur zu starr ist. Damit Unternehmen am Markt überleben, müssen sie sich aber sehr rasch an die gegebenen Marktbedingungen anpassen, was sehr oft auch eine Änderung der eingesetzten Systeme mit sich bringt. Das Management und der Fachbereich in Unternehmen verlangen daher nach immer mehr Flexibilität bei den Systemen.

Automatisierte Geschäftsprozesse und Process Engines

Automatisierte Geschäftsprozesse die mithilfe von *Process Engines* und Geschäftsprozess-Sprachen wie BPEL und entsprechenden grafischen Werkzeugen implementiert werden, können helfen *Agilität* in das Unternehmen zu bringen. Im Zusammenspiel von serviceorientierter Architektur und automatisierten Geschäftsprozessen ist ein rasches Anpassen der Systeme möglich. Die neu entwickelten Anwendungen entstehen auf einer höheren Ebene als bisher. Das Ergebnis sind *prozessorientierte Applikationen*. Sie zeichnen sich durch das von Geschäftsprozessen gesteuerte Zusammenspiel von Services aus und sind in ihrer Struktur nicht mehr so starr. Das ist auch darauf zurückzuführen, dass der Ablauf des Prozesses nicht mehr im Applikations-Code versteckt ist sondern relativ leicht angepasst werden kann. Abschnitt 9.7.1 beschreibt in diesem Zusammenhang den Einsatz von sogenannten Geschäftsprozess-Werkzeugen.

7.7 Ereignisgetriebene Architektur

Events

Zuletzt soll noch der Architekturstil der *Event Driven Architecture (EDA)* kurz vorgestellt werden. Unter einem Ereignis (*Event*) versteht man eine *signifikante Zustandsänderung* in einem System (siehe auch Abbildung 7.6). Geht z. B. eine neue Bestellung eines Kunden ein, so kann dies als Ereignis im entsprechenden IT-System interpretiert werden. Landet ein Flugzeug auf einem Flughafen, so ist vermutlich eine Vielzahl an IT-Systemen betroffen, und verschiedene Events werden ausgelöst, z. B. „Flugzeug erreicht Kontrollbereich der Flugsicherung“, „Anfrage nach Landeerlaubnis“, … „Flugzeug ist gelandet“, „Passagiere sind ausgestiegen“ usw.

Message

Ein solches Event (*Ereignis*) kann in einem IT-System durch eine Nachricht (*Message*) ausgedrückt und kommuniziert werden. Bei dieser Nachricht handelt es sich um ein Datenpaket, das die notwendige Information beinhaltet, die das Ereignis beschreibt (häufig im XML-Format ausgedrückt)[9].

[9] Möchte man ereignisgetriebene Architekturen umsetzen, so ist die Granularität der Events und der darauf aufbauenden Nachrichten eher fein zu halten (wie in den Beispielen in diesem Abschnitt).

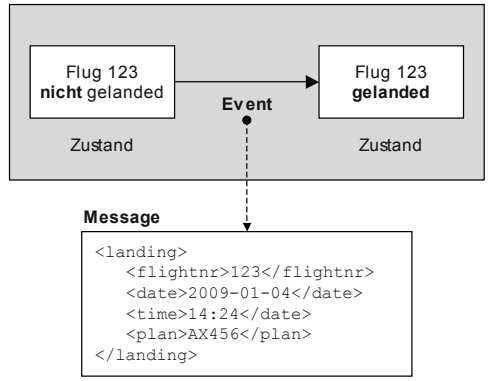


Abbildung 7.6 *Beispiel einer Event Message: Der Zustandsübergang vom ersten zum zweiten Zustand stellt ein Ereignis (Event) dar und kann durch eine Nachricht (Message) ausgedrückt werden, im Beispiel ist eine XML-Nachricht angedeutet.*

Um das obige Beispiel mit dem Flugzeug aufzugreifen, könnte die Nachricht für das Ereignis „Flugzeug ist gelandet“ folgende Daten beinhalten (siehe Abbildung 7.6):

- Flugplan,
- Zeit/Datum,
- Flugnummer,
- Kennung der Landebahn.

Integrations-Muster und Middleware

Um diese Nachrichten zu erstellen, zu versenden und zu verarbeiten, kann eine entsprechende Kommunikations-Middleware eingesetzt und auf passende *Integrationsmuster* zurückgegriffen werden. Integrationsmuster werden in Abschnitt 8.6 im Detail vorgestellt. In diesem Abschnitt werden Integrations-Muster und damit auch nachrichtenorientierte Integration beschrieben. In Abschnitt 9.7.2 wird ein Überblick über Middleware-Systeme gegeben, die z. B. auch in EDA-Szenarien zum Einsatz kommen können.

Event Message

Bei Nachrichten, wie im obigen Beispiel, spricht man auch von *Event Messages*, die im Gegensatz zu *Command Messages* zu sehen sind. Betrachtet man den Inhalt der Nachrichten in den beiden Beispielen, so stellt man fest, dass diese Nachrichten nur jene Daten enthalten, die das entsprechende Ereignis auszeichnen. Man findet aber keine weiteren Informationen darüber, was aus diesem Event zu folgern ist, bzw. welche Verarbeitungsschritte sich daraus ableiten.

Command Message

Bei *Command Messages* hingegen wird ebenfalls eine Nachricht versandt, aber es steht explizit eine *Anweisung* oder eine *Anfrage* im Vordergrund. Eine Nachricht, die an einen Kreditprüfungs-Service des Kreditkartenunternehmens gesandt wird, um die Kreditwürdigkeit des Kunden zu prüfen, wäre eine *Command Message*; ebenso eine Anfrage des IT-Systems einer

Fluglinie an ein Service der Flugsicherung, ob ein bestimmtes Flugzeug schon gelandet ist[10].

Architekturstil

Unter einer ereignisgetriebenen Architektur (*event driven Architecture, EDA*) [40] versteht man nun – vereinfacht gesagt – eine Architektur, bei der Ereignisse im Vordergrund der Betrachtung stehen. Das IT-System wird in einer solchen Architektur als eine Summe von Teil-Systemen betrachtet, die die Umgebung *beobachten*, d. h. über Änderungen in der Umgebung durch Event Messages informiert werden. Je nach Art der Änderung entscheidet das jeweilige Teil-System, welche weiteren Schritte zu treffen sind.

Umkehrung von Command and Control

Man kann sich das als Umkehrung des *Command-and-Control*-Musters vorstellen. Nicht das System, das eine Zustandsänderung wahrnimmt, löst explizit Aktionen in anderen Systemen aus, sondern versendet diese Tatsache in Form einer Event Message (meist an einen Message Broker). Alle Systeme, die von einer bestimmten Zustandsänderung (z. B. einer neuen Bestellung) betroffen sind, registrieren sich beim Broker für Nachrichten des entsprechenden Typs und werden dann von diesem benachrichtigt.

Eine EDA erlaubt also eine sehr elegante Entkopplung verschiedener (verteilter) Systeme. Einerseits ergibt sich eine Hol- und keine Bringschuld: Systeme müssen sich aktiv für die Events in Form von Messages interessieren, die sie für ihre Arbeit benötigen. Andererseits müssen Systeme, die entsprechende Messages versenden, nicht wissen, welche Zielsysteme an einer bestimmten Message interessiert sind. Auch werden ereignisgetriebene Systeme asynchron angelegt, was zu einer zusätzlichen Entkopplung der betroffenen Systeme führt.

EDA-Beispiel

Angenommen, man möchte einen Webshop implementieren. Dieses Shop besteht aus einer Reihe von Teilsystemen, z. B.:

- Web-Anwendung, mit der ein Kunde interagiert, z. B. Produkte suchen und bestellen.
- Webservice für andere Firmen, um Bestellungen aufzugeben.
- Customer Relationship Management (CRM) System.
- Enterprise Resource Planning (ERP) System für Abrechnung und Lagerhaltung.
- Kreditprüfungs-/Kundenprüfungssystem (entscheidet, ob an einen bestimmten Kunden geliefert wird).

[10] Eine Command Message könnte technisch gesehen z. B. ein Request/Reply-Aufruf eines SOAP-Services sein; das Nachrichtenformat wäre dann eine SOAP-Nachricht, die z. B. über SMTP verschickt wird.

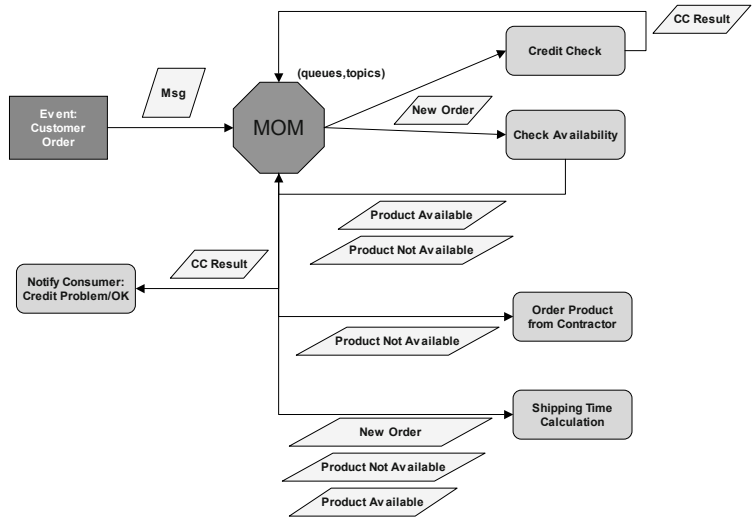


Abbildung 7.7 *Beispiel für eine ereignisgetriebene Architektur.*

Implementierung mit command and control

In einem konventionellen *command-and-control*-orientierten Ansatz könnte man das z. B. mit einer zentralen Applikation oder einer Prozess-Engine implementieren: Das Websystem oder die Webservice-Schnittstelle benachrichtigt diese Anwendung, und diese Anwendung kümmert sich um den weiteren Ablauf, überprüft also z. B. die Kreditwürdigkeit, stößt das Buchungssystem an, fragt den Lagerstand ab, berechnet die Lieferzeit und benachrichtigt den Kunden über den Status seiner Bestellung, initialisiert dann die Versendung usw. Der Nachteil dieses Ansatzes ist, dass alle Systeme relativ viel voneinander wissen müssen – sie sind ziemlich hart miteinander *verdrahtet*.

Implementierung mit EDA

Abbildung 7.7 deutet die Implementation im EDA-Stil an: Ein System, z. B. die Webanwendung oder das Webservice, sendet die Nachricht, dass eine neue Bestellung eingegangen ist. Sehr wichtig an dieser Stelle ist, dass diese Nachricht nicht an ein bestimmtes Zielsystem gerichtet ist, sondern an eine Topic oder Queue des Message Brokers[11]. Der Ersteller der Nachricht weiß also an dieser Stelle nicht, wer diese Nachricht konsumieren wird.

Verschiedene andere Systeme, z. B. *Credit Check* oder *Check Availability*, haben sich beim Broker registriert (Publish/Subscribe Pattern), dass sie an dieser Nachricht interessiert sind, und bekommen sie folglich zugestellt. Das Ergebnis der Kreditprüfung wird wieder als Nachricht an den Broker gesendet, und hier gilt dasselbe: Die *Credit-Check*-Komponente weiß nicht, wer an dem Ergebnis konkret interessiert ist. In dieser Weise kann man

[11] Im Realfall wird der Mechanismus der Verteilung vermutlich etwas komplexer sein; man wird vielleicht zusätzlich einen Enterprise Service Bus haben, der verschiedene Systeme integriert und zusätzlich ein komplexes Routing, Filtering oder Complex-Event-Processing usw. anbietet.

das Beispiel fortsetzen: Am Ende steht z. B. die *Shipping-Time-Calculation*-Komponente: diese registriert sich für Messages vom Typ *New Order*, *Product not available*, *Product available*, um aus diesen zusammengefassten Daten die Lieferzeit zu berechnen.

Vor- und Nachteile

Am Beispiel in Abbildung 7.7 kann man erkennen, wie stark die Entkopplung der Komponenten durch diesen Architekturansatz ist. Es ist beispielsweise jederzeit leicht möglich, die *Credit-Check*-Komponente aus Wartungsgründen durch eine andere Komponente zu ersetzen: man registriert die neue Komponente am Broker und entfernt die alte; die anderen Komponenten werden davon nicht berührt. Dasselbe trifft zu, wenn z. B. eine bestimmte Komponente mit der Last der Anfragen nicht mehr zurechtkommt, z. B. die *Check-Availability*-Komponente: In diesem Fall kann man recht einfach ein *Load Balancing* implementieren, indem man eine zweite identische Komponente auf einem anderen Server startet und am Broker registriert. Dieser kann dann einfach so konfiguriert werden, dass er eine *New-Order*-Nachricht immer nur einer der beiden Komponenten zukommen lässt. Auch hier gilt: die anderen Systeme bleiben von dieser Änderung unberührt.

Die Vorteile dieses Ansatzes sind klar: (a) hohe Flexibilität, (b) gute Entkopplung, (c) Asynchronität und (d) leichte Verteilbarkeit und Skalierbarkeit. Durch die starke Entkopplung und die Kommunikation über Event Messages müssen auch nur wenig Annahmen gemacht werden, wie Systeme kommunizieren und was für Fähigkeiten einzelne Systeme haben. Dies erleichtert eine Integration sehr heterogener Systeme.

Nachteilig an dieser Architektur ist jedoch, dass der Zustand des Systems nicht immer ganz leicht zu ermitteln ist. Verwendet man einen *command-and-control*-Ansatz, so hat man eine zentrale Stelle, die über den Status des Systems und aller Prozesse informiert ist. Bei der Planung und Implementation einer EDA muss daher immer sehr *defensiv* vorgegangen werden; es müssen mögliche Fehlerfälle, nicht vorhandene Komponenten sowie eine vernünftige Überwachung des gesamten Systems mitgeplant werden. Es gibt hier verschiedene Best-Practices, auf die an dieser Stelle verwiesen werden muss [41].

EDA und andere Paradigmen

EDA ist auch nicht unbedingt als *Gegensatz* zu anderen Architekturstilen zu sehen. EDA und andere Architekturen, z. B. die in Abschnitt 7.6 beschriebene serviceorientierte Architektur ergänzen einander sehr gut. Jack van Hoof beschreibt beispielsweise das mögliche Zusammenspiel von SOA und EDA [42]. Man wird im Realfall daher häufig eine Mischung dieser Architekturstile finden.

7.8 Zusammenfassung

Architektur und Sichten

Heutige Software-Systeme sind in der Praxis derart komplex, dass Software-Unternehmen in der Planung der technischen Entwicklung Ansätze benötigen, um leistungsfähige und wartbare Strukturen zu erzeugen bzw. die Eigenschaften unterschiedlicher Ansätze vor einer breiten Umsetzung zu evaluieren. Software-Architektur hat als Forschungs- und Entwicklungsfeld in den letzten Jahren einen starken Stellenwert gewonnen und stellt mehr als nur eine Methode zur Dokumentation dar. Software-Architektur ist ein zentrales Kommunikationsmedium zwischen den Projektbeteiligten. Die *4+1 Model View of Architecture* unterteilt die Software-Architektur in fünf Sichten: je eine logische, Implementierungs-, Prozess-, Verteilungs- und Anwendungsfallsicht. Damit können alle relevanten Bereiche einer Software-Architektur aus den verschiedenen Betrachtungsweisen beschrieben werden.

Technologie-Architekturen

Gute Software-Architekturen entstehen in kleinen iterativen Schritten. Auch heute werden viele Architekturentscheidungen immer noch aufgrund der aktuell verwendeten Technologie entschieden. Dies hat zur Folge, dass viele Software-Architekturen zu starr und unflexibel aufgesetzt werden. Dabei werden oft funktionale und nicht funktionale Anforderungen gar nicht beachtet, wenn diese mit der einzusetzenden Technologie nicht umsetzbar sind. Daher gilt als Grundregel, zuerst die Anforderungen zu verstehen und diese in eine technologieneutrale Architektur abzubilden. Erst im nächsten Schritt macht es Sinn, verschiedene Technologien in Betracht zu ziehen und jene auswählen, die eine technische Implementierung der Architektur erlauben.

Separation of Concerns

Im Zuge der Entwurfs- und Designphase entstehen Software-Komponenten, die dazu beitragen, funktionale und nicht funktionalen Anforderungen des Gesamtsystems abzudecken. Separation of Concerns ist ein Vorgehen, bei dem die Software in mehrere Komponenten zerlegt wird, die funktional möglichst gut voneinander getrennt sind und daher über klare und schlanke Schnittstellen kommunizieren können. Dabei kommen unterschiedliche Techniken wie Objektorientierung, aspektorientierte Programmierung oder etwa das Prinzip der losen Koppelung von Komponenten zum Einsatz.

Architekturstil

Nachdem die einzelnen Komponenten entworfen wurden, müssen diese wieder zu einem Gesamtbild zusammengeführt werden. Diese Zusammenführung zu einem Ganzen wird durch den eingesetzten Architekturstil beschrieben. Dabei gibt der Architekturstil Randbedingungen und Regeln vor, die bei der Umsetzung einzuhalten sind. Bekannte Architekturstile sind die Schichten- (Layered) Architektur, serviceorientierte Architekturen oder ereignisgetriebene Architekturen. Abhängig von den Anforderungen an das System wird die Wahl des Stils getroffen. Der beliebteste Architekturstil ist die Schichtenbildung, bei der das System in einzelne Schichten zerlegt wird. Jede Schicht nimmt dabei eine bestimmte Rolle (Daten, Logik, Präsentation) ein. Es gibt unterschiedliche Ausprägungen der Schichten-

Schichtenarchitektur

architektur, wie etwa die 2-Schichten-, 3-Schichten- und 5-Schichtenarchitektur. Mit der Anzahl der Schichten steigt die Flexibilität, aber auch die Komplexität des Gesamtsystems.

Service-orientierte Architekturen

Serviceorientierte Architekturen (SOA) zeichnen sich durch die lose Koppelung der einzelnen Services aus. Zentraler Bestandteil in einer SOA ist das Service, das bestimmte Dienste nach außen zur Verfügung stellt. Alle Services werden in einem zentralen Service-Repository kategorisiert abgelegt. Dies soll die Wiederverwendbarkeit von Services in einem Unternehmen fördern. Die Herausforderung bei der SOA liegt vor allem auch in der Umsetzung von generischen Services, die in unterschiedlichen Kontexten eingesetzt werden können. Durch die Orchestrierung von Services entstehen agile prozessorientierte Applikationen. Sie können im Vergleich zu herkömmlichen Applikationen rasch angepasst werden.

Ereignisgetriebene Architektur

Ein Architekturstil, der in letzter Zeit an Bedeutung gewonnen hat, ist die ereignisgetriebene Architektur. Unter einem Ereignis versteht man eine signifikante Zustandsänderung in einem System. Der große Unterschied zu herkömmlichen Systemen besteht darin, dass nicht das System eine Zustandsänderung wahrnimmt und explizit Aktionen in anderen Systemen auslöst, sondern nur ein Ereignis versendet. Die anderen Teilsysteme können sich für dieses Ereignis registrieren, um entsprechend darauf reagieren zu können. Damit ist es möglich, einen sehr hohen Grad der losen Koppelung zwischen den einzelnen Systemen zu erreichen. Ein besonderer Vorteil ereignisgetriebener Architektur ist, dass sie nicht notwendigerweise ein Gegensatz zu anderen Architekturstilen ist, sondern als Ergänzung zu einer bestehenden Architektur eingeführt werden kann.

8 | Entwurfs-, Architektur- und Integrationsmuster

Dieses Kapitel soll als Referenz auf bestehende Lösungsansätze für die im vorherigen Kapitel beschriebenen Architekturen für ein kleines bis mittelgroßes Projekt dienen. Die hier beschriebene Einführung in Entwurfs-, Architektur- und Integrationsmuster kann dazu verwendet werden, wiederkehrende Aufgaben in der Praxis zu implementieren.

Übersicht

8.1 Was ist ein Muster

Wiederkehrende Problemstellungen

In der Praxis der Software-Entwicklung hat sich gezeigt, dass bestimmte Problemstellungen immer wieder, wenn auch in leicht veränderter Form, vorkommen. Beispielsweise hat fast jede Anwendung, die in einem modernem Betriebssystem wie OS X oder Linux/KDE/Gnome läuft, bestimmte grafische Komponenten wie Buttons, Listen, Tabellen usw. Der Benutzer kann mit diesen Komponenten interagieren, also z. B. einen Button drücken, in einer Tabelle scrollen oder einen neuen Wert in eine Textbox eintragen. Der Programmierer einer Anwendung möchte daher sicher das Rad nicht neu erfinden, indem er diese Standardkomponenten selbst implementiert. Dasselbe gilt für die Entwicklung der Logik, die hinter den Komponenten steht z. B.:

Interaktion mit dem Benutzer

- Wie implementiert man eine Reaktion auf ein Benutzerverhalten, z. B. wenn ein Button gedrückt wird?
- Wenn sich „im Hintergrund“ Daten ändern, sollen diese in der Benutzerschnittstelle ebenfalls aktualisiert werden.
- Wenn der Benutzer Daten ändert, sollen diese nicht nur in der Benutzerschnittstelle, sondern z. B. auch in der Datenbank aktualisiert werden.

Beispiele für Problemfelder

Diese Problemstellungen findet man nahezu in jeder Anwendung, die über eine grafische Benutzerschnittstelle verfügt. Ähnlich wiederkehrende Problemstellungen findet man z. B. in diesen Bereichen:

- Wie kann man eine saubere Trennung von Geschäftslogik und Datenhaltung gewährleisten?
- Wie kann man die Definition von Schnittstellen von der konkreten Implementierung von Funktionalität trennen?
- Wie kann man die Interaktion mit komplexen Bibliotheken für bestimmte Anwendungsszenarien einfacher gestalten?
- Wie kann man Komponenten und Klassen möglichst locker verbinden (um sie austauschbar zu halten, z. B. Treiber und Plugins) und diese möglichst an zentraler Stelle konfigurieren?
- Wie kann man Objekte und Komponenten, deren Instanziierung „teuer“ ist (also viele Ressourcen oder Zeit benötigt, z. B. Datenbankzugriffs-Objekte) wiederverwenden?
- Wie kann man dynamisch zur Laufzeit Funktionalität zu Komponenten hinzufügen und gegebenenfalls auch wieder entfernen?

Tabelle 8.1 *Übersicht der hier verwendeten Entwurfsmuster nach Gamma et.al. [32].*

Muster	Seite	Problemstellung	verwandte Muster
Grundlegende Muster			
Interface	233	Trennung von Schnittstelle und Implementierung	Delegation, Strategy
Delegation	236	Erweitern von Funktionalität ohne Vererbung	Interface, Strategy, Proxy, Decorator
Strategy	236	Entkopplung von Objekten	Interface, Delegation, Factory
Immutable	243	Daten, die nicht verändert werden sollen	
Marker	243	Markieren einer Klasse	Annotation
Annotation	244	Anreichern von Objekten mit Metadaten	Marker
Erzeugungsmuster			
Singleton	247	Von einer Klasse darf es nur eine einzige Instanz geben	Factory
Factory	250	Erzeugen von Instanzen unter komplexen Rahmenbedingungen	Interface, Delegation, Strategy, Singleton, Dependency Injection
Object Pool	252	Wiederverwendung von Objekten	
Strukturmuster			
Fassade	253	Vereinfachter Zugriff auf komplexe APIs	Adapter
Iterator	254	Iterieren von verschiedenartigen Datenstrukturen	
Adapter	255	Klassen integrieren, die man nicht verändern kann	Fassade
Proxy	256	„Vortäuschen“ von Zuständigkeit	Decorator, Interceptor, Delegation
Data Access Object	263	Trennung von Geschäfts- und Persistenzlogik	Delegation, Factory, Dependency Injection
Generic DAO	267	Verhindern von sich wiederholender DAO-Logik	Data Access Object
Verhaltensmuster			
Observer	269	Benachrichtigen von interessierten Objekten im Fall von Zustandsänderungen	Event-Listener, Model View Controler
Decorator	274	Dynamisches Hinzufügen von funktionalen Aspekten zu Objekten	Proxy, aspektorientierte Programmierung, Delegation
Interceptor	278	Architektur, die dynamisches Hinzufügen von Funktionalität ohne Wissen der Basis-Objekte erlaubt	Aspektorientierte Programmierung, Observer, Proxy

Tabelle 8.2 *Übersicht über die hier beschriebenen Integrationsmuster nach Hohpe et.al. [41].*

Muster	Seite	Problemstellung
Integrations-Muster	281	Einführung
Integrations-Stile	284	Arten der Integration verschiedener Systeme
Messaging	289	Integration über Nachrichtenaustausch
Routing	294	Muster der Nachrichtenzustellung
Transformation	296	Transformation der Nachricht für verschiedene Empfänger
System Management und Testen	297	Muster, die System-Management und Testen von Messaging-Systemen erleichtern

> Wie kann man eine komplexe Landschaft von (Geschäfts-) Anwendungen so integrieren, dass diese einerseits flexibel bleibt, andererseits aber Wartbarkeit und Zuverlässigkeit nicht vernachlässigt werden?

Design-Patterns und Architekturmuster

In den letzten Jahrzehnten der Software-Entwicklung wurden daher solche immer wiederkehrenden Probleme analysiert, aus Fehlern gelernt und abstrakte Lösungsmuster entwickelt und beschrieben. Diese Muster (Pattern) kann man dann für eigene Probleme verwenden bzw. an diese anpassen. Die Beschreibung dieser Muster erfolgt in diesem Kapitel zwar in Form von Java-Beispielen, die *Muster an sich* sind aber sprachunabhängig und können in beliebigen Sprachen implementiert werden.

> „Pattern“ has been defined as „an idea that has been useful in one practical context and will probably be useful in others.“
> *Martin Fowler [28]*

Design-Patterns werden heute als Grundwissen jedes Software-Entwicklers angesehen und dienen daher in der Software-Entwicklung auch als „Sprachmittel“; d. h., die Kommunikation im Team über architektonische Konzepte wird einfacher.

Granularität

Muster können außerdem auf verschiedenen Ebenen der Granularität beschrieben werden. Eines der wesentlichsten Bücher, in dem die meisten hier besprochenen Entwurfsmuster eingeführt wurden, ist von Erich Gamma et. al. [32]. Sie beziehen sich im Wesentlichen auf architektonische Aspekte *innerhalb* einer Anwendung, sind also eher feingranular. Die erste Auflage des Buches stammt aus dem Jahr 1994.

In diesem Kapitel werden zunächst einige wesentliche Entwurfsmuster vorgestellt (Tabelle 8.1 gibt eine Übersicht). Außerdem werden grundsätzliche Architekturmuster eingeführt. Die Muster werden in Kategorien organisiert:

Grundlegende Muster

Grundlegende Muster sind ein essenzieller Teil jeder heutigen Software-Architektur und müssen daher verstanden werden, um die restlichen Kategorien verstehen zu können. Einige dieser grundlegenden Muster findet man bereits im Kern moderner Programmiersprachen (z. B. das Interface-Muster in Java).

Erzeugungs-Muster

Muster, die sich damit beschäftigen, Instanzen von Klassen zu erzeugen, werden als *Erzeugungs-Muster* bezeichnet. In vielen Fällen ist eine einfache Instanziierung von Klassen nicht möglich, weil Typ oder Konfiguration eines benötigten Objekts (oder einer Hierarchie von Objekten) mit dem konkreten Zustand des Programms variiert. In anderen Fällen möchte man vermeiden, dass mehrere Instanzen einer Klasse gebildet werden oder dass bestehende Instanzen wiederverwendet werden.

Struktur-Muster

Um die Struktur einer Anwendung möglichst flexibel und wartbar zu gestalten, haben sich eine Reihe von *Struktur-Mustern* herausgebildet. Das *Data-Access-Object*-Muster beispielsweise erlaubt eine saubere Trennung zwischen Persistenzschicht und Geschäftslogik.

Verhaltens-Muster

Muster, die sich mit der Interaktionen zwischen Objekten beschäftigen, werden als *Verhaltens-Muster* bezeichnet. Mit ihrer Hilfe kann die Kommunikation zwischen Objekten flexibler gestaltet werden. Das *Observer*-Muster beispielsweise hilft, Zustandsänderungen zwischen Objekten zu kommunizieren, ohne diese eng aneinander zu koppeln.

Notwendige Vorkenntnisse

Zum Verständnis der Patterns sind gute Kenntnisse objektorientierter Programmierung erforderlich. Viele Muster werden anhand von UML-Diagrammen erklärt, folglich ist auch ein gutes Verständnis von UML erforderlich. In den weiteren Abschnitten werden die Muster mit den üblichen englischen Namen benannt (in Klammern steht im Titel, sofern das Sinn macht, die deutsche Übersetzung). Das Verständnis der Patterns auf verschiedenen Ebenen ist auch eine Voraussetzung für das gute Verständnis der nächsten Kapitel.

Integrations-Muster

Die Software-Entwicklung und -nutzung hat sich aber schnell weiterentwickelt und heute müssen die meisten Anwendungen mit anderen Anwendungen interagieren. Daher wird in diesem Kapitel in Abschnitt 8.6 auch ein Grundverständnis der *Muster der Systemintegration* vermittelt, die in jüngerer Zeit vor allem von Hohpe et. al. systematisch beschrieben wurden [41]. Tabelle 8.2 gibt einen kurzen Überblick über die vorgestellten Integrationsmuster.

8.2 Grundlegende Muster

8.2.1 Interface (Schnittstelle)

Trennung von Schnittstelle und Implementierung

Das Interface-Pattern ist eines der einfachsten und auch fundamentalsten Muster. Es dient der Trennung von Schnittstellenbeschreibung und Imple-

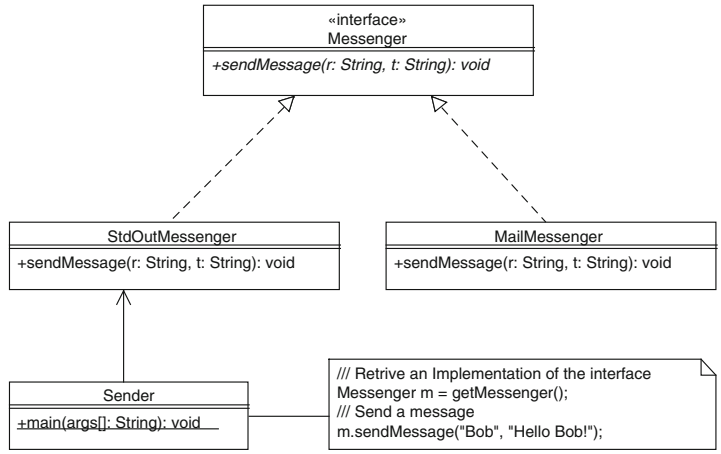


Abbildung 8.1 *Interface (Schnittstelle) Muster.*

mentierung. Dies erlaubt einerseits eine saubere Definition von Schnittstellen, die sich auch möglichst wenig ändern sollen, und verschiedenen Implementierungen dieser Schnittstellen. Klassen, die Implementierungen dieser Schnittstellen verwenden, müssen auch über konkrete Implementierungs--Details nicht Bescheid wissen. Dieses Muster ist also der erste Schritt in der Entkopplung von Klassen, wie man in weiterer Folge sehen wird[1].

Beispiel: Versenden von Nachrichten

Abbildung 8.1 zeigt ein Beispiel: Es soll eine Anwendung entwickelt werden, in der Nachrichten verschickt werden können. Nun gibt es verschiedene technische Methoden, Nachrichten zu versenden: Man könnte die Nachricht als E-mail verschicken, als Jabber XMPP Instant Message oder vielleicht sogar als JMS-Nachricht. In Fällen wie diesem ist es sinnvoll, nicht sofort eine konkrete Implementierung zu erstellen[2], sondern zunächst das Problem abstrakt zu betrachten:

Das Interface

Was ist die allgemeinste Betrachtung dieses Problems? Man könnte zu dem Schluss kommen, dass man eine Methode braucht, die eine einfache *Text-Nachricht* an *einen Empfänger* sendet.

Daher definieren wir zunächst wie im UML-Diagramm zu sehen ist ein Interface[3]:

[1] Details zur Entkoppelung von Objekten und Komponenten finden sich im Abschnitt 9.1. In diesem Abschnitt werden nochmals detailliert die Schritte von stark gekoppelten zu entkoppelten Systemen analysiert sowie deren Vor- und Nachteile beschrieben.

[2] Also nicht etwa gleich eine Klasse `MailMessage` mit der Methode `sendMail(...)` erstellen, das könnte man später nicht einfach auf ein anderes Protokoll portieren.

[3] Das komplette Beispiel mit Interface, Delegation, Proxy und Factory findet sich im Web in der Kategorie Designpatterns.

```
public interface Messenger {
  public void sendMessage(String receiver,
                          String text);
}
```

E-mail-Implementierung

Dieses Interface ist so allgemein wie möglich gehalten (für diese konkrete Problemstellung). Man beachte, dass dieses Interface keinerlei konkrete Annahmen über das zu verwendende Protokoll macht. Nun könnte man im nächsten Schritt eine oder mehrere konkrete Implementierungen erstellen:

```
public class MailMessenger implements Messenger {
  public MailMessenger() {
    // JavaMail System (Bibliothek) initialisieren
  }
  public void sendMessage(String receiver,
                          String text) {
    // zum Mailserver verbinden
    ...
    // Nachricht senden
    ...
    // Verbindung trennen
    ...
  }
}
```

Kommandozeilen-Implementierung

Oder eine Implementierung, die diese Nachricht auf die Kommandozeile ausgibt:

```
public class StdOutMessenger implements Messenger {
  public void sendMessage(String receiver,
                          String text) {
    System.out.println("Empfänger: " + receiver
                                    + ":");
    System.out.println(text);
  }
}
```

Nachrichten versenden

Möchte man nun in einer Klasse eine Nachricht per E-mail versenden, so wird man etwa folgenden Code schreiben:

```
public class Sender {
  public static void main(String args[]) {
    Messenger m = new MailMessenger();
    m.sendMessage("Bob",
                  "Hello Bob, this is Alice!");
  }
}
```

Wichtig ist es an dieser Stelle zu beachten, dass als Variablentyp das Interface `Messenger` und nicht etwa `MailMessenger` verwendet wurde.

D. h., durch einen Austausch einer Zeile kann statt dem E-mail-Versand nun auf Kommandozeile oder irgendeine andere Implementierung umgestellt werden:

```
1 public class Sender {
2   public static void main(String args[]) {
3     Messenger m = new StdOutMessenger();
4     m.sendMessage("Bob",
5                   "Hello Bob, this is Alice!");
6   }
7 }
```

Gleichzeitig drückt man damit aus, dass in diesem Codeteil nur die Funktionalität des Interfaces zu verwenden ist. Man unterbindet damit, dass etwa andere Methoden der `StdOutMessenger`-Klasse verwendet werden, was die Austauschbarkeit der Implementierung stark einschränken würde.

Anwendung

Das Interface-Pattern bietet sich also immer dann an, wenn man eine bestimmte Schnittstelle an andere kommunizieren möchte oder wenn zu erwarten ist, dass es für eine bestimmte Aufgabe mehrere unterschiedliche Implementationen geben wird. Schnittstellen sollten sich möglichst nicht verändern, während konkrete Implementierungen leicht verändert werden können.

Verwandte Patterns

Das Interface-Pattern wird sehr häufig in Kombination mit anderen Patterns verwendet. Besonders in Kombination mit dem Strategy-Pattern können Interfaces ihre Stärken ausspielen. Das Strategy-Pattern entkoppelt Klassen voneinander und erlaubt die einfachere Austauschbarkeit von konkreter Funktionalität (z. B. von Treibern, Import-, Export Filtern usw.).

Wendet man zusätzlich das *Factory* (siehe Abschnitt 8.3.2) oder das *Dependency-Injection*-Pattern (siehe Abschnitt 9.3) an, kann man überhaupt erst zur Laufzeit oder Startzeit der Anwendung entscheiden, welche Implementierung des Interfaces nun für die nächste Nachricht verwendet werden soll (z. B. durch Auslesen einer Konfigurationsdatei oder durch Nachfrage beim Benutzer).

8.2.2 Delegation

Problemstellung: Mehr Funktionalität ...

Das Delegations-Pattern wird dann eingesetzt, wenn eine Klasse Funktionalität benötigt, die innerhalb der Klasse (noch) nicht zur Verfügung steht. Dies ist grundsätzlich auf zwei Wegen möglich: Man kann eine Klasse ableiten und in der abgeleiteten Klasse weitere Funktionalität zur Verfügung stellen, oder man kann die benötigte Funktionalität in eine eigene Klasse auslagern und diese mittels Delegation verwenden. D. h., man holt sich eine Instanz dieser Hilfsklasse und verwendet diese in der Klasse, deren Funktionalität erweitert werden soll.

Beispielsweise könnte man in einer Anwendung die Klasse `Person` verwenden, um personenbezogene Daten zu verwalten (Name, Adresse, Kontaktinformation usw.). Nun könnte in der Anwendung der Bedarf auftreten, Benachrichtigungen an eben diese Personen zu versenden. Da die E-mail-Adresse und auch die Telefonnummer in der Klasse `Person` gespeichert

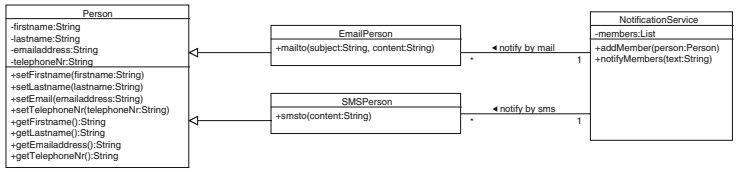


Abbildung 8.2 *Das Objekt* `Person` *wird mittels Vererbung um Notifikations-Funktionalität erweitert.*

sind, könnte man die Klasse ableiten und in der abgeleiteten Klasse diese Funktionalität ergänzen.

. . . mittels Vererbung?

Abbildung 8.2 illustriert diese Variante an einem Beispiel: Die Klasse `Person` hält personenbezogene Daten. Da man die Funktionalität der Benachrichtigung ergänzen möchte, könnte man die Klasse ableiten und zunächst `EmailPerson` erzeugen. Die Klasse `EmailPerson` verfügt nun über dieselbe Funktionalität wie Person, erlaubt aber zusätzlich das Versenden von Benachrichtigungen an eben diese Person.

Später könnte man nun feststellen, dass man nicht nur die Benachrichtigung mittels E-mail benötigt, sondern auch mittels SMS. Natürlich kann man auch diese Funktionalität implementieren, indem man wieder von `Person` ableitet; diesmal wird die Klasse `SMSPerson` erstellt. Auch diese enthält alle Daten und Funktionen wie `Person` und ermöglicht zusätzlich das Versenden von SMS.

Mehrfachvererbung und andere Probleme

Jetzt steht man aber vor dem Problem, dass die Klasse `NotificationService` entweder `EmailPerson` oder `SMSPerson` zur Datenspeicherung verwenden kann, also nicht beide Arten der Benachrichtigung gleichzeitig genutzt werden können. Dies ist natürlich in vielen Fällen ungünstig.

Grundsätzlich könnte man das Problem auch umgekehrt herum betrachten und zwei Klassen für die Benachrichtigung erstellen: `Email` und `SMS` und dann die `Person`-Klasse von einer der beiden oder von beiden Klassen ableiten, sofern die Sprache das unterstützt. Eine Mehrfachableitung ist zwar in manchen Sprachen wie C++ möglich, in vielen modernen objektorientierten Sprachen wie Java aber nicht.

Von dieser Einschränkung abgesehen, wäre das Ergebnis in beiden Fällen nicht optimal. Im ersten Fall muss man sich für eine Variante der Benachrichtigung entscheiden, im zweiten Fall leitet man eine Klasse, nämlich `Person` von einer anderen Klasse ab (`SMS` oder `Email`), die eigentlich inhaltlich nicht viel miteinander zu tun haben.

Lösung: Eleganter mit Delegation

In diesem Fall könnte man das Problem wahrscheinlich eleganter mittels Delegation lösen. Abb 8.3 zeigt *eine* Variante: Die Klassen `Email` und `SMS` implementieren nur die Benachrichtigungs-Funktionalität für das jeweilige Medium (E-mail, SMS). In der Klasse `Person` wird nun jeweils eine Instanz der Klasse `SMS` und `Email` verwendet, um die Benachrichtigung als SMS oder E-mail durchführen zu können; dabei kann natürlich innerhalb von `Person` auf alle Informationen zugegriffen werden und z. B.

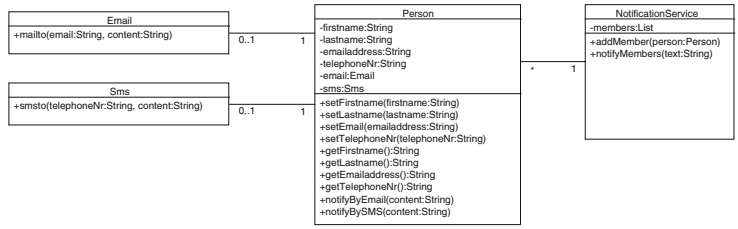


Abbildung 8.3 *Das Objekt* `Person` *delegiert die Notifikation an zwei konkrete Klassen* `Email` *und* `Sms`*. Schon besser, aber man erkennt immer noch eine recht enge Kopplung zwischen den Klassen.*

ein personalisiertes E-mail erstellt und dann mithilfe des `Email`-Objekts versendet werden[4].

Trends: Vererbung vs. Delegation

Delegation ist eine Möglichkeit, die Funktionalität einer Klasse zu erweitern und stellt eine häufig verwendete Alternative zur Vererbung dar. Es soll aber in diesem Abschnitt nicht der Eindruck entstehen, als wäre Vererbung grundsätzlich eine schlechte Wahl; dies trifft natürlich nicht zu.

Allerdings kann man durchaus auch im Software-Engineering und in Architekturansätzen gewisse „Modetrends“ erkennen. Vor vielleicht zehn Jahren wurde sehr gerne mit Vererbung gearbeitet. Analysiert man z. B. die von Sun mit dem Java Development Kit mitgelieferte API, so erkennt man, dass hier in vielen Bereichen (z. B. Swing) sehr viel Vererbung zu beobachten ist, was auch fallweise kritisiert wurde. Heute lässt sich eine Trendumkehr erkennen. Die Erfahrung zeigt, dass „exzessive“ Vererbungshierarchien eher zu vermeiden sind. Sie neigen dazu unflexibel und schwer verständlich zu werden.

Vererbung ist dann ein gutes Konzept, wenn es sich im Wesentlichen um eine *is-a*-Verbindung handelt. Also z. B. ein Kreis ist eine geometrische Figur, ebenso wie ein Rechteck; ein Student und ein Professor sind Personen. Eine Person kann Notifizierung verwenden wollen (oder umgekehrt), aber es ist schwer zu argumentieren, dass die beiden in einer *is-a*-Beziehung stehen. Daher ist besonders in solchen Fällen, und wo man ein höheres Maß an Flexibilität wünscht, häufig Delegation das bessere Mittel, um Verbindungen zwischen Klassen zu modellieren.

Verwandte Patterns

Strategy, Visitor und Observer-Pattern sowie Event Listener benötigen Delegation, um ihre Aufgaben zu erfüllen. Das Decorator-Pattern ist ebenfalls sehr ähnlich und verwendet auch Delegation. Das Proxy-Pattern (siehe Abschnitt 8.4.4) kann als Erweiterung des Delegations-Patterns verstanden werden.

[4] Ein anderer Ansatz wäre es, die Logik umzudrehen: `NotificationService` holt sich Instanzen der Benachrichtigungsobjekte `Email` und `SMS` und übergibt an diese die benötigten Daten des Personen-Objekts. In beiden Fällen würde man eine Delegation der Funktionalität durchführen.

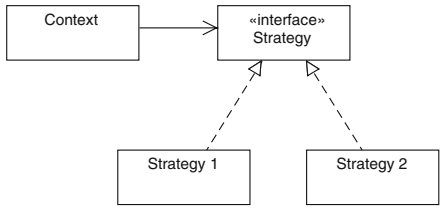


Abbildung 8.4 *Das Strategy-Pattern entkoppelt zwei Algorithmen voneinander: Das Context-Objekt arbeitet nur mit dem Interface, nicht mit einer konkreten Instanz.*

8.2.3 Strategy (Strategie)

Problemstellung

Die oben dargestellte Variante mittels Delegation ist schon eleganter als die Varianten mittels Vererbung. Dennoch hat auch die in Abbildung 8.3 skizzierte Architektur noch einige Schwachstellen:

- Die Bindung zwischen `Person` und `Sms` bzw. `Email` ist sehr eng, d. h., es kann nicht leicht eine weitere Variante der Benachrichtigung (z. B. für Instant Messenger) hinzugefügt werden, ohne dass der Code nachhaltig geändert werden muss.
- Für jede neue Benachrichtigungsvariante muss eine Methode in `Person` hinzugefügt sowie die entsprechende Benachrichtigungsklasse geschrieben werden.
- Die Entscheidung, welche Benachrichtigung zu erfolgen hat, ist in diesem Ansatz noch nicht wirklich gelöst: Im Augenblick muss die aufrufende Klasse (also z. B. `NotificationService`) bei jedem Aufruf entscheiden, welche Art der Benachrichtigung erfolgen soll.
- Dies ist besonders dann problematisch, wenn die Klasse `Person` von verschiedenen Klassen verwendet wird, aber nicht jede über die Präferenzen der jeweiligen Person Bescheid weiß; z. B. weil eine Klasse die Personen verwaltet, andere Klassen mit den Daten arbeiten und Benachrichtigungen versenden wollen.

Lösung

An dieser Stelle kann das Strategy-Pattern helfen: Die Grundstruktur ist in Abbildung 8.4 zu sehen. Das sogenannte *Context*-Objekt verwendet entweder das Objekt *Strategy 1* oder *2*. Die Verbindung ist aber nicht hart verdrahtet, sondern das *Context*-Objekt verwendet tatsächlich nur das *Strategy*-Interface. Damit kann die konkrete Strategie leicht ausgetauscht werden, evt. auch erst zur Laufzeit.

Ein häufig zitiertes Beispiel für die Anwendung des Strategy-Patterns ist ein Sortier- oder Suchalgorithmus. Angenommen das Context-Objekt benötigt einen Sortieralgorithmus. Um zu vermeiden, dass dieses Objekt hart mit einer bestimmten Implementierung, z. B. Bubblesort verbunden wird und damit ein späterer Austausch z. B. mit einem für das Problem effizienteren Algorithmus erschwert würde, wählt man den Weg des Strategy-Patterns:

- Zunächst wird ein generisches Interface, z. B. `Sort`, definiert.
- Das Context-Objekt verwendet intern *ausschließlich* dieses `Sort`-Interface.
- Dann werden ein oder mehrere konkrete Implementierungen des Interfaces programmiert, z. B. `BubbleSort` und `QuickSort`.
- Schließlich wird dem Context-Interface mitgeteilt, welche konkrete Implementierung verwendet werden soll.

Um auch diesen letzten Schritt möglichst flexibel zu halten, wird das Strategy-Pattern häufig mit anderen Patterns wie *Factory* oder *Dependency Injection* kombiniert.

Beispiel: Interface und Strategy für mehr Flexibilität

Nun kann das im vorigen Abschnitt begonnenen Beispiel noch verbessert werden, indem das Strategy-Pattern eingesetzt wird. Im letzten Schritt soll nun der Vorgang der Benachrichtigung generischer implementiert werden, sodass es leichter ist, neue Formen der Benachrichtigung hinzuzufügen. Auch die Entscheidung, welche Benachrichtigung für welche Person angemessen ist, soll innerhalb des Personen-Objekts möglichst elegant entschieden werden können. Abbildung 8.5 zeigt einen möglichen Ansatz mit dem Strategy-Pattern:

- Es wird zunächst ein Interface `INotification` definiert, das eine für alle Benachrichtigungen gleichermaßen gültige Methode `send()` definiert.
- Das Personen-Objekt wird übergeben, damit sich die jeweilige Implementierung des Interfaces die benötigten Informationen aktuell aus dem `Person`-Objekt holen kann.
- Die Klassen `Email` und `Sms` implementieren nun dieses Interface.
- Die `Person`-Klasse verfügt nur mehr über eine generische `notify()`-Methode.
- Die Entscheidung, welche Benachrichtigung für die jeweilige Person zu erfolgen hat, wird in der `setNotificationPreference()`-Methode definiert.

Java-Code

Durch diese Änderungen kann eine neue Methode der Benachrichtigung hinzugefügt werden, ohne dass Änderungen an der Klasse `Person` notwendig wären. Um diese Variante klarer zu machen, sollten die Kern-Aspekte noch in Java-Code deutlich gemacht werden:

Die Klasse `Person` ist ein einfaches Java Bean, das zur Datenhaltung verwendet wird. Interessant sind in diesem Zusammenhang nur folgende Methoden:

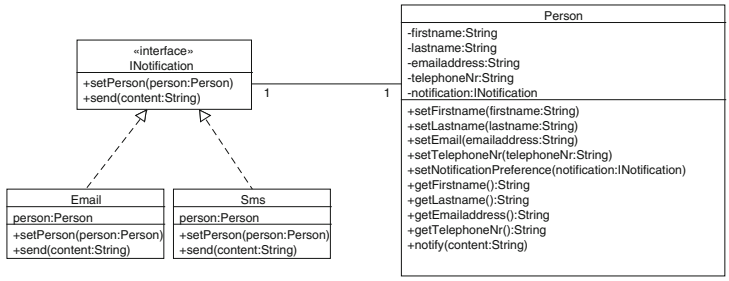


Abbildung 8.5 *Das `INotification`-Interface wird in Kombination mit dem Strategy-Pattern verwendet, um die Klassen zu entkoppeln und die Art der Benachrichtigung eleganter entscheiden zu können.*

```
public class Person {
  ...
  private INotification notification;
  ...
  public setNotificationPreference(
                        INotification notification) {
    this.notification = notification;
  }
  ...
  public notify(String content) {
    notification.send(content);
  }
}
```

Man erkennt hier, dass auch diese beiden Methoden nur mit dem Interface arbeiten und keinerlei Wissen über die konkrete Implementierung der Benachrichtigung benötigen. Das `INotification`-Interface ist ebenfalls einfach und implementiert nur zwei Methoden:

```
public interface INotification {
  public void setPerson(Person person);
  public void send(String content);
}
```

Die Klasse `Email` implementiert die E-mail-Funktionalität und holt sich die E-mail-Adresse aus dem Personen-Objekt, dasselbe gilt analog für die `Sms`-Klasse:

```
public class Email implements INotification {
  Person person;
  public String send(String content) {
    // send email here
    System.out.println("Sending Email to"
                       + person.getEmail() + ": "
                       + content);
    return "Email sent";
  }
  public void setPerson(Person person) {
    this.person = person;
  }
```

Zum Verständnis wichtig ist weiterhin noch, wie das Personen-Objekt verwendet werden könnte:

```
class NotificationManager {
  public enum NotificationMethod {SMS, EMAIL};
  private List<Person> members;
  ...
  public void addMember(Person person,
            NotificationMethod notificationPref) {
    members.add(person);
    INotification notification;
    if (notificationPref == NotificationMethod.SMS)
    {
      notification = new Sms();
    } else {
      notification = new Email();
    }
    notification.setPerson(person);
    person.setNotificationPreference(notification);
  }

  public void sendNotifications(String message) {
    for (Person p : members) {
      p.sendMessage(message);
    }
  }
  ...
}
```

Man erkennt hier, dass wieder nur die Interface-Variable verwendet wird; allerdings wird abhängig von der Präferenz entweder eine `Sms`-Instanz oder eine `Email`-Instanz gebildet und dem Person-`Objekt` übergeben[5]. Um alle Personen zu benachrichtigen reicht die einfache Schleife in der `sendNotifications` Methode. Da der jeweiligen Person schon vorher die entsprechende Instanz zugewiesen wurde, wir hier automatisch richtig benachrichtigt.

Diese Variante ist sichtbar flexibler als die vorige Lösung, hat aber immer noch einen offensichtlichen Nachteil: Für neue Notifizierungen muss die `NotificationService`-Klasse verändert werden, weil hier entschieden wird, welche konkrete Instanz gebildet wird. Diesen Nachteil kann man ebenfalls beheben.

Verwandte Patterns

Eine Möglichkeit bietet z. B. das Factory-Pattern in Abschnitt 8.3.2. Dieses wird gerne mit dem Strategy-Pattern kombiniert, um erst zur Laufzeit entscheiden zu können, welche konkrete Strategie verwendet werden soll. Ei-

[5] Diese Klasse ist hier deutlich vereinfacht. In der Beispielanwendung, die zum Download bereit steht, ist das Beispiel etwas umfangreicher implementiert und mit JUnit-Testmethoden versehen.

ne Erweiterung des Factory-Patterns ist das Dependency-Injection-Pattern (siehe Abschnitt 9.3), das in modernen Architekturen, die mit Komponenten-Frameworks arbeiten, gerne angewandt wird.

Fazit

Das Strategy-Pattern ist besonders in Kombination mit dem Interface sowie mit dem Factory (oder Dependency Injection) Pattern ein sehr mächtiges Werkzeug, um einzelne Komponenten einer Anwendung voneinander zu entkoppeln. Dies ist nützlich, um sich nicht frühzeitig für bestimmte Bindungen zwischen Klassen entscheiden zu müssen; z. B. welcher Transaktionsmanager, welcher Persistenz-Algorithmus, welcher Export-Filter, welcher Suchalgorithmus usw. nun in einem bestimmten Fall verwendet werden soll.

Es bietet sich daher an, Funktionalität, die potenziell von verschiedenen konkreten Implementierungen umgesetzt werden kann, immer zunächst über ein generisches Interface zu spezifizieren und dann erst mit dem Delegation-Pattern anzuwenden.

8.2.4 Immutable (Nicht veränderbares Object)

Problemstellung

In manchen Fällen möchte man verhindern, dass der Zustand eines Objekts nach der Erstellung verändert werden kann. Dies wird fallweise gemacht, wenn mehrere Threads mit einem Objekt arbeiten: dadurch wird sichergestellt, dass in dieser Instanz nur gelesen, nicht aber geschrieben werden kann. Auch bei Objekten, die z. B. Konfigurationseinstellungen speichern, die nur einmal bei Programmstart gelesen werden, macht es keinen Sinn, eine Änderung der Werte während der Laufzeit zu erlauben.

Lösung

Die Lösung dieses Problems ist einfach: Man definiert die Klasse so, dass die Werte im Konstruktor gesetzt werden, über Getter-Methoden gelesen, aber nicht mehr verändert werden können, da keine Setter-Methoden angeboten werden.

8.2.5 Marker (Markieren eines Objekts)

Problemstellung

Manchmal möchte man Codeteile auf eine bestimmte Weise markieren, also mit Metadaten ausstatten. Dies kann als Information für andere Klassen, für Code-Generatoren oder IDEs dienen. An einem Beispiel wird das verständlich: In der Java API finden sich Interfaces wie `Cloneable` oder `Serializable`. Liest man die Dokumentation, so stellt man fest, dass beide Interfaces *keine* Methoden definieren. Daher kann eine Klasse diese Interfaces implementieren ohne irgendwelche Methoden implementieren zu müssen (was normalerweise die Funktion eines Interfaces darstellt).

Umsetzung

In diesen beiden Fällen dient das Interface nur dazu, um Klassen auf eine bestimmte Weise zu markieren: Implementiert man `Serializable`, so zeigt man damit, dass die Klasse serialisierbar ist. Versucht eine andere

Klasse eine Klasse zu serialisieren, die das Interface nicht implementiert, so wird eine Exception ausgelöst. Der Hintergrund ist hier der, dass nicht jede Klasse geeignet ist, um serialisiert zu werden. Ist das möglich und erwünscht, kann dies durch das Interface ausgedrückt werden.

Verwandte Muster

Mit dem Marker-Pattern verwandt sind die im nächsten Abschnitt beschriebenen Annotationen.

8.2.6 Annotations (Metadaten für Objekte)

Problemstellung

Bereits im vorigen Abschnitt über Marker Interfaces kann man erkennen, dass es Fälle gibt, in denen man einen Codeabschnitt mit weiteren Informationen (Metadaten) anreichern möchte. Um das klarer zu machen, einige Beispiele:

- Man schreibt eine Bibliothek für andere Entwickler und stellt eine bestimmte Klasse aus Gründen der Abwärtskompatibilität *noch* zur Verfügung, möchte aber zeigen, dass diese Klasse eigentlich nicht mehr verwendet werden sollte (*deprecated functionality*).
- Eine Methode soll eine Methode der Überklasse überschreiben, und man möchte sicherstellen, dass diese Methode auch tatsächlich in der Überklasse vorhanden ist.
- Der Compiler oder die IDE geben Warnungen bei einem bestimmten Codeabschnitt aus (z. B. „deprecation warning“); man ist sich des Problemes bewusst, möchte die Warnung aber unterdrücken, um das Fehlerfenster nicht zu „fluten“.
- Eine Klasse soll einzelne Methoden als Webservice zur Verfügung stellen, und diese Methoden sollen kenntlich gemacht werden.
- Eine Klasse soll mit einem objektrelationalen Mapping Tool persistiert werden. Dafür muss die Verknüpfung zwischen Feldern einer Klasse und Spalten einer Tabelle der Datenbank oder die Abbildung von Java-Datentypen auf SQL Datentypen beschrieben werden.

Umsetzung

In all diesen Fällen benötigt man neben dem eigentlichen Sourcecode zusätzliche Informationen, um das jeweilige Problem zu lösen. Genau zu diesem Zweck kann man Annotationen[6] verwenden. Annotationen werden in verschiedenen modernen Programmiersprachen angeboten. In Java werden Annotationen durch das @-Zeichen ausgezeichnet, gefolgt vom Namen der Annotation. Dabei gibt es eine Reihe von Annotationen, die bereits vordefiniert sind (Beispiele s.u.). Daneben kann man sich beliebige eigene Annotationen definieren.

[6] Zum Thema Annotationen findet man weitere Infos in der Java-Dokumentation [51].

Um dies anhand von konkreten Beispielen zu zeigen:

```
/**
 * @deprecated  EmailService is deprecated,
 *              use NotificationService instead
 */
@Deprecated
public class EmailService {
  ...
}
```

Beispiel: Unterdrückte Warnungen

In diesem Beispiel zeigt man durch die Annotation `@Deprecated` an, dass die gekennzeichnete Klasse nicht mehr verwendet werden soll. Verwendet ein Entwickler diese Klasse immer noch, so erhält er eine „deprecation warning“ von der Entwicklungsumgebung oder vom Java Compiler, um auf diesen Umstand hinzuweisen. Möchte er diese Warnung unterdrücken, so kann er dies ebenfalls mit einer Annotation erledigen:

```
public class CustomerNotification {
  @SuppressWarnings({"deprecation"})
  public CustomerNotification() {
    EmailService es = new EmailService();
  }
}
```

Beispiel: Überschriebene Methoden

Sinngemäß ähnlich verhalten sich andere in Java 5 bereits eingebaute Annotationen. Mit der `@Override`-Annotation beispielsweise kann man sicherstellen, dass eine bestimmte Methode tatsächlich die gleichlautende Methode der Überklasse überschreibt. Ist dies nicht der Fall (z. B. weil ein Tippfehler passiert, oder weil eine der Methoden umbenannt wurde), so erhält man eine Fehlermeldung vom Compiler oder der Entwicklungsumgebung.

Beispiel: O/R Mapping

Neben diesen eher einfachen Anwendungsfällen gibt es auch komplexere Anwendungen, z. B. in der Definition von objektrelationalen Mapping (ORM). In Abschnitt 9.4.8 wird ORM genauer beschrieben. An dieser Stelle nur eine kurze Beschreibung des Problems sowie der möglichen Lösung mithilfe von Annotationen:

Angenommen man hat folgendes Domänenobjekt:

```
class Customer {
  private Long CustomerID;
  private String firstname;
  private String lastname;
  ...
}
```

Diese Klasse soll persistiert werden, d. h., die Daten der Objekte sollen in eine relationale Datenbank geschrieben bzw. von dort gelesen werden können. Nun könnte schon eine Datenbank vorhanden sein, und dort existiert eine Tabelle die etwa wie folgt aussieht:

```
Table: PERSON
+--------------------------+-----+
| Field      | Type        | ... |
+--------------------------+-----+
| ID         | int(8)      | ... |
| FIRST_NAME | VARCHAR(50) | ... |
| LAST_NAME  | VARCHAR(50) | ... |
+--------------------------+-----+
```

Nun muss natürlich definiert werden, welche Variable im Domänenobjekt auf welche Tabelle und welche Felder der Tabelle abzubilden sind. Hier gibt es unterschiedliche Möglichkeiten, dieses Problem zu lösen. O/R Frameworks bieten meist die Variante an, dieses Mapping in einer externen Datei (z. B. im XML-Format) zu beschreiben. Dies hat Vorteile, aber auch Nachteile: es kann schwieriger sein, das Mapping aktuell zu halten, und es ist mehr Schreibarbeit. Daher gibt es meist auch die Möglichkeit, dies mit Annotationen zu lösen, z. B. in dieser Form:

```
1  @Entity
2  @Table( name = "PERSON" )
3  class Customer {
4    private Long CustomerID;
5    private String firstname;
6    private String lastname;
7    ...
8    @Id
9    @Column( name = "ID" )
10   public int getCustomerID() {...}
11
12   @Column( name = "LAST_NAME", nullable = false )
13   public String getLastname() {...}
14   ...
15 }
```

In diese Kategorie von Problemen fallen auch Anwendungsfälle, in denen Methoden als Webservices zur Verfügung gestellt oder von J2EE-Komponenten beschrieben werden sollen.

Annotation-Prozessor

Dieses Beispiel soll nur die Idee vermitteln. Der Grundgedanke ist hier derselbe wie oben, nur in einem komplexeren Anwendungsfall; Annotationen *markieren* hier nicht nur Zustände, sondern bieten auch die Möglichkeit, *Parameter* zu definieren (z. B. den Name der Spalte in der Datenbank). Je nachdem welche Art von Annotationen man verwendet, benötigt man natürlich einen Annotation-Prozessor, der die Annotationen zur Laufzeit (oder zur Entwicklungszeit) auswertet und entsprechende Aktionen setzt. Im letzten Beispiel könnte dieser Prozessor von einem O/R-Mapping-Tool stammen, das damit zur Laufzeit das Mapping der Objekte zu den Datenbanktabellen ausliest und das O/R-Tool entsprechend konfiguriert.

Auch Codegenerierung anhand von Annotationen ist natürlich möglich: Viele O/R-Tools sind z. B. in der Lage, aus den Annotationen der Domänenobjekte das SQL-Skript zu erstellen, das man benötigt, um die Datenbanktabellen anzulegen.

Eigene Annotations

Natürlich ist es auch möglich, eigene Annotationen zu definieren, sowie einen eigenen Annotations-Prozessor zu schreiben, der diese auswertet. Dies wird zwar nicht sehr häufig notwendig sein, kann aber in Einzelfällen helfen.

Kritik

Annotationen werden aber nicht von allen Experten gleichermaßen positiv beurteilt. Sie haben einerseits den Vorteil notwendige Metadaten direkt im Sourcecode verwalten zu können. Andererseits werden hier auch verschiedene Aspekte vermischt: beispielsweise Methodendefintionenen in einer Klasse und Deklaration von Webservices oder Domänenobjekte und Mapping-Information für O/R Mapping.

8.3 Erzeugung

8.3.1 Singleton

Problemstellung

In vielen Software-Projekten stellt sich das Problem, dass bestimmte Objekte nur einmal instanziiert werden, und alle anderen Klassen mit dieser einen Instanz arbeiten sollen. Ein klassisches Beispiel ist die Kommunikation mit realer Hardware. Es gibt eben z. B. nur einen Drucker oder eine Maschine eines bestimmten Typs der von der Software angesteuert werden soll[7]. Das Objekt, das diese Maschine repräsentiert, soll daher auch nur in einer einzigen Instanz vorliegen. Bleiben wir beim Drucker-Beispiel:

Angenommen die Drucker-Klasse implementiert eine Warteschlange, in die sich alle Druckeinträge einordnen sollen. Dann wäre es natürlich fatal, wenn Objekte, die etwas Drucken wollen, mehrere Instanzen des Drucker-Objektes instanziieren und plötzlich mehrere Druckerinstanzen versuchen, gleichzeitig mit der einen Hardware zu kommunizieren.

In modernen Enterprise-Architekturen trifft man das Singleton-Pattern auch immer wieder an. Ein Beispiel sind Logging Services (siehe auch Abschnitt 9.8). Es muss sichergestellt werden, dass alle Statusmeldun-

[7] In dieser Pattern-Einführung bleiben wir bei dem häufigen Fall, dass das Singleton-Pattern sicherstellt, dass nur *eine* Instanz eines bestimmten Objekts angelegt wird. Gerade im Drucker- oder Maschinen-Beispiel sieht man aber sofort, dass die Anforderung fallweise (aber eben nicht so häufig) auch lauten kann: Es darf nur *eine bestimmte Anzahl* von Instanzen einer Klasse einer bestimmten Konfiguration erzeugt werden. Für diesen Spezialfall und andere Details, die zum Teil erheblich komplexer zu implementieren sind, sei auf die Pattern-Fachliteratur verwiesen.

gen genau über einen Kanal protokolliert werden. Auch Komponenten die keinen Status halten wie beispielsweise Data Access Objects (siehe Abschnitt 8.4.5) werden häufig von Komponentenframeworks als Singletons erzeugt und wiederverwendet. Ein Vorteil dieses Patterns ist dann, dass nicht laufend neue Instanzen gebildet und wieder verworfen werden müssen.

Herausforderungen

Die konkrete Implementation des Singleton-Patterns soll am Beispiel der `NotificationManager`-Klasse gezeigt werden. Es soll sichergestellt werden, dass alle Benachrichtigungen und die damit verbundene Ressourcenverwaltung genau an einer Stelle passieren. Um dies zu erreichen, wird eine `NotificationManagerService`-Klasse erstellt, die das Singleton-Pattern implementiert und sich um die Verwaltung der *Notification Manager* kümmert. Die Herausforderung besteht nun darin, eine Klasse zu erstellen, von der nur eine Instanz erstellt werden kann, die dennoch von allen Klassen der Anwendung verwendet werden kann. Zusätzlich sollte die Instanziierung auch Threadsafe sein. Der letzte Punkt ist leider eine im Detail sehr komplexe Angelegenheit, was man an folgender naheliegender und oft verwendeten Implementierung sieht:

```
1  public class NoficationManagerService {
2    private NoficationManagerService() {
3    }
4
5    private static NoficationManagerService
6                   notmanservice = null;
7
8    public static NoficationManagerService
9                  getInstance() {
10     if (notmanservice == null) {
11      notmanservice = new NoficationManagerService();
12     }
13     return notmanservice;
14   }
15   ...
16 }
```

Man erkennt hier einige wichtige Aspekte der Implementierung:

- Die Klasse hat einen `private`-Konstruktor. Dadurch wird verhindert, dass eine Instanz mit dem `new()`-Operator instanziiert wird, also `NoficationManagerService nms = new NoficationManagerService()` ist nicht möglich.
- Da es nur eine Instanz der Klasse geben darf, wird diese in einer statischen Variable in derselben Klasse gespeichert.
- Es gibt eine ebenfalls statische `getInstance()`-Variable, über die man die Instanz beziehen kann.

Lazy Initialisation threadsafe?

Es bleibt die Frage offen, *wann genau* nun die Instanz angelegt wird. Im obigen Codebeispiel wird *lazy initialisation* durchgeführt, d. h., erst beim

ersten Aufruf der Methode wird, falls die Instanzvariable noch `null` ist, die Variable initialisiert. Dies sieht praktisch aus, hat aber den erheblichen Nachteil, dass diese Variante nicht threadsafe ist (und in Java auch meist überflüssig, s. u.).

Es wäre denkbar, dass ein Thread gerade die `getInstance()`-Methode aufruft und die `if`-Abfrage „hinter sich“ hat, dann der nächste Thread zum Zug kommt und denselben Weg geht; in beiden Fällen ist `instance` ja noch `null`. In diesem Fall würden zwei Instanzen von `NoficationManagerService` erstellt werden. Es gibt lange und detailierte Abhandlungen über diese Problematik, z. B. von Mark Townsend [90].

Eine Lösung: Ein Classloader „Trick“

Zum Glück kann man für einfache Singleton-Klassen in Java einen Trick anwenden, der die Implementierung sehr einfach macht und viele potenzielle Probleme von vorneherein vermeidet. In den Beispielen wird eine `NotificationManagerService`-Klasse als Singleton wie folgt implementiert:

```
1  public class NoficationManagerService {
2    private NoficationManagerService() {
3    }
4
5    private static NoficationManagerService
6      notmanservice = new NoficationManagerService();
7
8    public static NoficationManagerService
9                  getInstance() {
10     return notmanservice;
11   }
12   ...
13 }
```

Man erkennt hier weitgehend dasselbe Schema wie in der ersten Implementierung, aber mit einem wichtigen Unterschied: die statische Variable, die die Objektinstanz hält, wird nicht zunächst auf `null` gesetzt und erst in `getInstance()` instanziiert, sondern die Instanziierung wird sofort in der Deklaration durchgeführt. Nun muss man wissen, dass Initialisierungen, die direkt bei der Deklaration von statischen Variablen definiert sind, in Java durchgeführt werden, wenn die Klasse *geladen* wird. Eine Klasse wird vom Classloader bei der ersten Verwendung geladen. Damit verhält sich diese einfachere Variante in den meisten Aspekten gleich der ersten Implementierung und ist zusätzlich threadsafe, was die Instanziierung betrifft.

Komponentenframeworks und Singletons

In modernen Architekturen werden häufig Komponentenframeworks, wie das Springframework, eingesetzt. In diesem Fall erspart man sich oft die Implementierung einer eigenen Singleton-Implementierung. Instanzen und Komponenten, die man vom Komponentenframework erhält, können je nach Konfiguration Singletons sein; darum kümmert sich dann das Framework (siehe auch Kapitel 9).

8.3.2 Factory (Fabrik)

Problemstellung

In manchen Anwendungen kommt es vor, dass man Objekte oder Komponenten benötigt, deren Erstellung (Instanziierung) von vielen Rahmenbedingungen abhängig ist, bzw. die selbst wiederum andere Instanzen benötigen, um korrekt zu funktionieren. Man möchte also eine bestimmte Klasse nicht direkt instanziieren, sondern verwendet eine Hilfsklasse/-methode, die einem die gewünschte Instanz zurückgibt. Im Falle komplexerer Instanziierungen (z. B. Data Access Objects, Datenbank-Zugriffsklassen, XML Parser-Klasse usw.) kümmert sich die sogenannte *Factory* (Fabrik) um die korrekten Initialisierungsschritte.

Zudem kann die Verwendung des Factory-Patterns helfen, Objekte voneinander zu entkoppeln. Die aufrufende Klasse muss dann nicht wissen, welche konkrete Instanz sie von der Factory-Methode zurückbekommt, sondern weiß nur, dass diese Instanz ein bestimmtes Interface implementiert oder von einer bestimmten abstrakten Klasse ableitet.

Mobiltelefon

Verwendet man zum Beispiel sein Mobiltelefon, so ist die Anforderung klar: man möchte eine Verbindung zu einer bestimmten Person bekommen. Das Telefon erwartet eine Telefonnummer. Das Telefon erwartet aber in der Regel *nicht*, dass wir angeben, welchen Mobilfunk-Anbieter wir verwenden wollen, welche Mobilfunk-Sendestation angesprochen oder über welche Leitungen das Gespräch geroutet werden soll. Dies übernimmt die Software im Telefon sowie in den anderen IT-Systemen des Mobilfunk-Anbieters. Sie entscheidet im Ausland, welcher Roaming-Partner verwendet wird, welcher Sendemast gerade geeignet ist und über welche Route die Gesprächsdaten geleitet werden usw. Die Instanziierung des Gespräches ist also im Detail eine komplexe Angelegenheit, die von verschiedensten Rahmenbedingungen abhängig ist.

Beispiel

Dies wird nun an einem konkreten Programm-Beispiel gezeigt, in dem man der Client-Klasse eine solche bequeme Erstellung einer Instanz ermöglichen möchte, und gleichzeitig im Hintergrund noch viele Möglichkeiten hat, die konkreten Instanziierung und Konfiguration zu variieren:

```
1  public class NotificationFactory {
2    ...
3    public static INotification
4                  createNotification(String type)
5    throws ... {
6      // berechne, welche konkrete Klasse für den
7      // gewünschten Notification type
8      // verwendet werden soll
9      ...
10     // Konfiguriere die entsprechende Instanz
11     INotification notification =
12     (INotification)Class.forName(classname).
           newInstance();
13     return notification;
14 }
```

In diesem Beispiel geht es darum, dass in der Anwendung verschiedene Möglichkeiten der Kommunikation mit dem Benutzer möglich sind. Der Benutzer kann z. B. per E-mail, SMS, Instant Messenger (IM) usw. benachrichtigt werden. Nun möchte man sowohl die Art der Benachrichtigungen flexibel gestalten (z. B. könnte man später leicht weitere IM-Formate hinzufügen), d. h., es soll idealerweise erst zur Laufzeit entschieden werden, welche konkrete Klasse für die Benachrichtigung verwendet wird. Andererseits kann auch die Konfiguration je nach Art der Benachrichtigung unterschiedlich sein: E-mail benötigt andere Parameter und Konfigurationsschritte als ein SMS Gateway etc.

In dieser Implementierung sieht man die verschiedenen Aspekte einer einfachen Factory-Implementierung:

- Es gibt ein Interface (`INotification`), das alle Benachrichtigungs-Klassen implementiert haben.
- Die Client-Klassen arbeiten nur mit diesem Interface, nicht mit konkreten Klassen.
- Die Factory-Methode `createNotification` entscheidet anhand des Kontextes (in diesem Fall der `type`-Variable), welche konkrete Implementierung verwendet werden soll.
- Diese Instanz wird in der Factory korrekt konfiguriert und zurückgegeben.

Man erkennt also einerseits, dass es die Factory dem Client einfach macht, mit Benachrichtigungs-Klassen umzugehen, und sie gleichzeitig die Clients von den Benachrichtigungs-Implementierungen entkoppelt; d. h., diese sind leicht austauschbar bzw. veränderbar.

Ein vollständiges Beispiel, das Factory, Interface, Delegation und Proxy beinhaltet, findet sich ab Seite 260.

Abstrakte Factories

Diese Factory-Implementierung ist nur eine einfache Variante des Factory-Patterns. Oft kommt es vor, dass zunächst eine abstrakte Factory-Klasse erstellt wird und dann verschiedene konkrete Factory-Implementierungen. Diese kann verwendet werden, wenn man Instanzen des gleichen Interfaces herstellen möchte, aber mit sehr unterschiedlicher Implementierung.

Ein Beispiel lässt sich im Zusammenhang mit dem Data-Access-Object-Pattern (siehe Abschnitt 8.4.5) finden: Angenommen man möchte in einer Anwendung mehrere Persistenz-Technologien unterstützen, z. B. relationale Datenbanken und XML. Abbildung 8.6 gibt ein Beispiel: Es werden mehrere Data Access Object Interfaces definiert, z. B. `PersonDAO`. Diese werden dann für verschiedene konkrete Persistenz-Methoden implementiert, z. B. für eine relationale Datenbank und für XML. Danach definiert man eine abstrakte `DAOFactory` sowie für jede konkrete Persistenztechnologie eine konkrete Factory. Nun kann zur Laufzeit entschieden werden,

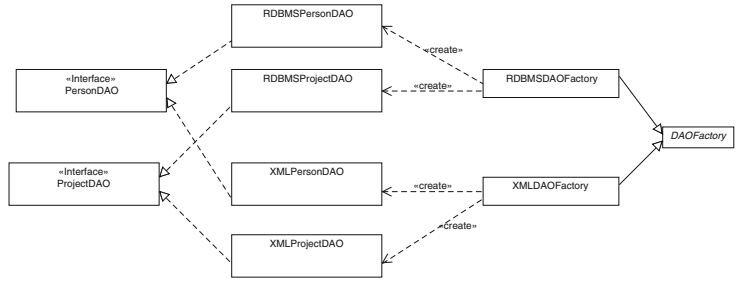


Abbildung 8.6 *Beispiel für abstrakte Factory: Es sollen verschiedene Persistenz-Technologien parallel unterstützt werden.*

welche Factory dem Client-Objekt zur Verfügung gestellt wird, je nachdem welche Persistenz-Technologie gerade verwendet werden soll. Für den Client ist dies transparent.

8.3.3 Objekt-Pool

Das Objekt-Pool-Pattern soll an dieser Stelle nur kurz in Kontext und Verwendung beschrieben, aber keine konkrete Implementierung besprochen werden. Der Grund ist, dass die korrekte Implementierung eines Objekt-Pools nicht trivial ist, und dass genügend (Open-Source-) Implementierungen für bestimmte Anwendungsbereiche vorhanden sind.

Problemstellung

Hat man das Problem, dass bestimmte Objekte häufig benötigt werden, die Erstellung der Objekte aber „teuer“ ist, so möchte man häufig erstellte Objekte wiederverwenden und nicht nach kurzer Verwendung „entsorgen“. Am einfachsten lässt sich das anhand von Datenbank-Verbindungen erklären: Entwickelt man z. B. eine Web-Anwendung, so können viele Benutzer gleichzeitig mit der Anwendung interagieren. Je nach Anfrage, die der Benutzer stellt, muss z. B. eine Datenbankverbindung hergestellt werden. Nun könnte man natürlich für jeden dieser Zugriffe ein entsprechendes *Connection*-Objekt instanziieren, verwenden und wieder freigeben. Dies funktioniert zwar, ist aber sehr ineffizient, weil das Erstellen einer Connection relativ zeit- und ressourcenintensiv ist.

Z. B. JDBC Connection Pools

In der Praxis versucht man daher solche Objekte wiederzuverwenden. D. h., man verwendet einen sogenannten Objekt-Pool oder in diesem konkreten Beispiel einen JDBC Connection Pool. Dieser Pool macht im Prinzip nichts anderes, als eine bestimmte Anzahl an instanziierten Connections offen zu halten: Wenn ein Objekt eine Connection benötigt, holt es sich eine Instanz vom Pool, benötigt es diese nicht mehr, gibt es sie an den Pool zurück.

Die Implementierung?

Das Prinzip ist einfach, die Implementierung aber im Detail schwierig; man sollte sich um „verloren gegangene“ Objekte kümmern, den Pool dynamisch vergrößern und verkleinern, Connections am Leben halten usw. Zum Glück muss man derartige Pools in den seltensten Fällen selbst schreiben. Oft sind sie schon Teil der Laufzeitumgebungen, oder man kann auf entsprechende Open-Source-Implementierungen zurückgreifen (als Beispiel

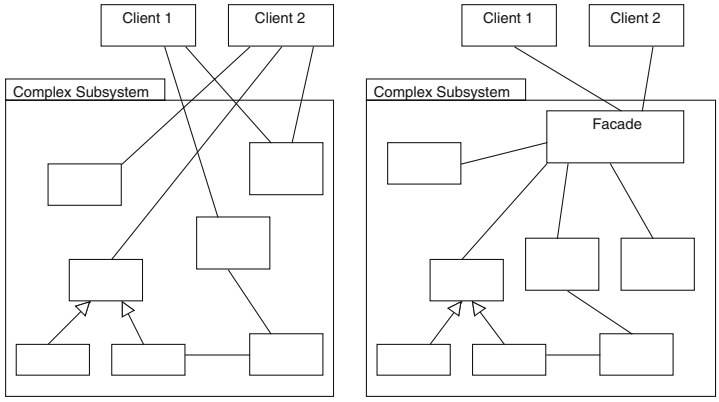


Abbildung 8.7 *Das Fassade-Pattern wird angewandt, um einen „vereinfachten“ Zugang zu komplexen Subsystemen zu ermöglichen.*

sei Apache dbcp genannt[8]). Wichtig ist aber die korrekte Interaktion mit dem Pool, also z. B. nicht zu vergessen, nicht mehr verwendete Objekte wieder an den Pool zurückzugeben.

8.4 Struktur

8.4.1 Fassade (Facade)

Problemstellung

Bei heutigen Software-Projekten muss häufig mit komplexen Bibliotheken oder Frameworks interagiert werden. Als Beispiel könnte man sich eine Server-Anwendung vorstellen, die (wie im Beispiel dieses Kapitels) Benachrichtigungen per E-mail versenden möchte. Zu diesem Zweck könnte man Suns JavaMail Bibliothek [9] verwenden. Sieht man sich nun die API-Dokumentation an, so stellt man fest, dass JavaMail in der Version 1.4 über mehr als 120 Klassen und Interfaces verfügt!

Nun möchte man nur eine einfache E-mail mit Betreff und textuellem Inhalt versenden und muss sich mit 120 Klassen auseinandersetzen, die alle möglichen und unmöglichen E-mail-Optionen abdecken; von IMAP Folder bis zu MIME-Multipart-Nachrichten. Nehmen wir weiter an, es arbeiten viele Entwickler an diesem Projekt und diese einfache E-mail-Benachrichtigungsfunktion wird von einigen Entwicklern benötigt.

Die Lösung: Eine Fassade!

In diesem Fall würde es sich anbieten, eine oder mehrere einfache „Komfort-Methoden“ zu schreiben, die man sich als Zwischenschicht zwischen dem eigenen Code und der JavaMail-API vorstellen kann:

[8] `http://commons.apache.org/dbcp/`
[9] `http://java.sun.com/products/javamail`

```
class SimpleMail {
  public static void sendMail(String address,
                              String subject,
                              String body) {
    // hier wird der Zugriff auf die JavaMail API
    // implementiert und mit verschiedenen
    // Klassen dieser API interagiert.
  }
  public static void sendMail(List addresses,
                              String subject,
                              String body) {
    ...
  }
  ...
}
```

Abbildung 8.7 verdeutlicht die Idee etwas verallgemeinert. Der linke Teil der Abbildung entspricht der direkten Interaktion mit der JavaMail (oder irgendeiner anderen komplexen) API. Jeder Client muss sich mit der Funktionalität der API auseinandersetzen. Im rechten Bild hätte ein Entwickler eine Hilfsklasse geschrieben, über die ein vereinfachter Zugriff auf dieses Subsystem ermöglicht wird. Die Möglichkeiten sind dann natürlich eingeschränkt, aber für die abgebildeten Spezialfälle ist die Arbeit mit dem Subsystem wesentlich einfacher geworden. Die Clients interagieren in diesem Fall nur mehr mit der einfachen Fassade-Klasse.

Verwandte Muster

Das Fassade-Pattern kann helfen, eine Anwendung in Schichten aufzubauen (siehe auch Abschnitt 7.5). Es unterscheidet sich vom Adapter-Pattern vor allem in der Intention: Das Fassade-Pattern soll den Zugriff auf bestimmte Subsysteme einfacher gestalten. Das Adapter-Pattern hingegen versucht, zwischen unterschiedlichen Interfaces, die aber denselben Zweck verfolgen, zu vermitteln.

8.4.2 Iterator

Kontext: Datenstrukturen

Je nach Anwendungsfall werden zur Verwaltung von Daten in Anwendungen unterschiedliche Datenstrukturen verwendet. Häufig verwendete Datenstrukturen sind verkettete Listen, (mehrdimensionale) Arrays, Hashmaps, Queues[10] usw.

Iterieren von Datenstrukturen

Das Iterator-Pattern erlaubt es nun auf eine einheitliche Weise (mittels eines gemeinsamen Interfaces), durch Datenstrukturen zu iterieren, ohne dass die darunterliegende Datenstruktur nach außen sichtbar wird. Das

[10] Moderene Programmierumgebungen bieten bereits umfangreiche Sammlungen von Datenstrukturen an, die man direkt verwenden kann ohne selbst Hashmaps oder verkettete Listen programmieren zu müssen. Im Fall von Java findet man diese im Package `java.util`.

Iterator-Pattern muss meist nicht extra implementiert werden, sondern ist heute eine Grundfunktion moderner Sprachen. Werden in einem Algorithmus beispielsweise zwei verschiedene Datenstrukturen verwendet, eine `LinkedList l` und eine `Queue q`, so können diese dennoch auf eine einheitliche Weise iteriert werden:

```
1 List<String> movies = new ArrayList<String>();
2 Queue<String> steps = new LinkedList<String>();
3 ... // Daten hinzufügen
4 Iterator<String> it = movies.iterator();
5 while (it.hasNext()) {...}
6 it = steps.iterator();
7 while (it.hasNext()) {...}
```

Iteratoren hinter die Kulissen!

In modernen Sprachen wie z. B. Python oder Java muss das Iterator-Pattern nicht mehr, wie oben gezeigt, explizit ausprogrammiert werden, sondern es werden erweiterte `for`-Befehle angeboten (auch for-each genannt), in Java:

```
1 for(Element e : myList)
```

Wobei `myList` ein Objekt sein muss, das das `java.lang.Iterable`-Interface implementiert. Diese Variante ist offensichtlich leichter zu schreiben und auch einfacher zu verstehen und sollte daher der ersten Variante vorgezogen werden. „Hinter den Kulissen“ wird aber weiterhin das *Iterator-Pattern* angewendet.

8.4.3 Adapter

Problem: Integration von nicht veränderbaren Klassen

Häufig möchte man Funktionalität in der eigenen Anwendung nutzen, die von externen Bibliotheken oder Klassen zur Verfügung gestellt wird, auf die man keinen direkten Zugriff hat oder deren Sourcecode man nicht verändern möchte. Dies können z. B. Bibliotheken sein (z. B. JavaMail) oder auch Funktionalität, die von proprietären Systemen zur Verfügung gestellt wird. Diese Bibliotheken verfügen manchmal über eine API, die sich nicht gut in die Architektur des eigenen Systems integrieren lässt. In einem solchen Fall kann man die Anbindung über eine Adapter-Klasse gestalten. Konkret gesagt möchte man in solchen Fällen fremde Klassen über eigene Interfaces ansprechen.

Eigenes Interface für fremde Klassen

Im Beispiel wurde ein Benachrichtigungs-Interface definiert (`INotification`, siehe Seite 241). Angenommen man hätte eine Bibliothek zur Verfügung, die in der Lage ist, aus Text Sprache zu erzeugen und die mit einer Telefonie-Software eine angegebene Telefonnummer anruft und dem zu Benachrichtigenden die Nachricht vorliest. Dies wäre ein eigenes Software-Produkt, das zugekauft wurde und sich über eine bestimmte API steuern lässt, die z. B. wie folgt aussieht:

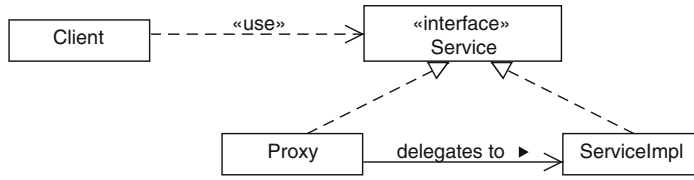


Abbildung 8.8
Grundidee: Service-Klasse und Proxy implementieren das gleiche Interface. Der Proxy kann als Stellvertreter für das Service verwendet werden.

```
public class VoiceCall {
  public String callNr(String phoneNr,
                       String text) {
    ...
    // an dieser Stelle würde die Bibliothek
    // den Anruf durchführen und die
    // Sprachausgabe erstellen
    ...
    return status;
  }
```

Die Adapter-Klasse

Diese Klasse ist nun nicht direkt zugänglich, weil man sie nicht verändern möchte, kann oder darf. Im eigenen Projekt sollen aber alle Benachrichtigungsklassen das `INotification`-Interface implementieren, um sich gut in die Architektur (Factory usw.) zu integrieren. Daher kann man den Weg über eine Adapterklasse gehen:

```
public class VoiceCallNotificationAdapter
implements INotification {
  private Person receiver = null;
  public String send(String content) {
    VoiceCall vc = new VoiceCall();
    String status =
    vc.callNr(receiver.getTelephoneNr(), content);
    return status;
  }
  public void setPerson(Person person) {
    receiver = person;
  }
}
```

Diese Adapterklasse implementiert das gewünschte Interface, lässt sich also in die eigene Architektur gut integrieren, delegiert aber die konkrete Funktionalität nur an die `VoiceCall`-Klasse weiter.

8.4.4 Proxy (Stellvertreter)

Struktur des Pattern

Das Proxy-Pattern ist leichter von der Struktur des Patterns her zu erklären und zu verstehen. Abbildung 8.8 zeigt die Idee des Proxy-Patterns:

Es gibt ein Service Interface `Service`: dieses definiert die Schnittstelle für eine bestimmte Funktionalität, die ein oder mehrere Client-Objekte benötigen. Dann gibt es noch eine konkrete Implementierung dieses Interfaces, `ServiceImpl`, die die Funktionalität des Interfaces implementiert. An dieser Stelle entspricht das einfach dem Delegation-Pattern ähnlich wie in Abbildung 8.5 gezeigt. D. h., das Client-Objekt könnte direkt das Service-Objekt verwenden, wie im Delegation-Pattern erklärt.

Das Proxy-Pattern ist eine Erweiterung dieser Idee: Die `Proxy`-Klasse implementiert dasselbe Interface wie die Service-Klasse also `Service`, stellt aber im Gegensatz zum Delegation-Pattern nicht die volle Service-Funktionalität wie vom Interface vorgesehen zur Verfügung, sondern reicht die Anfragen an das `ServiceImpl`-Objekt weiter, hat aber die Möglichkeit, vor oder nach dem Weiterreichen der Anfragen an das eigentliche Service-Objekt weitere Aktionen durchzuführen.

Der Stellvertreter

Der Proxy ist also ein Stellvertreter und verhält sich nach außen hin (für den Client) so, als wäre er das eigentliche Service, verwendet aber das eigentliche Service-Objekt, um Anfragen zu beantworten. Interessant an diesem Pattern ist, dass die Client-Klasse gar nicht wissen muss, ob sie mit einer *konkreten* Implementierung oder mit einem *Proxy* spricht. Dies ist besonders dann sehr flexibel, wenn man das Proxy-Pattern mit dem Factory-Pattern verbindet (wie im konkreten Beispiel unten gezeigt werden wird). Es gibt viele verschiedene Anwendungsbereiche, in denen das Proxy-Pattern eingesetzt werden kann, beispielsweise:

Beispiel: Cache

Die Service-Implementierung ist auf einem anderen Rechner. Greift ein Client auf das Service zu, muss also ein (teurer) Remote Procedure Call durchgeführt werden. Um zu vermeiden, dass *jeder* Zugriff auf das Service tatsächlich über das Netzwerk geht, kann man eine Proxy-Klasse schreiben, die als Cache dient. Die Client-Objekte interagieren dann tatsächlich mit dem Proxy und nicht mit dem Service. Der Proxy speichert Anfragen und Ergebnisse im lokalen Speicher und greift (je nach Cache-Strategie) z. B. bei gleichen Anfragen in einem bestimmten Zeitraum gar nicht auf das eigentliche Service zurück, sondern gibt sofort die im Proxy zwischengespeicherten Ergebnisse an den Client zurück. Dies kann die Anfragen beschleunigen und die Last auf das eigentliche Service reduzieren.

Beispiel: Zugriffsschutz

In vielen Geschäftsanwendungen gibt es klare Richtlinien, welche Objekte unter welchen Voraussetzungen mit bestimmten Services interagieren dürfen (z. B. abhängig davon, welche Person gerade eingeloggt ist). Nun könnte man diesen Zugriffsschutz direkt ins Service-Objekt einbauen, d. h., die Service-Klasse prüft, ob das Client-Objekt das Recht hat, auf das Service zuzugreifen. Dies ist in der Praxis nicht empfehlenswert. Einerseits werden zwei verschiedene Aspekte vermischt: die *Geschäftslogik* im Service und die *Sicherheitslogik*. Außerdem wird es in vielen Fällen mehrere Service-Objekte geben, für die ähnliche Sicherheitskriterien (access control lists usw.) existieren. Es macht also Sinn, Zugriffsprüfungen an einer zentralen Stelle zu implementieren und wiederzuverwenden. Ein weiterer wichtiger

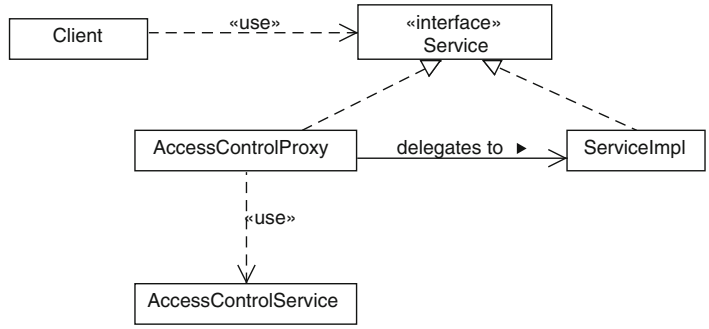


Abbildung 8.9
Einfaches Beispiel für Zugriffskontrolle mithilfe des Proxy-Patterns.

Punkt ist, dass häufig unterschiedliche Entwickler(gruppen) für die Implementierung von Geschäftslogik und Services und für Security zuständig sind. Eine Vermischung beider Aspekte in einer Klasse macht also die Entwicklung und Wartung schwieriger und fehleranfälliger.

Auch in diesem Fall kann man sich leicht mit dem Proxy-Pattern helfen. Abbildung 8.9 illustriert diese Idee: Die eigentliche Service-Klasse implementiert „nur" die Geschäftslogik und keinerlei Security-Mechanismen. Diese werden über den `AccessControlProxy` eingeführt. Der konkrete Ablauf sieht dann so aus:

- Die Clients sprechen nie direkt mit der `ServiceImpl`-Klasse wenn Zugriffsschutz erwünscht ist.
- Stattdessen verwenden die Clients die `AccessControlProxy`-Klasse über das `Service` Interface; die Client-Objekte müssen also gar nicht mitbekommen, dass sie de facto mit einem Proxy interagieren!
- Möchte nun ein Client eine Methode im Service aufrufen, so greift er tatsächlich zunächst auf diese Methode im Proxy zu.
- Der Proxy prüft nun, ob dieser Client überhaupt berechtigt ist, mit dem Service zu interagieren. Dabei kann er sich z. B. eines weiteren Services (`AccessControlService`) bedienen. Dieser Service kann die Security-Logik für viele verschiedene Interaktionen implementieren und daher wiederverwendet werden.
- Ist der Client berechtigt zuzugreifen, so leitet der `AccessControlProxy` die Anfrage nun unverändert an das Service weiter und gibt das Ergebnis ebenso unverändert an den Client zurück. Dieser hat also von dieser Überprüfung nichts mitbekommen.
- Ist der Client aber *nicht berechtigt* zuzugreifen, so reicht der `AccessControlProxy` die Anfrage natürlich nicht weiter, sondern löst stattdessen beispielsweise eine Security Exception aus.

Beispiel: Protokollierung von Zugriffen

Ein weiteres Beispiel ist die Möglichkeit, einen Proxy einzuführen, um Zugriffe mitzuprotokollieren. In den Patterns-Beispielen wird dies veranschaulicht. Wir hatten das Problem, dass das Programm Benachrichtungen auf verschiedenen Wegen (z. B. E-mail, SMS, Telefon) an den Benutzer senden können soll. Zu diesem Zweck wurde das `INotification`-Interface definiert:

```
1 public interface INotification {
2   public void setPerson (Person person);
3   public String send (String content);
4 }
```

Dieses Interface definiert vor allem eine generische `send- (String content)` Methode. Verschiedene Services wie `Email` und `Sms` implementieren dieses Interface.

Zudem gibt es eine Factory-Klasse `NotificationFactory`. Diese Factory liest aus einer Konfigurationsdatei aus, welcher Benutzer welche Form der Benachrichtigung wünscht und instanziiert dann die entsprechende Klasse (E-mail, SMS, Voice). Bis zu diesem Punkt wurde das Beispiel bereits in den vorigen Abschnitten Delegation (siehe Seite 236) sowie Factory (siehe Seite 250) erklärt.

Nun kommt die weitere Anforderung hinzu, dass es in manchen Fällen gewünscht ist, dass mitprotokolliert wird, wenn Benachrichtigungen an Benutzer versandt werden (z. B. um das später nachvollziehen zu können). Natürlich könnte man diese Funktionalität in die Benachrichtigungs-Klassen (E-mail, SMS, ...) einbauen, aber auch hier gilt: Das wäre wenig elegant, denn die Protokoll-Funktionalität und die Notifikations-Funktionalität haben eigentlich wenig miteinander zu tun. Die Protokoll-Funktionalität ist außerdem für alle Arten der Notifikation gleich und sollte somit wiederverwendet werden. Die Protokollierung ist nicht in allen Fällen gewünscht, muss also konfigurierbar sein. Im Beispiel wird das daher mit dem Proxy-Pattern gelöst und zunächst eine Proxy-Klasse implementiert:

```
1  public class RecordNotificationProxy
2  implements INotification {
3    private INotification notify;
4    public RecordNotificationProxy(
5            INotification notify) {
6      this.notify = notify;
7    }
8    public String send(String content) {
9      System.out.println("Proxy Interception:
10             forwarding '"
11             + content + "'");
12     return notify.send(content);
13   }
14   public void setPerson(Person person) {
15     notify.setPerson(person);
16   }
17 }
```

`RecordNotificationProxy` implementiert genau wie die `Email`- und `Sms`-Klasse das `INotification`-Interface. Im Konstruktor wird aber erwartet, dass eine Referenz auf eine *andere* `INotification`-Implementierung übergeben wird. Dies ist notwendig, weil es sich bei dieser Klasse ja nur um den Proxy handelt, der aber eine konkrete Implementierung des Interfaces benötigt, um die eigentliche Funktionalität der Notifikation durchführen zu können. Dies erkennt man sofort in der `setPerson()`-Methode, diese wird direkt an die konkrete Notifikations-Implementierung weitergereicht.

Wirklich interessant an diesem Proxy ist eigentlich nur die `send()`-Methode: Hier wird in der letzten Zeile wieder der Zugriff weitergereicht, aber zuvor der Inhalt der Nachricht auf die Konsole ausgegeben. Dies ist nur ein einfaches Beispiel. In einer realen Implementierung würde man natürlich in ein Logging Framework, eine Datei oder eine Datenbank protokollieren.

Beispiel: Proxy und Factory

Der letzte interessante Schritt in diesem Beispiel ist es, die Konfiguration für die Factory in einer Konfigurationsdatei zu halten. Damit kann über diese Konfiguration das Verhalten der Factory gesteuert werden. Es können beliebige weitere Notifikationsmethoden hinzugefügt werden. Die entsprechende Implementierung (Klasse) muss nur im Klassenpfad zu finden sein, und in der Konfigurationsdatei eingetragen werden. Im Sourcecode der Factory selbst sind keine Änderungen notwendig!

```
SMS, at...Sms, record_on
Email, at...Email, record_off
Voice, at...VoiceCallNotificationAdapter,
            record_off
```

Jede Zeile definiert eine Benachrichtigungsmethode, zunächst eine *Kennung* (SMS, E-mail, Voice), danach die *konkrete Klasse*, die verwendet werden soll (diese Klasse muss `INotification` implementieren) und zum Schluss einen Parameter, der angibt, ob die entsprechende Kommunikation *protokolliert* werden soll oder nicht. Für den Fall, dass keine Protokollierung notwendig ist, verhält sich die Factory wie in Abschnitt 8.3.2 beschrieben, es wird also je nach Benachrichtigungswunsch eine Instanz der Klasse `Email`, `Sms`, `VoiceCallNotificationAdapter` erstellt und an das Client-Objekt zurückgegeben.

Für den Fall, dass protokolliert werden soll, wird zusätzlich eine Instanz von `RecordNotificationProxy` erstellt und die konkrete Benachrichtigungsklasse dem Proxy übergeben. Die Kommunikation läuft dann

immer über den Proxy, ohne dass die Client-Objekte das merken. In der Factory-Klasse stellt sich das hier etwas umfangreicher wie folgt dar[11]:

```
public class NotificationFactory {
...
  private static NotificationFactory
    notificationFactory = new NotificationFactory();
  private HashMap<String, String>
    notificationClasses
            = new HashMap<String, String>();
  private HashMap<String, Boolean>
    notificationRecord
            = new HashMap<String, Boolean>();

  private void readCfg() throws IOException {
    // read config file
    ...
    // fill notificationClasses map with all classes
    // fill notificationRecord map with true/false
    ...
  }

  public static INotification
        createNotification(String type) throws ... {
    ...
    String classname
      = notificationFactory.notificationClasses
        .get(type);
    INotification notification
      = (INotification)Class.forName(classname)
        .newInstance();
    boolean doRecord
      = notificationFactory.notificationRecord
        .get(type);
    if (doRecord) {
      return new RecordNotificationProxy (
          notification);
    }
  return notification;
  }
...
}
```

Relevant ist hier die `createNotification()`-Methode: In dieser Methode werden zwei Hashmaps verwendet: `notificationClasses` und `notificationRecord`, die Daten aus der Konfigurationsdatei repräsentieren. Beide haben als Schlüssel die Kennung der gewünschte Methode, also im Beispiel „SMS“ oder „E-mail“ oder „Voice“. Die erste speichert

[11] Dieses Beispiel findet sich im Detail ausgeführt bei den Design-Patterns Beispielen im Web.

als Wert den *Namen der Klasse* inklusive Package, die die Benachrichtigungs-Funktionalität implementiert und die zweite Hashmap einen *Boolean*-Wert der entscheidet, ob die Kommunikation protokolliert werden soll oder nicht.

Folglich wird im ersten Schritt der `classname`, also der Name der benötigten Service-Klasse, ermittelt, danach diese Klasse instanziiert und der `notification`-Variable (vom Typ `INotification`!) übergeben. Danach wird überprüft, ob ein Protokollieren erforderlich ist. Falls nicht, wird direkt die `notification`-Variable zurückgegeben. Falls Protokollierung erforderlich ist, wird zusätzlich der Proxy `RecordNotificationProxy` instanziiert und diesem die Referenz auf die Service-Instanz `notification` übergeben. Zurückgegeben wird dann natürlich die Proxy-Instanz.

Mit dieser Vorgehensweise kann auch erst bei der Konfiguration oder gar erst zur Laufzeit entschieden werden, ob die Client-Objekte nun direkt mit der Implementierung oder über einen Proxy kommunizieren.

Kaskadierung

Das Proxy-Pattern kann natürlich auch kaskadiert eingesetzt werden. Um die obigen Beispiele aufzugreifen: Es wäre denkbar, mehrere Implementierungen der Benachrichtigung zu haben (E-mail, SMS, ...) sowie einen Proxy für Protokollierung und Logging, einen anderen Proxy zum Cachen (z. B. um mehrere Nachrichten zusammenkommen zu lassen, bevor gesendet wird) und einen anderen Proxy, der Zugriffsrechte implementiert. Diese können dann je nach Konfiguration verkettet eingesetzt werden. E-mail z. B. wird direkt versandt und nicht protokolliert, SMS wird gecached, protokolliert sowie Zugriffsschutz angewendet usw.

Verwandte Patterns

Das Proxy-Pattern ist eine Erweiterung des Delegation-Patterns. Außerdem macht es in der Praxis Sinn, das Proxy-Pattern mit einem Factory-Pattern, oder einem Komponentenframework mit Dependency Injection zu kombinieren. Sind wie im Beispiel Client und Service nicht „hart verdrahtet", sondern über ein Factory-Pattern erzeugt, so kann man auch später per Konfiguration noch Proxies für bestimmte Anwendungsfälle einführen, ohne dass die Client-Objekte davon etwas mitbekommen. Dies ist also eine sehr flexible Möglichkeit, neue Funktionalität einzuführen.

Weiterhin ist das Proxy-Pattern mit dem Decorator-Pattern (siehe Abschnitt 8.5.2) sowie dem Interceptor-Pattern (siehe Abschnitt 8.5.3) verwandt. Besonders das Interceptor-Pattern wird ähnlich implementiert, hat aber einen etwas anderen Anwendungsfall.

Das Proxy-Pattern ist auch de facto die Basis für verschiedene andere Architekturstile, z. B. der aspektorientierten Programmierung (siehe Abschnitt 9.5) oder für Frameworks, z. B. Cache-Implementierungen.

8.4.5 Data Access Object

Problemstellung

In den meisten modernen Anwendungen werden Daten nicht nur im RAM gehalten, sondern es besteht die Anforderung, Daten persistent zu halten sowie auf persistierte Daten zuzugreifen. Dazu können eine große Zahl verschiedener Persistenzmechanismen eingesetzt werden. Häufig verwendet werden:

- Relationale Datenbanken (verschiedener Hersteller bzw. verschiedenste Open-Source-Produkte[12]).
- XML-Dateien (mit unterschiedlichen Zugriffsmechanismen und Abfragemöglichkeiten).
- Binäre (proprietäre) Datenformate, z. B. Office-Dateien oder Bitmap-Grafiken.
- Webservices.

Selbst wenn man sich für eine bestimmte Persistenztechnologie entschieden hat, ist es aber wenig empfehlenswert, Persistenzcode und Geschäftslogik zu vermischen. Folgendes sollte man beispielsweise vermeiden (das Beispiel ist *Pseudocode* und dient nur der Illustration der Problematik):

```
1  double calculateBill (Customer cust) {
2    dbt = accessDB("SELECT TAX FROM TAXES WHERE ...");
3    double tax = dbt.getInt ("tax");
4    dbt
5      = accessDB("SELECT PRICE FROM ORDER WHERE ...");
6    double sum = 0;
7    for (...dbt...) {
8      sum += dbt.getDouble("price");
9    }
10   sum = sum * tax;
11   return sum;
12 }
```

An diesem Pseudocode-Beispiel lässt sich die Problematik einer solchen Vermischung schon leicht erkennen:

- Geschäftlogik (Berechnungen) und Zugriffslogik sind vermischt, werden aber häufig von unterschiedlichen Entwicklern programmiert und gewartet.
- SQL-Befehle sind über die ganze Anwendung verstreut und sehr schwer wartbar (z. B. wenn sich die Datenbank ändert).

[12] Im Produktivbetrieb werden gerne PostgreSQL oder MySQL verwendet, im Java-Umfeld sind hsqldb und Apache Derby sehr verbreitet.

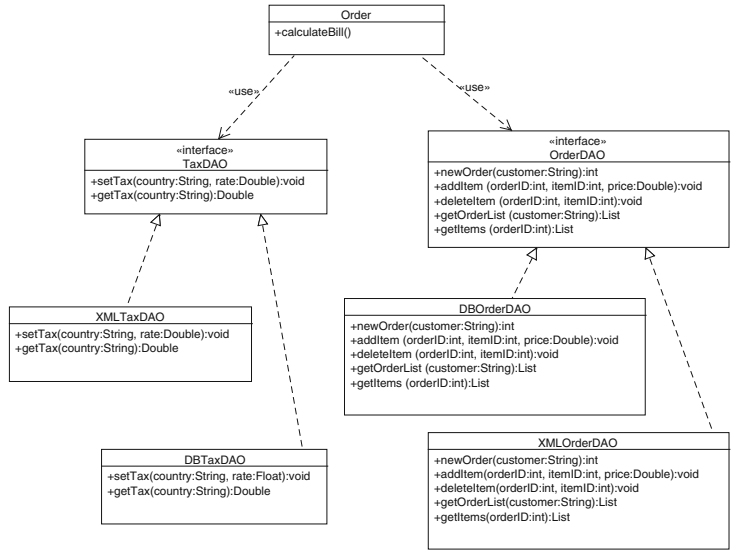


Abbildung 8.10 *Das DAO-Pattern erlaubt die saubere Trennung von Geschäftslogik und Persistenzcode. In diesem Beispiel gibt es zwei Implementierungen der DAO-Interfaces; eine für eine relationale Datenbank, eine für XML-Dateien.*

- Die Handhabung von Datenbankverbindungen und der Transaktionssteuerung wird deutlich erschwert, besonders wenn die Transaktion über mehrere Schritte erfolgt und nicht nur in einem (SQL) Statement.

- In diesem Beispiel gibt es noch überhaupt keine Fehlerbehandlung; würde man diese hinzufügen, wird es noch unübersichtlicher.
- Der Austausch der Persistenzschicht (z. B. zu XML-Persistenz oder einer anderen Datenbank) ist in einem solchen Fall sehr kompliziert und erfordert händischen Eingriff in viele Klassen.
- Das Testen wird erschwert, weil man Fehlerklassen nicht leicht trennen kann (warum ist die Berechnung falsch? Kein Datenbankzugriff? Falsche Testdaten in der Datenbank? Richtige Testdaten, aber ein Fehler in der Berechnung?).
- Konfiguration und Skalierung der Anwendung werden schwieriger.

Lösung

Die Konsequenz ist klar, in modernen Anwendungen versucht man *Geschäftslogik* und *Persistenzlogik* sauber voneinander zu trennen. Dabei helfen Ansätze wie eine Schichtenarchitektur (siehe auch Abschnitt 6.2.2 (Modellierung) sowie Abschnitt 6.2.2 (Architektur)) sowie der Einsatz des *Data-Access-Object*-Patterns (DAO), am besten in Kombination mit einem Komponentenframework und Dependency Injection (siehe Abschnitt 9.2 ab Seite 315). In diesem Abschnitt wird zunächst das DAO-Pattern erklärt.

DAO-Pattern-Idee

Die grundlegende Idee des DAO-Patterns ist in Abbildung 8.10 ersichtlich: Die `Order`-Klasse enthält ausschließlich Geschäftslogik, also z. B.

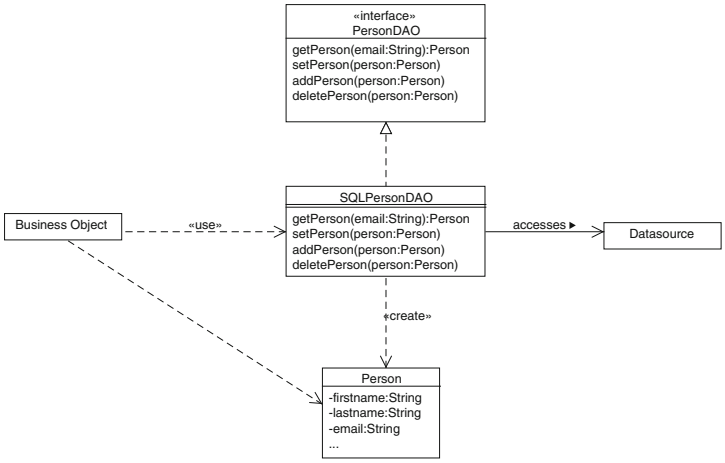


Abbildung 8.11
DAO-Pattern im Detail: Speichern/Laden von Domänenobjekten. Zugriff auf Datenquelle und Transfer-Object-Pattern.

die Methode zum Berechnen der Rechnung eines Kunden. Wichtig ist dabei, dass sich in Geschäftsmethoden *keinerlei* direkte Interaktion mit Persistenztechnologien wie relationalen Datenbanken (wie im obigen Beispiel) findet. Um Daten (Domänenobjekte) zu persistieren, definiert man Data Access Object Interfaces, wie in diesem Beispiel `TaxDAO` und `OrderDAO`. In diesen Interfaces definiert man möglichst generische „neutrale" Methoden, die zum Daten speichern, laden oder suchen verwendet werden können. Man erkennt z. B. im `OrderDAO`-Interface Methoden wie `addItem(...)` oder `getItems(...)`, mit denen neue Items gespeichert oder geladen werden können, die aber keine bestimmte Persistenztechnologie voraussetzen.

Im Beispiel erkennt man auch, dass beide Interfaces in jeweils zwei Varianten implementiert werden, z. B. `DBOrderDAO` würde in eine relationale Datenbank persistieren und `XMLOrderDAO` in ein XML File[13].

In Abbildung 8.11 wird das gesamte Szenario wie ein Domänenobjekt dargestellt, wobei `Person` mittels DAO-Pattern persistiert oder aus der Datenbank geladen wird. Man erkennt links wieder ein Geschäftsobjekt, das mit dem Data Access Object `SQLPersonDAO` interagiert, um das Domänenobjekt `Person`[14] zu laden oder zu speichern. Das `SQLPersonDAO`-Objekt ist dabei wieder nur *eine mögliche* Implementierung des `PersonDAO`-Interfaces. Diese Implementierung verwendet

[13] Es wäre ebenso denkbar, dass z. B. `OrderDAO` mit einer relationalen Datenbank arbeitet, während `TaxDAO` auf ein Webservice zugreift.

[14] Das `Person`-Objekt wird aus Sicht des DAO-Patterns auch als Transfer-Objekt bezeichnet, weil es dazu dient, die Daten zwischen Geschäftslogik und Persistenzlogik zu transportieren.

z. B. eine relationale Datenbank und greift auf diese über eine (JDBC-) `Datasource` zu.

Separation of Concerns

Auch am zweiten Beispiel erkennt man deutlich, dass die Zuständigkeiten nun sauber getrennt sind (siehe auch Abschnitt 7.4). Zusammenfassend kann man sagen:

- Das DAO-Interface definiert die benötigten Zugriffsmethoden auf die Persistenzschicht in einer möglichst technologieneutralen Weise.
- Die Implementierung des DAO-Interfaces stellt die Verbindung zu einer konkreten Persistenztechnologie her.
- Die Persistenz-Schicht kann weitgehend entkoppelt von der Geschäftslogik (evt. auch von einem eigenen Team) entwickelt und gewartet werden.
- Die Entwickler, die an der Geschäftslogik arbeiten, müssen keine Details der Persistenztechnologie kennen und arbeiten immer nur gegen das DAO-Interface.
- Die konkrete „Verdrahtung" der DAO-Implementierungen mit der Geschäftslogik kann leicht mit einem Komponentenframework und Dependency Injection erledigt werden (siehe Abschnitt 9.3).
- Sowohl Persistenzschicht als auch Geschäftslogik können leichter getrennt voneinander getestet werden (z. B. mit Unit-Tests und Mockup-Objekten, siehe Kapitel 5).

DAO als Singleton

Data Access Objects werden auch häufig als Singletons (siehe Abschnitt 8.3.1) eingesetzt. Einer der Gründe ist, dass DAOs sehr häufig und an verschiedenen Stellen in einer Applikation verwendet werden. Durch die Verwendung des Singleton-Patterns kann man verhindern, dass laufend neue DAO-Objekte erstellt und wieder verworfen werden. Verwendet man ein Komponentenframework wie Spring, so kümmert sich das Framework um den Lifecyle der Objekte und auch um die Implementierung des Singleton-Patterns (siehe auch Abschnitt 9.2). Allerdings ist zu beachten, dass die DAO-Implementierung möglichst keine Instanzvariablen deklarieren sollte, um Synchronisationsprobleme zu vermeiden, falls das DAO in einer Umgebung mit mehreren Threads arbeitet.

Weitere verwandte Patterns

Das DAO-Pattern beruht auf grundlegenderen Patterns, wie Interface und Delegation, die am Beginn dieses Kapitels erklärt wurden. Es ist zu beachten, dass das DAO-Pattern, wie es in diesem Abschnitt eingeführt wurde, nur die elementare Idee darstellt. In komplexeren Fällen kann man das sogenannte *generische DAO*-Pattern, das im nächsten Abschnitt beschrieben ist, einsetzen. Das DAO-Pattern wird auch meist mit dem Dependency-Injection-Pattern sowie mit Komponentenframeworks wie dem Springframework eingesetzt, siehe Abschnitt 9.3. Das DAO-Pattern kann alternativ zu einem Komponentenframework auch mit dem Abstract-Factory-Pattern kombiniert werden (siehe Abschnitt 8.3.2).

8.4.6 Generisches DAO

Problemstellung

Im vorigen Abschnitt wurde die grundsätzliche Idee des Data-Access-Object-Pattern vorgestellt. In vielen Projekten stellt man aber fest, dass man eine größere Anzahl von Domänenobjekten persistieren möchte, und dabei häufig dieselben Methoden mehrfach implementiert werden müssen. Dies soll an einem Beispiel illustriert werden:

- `Person`: Dies ist die Basisklasse, in der Personenstammdaten gespeichert werden.
- `Professor`: Dieser Klasse wird von `Person` abgeleitet und speichert zusätzliche, für Professoren wichtige Daten.
- `Student`: Sinngemäß wie bei `Professor`.
- `Course`: Diese Klasse hält Daten eines bestimmten Kurses.
- `Exam`: In dieser Klasse werden Daten für eine bestimmte Prüfung abgelegt.

Für alle diese Domänenobjekte braucht man typische CRUD- (create, read, update und delete) Methoden wie: `getProfessor(...)`, `saveProfessor (...)`, `deleteProfessor(...)` usw., dann sinngemäß dieselben Methoden nochmals für Student, also `getStudent(...)`, `saveStudent (...)`, `deleteStudent(...)` usw.

Es ist nicht nur mühsam, immer wieder dieselben Methoden für verschiedene DAOs zu schreiben, sie müssen natürlich auch mehrfach getestet werden. Die Fehlerhäufigkeit steigt daher, und die Wartung ist aufwendiger, weil Änderungen an der Persistenzlogik an vielen DAOs wiederholt werden müssen.

Lösung

Um dieses Problem zu umgehen, kann man ein *generisches DAO* erstellen, das diese repetitiven Methoden implementiert und in den konkreten DAOs wiederverwendet. Im Java-Umfeld funktioniert das mit Java Generics sehr elegant. In Kombination mit Java Generics kommt in der Persistenzschicht auch immer häufiger die Java Persistence API zum Einsatz.

Wie es bei einem DAO üblich ist, wird auch für das generische DAO ein Interface definiert. Dieses Interface wird jedoch parametrisiert. Der Parameter `T` steht stellvertretend für das Domänenobjekt, zum Beispiel `Student` oder `Professor`. Der Parameter ID repräsentiert den Primärschlüssel für ein Objekt. In beiden Fällen sind die Parameter `Serializable`. Nun kommt der eigentlich Trick: Bei allen Methoden werden statt den Domänenobjekten die stellvertretenden Parameter `T` und/oder ID verwendet. Die `saveOrUpdate`-Methode beispielsweise bekommt als Paramter ein Domänenobjekt und liefert das gespeicherte Domänenobjekt wieder zurück. Die Parameter können auch auf Listen

angewendet werden, um typsichere Listen zu definieren.

```
public interface
        GenericDao<T, ID extends Serializable> {
 public T saveOrUpdate(T entity);
 public void delete(T entity);
 public T findById(ID id, boolean lock);
 public List<T> findAll();
 public List<T> findByExample(T exampleInstance);
}
```

Implementierung mit Hibernate

Im folgenden Listing ist nun eine Implementierung des generischen DAOs mit Hibernate realisiert. Entscheidend in dieser Implementierung ist das Attribut `type` vom Typ `Class` und der Konstruktor, der als Parameter die Klasse bekommt. Genau hier erfolgt die tatsächliche Verknüpfung zwischen dem Domänenobjekt und dem generischen DAO. Möchten wir nun ein DAO für die Studenten-Klasse, dann kann mit der Deklaration `GenericDao personDAO = new HibernateGenericDao(domain.Person)` ein neues Personen-DAO erstellt werden.

```
public class HibernateGenericDao extends
    HibernateDaoSupport implements GenericDao {
 private Class<T> type;

 public HibernateGenericDao(Class type) {
     this.type = type;
 }

 public T saveOrUpdate(T entity) {
     getHibernateTemplate().saveOrUpdate(entity);
     return entity;
 }

 public void delete(T entity) {
     getHibernateTemplate().delete(entity);
 }

 public List<T> findAll() {
     return getHibernateTemplate().loadAll(type);
 }

 public List<T> findByExample(T exampleInstance) {
     return getHibernateTemplate()
             .findByExample(exampleInstance);
 }

 public T findById(ID id, boolean lock) {
     return (T)getHibernateTemplate()
             .load(type.getClass(), id);
 }
}
```

Das eben vorgestellte DAO bietet nun domänenübergreifende CRUD-Operationen an. Es befinden sich auch Standard-Suchmethoden, wie `findAll`, `findByExample` oder `findById` im generischen DAO. Natürlich kann das generische DAO auch erweitert werden. Zum Beispiel soll ein Student nicht nur nach der `ID`, sondern auch nach dem Namen und der Matrikelnummer gesucht werden können. Für diesen Zweck wird das generische DAO erweitert:

Erweiterung des generischen DAOs

```
1 public interface PersonDao
2         extends GenericDao<Person, Long> {
3     List<Person> findByName(String name);
4     Person findByMatrNr(String matrNr);
5 }
```

Das erweiterte DAO ist aber bereits mit den richtigen Parametern befüllt, da ja nur der Student über eine Matrikelnummer verfügt [22].

8.5 Verhalten

8.5.1 Observer

Problemstellung

Ändert sich der Zustand eines Objekts (d. h. ändert sich der Wert einer oder mehrerer Instanzvariablen), so möchte man in vielen Fällen bestimmte Aktionen ausführen, z. B. eines oder mehrere andere Objekte von der Zustandsänderung benachrichtigen. Dieses Problem findet man beispielsweise bei der Entwicklung von Benutzerschnittstellen: Wählt ein Benutzer einen Eintrag in einer Liste aus, so sollen bestimmte Aktionen durchgeführt werden (z. B. die Aktualisierung eines Balkendiagramms und einer Tabelle).

Beispiel: Staumauer

Als Beispiel betrachten wir eine Staumauer mit Kraftwerk: Sie soll so modelliert werden, dass Sensordaten zur Steuerung (Regelkreis), aber auch für andere Zwecke verwendet werden können[15]:

In Abbildung 8.12 ist ein Fluss mit einer Staumauer angedeutet. Ein Sensor-Objekt ist durch eine Messuhr symbolisiert. Dieses registriert die Messwerte des Wasserstandes *vor* und *nach* der Staumauer. Außerdem sollen andere Objekte benachrichtigt werden, wenn sich der Wasserstand an irgendeiner Stelle ändert. In dem Beispiel sind dies die Stadt vor und hinter der Staumauer um rechtzeitig auf Hochwasser reagieren zu können, sowie natürlich das Objekt das die Regeleinheit in der Staumauer repräsentiert, die den Wasserdurchfluss zu steuern hat. Zuletzt soll der Wasserstand für statistische Zwecke auch in einer Datenbank gespeichert werden.

[15] Dieses Beispiel findet sich im Detail ausgeführt bei den Design-Patterns Beispielen im Web.

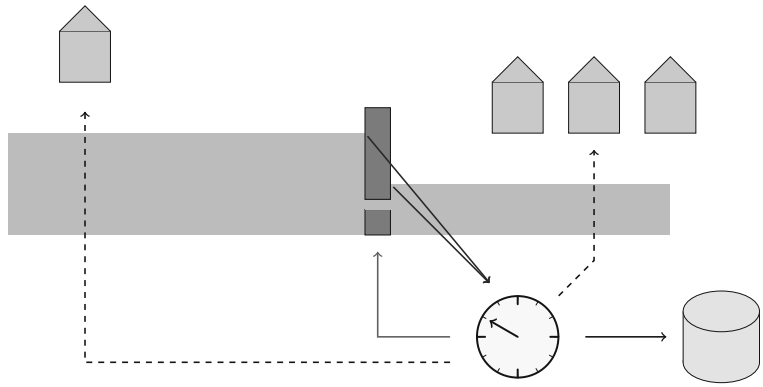


Abbildung 8.12 *Der Sensor in der Mitte registriert den Wasserstand vor und nach der Staumauer. Bei Änderung des Wasserstands sind verschiedene interessierte Objekte zu informieren.*

Zunächst könnte man eine `Sensor`-Klasse erstellen, die mit der Hardware interagiert und die Messwerte speichert. Ganz vereinfacht könnte die Klasse wie folgt aussehen:

```
1  public class Sensor {
2    private int waterLevelUpstream;
3    private int waterLevelDownstream;
4    public Sensor(int waterLevelUpstream,
5                  int waterLevelDownstream) {
6      this.waterLevelUpstream = waterLevelUpstream;
7      this.waterLevelDownstream
8              = waterLevelDownstream;
9    }
10   public int getWaterLevelUpstream() {
11     return waterLevelUpstream;
12   }
13   public void setWaterLevelUpstream(
14                 int waterLevelUpstream) {
15     this.waterLevelUpstream = waterLevelUpstream;
16   }
17   public int getWaterLevelDownstream() {
18     return waterLevelDownstream;
19   }
20   public void setWaterLevelDownstream(
21                 int waterLevelDownstream) {
22     this.waterLevelDownstream
23             = waterLevelDownstream;
24   }
25 }
```

Man kann sich vorstellen, dass die zwei Wasserstandssensoren in regelmäßigen Intervallen die aktuellen Messwerte in den `waterLevelX`-Variablen ablegen (sowie etwa einen Zeitstempel, wann dies das letzte Mal geschehen ist).

Damit ist nur die Speicherung der jeweils letzten Messwerte garantiert. Gefordert ist aber auch – wie oben beschrieben – dass Änderungen der Messwerte an verschiedene andere Objekte weitergereicht werden. Eine Variante

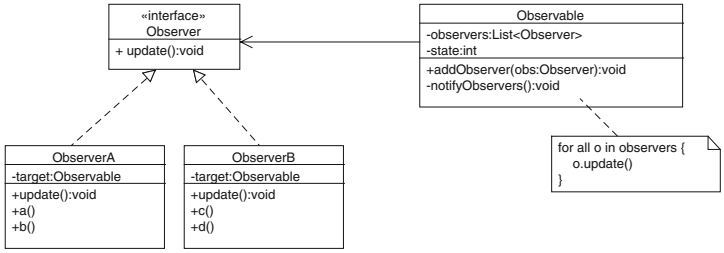


Abbildung 8.13 *Prinzip des Observer-Patterns.*

dies zu lösen wäre, dass das Sensor-Objekt selbst diese Benachrichtigungen durchführt, z. B. in folgender Weise (nur die Änderung in einer Methode werden gezeigt, der Rest ist sinngemäß wie oben):

```
1  public class Sensor {
2    ...
3    public void setWaterLevelUpstream(
4                  int waterLevelUpstream) {
5      this.waterLevelUpstream = waterLevelUpstream;
6      notifyOthers();
7    }
8    private void notifyOthers() {
9      dam.update (this);
10     cityAlarm.update (this);
11     ...
12   }
13   ...
14 }
```

Wird also ein neuer Wert gesetzt, kümmert sich das `Sensor`-Objekt darum, dass andere interessierte Objekte benachrichtigt werden. Dies funktioniert natürlich, hat aber den Nachteil, dass die `Sensor`-Klasse sehr hart an die anderen Klassen gebunden ist. Die Aufgabe der `Sensor`-Klasse ist es ja eigentlich, nur mit dem Sensor zu interagieren sowie die Daten zu speichern. Nun bekäme sie zusätzlich eine operative Funktion. Ein weiterer Nachteil dieses Ansatzes ist folgender: Kommen neue Klassen hinzu, die benachrichtigt werden sollen, so ist dies immer mit Änderungen in der Sensor-Klasse verbunden, was häufig nicht wünschenswert oder gar nicht möglich ist. Außerdem ist es in diesem Fall nicht möglich, zur Laufzeit neue Objekte für eine Benachrichtigung hinzuzufügen oder bestehende zu entfernen.

Lösung

Um diese Problemstellung flexibler zu lösen, kann man sich des Observer-Patterns bedienen: Abbildung 8.13 stellt das Prinzip dar: Die `Observable`-Klasse ist „unter Beobachtung“, d. h., ihre Zustandsände-

rungen werden weitergeleitet. Das Interface `Observer` definiert die eine Methode: `update()`[16].

Klassen, die von der Zustandsänderung der `Observable`-Klasse benachrichtigt werden wollen, implementieren das `Observer`-Interface und registrieren sich beim Observable-Objekt mithilfe der `addObserver()`-Methode. Damit ist es möglich, neue Klassen auch zur Laufzeit zu registrieren bzw. diese Registrierung wieder zu entfernen. Tritt eine Zustandsänderung im `Observable`-Objekt ein, so benachrichtigt dieses einfach alle Objekte der `observers`-Liste über den Aufruf der `update()`-Methode ohne im Vorhinein wissen zu müssen, welche Objekte dies betrifft.

Observer und Observable in Java

In Java ist dies besonders leicht zu implementieren, weil es sowohl das `Observer`-Interface als auch die `Observable`-Klasse schon in der Klassenbibliothek im `java.util`-Packet gibt. Das oben begonnene Beispiel kann jetzt mit diesem Pattern elegant umgesetzt werden:

Flusskraftwerk-Beispiel mit Observer-Pattern

Zunächst muss die Sensor-Klasse etwas erweitert werden, denn sie wird ja zum `Observable`, also zum Objekt unter Beobachtung:

```
1  import java.util.Observable;
2  public class Sensor extends Observable {
3    private int waterLevelUpstream;
4    private int waterLevelDownstream;
5    public Sensor(int waterLevelUpstream,
6                  int waterLevelDownstream) {
7      this.waterLevelUpstream = waterLevelUpstream;
8      this.waterLevelDownstream
9              = waterLevelDownstream;
10   }
11   public int getWaterLevelUpstream() {
12     return waterLevelUpstream;
13   }
14   public void setWaterLevelUpstream(
15                 int waterLevelUpstream) {
16     this.waterLevelUpstream = waterLevelUpstream;
17     setChanged();
18     notifyObservers(this);
19   }
20   ...
```

Die Grundidee ist dieselbe wie im ersten Listing; hier wird beispielhaft nur die `setWaterLevelUpstream()`-Methode im Detail ausgeführt. Es gib folgende Änderungen:

[16] Oft wird der Methode auch als Übergabeparameter eine Referenz auf das Observable-Objekt sowie ein Datenobjekt mitgegeben.

- Eine `notifyOthers()`-Methode ist nicht mehr erforderlich, darum kümmert sich schon die `Observable`-Klasse von der abgeleitet wird.
- Tritt eine Zustandsänderung ein, so muss die `setChanged()`-Methode aufgerufen werden, die ein Flag setzt, dass sich der Zustand des Objekts geändert hat.
- Die eigentliche Benachrichtigung erfolgt durch den Aufruf der `notifyObservers()`-Methode, dabei wird eine Referenz auf die eigene Instanz (`this`) mitgegeben, damit die benachrichtigten Objekte auf die Daten zugreifen können.

Nun gibt es zudem die Klassen `CityAlarm` (Benachrichtigung der Städte) und `Dam` (Kontrolle des Wasserflusses) sowie `WaterLevelLogger` (speichern der Wasserstände in einer Datenbank). All diese Klassen (und vielleicht noch weitere) sind `Observer`, also „Beobachter“. Die konkrete Implementierung ist einfach, wie man am Beispiel von `CityAlarm` leicht sehen kann:

```
1  import java.util.Observable;
2  import java.util.Observer;
3  public class CityAlarm implements Observer {
4    public void update(Observable o, Object arg) {
5      Sensor sensor = (Sensor) o;
6      checkFloodWarning(
7        sensor.getWaterLevelUpstream(),
8        sensor.getWaterLevelDownstream());
9    }
10   private void checkFloodWarning(
11                 int waterLevelUpstream,
12                 int waterLevelDownstream) {
13   ...
14   }
15 }
```

Die Klasse implementiert das `Observer`-Interface und die vom Interface vorgeschriebene `update()`-Methode. Diese wird vom `Observable`-Objekt aufgerufen, wenn eine Änderung eingetreten ist, und die `CityAlarm` Klasse kann nun z. B. prüfen, ob ein Alarm ausgelöst werden muss.

Damit die Benachrichtigung funktioniert, müssen natürlich alle `Observer`-Objekte beim `Observable` registriert werden, z. B. bei der Instanziierung[17]:

[17] Diese Registrierung kann je nach konkreter Problemstellung entweder wie hier extern oder eventuell auch im Konstruktor der Observer erfolgen. Dann binden sich die Observer selbst an das Observable-Objekt.

```
Sensor sensor = new Sensor(...);
Dam dam = new Dam (...);
CityAlarm cityAlarm = new CityAlarm(...);
waterLevelLogger = new WaterLevelLogger(...);
sensor.addObserver(dam);
sensor.addObserver(cityAlarm);
sensor.addObserver(waterLevelLogger);
```

Verwandte Patterns

Mit dem Observer-Pattern eng verwandt sind das Model-View-Controler (MVC) sowie das Event-Listener-Pattern in der Entwicklung von Benutzerschnittstellen. Eigentlich sind dies fast Synonyme. Das Observer-Pattern ist die allgemeine Beschreibung des Konzepts, während MVC/Event Listener konkrete Einsatzbereiche darstellen. Weitere verwandte Patterns sind das Producer/Consumer- und Publisher/Subscriber-Pattern (siehe auch Abschnitt 8.6.2).

8.5.2 Decorator

Problemstellung

Wie schon in den vorigen Abschnitten immer wieder gezeigt wurde (z. B. bei der Delegation), ist Vererbung nicht immer der geeignete Mechanismus, um einem Objekt oder einer Klasse neue Funktionalität hinzuzufügen. Man kann sich das am Beispiel einer Bäckerei, die Torten verkauft, veranschaulichen: Zunächst gibt es eine Anzahl verschiedener Torten, z. B. Schokolade-, Nuss- oder Cremetorten. Möchte man dieses System modellieren, könnte man sich vorstellen, dass man zunächst eine Basisklasse `Cake` definiert, die bestimmte Grundeigenschaften aufweist. Die Spezialisierung in Schokolade- und Cremetorte wird dann durch abgeleitete Klassen definiert. Nüsse können aber vielleicht sowohl in Schokolade- als auch in Cremetorten vorhanden sein. Selbstverständlich möchte man aber auch Torten für verschiedene Anlässe anbieten: mit Figuren für Hochzeiten, mit Schrift für Geburtstage, mit Kerzen, mit Verzierung einzelner Tortenstücke, mit einem Feuerspucker in der Mitte usw.

Möchte man dies mit Vererbung modellieren, wird es unübersichtlich und kompliziert. Denn all diese verschiedenen Dekorationen können auf alle Torten angewandt werden. So würde das dazu führen, dass man jede Variante x-mal ableiten muss; also eine Schokotorte mit Kerzen, eine Nusstorte mit Kerzen etc. Dann hat man immer noch das Problem, dass auch noch andere Kombinationen möglich sind, also z. B. Schokotorte mit Schrift, Verzierung und Kerzen, Nusstorte mit Hochzeitspaar, Nusstorte mit Hochzeitspaar und Verzierung usw.

Man erkennt schnell, dass dies in der Praxis mit Vererbung nicht elegant zu lösen ist, sondern dass man vielmehr einen Mechanismus benötigt, der in der Lage ist, Objekte mit Basisfunktionalität zur Laufzeit flexibel mit weiteren Funktionen anzureichern. Probleme dieser Art treten durchaus häufig auf, beispielsweise im Fall von Benutzerschnittstellen: Fenster mit Rahmen, Button mit Rahmen, Button mit Tooltip, Drop-down-Element mit

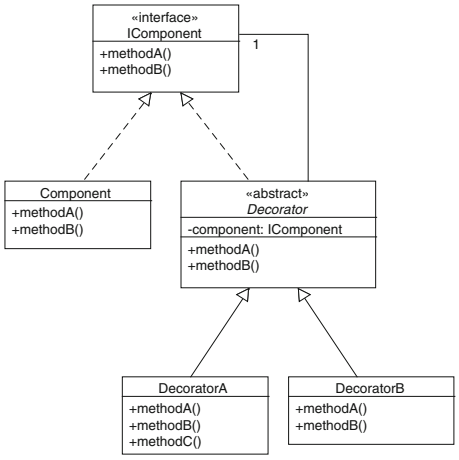


Abbildung 8.14
Konzept des Decorator-Patterns.

Tooltip etc. In der Java-API findet man auch das Beispiel der Streams, bei denen verschiedenste Kombinationen möglich sind: z. B. kann ein Stream komprimiert oder gepuffert sowie komprimiert *und* gepuffert werden.

Lösung

Um derartige Problemstellungen elegant zu lösen, wird gerne das Decorator-Pattern (auch Wrapper genannt) eingesetzt. Dieses Pattern ist in der Implementierung dem Proxy-Pattern (siehe Abschnitt 8.4.4) ähnlich, der Einsatzbereich ist aber anders. Abbildung 8.14 zeigt die Grundstruktur dieses Patterns:

- Es gibt ein Interface `Component`.
- Eine oder mehrere konkrete Komponenten implementieren dieses Interface.
- Daneben gibt es eine abstrakte `Decorator`-Klasse. Diese implementiert das Interface ebenfalls und hält eine Referenz auf das Interface, um Funktionalität weiterreichen zu können (dies ist von der Implementierung dem Proxy-Pattern sehr ähnlich).
- Dann gibt es eine oder mehrere Decorator-Implementierungen, die bestimmte ergänzende Funktionalität definieren können.

Beispiel: Torte

Anhand des Torten-Beispiels in Abbildung 8.15 sollte der Einsatz klar werden: Zunächst wird ein Interface `Cake` definiert, das die Operationen darstellt, die zum Backen einer Torte notwendig sind. In diesem Beispiel ist das stark vereinfacht nur die Methode `bake()`. Links sieht man die Basis-Tortenklasse `ConcreteCake` sowie davon abgeleitet verschiedene Arten von Torten, die recht unterschiedlich hergestellt werden, z. B. `ChocolateCake` und `CreamCake`.

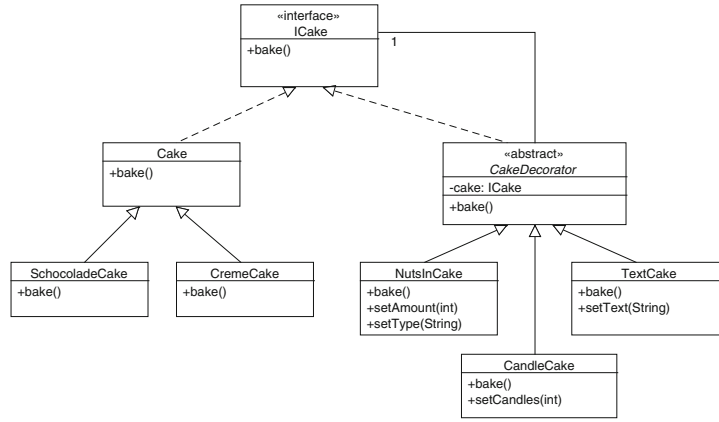


Abbildung 8.15 *Beispiel für das Decorator-Pattern: Backen verschiedener Torten, ausgehend von Basisvarianten (links) angereichert mit verschiedenen „Dekors“.*

Rechts sieht man das Decorator-Pattern; zunächst eine abstrakte Basisklasse `CakeDecorator`, von der alle Varianten der Dekoration abgeleitet werden. `CakeDecorator` hält eine Referenz auf das `Cake`-Interface. Es gibt zudem eine Reihe von Dekorationsvarianten, z. B. `CandleCake`.

Wie würde man nun das Objekt erstellen, das eine Schokotorte repräsentiert, die aber auch Nüsse und Kerzen haben soll?

```
Cake cake = new ChocolateCake();

NutsInCake nic = new NutsInCake(cake);
nic.setAmount(15);
nic.setType("hazelnut");
CandleCake cc = new CandleCake (nic);
cc.setCandles(13);
cake = (Cake)cc;

cake.bake();
```

D. h., es wird zunächst die konkrete Klasse instanziiert und dann die Decorator-Klasse, der die Referenz auf die konkrete Klasse übergeben wird. Sollen mehrere Dekorationen durchgeführt werden, kann dies mehrfach kaskadiert werden. Am Ende wird die `bake()`-Methode aufgerufen und automatisch alles ausgeführt, was für das Backen erforderlich ist, da diese Methode durch die Dekorator-Klassen durchgereicht wird (siehe auch Abbildung 8.16). `CandleCake` könnte beispielsweise zunächst die `bake()`-Methode der übergeordneten Klasse aufrufen und danach auf die fertige Torte Kerzen stecken.

In der Praxis kann man die Dekoration des Basisobjekts in eine eigene Methode verlagern und damit den ganzen Prozess übersichtlicher machen:

```
public void bakeCake(DecorationOptions options) {
  Cake cake = makeDecoration(new ChocolateCake(),
                             options);
  cake.bake();
```

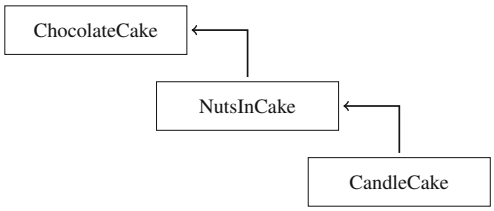


Abbildung 8.16 *Im Beispiel werden die Objekte verschachtelt um in jedem Schritt Funktionalität hinzuzufügen.*

```
}
private Cake makeDecoration(Cake cake,
                            DecorationOptions o) {
  // if nuts
    NutsInCake nic = new NutsInCake(cake);
    nic.setAmount(15);
    nic.setType("hazelnut");
    cake = (Cake)nic;
  // if candles
  ...
  // if...
  ...
  return cake;
}
```

In der Methode, in der gebacken wird, wird das Basisobjekt instanziiert. In die `makeDecoration()`-Methode wird dann der Algorithmus ausgelagert (hier nur als Pseudocode angedeutet), der z. B. anhand der Bestellung erkennt, welche „Sonderausstattung“ erforderlich ist. Er ergänzt dann, wie im vorigen Codebeispiel, die entsprechende Funktionalität.

Beispiel: Stream

Ein anderes – vorhin schon erwähntes Beispiel – ist die Stream-Implementierung in der Java API. Dort ist sinngemäß Vergleichbares möglich. Man kann einen einfachen Filestream erstellen:

```
Stream s = new FileStream(filename);
```

Diesen Filestream kann man aber ebenfalls mit verschiedenen weiteren Funktionen anreichern (dekorieren), z. B.:

```
Stream s = new CompressingStream(
             new BufferedStream(
               new FileStream(filename)
             )
           );
```

Auch hier kann wieder beliebig kombiniert werden: der `FileStream` kann nur komprimiert oder nur gepuffert werden oder beides. Die Erweiterungsmöglichkeiten sind also voneinander unabhängig.

Kontext

Das Decorator-Pattern ist also in solchen Fällen nützlich, wo Objekte mit verschiedensten Funktionen angereichert, bzw. zur Laufzeit diese auch wieder entfernt werden sollen, und dies mit Vererbung nicht elegant zu lösen ist. Durch die Möglichkeit der Kaskadierung können Funktionsblöcke auch kombiniert werden (Kerzen, Schrift, Nüsse). Damit ist es möglich erst *zur*

Laufzeit aus einer größeren Anzahl von Möglichkeiten die gewünschten Objekte dem Problem angepasst zu konfigurieren.

Verwandte Patterns

Das Decorator-Pattern ist die Basis für aspektorientierte Programmierung (siehe Abschnitt 9.5). Weiterhin ist auch das Proxy-Pattern in der Art der Implementierung dem Decorator-Pattern ähnlich. Allerdings ist der Anwendungsfall ein gänzlich anderer. Während das Proxy-Pattern dem Client vortäuscht „das richtige" Objekt zu sein, um damit Zugriffsschutz und ähnliche Funktionalität zu ergänzen, ist das Decorator-Pattern geradezu offensiv: Dem Entwickler werden alle Bausteine in die Hand gegeben, um daraus ein Objekt mit der gewünschten Funktionalität zu basteln.

8.5.3 Interceptor

Problemstellung

Versucht man für komplexere Systeme eine geeignete Architektur zu entwickeln, so steht man häufig vor zwei grundsätzlichen Design-Optionen: (1) Man versucht *alle* Funktionen, die das System bieten soll, zu erfassen, zu modellieren und zu implementieren oder (2) man stellt fest, dass einige Funktionen nur fallweise benötigt werden oder orthogonal zu anderen stehen (z. B. Security, Logging von Aktivitäten). Außerdem ist es besonders in modernen Systemen häufig erwünscht, ein hohes Maß an Flexibilität zu haben, um auch später während der Laufzeit noch neue Funktionen einhängen zu können, die zur Designzeit noch nicht bekannt waren.

Man könnte versuchen, alle am Ablauf beteiligten in die Modellierungsphase miteinzubeziehen und alle möglichen Abläufe zu erfassen und zu modellieren. Mit diesem Ansatz wird man wahrscheinlich scheitern, weil es kaum gelingen wird, alle möglichen Funktionen und Aktivitäten korrekt zu erfassen. *Selbst wenn* es gelänge, wäre das entstehende System sehr starr und könnte später nur schwer erweitert werden.

Beispiel: Flughafen

Als Beispiel könnte man sich die Aufgabe stellen, einen Flughafen zu modellieren. Hier gibt es verschiedene Kernfunktionen wie Check-in, Gepäck-Verwaltung, Passkontrolle, Überwachung des gesamten Flughafens um Sicherheit zu gewährleisten, Geschäfte, die Dienste am Gelände anbieten usw. Fliegt man beispielsweise von Wien nach New York, so geht man zuerst in den Flughafen, dann in den Check-in-Bereich, dann in den Abflug-Bereich und betritt letztlich das Flugzeug. Dazwischen liegen verschiedene vorgeschriebene Interaktionen mit Flughafen- oder Sicherheitspersonal, z. B. beim Check-in, der Pass-Kontrolle oder der Ticket-Kontrolle.

Daneben gibt es aber einige Aktivitäten, die nur in bestimmten Fällen vorgenommen werden, z. B. eine Handdurchsuchung des Gepäcks oder Aktivitäten, die Sicherheitskräfte einleiten, wenn sie auf den Überwachungsmonitoren verdächtige Aktivitäten beobachten usw. Besondere Sicherheitsmaßnahmen können in bestimmten Situationen verordnet werden, ohne dass der Regelbetrieb davon berührt wird. Vielleicht erfordern bestimmte Flüge auch besondere Services (z. B. bei Staatsgästen). Möglicherweise kommt

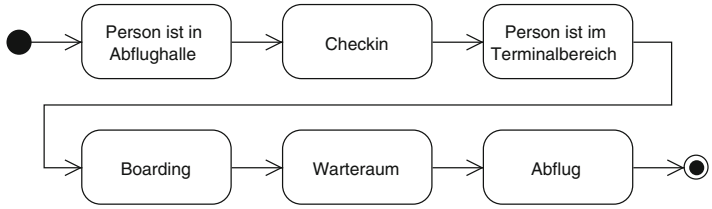


Abbildung 8.17
Einfaches Flughafen-Beispiel für Interceptor.

das Management auf die Idee, die Personenbewegungen im Flughafenareal zu beobachten und statistisch auszuwerten, um die Logistik zu verbessern etc.

Elementare Schritte des Prozesses

Abbildung 8.17 zeigt ein sehr vereinfachtes Zustandsdiagramm einer Person, die einen Flug antritt. In diesem einfachen Beispiel sind nur die Zustände eingetragen, die unmittelbar für den Abflug erforderlich sind. Es fehlen aber z. B. alle Sicherheitsmaßnahmen, die auf einem Flughafen erforderlich sind.

Im nächsten Schritt, der in Abbildung 8.18 illustriert ist, werden Sicherheitsüberprüfungen in den Prozess aufgenommen. Aber schon an diesem einfachen Beispiel kann man sehen, dass eine solche Architektur nicht sehr elegant ist. Die Aktionen der Sicherheitskräfte wiederholen sich mehrfach, wir haben außerdem ein Vermischen verschiedener Aspekte: ein Aspekt betrifft die Abwicklung des Fluges, der andere die Gewährleistung der Sicherheit.

Es kommt noch hinzu, dass eventuell nicht alle Überprüfungen in jedem Fall erfolgen müssen, zum Beispiel der Check am Flugzeug könnte im Regelfall gar nicht erfolgen, im Spezialfall vielleicht nicht per Video, sondern mit Beamten am Eingang des Flugzeugs. Im Falle von besonderen Bedrohungen könnten weitere Maßnahmen verordnet werden, die in diesem Prozess noch gar nicht bedacht sind. Hat man aber das Modell wie in Abbildung 8.18 implementiert, ist eine Erweiterung unter Umständen schwierig.

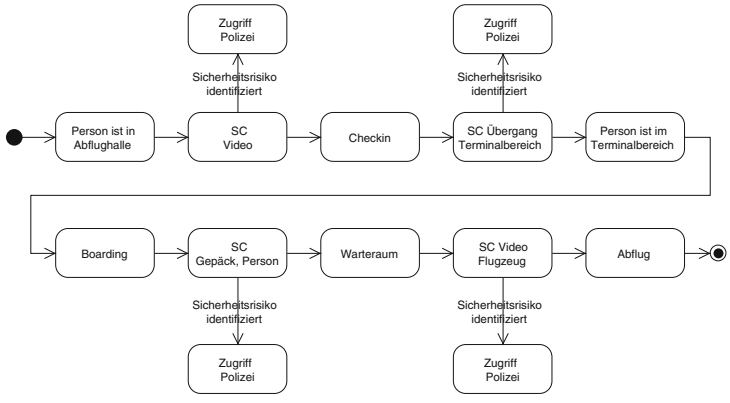


Abbildung 8.18
Erweitertes Beispiel: In diesem Beispiel wurden Sicherheitsüberprüfungen hinzugefügt. SC steht für Sicherheits--Check.

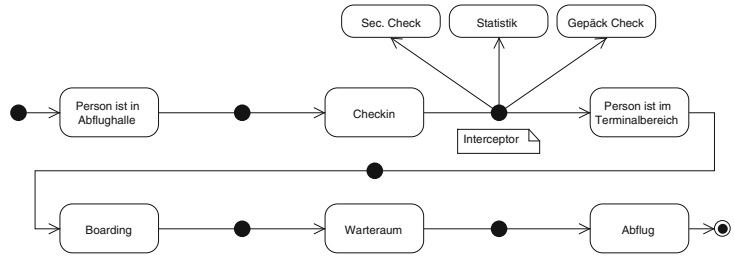


Abbildung 8.19 *An allen relevanten Übergängen werden Interceptoren definiert. An diesen Punkten können später beliebige Services eingehängt werden.*

Lösung: Separation of Concerns

Eine Möglichkeit, eine solche Architektur eleganter zu gestalten, ist der Einsatz des Interceptor-Patterns. Das Interceptor-Pattern ist eines der komplexeren Patterns und nicht ganz einfach zu implementieren. In dieser Einführung soll zunächst nur die Idee vermittelt werden. Das Interceptor-Pattern basiert auch auf der Idee der *Separation of Concerns*, Details dazu in Abschnitt 7.4.

Das Beispiel mit Interceptoren

Abbildung 8.19 illustriert die Idee des Interceptor-Patterns: Die „hart verdrahteten" Sicherheits-Checks wurden zunächst aus dem Modell entfernt. Allerdings wurden zwischen allen Übergängen *Interception Points* definiert. Konkret bedeutet dies, dass das Interceptor-Pattern dafür sorgt, dass sich auch noch zur Laufzeit Services an beliebige Punkte einhängen können. Dies wird am Übergang zwischen *Check-in* und *Person ist im Terminalbereich* gezeigt. Security-Check, ein Statistik-Service, das z. B. Personenströme registriert, sowie der Gepäck-Check haben sich an dem Interceptor registriert.

Dies bedeutet nun, dass das Framework dafür sorgt, dass diese registrierten Services aufgerufen werden, *bevor* der nächste Systemzustand *Person ist im Terminalbereich* erreicht wird. Diese Services könnten nun verhindern, dass die Person den Terminalbereich betritt, z. B. wenn es Sicherheitsbedenken gibt, sie könnten aber auch einfach irgendwelche anderen Aktivitäten ausführen, die an dieser Stelle angebracht sind (z. B. eine Aufzeichnung der Aktivität für die Statistik).

Verdrehte Logik

Die Logik ist hier gewissermaßen umgedreht: Der Designer der Geschäftsanwendung kümmert sich um Aspekte wie Sicherheit zunächst gar nicht, schafft aber die Voraussetzung mit dem Interceptor-Pattern, dass derartige Funktionen später eingehängt werden können. Die konkrete Verdrahtung, also wann beispielsweise welche Sicherheits-Checks durchgeführt werden, kann zu einem späteren Zeitpunkt und eventuell auch von ganz anderen Stakeholdern entschieden werden, z. B. von den Sicherheitskräften selbst. Diese können je nach Kontext entscheiden und Sicherheitsprüfungen einfügen oder wieder entfernen und dies in den restlichen Prozess einhängen, ohne dass dies zur Designzeit bekannt sein muss.

Beispiel: Webapplikation

Auch bei der Entwicklung von Webapplikationen stößt man häufig auf solche Probleme. Beispielsweise soll ein Login-Mechanismus dafür sorgen, dass nur bestimmte Benutzergruppen bestimmte Aktivitäten durchführen

können. Traditionellerweise wird in den Methoden, die aufgerufen werden, um Anfragen zu beantworten, geprüft, ob die zugreifende Person auch eingeloggt ist und über hinreichende Rechte für die gewählte Aktion verfügt.

Dies ist nicht sehr elegant, da in diesen Objekten zwei verschiedene Aspekte vermischt werden: die eigentliche Geschäftslogik, sowie der Zugriffsschutz, der besser an einer zentralen Stelle implementiert wird. Mithilfe des Interceptor-Patterns oder der unten erwähnten aspektorientierten Programmierung kann man diese beiden Aspekte voneinander trennen. Der Entwickler, der für die Geschäftslogik zuständig ist, implementiert genau diese, ohne den Zugriffsschutz berücksichtigen zu müssen. Der Zugriffsschutz wird dann über einen Interception-Mechanismus später hinzugefügt. Die Zugriffslogik kann von einem Spezialisten implementiert werden, der sich genau mit dem Problem, welcher Benutzer darf wann und in welchem Kontext mit welchen Objekten interagieren, auseinandersetzt. Er kann dies implementieren, ohne Details der Geschäftslogik verstehen zu müssen. Auf dieselbe Weise kann man auch Logging und andere Mechanismen an diesen Interception Points zu einem späteren Zeitpunkt integrieren.

Verwandte Patterns und weitere Informationen

In dieser Einführung wurde nur die Idee des Interceptor-Patterns erklärt. Die eigene Implementierung ist nicht ganz trivial. In Kapitel 9.5 wird aber in die aspektorientierte Programmierung eingeführt. Dieser Architekturstil bedient sich unter anderem des Interceptor-Patterns, und Frameworks wie Spring oder AspectJ erlauben dem Entwickler eine vergleichsweise einfache Implementierung solcher Szenarien.

Weitere verwandte Patterns sind das Observer-Patterns (siehe Abschnitt 8.5.1) sowie das Proxy-Pattern (siehe Abschnitt 8.4.4) und das Decorator-Pattern (siehe Abschnitt 8.5.2).

8.6 Integration

Muster (Patterns) kann man auf verschiedenen Ebenen der Anwendungsentwicklung finden. Die bisher genannten Muster haben eine eher feine Granularität, beschreiben also z. B. nur die Interaktion weniger Objekte miteinander, wie z. B. das Interface, Delegations oder Proxy-Pattern. Andere sind schon durchaus komplexer und oft auch Teil einer Middleware oder eines Komponentenframeworks, wie z. B. das Factory oder Interceptor-Pattern.

Anwendungen „im Konzert“

In diesem Abschnitt wird noch einen Schritt weitergegangen: Zeitgemäße Software-Entwicklung unterscheidet sich erheblich von der Software-Entwicklung der 1980er oder auch noch der 1990er Jahre. „Monolithische“ Anwendungen, die für sich alleine stehen und kaum mit anderen Systemen interagieren müssen, findet man heute kaum mehr. Zeitgemäße Software – ganz besonders im betrieblichem Umfeld – muss in fast allen Fällen gut mit anderen Produkten und Services zusammenarbeiten.

Es stellt sich also die Frage, wie man verschiedene Systeme – die möglicherweise auch von verschiedenen Herstellern stammen – miteinander integrieren kann, sodass einerseits die heute geforderte Flexibilität gewährleistet ist, andererseits aber die Systeme wartbar bleiben. Um diese Ziele zu erreichen, müssen verschiedene Bedingungen erfüllt sein.

Dieser Abschnitt steht inhaltlich zwischen den grundlegenden Mustern, die in den vorigen Abschnitten besprochen wurden, und gleichzeitig im Kontext komponentenorientierter Software-Entwicklung (siehe Kapitel 9). Eine Einführung in die Begriffe *Komponente* und *Service* bietet Abschnitt 9.1 „Vom Objekt zum Service“.

Grundsätzliche Überlegungen

Möchte man Anwendungen miteinander integrieren oder eine neue Anwendung in eine bestehende „Landschaft“ von Anwendungen einbetten, dann gibt es grundsätzlich viele verschiedene Möglichkeiten, wie diese Integration stattfinden kann. Einige grundsätzliche Überlegungen sollte man sich vorneweg jedoch stellen, z. B.: Ist die Landschaft sehr heterogen? Werden also verschiedene Betriebssysteme, Programmierplattformen, Protokolle eingesetzt, oder ist sie sehr homogen? Soll nur innerhalb einer Organisation/Firma kooperiert werden oder auch mit Anwendungen, die außerhalb der Organisation liegen?

Selbst für den Fall einer homogenen Landschaft sollten weitere Aspekte bedacht werden, unter anderem folgende:

- Wie eng können oder sollen Anwendungen miteinander gekoppelt sein?
- Ist es notwendig, dass die Kommunikation zwischen Anwendungen synchron erfolgt, oder ist eine asynchrone Kommunikation möglich?
- Wie einfach ist es, diese Integration durchzuführen und vor allem in Zukunft weitere Anwendungen zu integrieren oder bestehende Verbindungen zu ändern?
- Sollen hauptsächlich Daten ausgetauscht werden oder auch Funktionalität (etwa Methoden- und Serviceaufrufe)?
- Gibt es eine bestehende Software-Landschaft (z. B. Middleware), die zu verwenden oder zu berücksichtigen ist?
- Granularität des Datenaustausches: soll häufiger und schon bei kleineren Datenänderungen kommuniziert werden oder seltener und dafür mit größeren Datenmengen.

Kopplung

Gehen wir auf einige Punkte genauer ein: Eine sehr wichtige Frage ist, wie eng zwei Systeme miteinander gekoppelt sein sollen, und damit auch verbunden die Frage, ob Kommunikation synchron erfolgen muss oder auch asynchron implementiert werden darf. Was versteht man unter enger und loser Kopplung (*loose coupling*)?

Eng gekoppelte Systeme

Eng gekoppelte Systeme sind solche, die – vereinfacht gesagt – viel voneinander wissen müssen, um richtig zu funktionieren, und daher eine starke Abhängigkeit voneinander haben. Diese Abhängigkeit kann verschiedene konkrete Ursachen haben. Um ein Beispiel zu nennen: Es können Systeme sein, die in derselben Programmiersprache geschrieben sind, über ein plattformspezifisches Protokoll (wie z. B. Java RMI) kommunizieren, deren Verbindung „hart verdrahtet" ist und die über einen Request-Reply-Mechanismus miteinander kommunizieren, bei dem also der Client auf eine Antwort vom Server wartet.

Lose gekoppelte Systeme

Es gibt Fälle, in denen es eine gute Entscheidung ist, Systeme eng zu koppeln, weil diese dann einfacher zu entwickeln sind und in der konkreten Konfiguration auch sehr effizient miteinander arbeiten können. Es gibt aber auch mehr und mehr Anwendungsfälle, in denen man sich stärker *entkoppelte* Systeme wünscht, z. B.:

- Die Weiterentwicklung der beiden Systeme soll möglichst unabhängig erfolgen können.
- *Load Balancing*: Wenn eines der Systeme unter starker Last steht, soll es einfach möglich sein, dieses Service zu skalieren.
- Fehlertoleranz: Es soll möglich sein, Systeme neu zu starten oder durch andere zu ersetzen, ohne dass davon abhängige Services zu stark beeinträchtigt werden oder gar geändert oder umkonfiguriert werden müssen.
- Systeme sollen in unterschiedlichen Sprachen/Plattformen implementiert werden können (bzw. es sollen Systeme verschiedener Hersteller verwendet werden können, ohne dass man zu stark von einem bestimmten Anbieter abhängig wird).
- Der Austausch eines Systems soll leicht möglich sein, z. B. durch ein System eines anderen Anbieters oder gar durch ein Outsourcing des entsprechenden Services an einen Dienstleister.
- Die Vernetzung zwischen Systemen über Systemgrenzen hinweg soll möglich sein (z. B. zwischen Firmen), ohne dass man zu viele Voraussetzungen technischer oder organisatorischer Art auf der jeweils anderen Seite machen muss.

Die Implementierung entkoppelter Systeme ist allerdings fast immer mit höherem (initialem) Aufwand verbunden. Denn die Kommunikation und die Definition der Schnittstellen zwischen den Systemen dürfen nicht nur für eine Plattform funktionieren und müssen daher auf einem höheren Abstraktionsniveau passieren.

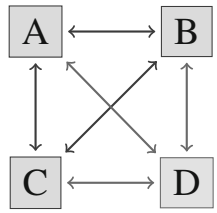


Abbildung 8.20
Point-to-Point-Integration: Schon bei wenigen Systemen wächst die Anzahl der notwendigen Verbindungen und Abhängigkeiten zwischen den Systemen.

Point to Point Integration

Ein zusätzliche Herausforderung tritt auf, wenn viele Systeme miteinander kommunizieren müssen. Selbst wenn Protokoll und Schnittstellen plattformunabhängig sind, bleibt das Problem, dass man leicht bei einer sogenannten Point-to-Point-Integration landet, die oft aus kurzfristigen Notwendigkeiten als „Provisorium“ entsteht, sich dann langfristig aber häufig als schwer erweiterbar und schwer wartbar herausstellt. Abbildung 8.20 illustriert das Problem: Schon bei vier Systemen, die miteinander kommunizieren wollen, müssen sechs verschiedene Verbindungen implementiert werden; jedes involvierte System muss sich (potenziell) mit den API aller anderen auseinandersetzen. Kommt ein weiteres System hinzu, muss in die bestehenden vier Systeme eingegriffen werden usw. Man erkennt, dass eine solche Architektur schnell unübersichtlich wird.

Hohpe: EI-Patterns

In diesem Abschnitt werden einige wichtige Integrations-Möglichkeiten beschrieben, die häufig eingesetzt werden. In Kombination mit den oben beschriebenen Patterns sowie einer geeigneten Architektur (siehe Kapitel 7 und 9) hat man gute architektonische Voraussetzungen, um stabile, gut integrierbare und auch flexible Anwendungen zu entwickeln. Die *Enterprise-Integration-Patterns* wurden von Hohpe und Wolf in dem gleichnamigen Buch [41] sehr gut verständlich beschrieben. Hohpe betreibt auch eine Webseite[18], auf der die wesentlichen Informationen, die im Buch detailliert dargestellt sind, im Überblick zu finden sind.

8.6.1 Integrations-Stile

Integrations-Strategien in der Praxis

Um die Kommunikation zwischen verschiedenen Systemen zu ermöglichen, haben sich in der Praxis verschiedene Möglichkeiten etabliert:

- File Transfer: Kommunikation über gemeinsame Dateien.
- Shared Databases: Kommunikation über gemeinsame (relationale) Datenbanken.
- Remote Procedure Invocation (RPI): Methodenaufruf von Services, die auf anderen Rechnern laufen, mit plattformspezifischen Standards.

[18] `http://www.enterpriseintegrationpatterns.com`

- Webservices (W3C-Standards, REST): RPI oder Datenintegration mittels plattformunabhängigen Standards.
- Messaging: Integration über asynchrone Nachrichten und nachrichtenbasierter Middleware.

File Transfer

Der Datenaustausch über Dateien ist eine der ältesten Integrationspraktiken. Eine Anwendung schreibt dabei die auszutauschenden Daten in eine Datei (meist eine Text- oder XML-Datei), eine oder mehrere andere Anwendungen lesen diese Datei. Dies erfolgt häufig im *Batch Mode*, also z. B. zeitgesteuert einmal pro Tag. Manchmal wird auch von den lesenden Anwendungen gepollt, d. h. in regelmäßigen Abständen geprüft, ob es neue Daten gibt. Dieses Verfahren ist eher als „historisch" zu betrachten und wird heute nur mehr in Ausnahmefällen für neue Integrationsprobleme eingesetzt.

Integration mittels File-Transfer hat auch das Problem der Aktualität, sowie der Granularität. D. h., die Daten in den verschiedenen Anwendungen sind vergleichsweise lange nicht synchronisiert. Neue Daten werden zu bestimmten Zeitpunkten in relativ großen Blöcken aktualisiert. Das Verfahren setzt zudem voraus, dass alle Anwendungen entweder auf gemeinsame Laufwerke zugreifen können, oder die Dateien müssen über andere Mechanismen verschickt werden, dann bewegt man sich aber schon in die Richtung von Messaging-Lösungen.

Shared Databases

Applikationsintegration über gemeinsame (relationale) Datenbanken ist ebenso eine immer noch häufig anzutreffende Vorgehensweise. Dabei greifen mehrere Anwendungen auf dieselbe Datenbank zu. Das Prinzip ist ähnlich wie bei Shared-Files, allerdings ist die Granularität natürlich viel feiner, da eine erfolgreiche Transaktion einer Anwendung (auch wenn sie nur einen einzelnen Datensatz betrifft) sofort von den anderen Anwendungen „gesehen" wird, da sich diese ja dasselbe Datenmodell teilen.

Genau dabei liegt aber auch eines der Probleme dieses Ansatzes. Eine wesentliche Methode im modernen Software-Engineering, um Systeme zu organisieren und klare Schnittstellen zwischen Systemen zu schaffen, ist die Kapselung, die auch eine der wesentlichen Grundprinzipien objektorientierter Sprachen ist. Kapselung bedeutet nun aber, dass *nicht* jede Anwendung *direkt* mit Daten interagiert, sondern dass eine logische Schicht dazwischengeschaltet wird. Dieser Layer kontrolliert und strukturiert den Zugriff auf die dahinterliegenden Daten. Eine neue Bestellung bedeutet eben nicht nur eine Update oder ein Insert in die Datenbank. Vielleicht müssen bestimmte Prüfungen durchgeführt werden, bevor die Bestellung in die Datenbank geschrieben wird, oder es müssen andere Aktivitäten angestoßen werden. Auch können sich Datenstrukturen ändern, was im Falle gemeinsamer Nutzung wesentlich komplizierter umzusetzen ist.

Eine Integration über *shared databases* kann daher im Einzelfall noch eine Möglichkeit sein, verletzt aber meist das Prinzip der Kapselung und kann

zu sehr schwer durchschaubaren Abhängigkeiten führen: „Wer liest und schreibt eigentlich unter welchen Voraussetzungen in welche Tabellen?“ Möchte man die Datenstruktur einmal ändern oder Anwendungen entfernen oder aktualisieren, so steht man häufig vor einem erheblichem Problem.

Shared Databases findet man manchmal auch als „Notlösung“, wenn mit Anwendungen interagiert werden musste, die über keine saubere API verfügen, und die einzige Möglichkeit darin bestand, die Datenbank „anzubohren“. Werden auch noch Stored Procedures und ähnliches auf Datenbankebene verwendet, entsteht häufig ein kaum mehr zu durchblickendes Chaos. Aus diesen Gründen sollte man von der Integration über gemeinsame Datenbanken nach Möglichkeit absehen.

Remote Procedure Invocation

Während die beiden vorhin genannten Methoden eher nur mehr in Sonderfällen eingesetzt werden, ist Remote Method Invocation (RPI), also der Aufruf einer Methode an einem entfernten Rechner mit Standards wie Java RMI oder CORBA, durchaus eine häufig eingesetzte Methode, die auch in vielen Fällen Sinn macht.

Der Vorteil von RPI ist, vor allem im Gegensatz zu shared Databases, dass die Kapselung der Daten erhalten bleibt. Jede Anwendung hat ihr eigenes Datenmanagement und stellt explizit Interfaces zur Verfügung, die zur Kommunikation mit anderen Systemen dienen.

Eine weiterer Vorteil von einigen RPI-Mechanismen ist, dass sie innerhalb einer bestimmten Plattform sehr effizient sein können. Java RMI beispielsweise ist genau auf die Java-Plattform angepasst und erlaubt damit eine effiziente Remote Procedure Invocation zwischen verschiedenen auf Java basierten Systemen, ohne „teure“ Zwischenschritte, wie dem Erstellen, Versenden und Parsen von XML-Nachrichten, wie dies bei den später erklärten SOAP Webservices notwendig ist. Allerdings haben viele RPI-Mechanismen Probleme, wenn Plattformgrenzen zu überschreiten sind oder wenn man das System stärker entkoppeln möchte.

Performante plattformspezifische RPI sollten daher in Betracht gezogen werden, wenn die engere Kopplung der Systeme kein Problem darstellt. Dies trifft beispielsweise zu, wenn bestimmte Systeme eigentlich immer nur miteinander kommunizieren und Flexibilität nicht erforderlich ist sowie wenn das Gesamtsystem einfach und überschaubar ist.

Webservices

Webservices basieren auf plattformunabhängigen XML-Standards wie SOAP (simple object access protocol) [81] und WSDL (webservice description language) [93]. SOAP kann man sich als den Briefumschlag vorstellen, in dem die zu übertragenden Daten oder Funktionsaufrufe verpackt werden. WSDL dient zur Beschreibung eines Services (angebotene Methoden, Ports usw.). Das konkret zu verwendende Transportprotokoll ist von den Standards nicht spezifiziert. Webservices können für Remote Procedure Invocation verwendet werden. In diesem Fall wird als Protokoll gerne http verwendet. Gleichzeitig bieten die neueren Standards wie WSDL

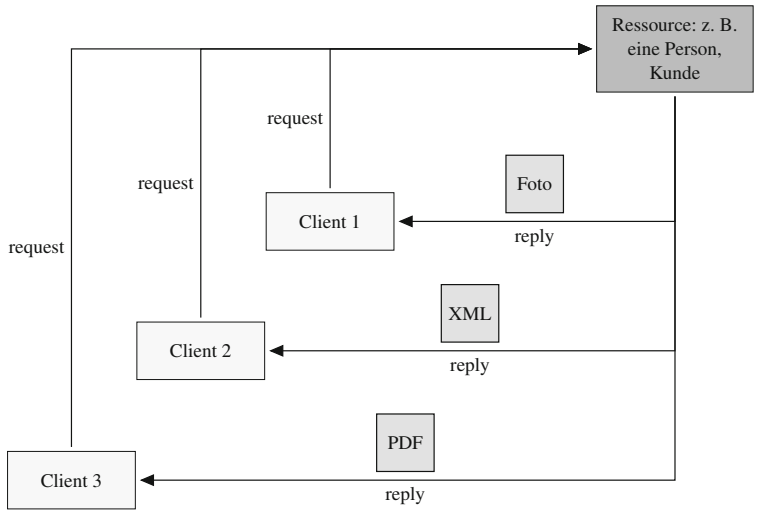


Abbildung 8.21 *REST Webservices: Clients fordern eine Ressource an. Die eindeutige Adressierung erfolgt über eine URL. Der Server gibt eine Repräsentation dieser Ressource zurück. Diese Repräsentation könnten Daten im XML-Format, ein JPG-Foto oder ein PDF-Dokument sein.*

2 aber auch Kommunikationsmuster an, die deutlich über Request/Reply hinausgehen. Es können z. B. Notifikationen versandt und asynchron gearbeitet werden. Da SOAP und WSDL kein bestimmtes Transportprotokoll definieren, können Webservice-Protokolle mit Messaging-Plattformen kombiniert werden und Messaging-Protkolle wie JMS eingesetzt werden.

REST

REST-Style Webservices sind eine in letzter Zeit immer beliebter werdende Möglichkeit, Kommunikation zwischen verteilten Systemen zu ermöglichen. REST ist wesentlich leichtgewichtiger als Webservices auf SOAP/WSDL-Basis. Die wesentlichen Ideen von REST sollen kurz angedeutet werden (es gibt eine größere Anzahl an Tutorials die hier ins Detail gehen[19]):

- Eine zentrale Rolle spielen *Ressourcen*; diese sind Endpunkte von Anfragen, oder anders ausgedrückt: Ein Client interessiert sich für eine Ressource.
- Jede Ressource ist eindeutig über eine *URL* adressierbar[20].
- Jede Ressource kann über eine oder mehrere *Repräsentationen* verfügen. Repräsentationen können je nach Anwendungsfall gewählt werden. Häufig ist die Repräsentation ein XML-Dokument, es kann aber auch ein Bild, eine Audio-Datei, ein HTML-, PDF-Dokument usw. sein (Abbildung 8.21 illustriert die Idee).
- REST Services verwenden ausschließlich http. Eines der Kernargumente lautet hier, dass das World Wide Web auf Basis des http-Proto-

[19] `http://del.icio.us/aschatt/rest`
[20] z. B. `http://www.example.com/order/o12345`

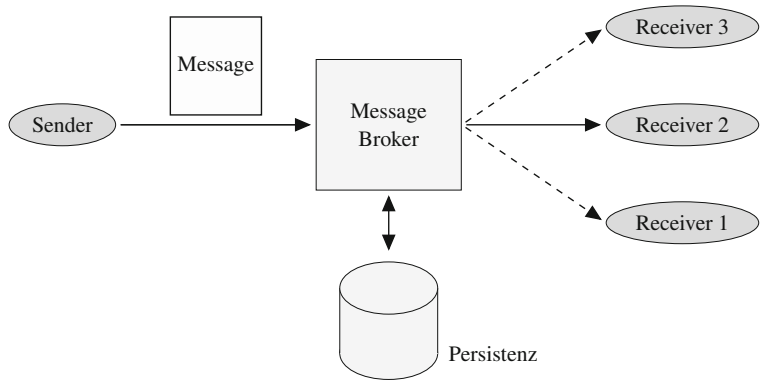


Abbildung 8.22
Integration mit Messaging: Ein Client sendet eine Nachricht an einen Broker, dieser leitet sie an einen oder mehrere Empfänger (Receiver) weiter.

kolls in den letzten 15 Jahren gezeigt hat, dass es robust, fehlertolerant und skalierbar ist.

- Es werden die http-Befehle `GET`, `POST`, `PUT` und `DELETE` verwendet.
- Mit `GET` wird eine Repräsentation einer Ressource vom Server angefordert[21].
- Mit `PUT` und `POST` wird der Zustand einer Ressource geändert[22].
- Mit `DELETE` wird eine Ressource gelöscht[23].

Man kann also erkennen, dass REST-Services eher datenorientiert funktionieren und daher einfach zu implementieren und zu verwenden sind. Bei SOAP-Services stehen eher Funktionen im Vordergrund. SOAP-Services sind tendenziell schwergewichtig und komplexer zu implementieren, erlauben aber auch eine klarere Spezifikation der Services.

Man hat bei Verwendung von Integrations-Middleware auch die Möglichkeit, ein Service in verschiedenen Technologien anzubieten, um den Kunden die freie Wahl zu lassen. So könnte man für ein Service nach außen hin sowohl eine CORBA-, eine SOAP-Webservice- als auch eine REST-Schnittstelle anbieten, wobei alle mit demselben Service im Backend interagieren.

Kommunikation über asynchrone Nachrichten

Eine der wesentlichsten modernen Integrationsstile ist das Messaging, also die Kommunikation über asynchrone Nachrichten, die von einem Message Broker vermittelt werden. Messaging ist besonders dann sehr elegant, wenn heterogene Systeme verbunden werden sollen, hohe Flexibilität und gleichzeitig starke Entkopplung der Systeme erwünscht sind.

[21] z. B. `GET http://www.example.com/order/o12345`
[22] z. B. `PUT http://www.example.com/person/adam-smith.jpg`
[23] z. B. `DELETE http://www.example.com/person/adam-smith`

Abbildung 8.22 stellt die Idee des Messaging vor. Ein Client möchte mit einem oder mehreren Empfängern kommunizieren. Er erstellt zunächst eine Nachricht (häufig Text oder XML). Diese sendet er an einen Message Broker (dies ist ein ganz ähnliches Prinzip wie E-mail, wo ein Mailserver sich um die Zustellung kümmert). Dieser Message Broker nimmt die Nachricht an und stellt sie an den oder die gewünschten Empfänger zu. Der Broker ist dafür verantwortlich, dass die Nachricht nicht verloren geht, sondern zuverlässig zugestellt wird.

8.6.2 Messaging

Beim Messaging werden Daten, die zwischen Anwendungen ausgetauscht werden sollen, in Pakete (Messages) verpackt und mithilfe eines Message Brokers ausgetauscht. Die Art der Daten, die ausgetauscht werden soll, ist im Prinzip beliebig. Das Nachrichtenformat ist jedoch häufig textbasiert bzw. XML. In manchen Anwendungsfällen werden tatsächlich Daten verschickt (z. B. eine neue Bestellung, ein Messwert usw.), in anderen Fällen wird die Messaging Middleware verwendet, um die Kommunikation zwischen Partnern zu entkoppeln und auch robuster gegen Störungen (wie z. B. dem zeitweisen Ausfall einzelner Systeme) zu machen. Als konkretes Beispiel könnte man hier SOAP over JMS nennen:

Beispiel: SOAP over JMS

Auf Seite 286 wurde kurz die Integration über Webservices erklärt. Das SOAP-Protokoll schreibt explizit kein konkretes Transportprotokoll vor. In Geschäftsanwendungen werden gerne Messaging-Systeme für den Transport der SOAP-Nachrichten verwendet. Die SOAP-Nachricht wird dann z. B. in eine JMS- (Java Messaging Service) Nachricht verpackt und über einen JMS Broker versandt. Dies ist zwar schwergewichtiger als http, dafür ist das System zuverlässiger und arbeitet asynchron. In diesem Fall würde die Messaging-Lösung nicht unbedingt reine Daten übertragen, sondern einfach die SOAP-Nachricht transportieren, und diese könnte z. B. einen Methodenaufruf darstellen.

Die Nachricht (Message)

Besprechen wir jetzt die Grundlagen von Messaging-Ansätzen systematisch: Das erste wesentliche Element ist die Nachricht (Message). Dabei handelt es sich, wie schon erwähnt, häufig um ein reines Textformat; gängige Messaging-Middleware-Lösungen können aber auch mit Binär-Nachrichten umgehen, teilweise auch mit kombinieren Formaten, ähnlich wie bei E-mails (Textnachrichten mit binären Attachments usw.).

In diesem Kapitel wird Messaging am Beispiel des sehr häufig verwendeten JMS- (Java Messaging Standard) Format erklärt, das zwar aus der Java-Welt kommt, aber mittlerweile in vielen Integrationslösungen auch mit nicht Java-Anwendungen verwendet wird. Andere Messaging Systeme haben ähnliche Funktionalität. Zunächst zu den Messages. Eine JMS-Message besteht aus mehreren Teilen:

- einem *Header*, in dem bestimmte formatspezifische Metadaten gespeichert werden (z. B. Message ID, Zeitstempel, „Ablaufdatum“, usw.),

- dem *Body*, das ist der Bereich, der beliebige (textuelle) Daten zum Inhalt haben kann, also die zu übermittelnden Daten, auch Nutzlast (Payload) genannt, beinhaltet

- und optionalen Eigenschaften (Properties), die der Benutzer beliebig für anwendungsspezifische Metadaten verwenden kann.

(JMS) Messages werden im Allgemeinen nicht „händisch“ zusammengebaut, sondern man verwendet spezifische Bibliotheken, die den Entwickler dabei unterstützen. Diese APIs erlauben das Setzen der Header-Informationen, das Erstellen des Bodies sowie der optionalen Properties. Die JMS API bietet hierfür verschiedene Hilfsklassen für verschiedene Arten von Nachrichten an: z. B. für Text-Nachrichten oder binäre Nachrichten. Andere Messaging Systeme unterscheiden sich in Details.

Integration: Plattformunabhängige Messages

Diese Message-Objekte werden dann der (JMS-) Bibliothek übergeben, die sich dann um das Versenden an den Message Broker kümmert. Möchte man heterogene Systeme über Messaging integrieren, so empfiehlt es sich natürlich, die Nachrichten in einem plattformunabhängigem Format zu halten; als z. B. JMS-Textnachrichten, die im Body XML-Daten transportieren. Verwendet man Features, wie serialisierte Java-Objekte im Body, oder Features, die das Messaging-System anbietet, die aber nicht auf allen Plattformen unterstützt werden, erschwert man natürlich die Integration über Systemgrenzen hinweg oder macht sie unmöglich. Außerdem bindet man sich dann sehr stark an ein bestimmtes (Middleware-) Produkt.

Kanal (Channel)

Die nächste offene Frage ist, wie die Anwendungen, die miteinander kommunizieren sollen, miteinander verbunden werden. Bei der Messaging-Integration nennt man die Verbindung, über die Nachrichten zwischen zwei oder mehreren Anwendungen ausgetauscht werden, einen *Kanal (Channel)*. Aus Sicht der Middleware befindet sich der Channel im Message Broker. Im Messaging haben sich zwei wesentliche Arten von Channels etabliert[24].

- Point-to-Point: eine 1:1-Verbindung zwischen zwei Anwendungen; wird meist im Message Broker über eine Queue realisiert.

[24] Daneben sind eine Reihe von anderen Channels beschrieben, z. B. *Invalid Message Channel*, ein Kanal, an den ein Empfänger Nachrichten senden kann, mit denen er nichts anfängt, oder ein *Dead Letter Channel*, ein Kanal, an den die Messaging Middleware Nachrichten sendet, die nicht zustellbar sind. Details kann man in [41] nachlesen.

- Publish-Subscribe: einer oder mehrere Sender senden, ein oder mehrere Empfänger erhalten die Nachricht; wird im Message Broker üblicherweise über ein sogenanntes *Topic* realisiert.

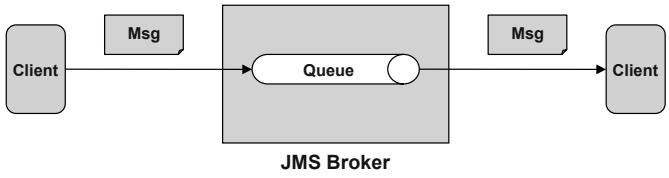


Abbildung 8.23
Point-to-Point-Messaging mit JMS.

Point-to-Point

Im ersten Szenario (siehe Abbildung 8.23) sieht man einen Client, der eine Nachricht Point-to-Point an einen anderen Client sendet. Der Message Broker stellt dafür eine Queue zur Verfügung, an die der Client eine Nachricht sendet. Diese (und weitere Nachrichten) bleiben solange in der Queue, bis sie der Empfänger abholt oder sie dem Empfänger zugestellt werden können (bzw. bis das „Ablaufdatum" (*Expiration*) der Nachricht erreicht ist). Man erkennt hier die Entkopplung der beiden Systeme: beide müssen z. B. nicht genau zum gleichen Zeitpunkt verfügbar sein. Der Broker hält die Nachricht vor, bis der Empfänger Zeit hat, sie zu empfangen.

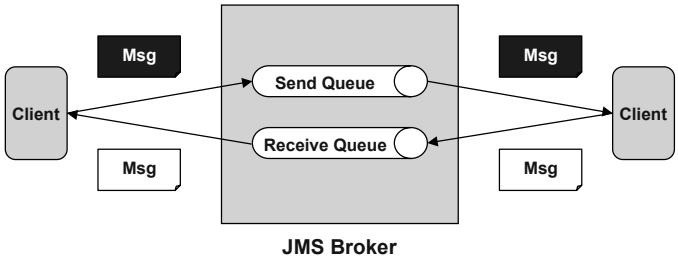


Abbildung 8.24
Request/Reply mit zwei JMS-Queues.

Request/Reply

Das Point-to-Point Schema ist auch gut geeignet, wenn zwei Anwendungen miteinander kommunizieren müssen, um ein Request/Reply-Schema abzudecken (z. B. einen Remote Procedure Call mittels SOAP over JMS): Das zweite Szenario in Abbildung 8.24 illustriert wie ein Request/Reply-Zyklus mit zwei Queues abgebildet werden kann.

Publish-Subscribe

Besonders interessant ist aber das Publish-Subscribe-Prinzip, wie es das dritte Szenario zeigt (siehe Abbildung 8.25): Hier wird ein sogenanntes *Topic* (also „Thema") verwendet. Im Prinzip ist ein Topic einer Queue sehr ähnlich, nur mit einem wesentlichen Unterschied: für ein Topic können sich mehrere Empfänger interessieren. Das bedeutet, ein Sender könnte eine Nachricht an ein Topic senden. Beliebig viele Empfänger können sich für dieses Topic interessieren und registrieren und bekommen neue Nachrichten, die an dieses Topic geschickt werden, automatisch zugestellt. Ein Vorteil dieses Ansatzes ist, dass der oder die *Sender* nicht wissen müssen, wer sich für das entsprechende *Topic* interessiert und die Nachricht daher

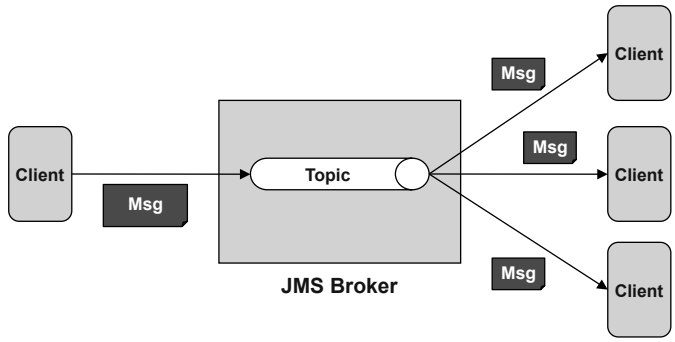


Abbildung 8.25
Publish/Subscribe mit JMS-Topics.

zugestellt bekommt. Daher ist dies eine sehr starke Form der Entkopplung.

Event-Driven Architecture

Zur Vertiefung der Idee der Entkopplung siehe Abschnitt 7.7. Dort werden ereignisgetriebene Architekturen (event driven architectures) vorgestellt, die auf diesem Prinzip aufbauen.

Beispiel: Pub/Sub, Observer

In Abbildung 8.12 wurde das Beispiel der Staumauer gebracht, in der Sensoren den Wasserstand messen und Änderungen an interessierte Objekte mittels Observer-Pattern weiterleiten. Den Publish-Subscribe-Mechanismus mittels Topic kann man sich wie ein verteiltes Observer-Pattern vorstellen: Der Sensor würde Datenänderungen als (JMS-) Message an ein Topic im Broker senden, verschiedene andere Systeme können sich am Topic anmelden und bekommen dann alle Nachrichten vom Broker zugestellt, die an dieses Topic gesandt werden.

Endpoints

Zuletzt muss man bei der Implementierung einer Messaging-Lösung entscheiden, wie Empfänger (Consumer) an das Nachrichtensystem angebunden werden, d. h., in welcher Weise sie eine neue Nachricht erhalten. Hier gibt es verschiedene Möglichkeiten; die wichtigsten sind:

- Polling Consumer,
- Event driven Consumer,
- Competing Consumer,
- Selective Consumer.

Der *polling Consumer* ist ein Client, der in regelmäßigen Abständen oder nur bei Bedarf beim Server nachfragt, ob neue Nachrichten für ihn vorhanden sind. Ein E-mail- Client, der per POP mit dem Mailserver kommuniziert, wäre ein Beispiel. Das E-mail-Programm kontaktiert z. B. alle fünf Minuten den Mailserver und prüft, ob neue Mails angekommen sind.

Beim *event driven Consumer* ist die Logik umgekehrt: Hier verfügt der Server über eine Möglichkeit, potenzielle Empfänger über neue Nachrichten zu informieren. Verwendet man JMS, kann die Klasse, die Nachrichten

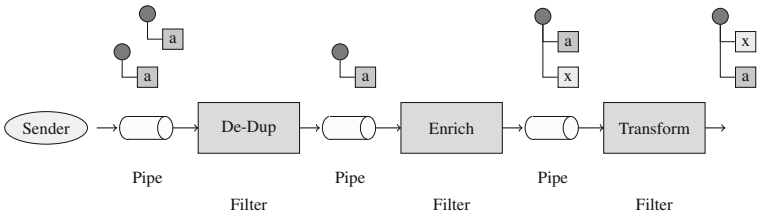


Abbildung 8.26 *Beispiel für das Pipes- und Filter-Pattern.*

empfangen soll, das `MessageListener`-Interface implementieren. Vom Framework wird dann die `onMessage(Message m)`-Methode aufgerufen, sobald eine neue Nachricht eingegangen ist.

Competing Consumer sind ein Spezialfall von Point-to-Point-Verbindungen. Gibt es nur einen einzelnen Consumer an einer Queue, so kann der Fall eintreten, dass mehr Nachrichten in die Queue geschrieben werden als der einzelne Empfänger verarbeiten kann (z. B. weil die Verarbeitung lange dauert oder weil mehrere Sender in diese Queue schreiben). Für diesen Fall kann man mehrere Konsumenten an einen Kanal hängen, und der nächste verfügbare Konsument (Client) nimmt die nächste Nachricht entgegen. Dies kann gegebenenfalls auch eine sinnvolle Failover-Technik sein, um dem Ausfall einzelner Konsumenten leicht entgegensteuern zu können.

Der *selective Consumer* schließlich ist eine Variante, die viele Messaging-Lösungen (auch JMS) anbieten: Hierbei hängt sich ein Empfänger nicht nur an einen Kanal und empfängt alle Nachrichten dieses Kanals, sondern schränkt zusätzlich ein, an welchen Nachrichten dieses Kanals er konkret interessiert ist. Dies entspricht einem Filter; es können z. B. Bedingungen formuliert werden, die auf Header- und Nachrichteninformationen zurückgreifen.

Message Broker Features

In diesem Abschnitt wurden nur die wesentlichsten Ideen des Messaging erklärt. Moderne Message Broker verfügen über eine Vielzahl an weiteren Möglichkeiten. Es werden z. B. auf Wunsch Messages persistiert, um im Failover-Fall keine Datenverluste zu haben; es können Cluster von Brokern erstellt werden, um zu skalieren oder für Ausfallsicherheit zu sorgen; Broker können Protokolltransformationen durchführen und vieles mehr.

Middleware

Im Abschnitt 9.7.2 wird ein Überblick über verschiedene Middleware-Komponenten gegeben, zu denen auch Message Broker zählen.

8.6.3 Pipes and Filters

Das *Pipes and Filters* Pattern beschreibt, wie ereignisverabeitende Systeme effizient und gleichzeitig flexibel Nachrichten verarbeiten können: Eine Pipe entspricht dabei einem Kanal (Channel), wie im vorigen Abschnitt beschrieben. Ein Filter ist eine Komponente, die ein einfaches Interface implementiert. Ein Filter akzeptiert eine Message, und das Ergebnis des Filters ist wieder eine Message.

Implementierung

Mit diesem Pattern ist es daher möglich, im Prinzip beliebige Verarbeitungsketten zu implementieren und generisch implementierte Filter wiederzuverwenden. Dieses Pattern kann man mit nachrichtenorientierter Middleware und gegebenenfalls ESBs (Enterprise Service Bus) umsetzen. Aber auch andere Systeme basieren auf diesem Konzept, z. B. das XML Framework Apache Cocoon[25]. Auch Unix Pipes folgen einem sehr ähnlichen Ansatz.

Beispiel

Abbildung 8.26 zeigt ein Beispiel: Eine Nachricht wird an eine eingehende Pipe geschickt und durchläuft dann eine Verarbeitungskette. In diesem Fall soll zunächst sichergestellt werden, dass doppelte Nachrichten entfernt werden, dann wird die Nachricht mit weiteren Daten angereichert und zuletzt in ein anderes Format konvertiert.

8.6.4 Routing

Nachrichtentypen

Folgt man der Idee der Integration über Messaging, wie in den vorigen Abschnitten erklärt, so stößt man in der Praxis häufig auf ein Problem: Viele integrierte Systeme erzeugen eine große Anzahl an verschiedenen Nachrichtentypen. Auch ein einzelnes System kann verschiedenste Nachrichtentypen erzeugen, z. B. könnte ein Webshop folgende Arten von Nachrichten erzeugen:

- Neue Bestellung,
- Bestellung löschen,
- Neuer Kunde,
- Anfrage eines Kunden.

Eine Nachricht vom Typ „Neue Bestellung“ könnte wiederum einige Unterarten haben, z. B. könnte der Kunde, der die Bestellung aufgegeben hat, ein Neukunde, ein Stammkunde, ein „Premium-Kunde“ oder ein Kunde sein, der seine alten Rechnungen nicht bezahlt hat. Eine Anfrage eines Kunden kann eine Beschwerde sein, eine Anfrage über einen Lieferstatus oder eine Produktanfrage.

One channel per message type

Wie geht man nun in einem Messaging-System mit vielen verschiedenen Arten von Nachrichten um? Eine offensichtliche Möglichkeit ist die Strategie: *one channel per message type*. Dabei werden für alle Arten von Nachrichten eigene Kanäle (Queues oder Topics) im Broker definiert. In manchen Fällen kann dies ein vernünftiger Lösungsansatz sein, häufig ist dies aber nicht unproblematisch:

[25] `http://cocoon.apache.org`

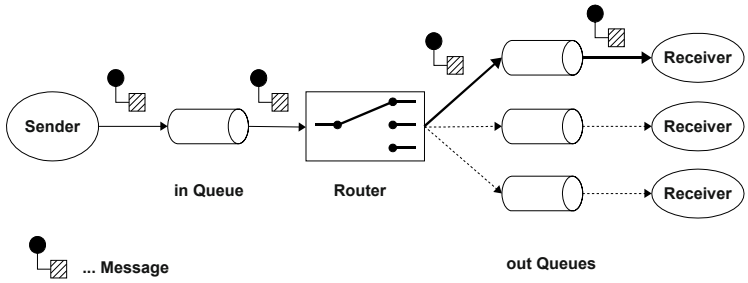


Abbildung 8.27
Illustration der Router-Komponente: Der Router entscheidet, an wen eine Nachricht weiterzuleiten ist.

- Bei vielen Systemen und vielen verschiedenen Arten von Nachrichten kommt es zu einer großen Anzahl an Kanälen. Dies kann schnell unübersichtlich werden.
- Die Kopplung zwischen Sender und Empfänger ist relativ stark.
- Der Sender muss viel Wissen über das System haben, denn er muss entscheiden, um welche Art von Nachricht es sich im Detail handelt und diese an den entsprechenden Kanal senden.
- Die Entscheidung für einen bestimmten Empfänger kann vom Kontext abhängen und unter Umständen vom Sender gar nicht entschieden werden.

Router

Aus diesen Gründen wird häufig eine weitere Komponente eingeführt, der sogenannte *Router*. Der Router kann in die Kette der Nachrichtenverarbeitung am Broker oder im Enterprise Service Bus (siehe Abschnitt 9.7.2) eingehängt werden und trifft die Entscheidung über die Weiterleitung einer bestimmten eingehenden Nachricht. Um das obige Beispiel wieder aufzunehmen: Es könnten am Broker vier verschiedene Kanäle definiert sein („Neue Bestellung“, „Bestellung löschen“, „Neuer Kunde“, „Anfrage“) und zusätzlich ein oder mehrere Router definiert werden, die dann weitere Unterscheidungen vornehmen. Also z. B. ein Router der anhand des Inhaltes der „Anfrage Nachricht“ entscheidet, um welche Art von Anfrage es sich handelt, und diese dann an den entsprechenden Empfänger weiterleitet.

Routing Varianten

Abbildung 8.27 zeigt das Prinzip eines Routers. Es gibt verschiedenste Kriterien nach denen ein Router die Routing-Entscheidung vornimmt, und Router werden auch nach dieser Funktionsweise kategorisiert. Die häufigsten Routing-Varianten sind:

- *Content-based Routing*: hierbei entscheidet der Inhalt der Nachricht.
- *Message Filter* sind ein Spezialfall des Routings, dabei werden nicht erwünschte Nachrichten des Kanals ausgefiltert.
- Ein *Splitter* teilt eine Nachricht in mehrere kleinere Nachrichten auf (z. B. aus einer Nachricht, die eine Bestellung mit mehreren Artikel enthält, werden n-Nachrichten, eine pro Artikel).

- Ein *Aggregator* ist das Gegenstück zum Splitter, hier werden mehreren Nachrichten zu eine Nachricht zusammengefasst.
- Ein *Resequencer* ist eine Komponente, die Nachrichten wieder in die richtige Reihenfolge bringt[26].

8.6.5 Transformation

In Integrationsszenarien kommt es immer wieder vor, dass ein System Nachrichten in einem Format versendet, das andere Systeme nicht verstehen. Manchmal sind nur Teile der geschickten Nachricht für das Zielsystem von Interesse, aber auch das Gegenteil kommt vor: Die Nachricht des Quellsystems enthält nicht alle Informationen, die das Zielsystem benötigt.

Um Probleme dieser Art zu lösen, kann man Transformations-Patterns verwenden. Oft verwendete Transformationen sind:

- *Message Translator*: Also das Übersetzen eines Formats in ein anderes, gleichwertiges Format[27].
- *Content Enricher*: Hier wird eine Nachricht um Information angereichert.
- Ein *Content Filter* kann verwendet werden, um eine (große) Nachricht eines Quellsystems zu verkürzen, wenn die Zielsysteme nicht die komplette Information benötigen.

Chaining

In einigen Fällen muss für die Transformationen kein eigener Code geschrieben werden, sondern es können bei XML-Nachrichten XSLT-Transformationen verwendet werden. Häufig bietet die Middleware auch die Möglichkeit des sogenannten *Chaining* an, also der Verkettung von Transformationsschritten. Existiert z. B. ein Transformer X, der Nachrichtentyp B in C und eine Transformer Y, der Nachrichtentyp A in B umwandelt, so können X und Y verkettet und damit A in C tranformiert werden. Damit kann man ein größeres Maß an Flexibilität in der Konversion zwischen verschiedenen Formaten erreichen.

[26] Es ist wichtig zu verstehen, dass in einer nachrichtenorientierten Architektur praktisch nie garantiert werden kann, dass Nachrichten beim Empfänger genau in der Reihenfolge ankommen, in der sie versendet wurden. Dies kann auf Effekte des Netzwerks zurückzuführen sein oder viele andere Ursachen haben.

[27] Dieser Ansatz entspricht etwa dem Adapter-Pattern, das in Abschnitt 8.4.3 beschrieben wurde.

8.6.6 System-Management und Testen

Integration von Systemen über nachrichtenorientierte Middleware hat Vorteile, z. B. die stärkere Entkopplung der Systeme, die Möglichkeit, Komponenten wiederzuverwenden sowie leichter skalieren zu können. Gleichzeitig erkauft man sich diese Vorteile mit einem Preis: Die Übersicht und Kontrolle in solchen verteilten Systemen nicht zu verlieren und besonders auch Fehlerfälle zu identifizieren und entsprechend darauf reagieren zu können, kann eine Herausforderung darstellen. In dieser kurzen Einführung kann nicht im Detail beschrieben werden, wie nachrichtenbasierte Systeme zu entwickeln, testen und warten sind, aber einige grundsätzliche Ideen sollen erwähnt werden:

Control Bus

Eine wesentliche Vorgehensweise, um verteilte nachrichtenorientierte Systeme unter Kontrolle zu halten, ist das Einführen eines *Control Bus*. Dabei wird häufig dieselbe Infrastruktur wie für die Behandlung „normaler" Nachrichten verwendet, aber die Komponenten am System verwenden eigene Kanäle für Kontroll-Nachrichten. D. h., eine Komponente im System verfügt dann typischerweise über drei Interfaces: eines für eingehende *operative* Nachrichten, eines für ausgehende *operative* Nachrichten sowie eines für Kontroll-Nachrichten. Bei Kontroll-Nachrichten handelt es sich meist um Nachrichten, die die Konfiguration einer Komponente verändern sollen (und daher besonderen Sicherheitsanforderungen unterliegen). „Heartbeat-Nachrichten" dienen dazu, festzustellen, ob eine Komponente noch „am Leben ist", also auf Anfrage reagiert. Ist dies nicht mehr der Fall können entsprechende Maßnahmen eingeleitet werden. Zudem werden häufig auch statistische Daten über diesen Kanal ausgetauscht; die Komponente könnte also über den Nachrichtendurchsatz oder ähnliche Informationen über diesen Kanal berichten. Der Control Bus kann also eine wesentliche Hilfe für die Administration und die Überwachung des Verhaltens eines laufenden Systems darstellen und sollte schon bei der Planung entsprechend berücksichtigt werden.

Wire-Tap

Verhalten sich (Gruppen von) Komponenten nicht so, wie man das erwarten würde, kann man Kommunikationskanäle abhören. Dies ist im Wire-Tap-Pattern beschrieben. Die Umsetzung dieses Patterns ist beispielsweise vom Apache Camel Framework unterstützt. Ein *Wire-Tap* ist also eine „Abhörmaßnahme", dabei wird z. B. aus einem Point-to-Point-Kanal, der einen Sender und einen Empfänger verbindet, ein Kanal, der einen Eingang und zwei Ausgänge hat: ein Ausgang richtet sich wie gehabt an den ursprünglichen Empfänger, der zweite Ausgang wird von der Komponente verwendet, die die Kommunikation beobachtet, um beispielsweise Fehler zu beheben.

Test-Message

Test-Messages können auch im laufenden Betrieb eines Systems eingesetzt werden, um z. B. in regelmäßigen Intervallen die korrekte Funktion des Systems zu prüfen. Dabei kann man wie folgt vorgehen:

- Man führt einen *Nachrichten-Generator* ein, der Test-Nachrichten generiert, bei denen das Ergebnis der Verarbeitung bekannt ist.
- Diese Komponente sendet z. B. in bestimmten Intervallen Testnachrichten in das System; dabei ist aber zu beachten, dass die Testnachricht als solche zu kennzeichnen ist (z. B. durch ein entsprechendes Feld im Header).
- Die zu testende(n) *operative(n) Komponente(n)* verarbeiten diese Testnachrichten, kennzeichnen aber die Ausgabenachrichten ebenso als Test-Nachricht.
- Ein *Content based Router* entfernt schließlich die Nachrichten der Test-Ergebnisse aus dem Nachrichtenstrom und führt sie einer Validierungskomponente zu.
- Diese *Validierungskomponente* prüft nun, ob das Ergebnis der Erwartung entspricht.

Message History

Zuletzt sollen noch *Message History* und *Message Store* erwähnt werden: In den vorigen Abschnitten wurde besprochen, wie Nachrichten zur Kommunikation zwischen heterogenen Systemen eingesetzt werden können. In der Kommunikation zwischen den Systemen geht aber häufig die Geschichte des Nachrichtenflusses verloren. Sendet beispielsweise die Web-Anwendung eine Nachricht an das Topic *NewOrder* des MessageBrokers, dass eine neue Bestellung eingegangen ist, so kann das Bestellsystem und verschiedene andere Systeme an dieser Nachricht interessiert sein und entsprechend dieses Topic subskribiert haben. Das Bestellsystem könnte in weiterer Folge neue Nachrichten generieren, z. B. solche, die eine Prüfung der Kreditwürdigkeit veranlassen. Nun gibt es aber mehrere Systeme, die die Kreditwürdigkeit prüfen, abhängig davon, welche Bezahlmethode der Kunde gewählt hat. Die entsprechende Antwort wird als *CreditCheck*-Nachricht versandt. Das Bestellsystem wartet auf die entsprechende Information, um die Bestellung weiterverarbeiten zu können.

Nehmen wir nun an, dass die CreditCheck-Nachrichten fallweise Fehler aufweisen, z. B. in einem fehlerhaften Format ankommen. Bei der Fehlersuche hat man nun die Schwierigkeit, dass aus der Nachricht selbst nicht leicht festgestellt werden kann, welchen Weg die Information durch das System genommen hat: War der Weg: *Web-Anwendung – Bestellsystem – Kreditkartenprüfung – Bestellsystem*, oder *Web-Anwendung – Bestellsystem – Kontoprüfung – Bestellsystem* oder etwa *Kundenbetreuer mit Fat-Client – CRM-System – Kreditkarte – Bestellsystem*?

Um derartige Fragestellungen beantworten zu können, kann man den Weg der Nachricht oder den Ursprung der Information in der neuen Nachricht im Header abspeichern, dann könnten spätere Komponenten den Weg der Nachricht(en) rekonstruieren und dann z. B. feststellen, dass der Fehler nur unter bestimmten Konstellationen auftritt. Diese Lösung zeigt sehr elegant

den Weg moderner IT-Architekturen: einerseits bleiben sie entkoppelt und flexibel, man führt keine neue Top-down-Kontrolle ein, schafft aber die Möglichkeit, im Nachhinein das Verhalten des Systemes leicht analysieren und damit das tatsächliche Verhalten des Systemes mit dem gewünschten Verhalten vergleichen zu können.

Message Store

Ein *Message Store* ist ähnlich der Idee der Message History; allerdings wird dabei eine zentrale Datenbank eingerichtet. Jede Nachricht wird (ähnlich wie beim Wire-Tap-Pattern) dann nicht nur an den Kanal für den oder die Empfänger gesendet, sondern zusätzlich auch an den Kanal der Datenbank, die die entsprechende Nachricht abspeichert. Damit kann an einer zentralen Stelle der Lauf der Nachrichten sowie die Nachrichten selbst nachvollzogen werden.

8.7 Zusammenfassung

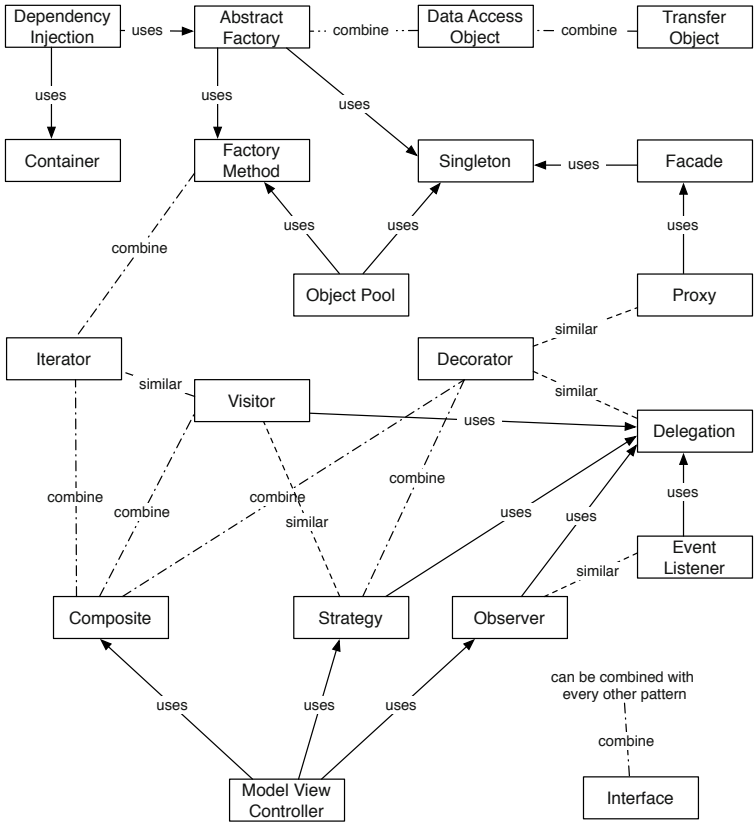


Abbildung 8.28
Pattern-Map nach Zimmer mit drei Arten von Beziehungen: (a) „uses“ (Muster wird verwendet von), (b) „combine“ (Muster kann kombiniert werden mit), und (c) „similar“ (Muster sind vom Design ähnlich).

Design-Patterns oder Muster sind bewährte Vorlagen für Lösungen zu wiederkehrenden Problemstellungen in der Software-Entwicklung. Gut kon-

struierte Software verwendet eine Kombination von verschiedenen Mustern, die sich gegenseitig unterstützen und ergänzen. Diese Muster fallen in die Kategorien: (a) grundlegende Muster, (b) Erzeugungsmuster, (c) Strukturmuster, (d) Verhaltensmuster und (e) Integrationsmuster.

Grundlegende Muster

Grundlegende Muster sind ein essenzieller Teil von modernen Applikationen und werden von den meisten anderen Mustern verwendet, um ihre Aufgaben zu erfüllen. Manche dieser Muster sind bereits fester Bestandteil von neueren Programmiersprachen, beispielsweise wurde bei Java das Interface-Pattern von Anfang an als „Keyword“ eingeplant. In aktuellen Versionen von Java ist es auch üblich, Annotationen zu verwenden. In der Enterprise-Version von Java sind sie sogar notwendig, um dessen Funktionalität vollständig auszunutzen.

Erzeugungsmuster

Alle Erzeugungsmuster beschäftigen sich mit dem besten Weg, Instanzen von Klassen zu erzeugen. In vielen Fällen ist die genaue Version oder Konfiguration eines benötigten Objekts von den Bedürfnissen der Applikation und dem jeweiligen Kontext während der Erzeugung abhängig. Den Erzeugungsprozess in eine spezielle „Erstellungs-Klasse“ zu abstrahieren, macht die Applikation flexibler und generischer.

Strukturmuster

Strukturmuster vereinfachen das Design einer Applikation, indem sie neue Wege identifizieren, um Beziehungen zwischen Entitäten zu realisieren. Durch Vererbung werden nützliche Schnittstellen angeboten, wie beispielsweise im Data Access Object. Andere Patterns, wie Decorator oder Proxy, beschreiben, wie Objekte miteinander verschachtelt und zur Laufzeit ausgetauscht werden können, um völlig neue Schnittstellen anzubieten.

Verhaltensmuster

Verhaltensmuster beschäftigen sich ausdrücklich mit der Kommunikation zwischen Objekten. Sie erhöhen die Flexibilität, gewährleisten aber dennoch die erwünschte lose Kopplung.

Integrationsmuster

Integrationsmuster finden sich auf einer höheren Ebene als die bisher besprochenen Muster wieder. Sie erleichtern die Kommunikation zwischen verschiedenen Systemen, die möglicherweise auch von verschiedenen Herstellern stammen, sodass die heute geforderte Flexibilität gewährleistet bleibt.

Übersicht

Abbildung 8.28 zeigt einen Überblick über die beschriebenen Muster und wie sie miteinander in Beziehung stehen. Für die Darstellung der Beziehungen wurde die Notation von Zimmer [94] verwendet.

9 | Komponentenorientierte Software-Entwicklung

Sobald die Anforderungen feststehen, überlegt der Software-Architekt, wie das Software-System gebaut werden soll. Kernfragestellungen sind, in welche Teile (Komponenten) sich die Software zerlegen lässt, und ob manche Teile mit bereits bestehenden Software-Produkten realisiert werden können. Generell zielen Komponenten auf die Verbesserung der Wiederverwendung von Software-Investitionen ab und ermöglichen die Komposition komplexer Software-Systeme aus einfacheren Teilkomponenten sowie den einfacheren Austausch von Systemteilen im Laufe der Wartung und Evolution eines Anwendungssystems. Dieses Kapitel fasst zusammen, an welchen Stellen einer Software-Architektur welche Konzepte anzuwenden sind und wie die genannten Begriffe zueinander in Bezug stehen.

Übersicht

9.1 Vom Objekt zum Service: Schritte der Entkopplung

Schon in den vorigen Abschnitten – etwa bei der Erklärung verschiedener Muster – sind die Begriffe *Objekt* und *Interface* sowie fallweise auch *Komponente* und *Service* gefallen. Dieser Abschnitt fasst zusammen, an welchen Stellen einer Software Architektur welche Konzepte anwendbar sind, und wie die genannten Begriffe zueinander in Bezug stehen. Denn beim Entwurf von Systemen hat man die Wahl, wie eng man bestimmte Objekte innerhalb einer Anwendung aneinander koppelt und wie weit bestimmte Objekte zu größeren Einheiten mit klar definierten Schnittstellen nach außen aggregiert werden (Komponentenbildung). Sollen die Komponenten verteilt arbeiten oder mit externen Systemen interagieren, so können sie entweder über plattformspezifische Protokolle (wie RMI) kommunizieren oder über plattformunabhängige Serviceschnittstellen (z. B. Webservices, REST). In diesem Abschnitt wird diskutiert, auf welcher Abstraktionsebene welches Konzept angesiedelt ist und mit welchen *Kosten* und *Nutzen* man beim Einsatz zu rechnen hat. Abbildung 9.1 dient dabei als roter Faden durch diesen Abschnitt (die Abbildungen 9.2 bis 9.5 greifen jeweils einen Fall im Detail heraus).

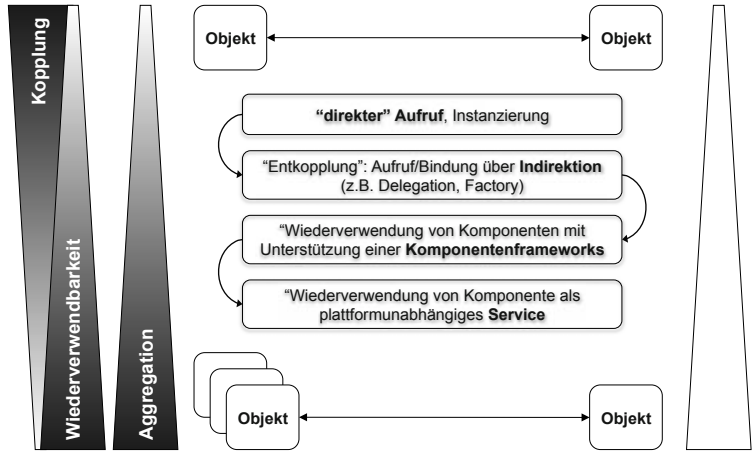


Abbildung 9.1 *Vom Objekt zum Service; Entkopplung und Abstraktion über mehrere Schritte.*

9.1.1 Objekte und Schnittstellen (Interfaces)

Objekte

Abbildung 9.2 deutet den *einfachsten* Fall an: *ein Objekt* bildet eine Instanz von einem oder mehreren anderen Objekten und verwendet diese Instanz in weiterer Folge, um mit diesen Objekten zu kommunizieren.

Dies ist der häufigste Fall von Interaktion zwischen Objekten in einer objektorientierten Sprache und entspricht etwa dem Delegationsmuster (siehe Abschnitt 8.2.2). Dabei wird (in Java) der `new()`-Operator verwendet, um

Abbildung 9.2
„Direkter“ Aufruf, d. h. enge Kopplung zweier Objekte über Instanziierung.

eine Instanz einer anderen Klasse zu bilden und eine Referenz auf diese Instanz z. B. einer lokalen Variable oder einer Instanzvariable zuzuweisen:

```
class KlasseA() {
  private KlasseD instanzD = new KlasseD();
    // Instanz von KlasseD wird über
    // Instanzvariable instanzD verwendet
  public void x() {
    KlasseB instanzB = new KlasseB();
    // Instanz von KlasseB wird über
    // lokale Variable instanzB verwendet

  }
}
```

Vorteile

Dieser Fall kommt in jedem Programm, das in einer objektorientierten Sprache geschrieben ist, sehr häufig vor. Was sind nun die Vor- und Nachteile der direkten Bindung von Objekten mittels Delegation? Zunächst zu den Vorteilen: Dieses Muster ist leicht zu implementieren, der Zugriff auf die andere Instanz ist einfach und effizient. Weiter ist die Bindung zwischen den Objekten zum Zeitpunkt der Übersetzung bekannt. Dies hat mehrere Vorteile: Einerseits kann der Compiler Optimierungen am Code vornehmen, andererseits kann eine Entwicklungsumgebung (IDE) oder ein Reverse-Engineering-Werkzeug die Beziehungen zwischen den Objekten erkennen und mit dieser Information den Entwickler unterstützen. Auch für den Entwickler ist dieser Code leicht verständlich und nachvollziehbar: Instanzen von Klasse `KlasseA` verwenden Instanzen von Klasse `KlasseB` und `KlasseD`. Dies ist bekannt, und daran ändert sich auch später zur Laufzeit nichts mehr.

Nachteile

Aus dem bisher gesagten ergeben sich aber gleichzeitig auch schon die Nachteile[1], die gerade in der sehr engen Bindung der Klassen zueinander liegen: Die Klassen `KlasseA`, `KlasseB` und `KlasseD` wissen sehr viel voneinander; möchte man beispielsweise `KlasseD` später durch die Klasse `KlasseE` austauschen muss man eventuell größere Änderungen im Sourcecode vornehmen. Möchte man es gar dem Benutzer der Software überlassen, ob Klasse `KlasseD` oder `KlasseE` verwendet werden soll (also eine Konfiguration zur Laufzeit, z. B. für einen Gerätetreiber), so ist dies mit der engen Bindung gar nicht möglich.

[1] Die Problematik der Delegation wurde im Detail bei der Erklärung der Muster-Delegation, Interface und Strategy analysiert (siehe Abschnitte 8.2.1 bis 8.2.3).

9.1.2 Erste Schritte der Entkopplung

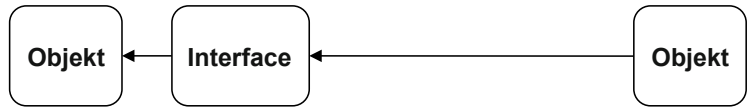


Abbildung 9.3
Entkopplung: Aufruf und Bindung mittels Indirektion über Interfaces.

Interfaces

Abbildung 9.3 illustriert den ersten Schritt zur Entkopplung. Dieser besteht meist darin, die Funktionalität, die benötigt wird, zunächst in Form eines *Interfaces* zu definieren und damit die Schnittstelle unabhängig von einer bestimmten Implementierung zu beschreiben. Danach wird dieses Interface durch eine oder mehrere Klassen implementiert.

```
1 public interface TextImport {
2   public Document read (String filename);
3 }
```

In diesem Beispiel wird ein Interface definiert, das dazu dient, Dokumente zu importieren. Auf Ebene des Interfaces ist es nur wesentlich, den Dateinamen zu wissen sowie eine Referenz auf eine `Document`-Instanz zurückzugeben; die `Document`-Klasse würde die interne Repräsentation des Dokumentes darstellen, die Datei die externe Repräsentation. Dann könnte man in verschiedenen Klassen dieses Interface implementieren, z. B.:

```
1  public class WordImport implements TextImport {
2    public Document read (String filename) {
3      // hier erfolgt die konkrete Implementierung
4      // des "Word"-Import-Filters
5    }
6  }
7  ...
8  public class HtmlImport implements TextImport {
9    public Document read (String filename) {
10   // hier erfolgt die konkrete Implementierung
11   // des HTML-Import-Filters
12   }
13 }
```

Benötigt man nun in einer Klasse einen HTML-Import, so kann man dies wie folgt implementieren:

```
1 class A {
2   TextImport ti = new HtmlImport();
3   public processDocument (String filename) {
4     Document doc = ti.read(filename);
5     ...
6   }
7 }
```

Implementierung des Interfaces

Das Entscheidende an diesem Beispiel ist die Tatsache, dass in der Klasse `A` die Variable `ti` vom Typ des `TextImport`-Interface ist, und an dieser Stelle keine konkrete Bindung an die `HtmlImport`-Klasse gemacht

wird. Tatsächlich ist es für den Rest des Codes, z. B. für die `processDocument()`-Methode, irrelevant, welche konkrete Klasse für den Import verwendet wird, solange diese das Interface implementiert.

In diesem Beispiel besteht zwar immer noch eine Bindung zwischen der Klasse `A` und der Klasse `HtmlImport`, aber der erste Schritt zu einer stärkeren Entkopplung (z. B. in einer späteren Version der Software) ist bereits gemacht. Möchte man die `TextImport`-Klasse ändern, muss man nur noch eine Zeile im Sourcecode tauschen. Außerdem ist die Bindung insofern gelockert, als nur noch das Interface verwendet wird, und nicht mehr eine konkrete Implementierung.

9.1.3 Software-Komponenten

Mehr Freiheit!

In manchen Fällen möchte man nun tatsächlich mehr Freiheitsgrade in der Bindung zwischen Klassen haben, d. h. die Information, welche konkrete Implementierung verwendet wird, soll nicht im Code stehen, sondern durch den Entwickler konfiguriert werden können. An dieser Stelle kommt die *Software-Komponente* ins Spiel, wie in Abbildung 9.4 dargestellt wird.

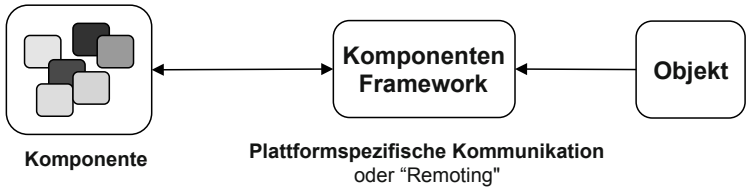


Abbildung 9.4
Aggregation von Funktionalität in Komponenten und Komponentenframeworks zur Entkopplung.

Der Begriff Komponente ist nicht eindeutig definiert, Konsens der Definitionen ist eine oder mehrere Klassen, die folgende Eigenschaften haben:

- Eine Komponente verfügt über eine *klare* und *stabile Schnittstelle*.
- Eine Komponente ist meist von *höherer Granularität* als eine einzelne Klasse, d. h., sie bietet meist Funktionen höherer Abstraktion an, z. B. durch Aggregation der Funktionalität mehrerer Klassen.
- Die Schnittstelle und die Komponente sind auf *Wiederverwendbarkeit* ausgelegt, d. h., die Komponente sollte in verschiedenen Programmteilen oder auch in verschiedenen Programmen verwendet werden können.
- Eine Komponente ist mit anderen Programmteilen *locker verbunden*, kann also bei Bedarf leicht ausgetauscht werden, ohne den Programmcode zu verändern (z. B. durch Änderung der Konfiguration).

Granularität

Diese Definitionen sind allerdings, wie gesagt, nicht sehr scharf, vor allem was die Granularität betrifft. Es kann durchaus im Einzelfall sein, dass eine

einzelne Klasse, die über recht einfache Funktionalität verfügt, als Komponente bezeichnet wird (z. B. ein Sortieralgorithmus oder eine Komponente, die Währungen umrechnet), weil sie an vielen Stellen im Programm verwendet wird oder aus Wartungsgründen leicht austauschbar sein soll.

Wartbarkeit

Gerade die Wartbarkeit von großen Programmen spielt in dieser Hinsicht eine wesentliche Rolle: wird ein Programm von verschiedenen Teams entwickelt, so erleichtert es die Entwicklung und die Qualitätssicherung, wenn die einzelnen Teile (Komponenten) des Programms möglichst entkoppelt voneinander entwickelt und getestet werden können.

Komponenten-Frameworks

Software Frameworks (siehe Abschnitt 9.2), konkret Komponenten-Frameworks, spielen eine wichtige Rolle bei der Entwicklung von Software, die sich aus Komponenten zusammensetzt: Ein Komponenten-Framework kümmert sich um die *Verdrahtung* der Komponenten, meist auch um die *Konfiguration* sowie um den *Lebenszyklus* der Komponenten. Dies soll an einem Beispiel betrachtet werden:

```
1  class KlasseA {
2    TextImport ti;
3    public KlasseA() {
4      // pseudocode:
5      ti = componentFramework.
6        getComponent ("htmlimport");
7    }
8    public processDocument (String filename) {
9      Document doc = ti.read(filename);
10     ...
11   }
12 }
```

In diesem Beispiel wird die Idee illustriert, Details sind im nächsten Abschnitt zu finden: In dieser Implementierung der Klasse `KlasseA` findet sich kein konkreter Bezug mehr zu einer der Import-Klassen. Die einzige Abhängigkeit besteht zur Schnittstelle (Interface). Welche Klasse nun für den Import herangezogen wird, entscheidet in diesem Beispiel das Komponenten-Framework: Der verwendende Entwickler gibt an, dass eine *htmlimport*-Komponente benötigt wird. Das Komponenten-Framework entscheidet nach einer wohl definierten Logik (z. B. durch Nachschlagen in einer Konfigurationsdatei), welche konkrete Klasse instanziiert werden soll. Auch andere Möglichkeiten sind denkbar: Das Komponenten-Framework könnte Bindungen anhand benötigter Interfaces automatisch vornehmen etc. Wichtig ist, an dieser Stelle folgende Aspekte zu verstehen:

- Die einzige konkrete Abhängigkeit der Klasse `KlasseA` besteht zum Interface (`Textimport`).
- Die Instanz einer Klasse, die das Interface implementiert, holt sich `KlasseA` vom Komponenten-Framework (KFw), ohne genau wissen zu müssen, welche Klasse das ist.

- Das KFw entscheidet anhand externer Logik oder Konfiguration, welche konkrete Klasse verwendet werden soll[2].
- Das KFw kümmert sich um die Konfiguration der Komponente.
- Außerdem verwaltet das KFw meist auch den *Lebenszyklus* der Komponente, d. h., es wird eine Instanz gebildet, gegebenenfalls das Singleton-Muster 8.3.1 verwendet; der Speicher nicht benötigter Komponenten wird wieder freigegeben.

Komponenten-Framework und das Factory Pattern

Ein Komponenten-Framework ist also locker formuliert eine bessere Factory (siehe Abschnitt 8.3.2). Ein Komponenten-Framework ist meist jedoch auf eine bestimmte Plattform beschränkt. Das Spring-Framework[3] beispielsweise integriert Java Komponenten[4]. In einigen Fällen werden für eine größere Anwendung Komponenten verschiedener Plattformen benötigt. Auch wenn eine stärkere Verteilung der Anwendung gewünscht oder notwendig ist, könnte die Funktionalität bestimmter Komponenten in einer plattformunabhängigen und noch stärker entkoppelten Weise zur Verfügung gestellt werden.

9.1.4 Software-Services

Der letzte Schritt ist in Abbildung 9.5 dargestellt und liegt in der Definition und Verwendung von *Services*, um Funktionen unabhängig von deren Implementierung zur Verfügung zu stellen.

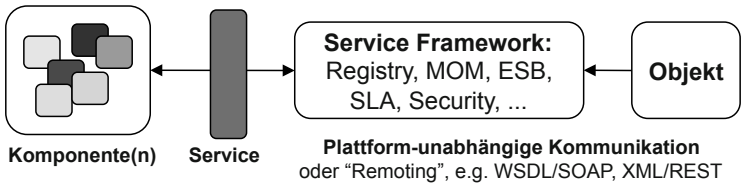


Abbildung 9.5 *Wiederverwendung von Komponenten als plattformunabhängige Services.*

Auch die Definition des Begriffs *Service* ist nicht immer ganz scharf. In diesem Kontext verstehen wir unter einem Service:

[2] Diese konkrete Klasse muss oftmals nicht einmal lokal vorliegen, das KFw könnte ggfs. auch einen Remoting-Mechanismus wie RMI oder Webservices verwenden, um die gewünschte Funktionalität von einem anderen Rechner aus zur Verfügung zu stellen.

[3] `http://www.springframework.org`

[4] Komponenten-Frameworks wie das Spring-Framework bieten auch die Möglichkeit an, Komponenten über Remoting-Mechanismen wie Webservices zu integrieren. Die Kernaufgabe von Komponenten-Frameworks wird jedoch meist im Management von Komponenten *innerhalb* einer Plattform gesehen.

- Es gibt eine klar definierte Schnittstelle, die die Funktionalität (oder einen Teil der Funktionalität) einer oder mehrerer Komponenten beschreibt, wobei die Verwendung in verschiedenen Kontexten (z. B. Plattformen) eine wesentliche Rolle spielt.
- Die Beschreibung dieser Schnittstelle erfolgt plattformunabhängig, d. h., es sind keine für die Plattform spezifischen Informationen enthalten (z. B. in WSDL [93]).
- Das Service selbst ist plattformunabhängig, d. h., dieses Service kann auf verschiedensten Systemen (Plattformen, Sprachen, Betriebssystemen, Netzwerkprotokollen) verwendet werden (z. B. durch Kommunikation mittels SOAP-Protokoll [81]).
- Das Service wird über ein Netzwerk angeboten (z. B. HTTP).

Kommunikation über Systemgrenzen

Damit ist der letzte Abstraktionsschritt vom Objekt zum Service passiert. Ein Service kann aus verschiedenen Systemen heraus verwendet werden, auch über Systemgrenzen hinweg, z. B. in der Kommunikation zwischen Firmen. Wichtig ist zu verstehen, dass mit der Definition von plattformunabhängigen Services ein hohes Maß an zusätzlicher Komplexität ins Spiel kommt: Da es sich um verteilte Architekturen handelt, die oft auch über Systemgrenzen hinweg angeboten werden, muss sich der Architekt und Entwickler meist auch mit weiteren komplexen (Webservice-) Frameworks auseinandersetzen: Oft sind Registries erforderlich, um Services zu verwalten. Security-Aspekte sind zu bedenken, und Service Level Agreements werden fallweise gefordert, die die Verfügbarkeit und Leistungsfähigkeit von angebotenen Services beschreiben und nachvollziehbar gestalten. Weiterhin bringt die Verwendung plattformunabhängiger Standards, wie SOAP [81], auch einen Mehraufwand mit sich, der bei unreflektiertem Einsatz von Services die Leistung des Systems empfindlich verringern kann.

Plattformunabhängige Standards

Um das an einem Beispiel zu illustrieren:

- Ein Objekt ruft ein SOAP-Service auf; dabei verwendet das Objekt eine Client-SOAP-Bibliothek.
- Diese Bibliothek wandelt den Aufruf in eine XML/SOAP-Nachricht um.
- Diese Nachricht wird über ein Netzwerk an das SOAP-Framework des Services geschickt.
- Eventuell wird der Aufruf noch über ein Service-Registry aufgelöst.
- Das Service-Framework muss diese XML-Nachricht wieder parsen und in einen plattformabhängigen Methodenaufruf umwandeln.
- Die Service-Methode wird ausgeführt und das Ergebnis wieder an das SOAP-Framework zurückgegeben.

- Das SOAP-Framework wandelt das Ergebnis wieder in eine XML-Nachricht um und sendet diese zurück an den Client.
- Die SOAP-Bibliothek am Client parst die XML-Nachricht und gibt das Ergebnis an das aufrufende Objekt zurück.

Man erkennt also, dass hier im Vergleich zur Verwendung eines Objekts oder einer Komponenten ein deutlicher Mehraufwand bei jedem Service-Aufruf zu berücksichtigen ist, der aber gerechtfertigt werden kann, wenn die Integration heterogener Systeme sowie die Kommunikation über Systemgrenzen hinweg im Vordergrund steht.

Es soll aber nicht unerwähnt bleiben, dass über die konkrete Bedeutung des Begriffs *Service* unterschiedliche Ansichten herrschen. Ein erster wesentlicher Aspekt ist die Frage der Granularität: ein Service muss nicht immer eine grob-granulare Komponente verbergen, sondern kann im Einzelfall auch nur eine einfache Funktion abdecken, z. B. das immer wieder zitierte Service, das Wechselkurse berechnet. Es steht, wie oben erwähnt, bei einem Service die Wiederverwendung in verschiedenen Kontexten zumeist im Vordergrund. Derartige Funktionen sind meist eher auf einer höheren Aggregationsstufe zu finden, aber eben nicht immer.

Daher wird die Diskussion geführt, ob ein Service letztlich die Sicht auf eine Komponente ist oder einfach als *Integration Gateway* verstanden wird, also ein von wenigen Rahmenbedingungen abhängiger Zugang zu bestimmten Funktionen oder Daten. Eine gute Zusammenfassung der Diskussion um die Definition von Services diskutieren Martin Fowler und David Ing [48].

Synchrone vs. asynchrone Kommunikation

Ein letzter Aspekt der Entkopplung wird hier kurz erwähnt: In den meisten Fällen erfolgen Methodenaufrufe sowie Kommunikation mit Komponenten und oft auch Kommunikation mit Services *synchron*. Das bedeutet, dass der Client wartet, bis die Komponente oder das Service den Aufruf bearbeitet und das Ergebnis zurückgegeben hat. In einigen Fällen, vor allem wenn es um die Integration verschiedener Software-Systeme geht, versucht man nach Möglichkeit synchrone Aufrufe zu vermeiden und auf *asynchrone* Kommunikation zu setzen. Damit werden die Client- und Server-Systeme noch stärker voneinander entkoppelt. Diese Entkoppelung erfordert sowohl neue Middleware-Komponenten als auch eine andere Architektur der zu integrierenden Systeme. Details zu Integrationsmustern und asynchroner Kommunikation finden sich in Abschnitt 8.6, eine kurze Einführung in ereignisgetriebene Architekturen in Abschnitt 7.7.

9.1.5 Vor- und Nachteile verschiedener Stufen der Entkopplung

Zunehmende Abstraktion und Entkoppelung ...

Anhand von Abbildung. 9.1 sollen nun nochmals zusammenfassend die Vor- und Nachteile verschiedener Stufen der Entkopplung zwischen Ob-

jekten bzw. Komponenten und Services sowie eine zunehmende Abstraktion in der Beschreibung diskutiert werden: Es findet eine schrittweise Entkopplung und Abstraktion statt; beginnend von der direkten Objekt-Objekt-Kommunikation (über Instanzbildung und direkten Aufruf) hin zur Kommunikation über Service-Frameworks (evtl. nachrichtenorientierte Middleware oder Indirektion über Service Registries) sowie der Verwendung plattformunabhängiger Standards.

... und ihr Preis

Es soll an dieser Stelle nochmals betont werden, dass ein höherer Grad an Abstraktion nicht immer nur Vorteile bringt, sondern auch potenzielle Nachteile hat, die der Architekt den Vorteilen in der jeweiligen Anwendung gegenüberstellen muss. In der Berichterstattung werden neue Technologien und Konzepte gerne in den buntesten Farben beschrieben, und die alten Vorgehensweisen sehen dagegen oft etwas verstaubt aus. Man kann dann bei oberflächlicher Betrachtung den Eindruck bekommen, dass man in *modernen* Anwendungen einfach eine bestimmte Technologie verwendet (z. B. Webservices), weil sie besser sei als die alten Technologien. Dies kann natürlich zutreffen, dennoch sollten aber neue Ansätze immer der notwendigen kritischen Betrachtung unterzogen und vor allem auch immer die Frage gestellt werden, *für welche Arten von Problemen* ein bestimmtes Konzept am besten geeignet ist, und welche neuen Probleme ein neuer Ansatz aufwerfen kann.

Wiederverwendbarkeit bzw. Austauschbarkeit

Bei der Frage der Entkopplung wird auf der linken Seite in Abbildung 9.1 angedeutet, dass die Kopplung zwischen einzelnen Komponenten nach unten hin immer kleiner wird, gleichzeitig sollte aber auch das Potenzial der Wiederverwendbarkeit bzw. der Austauschbarkeit steigen, denn wenn zwei Komponenten oder Objekte eigentlich *immer* miteinander arbeiten und auch sehr stark voneinander abhängig sind, gibt es keinen Grund hier eine Entkopplung vorzunehmen. Das würde nur die Komplexität erhöhen sowie einen unnötigen Aufwand verursachen.

Soll aber das Potenzial einzelner Komponenten zur Wiederverwendung erhöht werden bzw. die Möglichkeit geschaffen werden, einzelne Komponenten leicht durch andere auszutauschen (dies muss nicht unbedingt den operativen Betrieb betreffen, sondern kann auch „nur“ das Testen erleichtern), so kann es Sinn machen, Schritte der Entkopplung zu gehen. An dieser Stelle muss man sich dann die Frage stellen, wie weit die Entkopplung gehen sollte: Die Verwendung von Interfaces ist sicherlich sehr häufig eine gute Idee. Gleichzeitig wird man aber feststellen, dass Wiederverwendung auch ein höheres Maß an Abstraktion voraussetzt und meist damit verbunden ist, dass mehrere Objekte zu funktionalen Einheiten (Komponenten) verbunden werden. Hier kann auch der Einsatz von leichtgewichtigen Komponenten-Frameworks die Bindung von Komponenten flexibler machen sowie deren Konfiguration (z. B. über Dependency Injection, siehe Abschnitt 9.3) erleichtern.

Overhead und Komplexität

Die rechte Seite von Abbildung 9.1 illustriert, dass der *Aufwand*, bzw. die *Komplexität* des Systems mit zunehmender Entkopplung zunächst einmal

steigt, denn es werden zusätzlich neue Konzepte, neue Frameworks und Bibliotheken verwendet. Das bedeutet, dass sich die Entwickler mit diesen neuen Konzepten vertraut machen müssen (z. B. mit den neuen Frameworks); auch neue Abhängigkeiten entstehen! Die Idee ist aber, dass diese *initial* höhere Komplexität in einem Projekt durch die Vorteile während der Projektabwicklung wieder kompensiert wird: Hat man z. B. das Komponenten-Framework im Griff, so stellt man meist fest, dass sich auch immer größer werdende Anwendungen leichter warten und testen lassen bzw. dass man sie flexibler gestalten kann. Dennoch sollte man sich schon zu Beginn der „Kosten" bewusst sein und den potenziellen Nutzen den potenziellen Problemen gegenüberstellen.

Kommunikation über Systemgrenzen

Dies trifft ganz besonders im letzten Schritt – der in Abbildung 9.1 gezeigt wird – zu: Gerade in den letzten Jahren erleben wir den starken Trend zur Kommunikation über Systemgrenzen hinweg, z. B. zwischen Firmen oder in Integrationsprojekten. Für derartige Einsatzbereiche stellen Technologien wie Webservices, nachrichtenorientierte Middleware etc. sehr leistungsfähige Konzepte zur Verfügung. Gleichzeitig handelt man sich auf dieser Ebene ein erhebliches Maß an Komplexität in den Frameworks und der Infrastruktur ein. Auch das Beherrschen der vielen verschiedenen Standards und Werkzeuge ist alles andere als trivial und angemessen zu berücksichtigen. Man kann sich auch die Frage stellen, ob es immer vergleichsweise schwergewichtige Ansätze wie Webservices auf der Basis von WSDL/-SOAP sein müssen oder ob nicht in einigen Fällen auch leichtgewichtigere Konzepte z. B. auf REST-Basis (siehe Abschnitt 8.6.1) sowie mit Protokollen wie JSON [54] oder Atom [2] angemessener sind. Hier empfiehlt sich meist eher ein pragmatischer als ein dogmatischer Zugang.

9.2 Software-Frameworks als Basis für die Komponentenbildung

Framework als Grundbaustein

Software-Frameworks haben sich besonders im Java-Umfeld als Grundbaustein für viele Anwendungen etabliert. Vieler Software-Projekte greifen dabei auf das breite Angebot an kommerziellen oder Open-Source-Frameworks zurück. Es gibt etwa kaum noch Projekte, bei denen beispielsweise die Datenzugriffsschicht (Persistenz) vollständig neu entwickelt wird; hier werden etablierte Open-Source-Lösungen, wie z. B. Hibernate[5], eingesetzt (siehe auch Abschnitt 9.4).

Versteckte Komplexität

Aufgrund der Komplexität und der hohen Einarbeitungszeit in Software-Frameworks scheuen viele Entwickler, sich mit einer neuen Technologie auseinanderzusetzen. Beim Einsatz von Frameworks ist es sehr wichtig,

[5] `www.hibernate.org`

Definitionen

die Grundkonzepte der Frameworks zu verstehen, da es hier einige wiederkehrende Prinzipien gibt. Werden einmal die Grundkonzepte eines Frameworks verstanden, so ist die Einarbeitung in neue Technologien vergleichsweise einfach. Bevor nun auf die Details und die Architektur von Software-Frameworks eingegangen wird, sollen zwei Definitionen des Framework-Begriffs gegeben werden:

> „A framework is a reusable design that requires software components." *Strooper [87]*

Wiederverwendbarkeit

Nach Strooper et. al [87] besteht ein Software-Framework selbst aus einer Menge von Komponenten und fördert auch die Bildung von wiederverwendbaren Komponenten. Die Wiederverwendbarkeit ist dabei eine wesentliche Eigenschaft, da Frameworks in unterschiedlichsten Kontexten von Software-Systemen zum Einsatz kommen können: z. B. kann ein Persistenz-Framework sowohl in verteilten Web-Anwendung, wie einem Webshop, in einer Desktop-Anwendung, wie einem Grafikprogramm, oder einem Spiel eingesetzt werden.

Die zweite Definition ist konkreter und beschreibt ein Framework wie folgt:

> „A framework is a set of classes that embodies an abstract design for solutions to a family of related problems. An object-oriented application framework is a promising technology for reifying proven software designs and implementations in order to reduce the costs and improve the quality of software." *Jonson und Foote [53]*

Vorgabe der Basisfunktion

Ein Software-Framework ist demnach eine wiederverwendbare Komponente und besteht aus einer Menge von Klassen, die bereits eine Basisfunktionalität vorgeben, die nun an die eigenen Bedürfnisse angepasst werden muss. Jonson und Foote [53] gehen auch auf die *Wirtschaftlichkeit* von Software-Frameworks ein. Bei richtigem Einsatz von Frameworks können die Entwicklungszeit und folglich auch die Kosten deutlich reduziert werden, da sich die Software-Architekten und Software-Entwickler besser auf den Kern ihrer Arbeit, nämlich *die Umsetzung der Problemdomäne*, konzentrieren.

Hot Spots und Frozen Spots

Ein Framework verfügt über sogenannte *Hot Spots* und *Frozen Spots*. Hot Spots sind jene Teile, die mit applikationsspezifischem Code befüllt werden oder angepasst werden müssen. Dies können beispielsweise abstrakte Klassen, Schnittstellen oder andere Komponenten sein. Gesteuert werden diese Hot Spots jedoch wieder vom Framework selbst. Man spricht dabei auch vom *Hollywood-Prinzip*: *„Don't call us, we call you"*.

„Don't call us, we call you"

Dem Framework wird nur die Logik zur Verfügung gestellt, und die Steuerung der einzelnen Komponenten übernimmt das Framework. Damit dieses Prinzip funktioniert, arbeitet der *Kernel* des Frameworks ausschließlich

mit den Schnittstellen und nicht mit den Implementierungen (diese unterscheiden sich ja in jeder Applikation). Frozen Spots hingegen sind feste Bestandteile des Frameworks und können nicht verändert werden. Der folgende Code implementiert beispielsweise ein Interface aus einem Framework, und in der Implementierung ist das spezielle Verhalten für die Applikation enthalten:

```
class MeineApplikationImplementierung
implements FrameworkInterface {

  public frameworkMethode1() {
    // Implementierung der Methode 1
    ...
  }
  public frameworkMethode2 (String parameter) {
    // Implementierung der Methode 2
    ...
  }
}
```

Eine Applikation kommt oft mit einem kleinen Teil der Komponenten aus, die das Framework anbietet. Um die Komplexität für den Software-Entwickler zu reduzieren sind viele Frameworks (wie beispielsweise Spring[6]) modular aufgebaut. Man verwendet dann nur diejenigen Bibliotheken (Komponenten) des Frameworks die man im konkreten Anwendungsfall auch tatsächlich benötigt und vermeidet damit überflüssigen Ballast. Die Abbildung 9.6 verdeutlicht das Zusammenspiel zwischen dem Framework und einer Applikation.

Framework Bestandteile

Der Kern (*Framework-Core*) setzt sich aus einer Menge von abstrakten Klassen und Schnittstellen zusammen. Diese Klassen geben die Basis und auch die Architektur für das zu implementierende System vor. Die *Framework-Library* ist eine Ansammlung von konkreten Komponenten für das Framework, wie beispielsweise Hilfsklassen. Diese Hilfsklassen werden auch oft vom Framework selbst verwendet. Der Entwickler kann ebenfalls auf dieses Angebot zurückgreifen. Die *Application Extensions* sind die implementierten Hot Spots des Frameworks, wie beispielsweise die Implementierung von abstrakten Klassen oder Schnittstellen.

Patterns und Frameworks

Wie in Kapitel 8 im Detail beschrieben, stößt man bei der Entwicklung von Software-Systemen man auf immer wiederkehrende, ähnliche Probleme. Frameworks entstehen durch *Best-Practice Ansätze* und haben viele Gemeinsamkeiten mit Patterns. Es bestehen aber auch konkrete Unterschiede zwischen Software-Frameworks und Patterns. Frameworks sind deutlich konkreter, da sie bereits in einer *bestimmten Sprache* implementiert sind. Patterns hingegen sind neutral gehalten und können in verschiedenen Spra-

[6] www.springframework.org

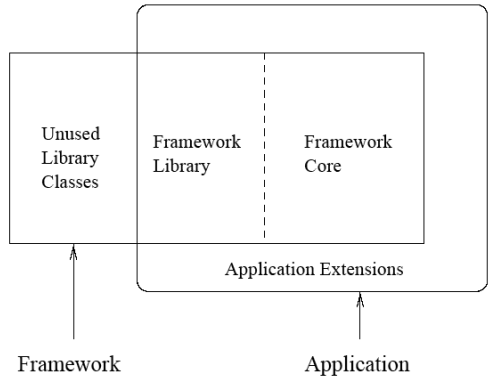


Abbildung 9.6
Framework-Komponenten [66]

chen implementiert werden. Durch ihre Komplexität geben Frameworks bereits eine Architektur vor. Dabei werden Pattern oft für diese Architektur verwendet. Pattern können somit als *Teilmenge* des Frameworks gesehen werden. In vielen Fällen sind Frameworks sind anwendungsspezifisch und decken einen bestimmten Aspekt in der Entwicklung ab, wie etwa die Persistenz.

Nachteile

Zusammenfassend soll noch auf die Vor- und Nachteile beim Einsatz von Frameworks eingegangen werden. Betrachten wir zuerst die Nachteile die bei Verwendung von Software-Frameworks auftreten können:

- *Komplexität*: Frameworks kann man sich wie einen Werkzeugkasten vorstellen, der eine bestimmte Komplexität mit sich bringt. Dieser Werkzeugkasten muss gut verstanden werden, bevor er effizient eingesetzt werden kann.
- Es ergeben sich *Abhängigkeiten* der Applikation vom Framework direkt, aber auch von den vorgegebenen Architekturen.
- Manche Frameworks (gerade im Open Source Umfeld) leiden an mangelhafter Dokumentation, die den Einstieg in das Framework erschweren.

Vorteile

Die Komplexität und die Abhängigkeit zum Framework können einerseits als Nachteil gesehen werden. Andererseits kann durch das umfangreiche Funktionsangebot eines Frameworks auch viel Zeit und Aufwand gespart werden:

- *Modularität:* Da ein Framework einen gewissen Grad an Architektur vorgibt, sind die Entwickler gezwungen ihre Komponenten modular aufzubauen.
- *Wiederverwendbarkeit:* Eine Technologie kann für eine Menge von Software-Systemen eingesetzt werden.

- *Erweiterbarkeit:* Frameworks sind so konzipiert, dass sie erweiterbar sind und somit für eine längerfristig stabile Architektur sorgen.
- *Inversion of Control:* Der Kontrollfluss bleibt beim Framework, der Anwendungsentwickler kann sich somit auf das Wesentliche konzentrieren: Die Implementierung der Geschäftslogik.
- *Standardisierung:* Bestimmte Frameworks sind zum Standard in Anwendungsdomänen geworden. Damit werden Standardprobleme auch immer auf dieselbe Weise gelöst, wodurch z. B. die Einarbeitungszeit neuer Entwickler verkürzt werden kann.

9.3 Dependency-Injection

Kontrollfluss

Im vorigen Abschnitt wurden grundsätzliche Konzepte von Software-Frameworks eingeführt. Ein Aspekt dabei war der *Kontrollfluss*: Frameworks bedienen sich dabei häufig des *Inversion of Control* oder *Dependency-Injection* Patterns. Diese beiden Begriffe werden häufig synonym verwendet. Die Grundproblematik besteht darin, dass man Komponenten möglichst wiederverwendbar und austauschbar gestalten möchte. Greift eine Komponente direkt auf eine andere zu, so sind diese beiden Komponenten eng miteinander verknüpft und können nicht mehr leicht ausgetauscht werden. Diese Problematik wurde bereits am Anfang dieses Kapitels, besonders in Abschnitt 9.1.3 genauer erklärt. Komponentenframeworks versuchen daher die Verbindung zwischen den Komponenten zu übernehmen und damit die Komponenten voneinander zu entkoppeln. Dies kann an einem Beispiel illustriert werden:

Beispiel

Bis vor wenigen Jahren waren Vorwahlen von Mobiltelefonen an bestimmte Anbieter gebunden. Dies hat für viele Kunden den Wechsel zu einem anderen Anbieter erschwert, weil damit immer auch ein Wechsel der Telefonnummer (zumindest der Vorwahl) verbunden war. Betrachtet man Personen die über ein Mobiltelefon verfügen als Komponenten, so kann man leicht erkennen, dass diese Komponenten im alten System sehr starr miteinander verbunden waren. Jede notierte oder gespeicherter Telefonnummer wurde obsolet, wenn der entsprechende Kontakt den Telefonie-Anbieter und damit die Telefonnummer gewechselt hat. Die Verbindung zwischen den Komponenten musste neu „verdrahtet“ werden.

Seit einigen Jahren gibt es aber die Möglichkeit Rufnummern von einem Anbieter zu einem anderen Anbieter mitzunehmen. Damit verliert die Telefonnummer de facto die Bindung an einen bestimmten Anbieter. Es wurde eine *Abstraktionsschicht* eingezogen, die man sich im Kontext des Software-Engineering als Komponentenframework vorstellen kann: Wechselt nun eine Person den Anbieter und nimmt die alte Nummer zum neuen Anbieter mit, so muss darüber niemand darüber informiert werden. Die Telefonnummer ist nicht mehr der *direkte Draht* zum gewünschten Kon-

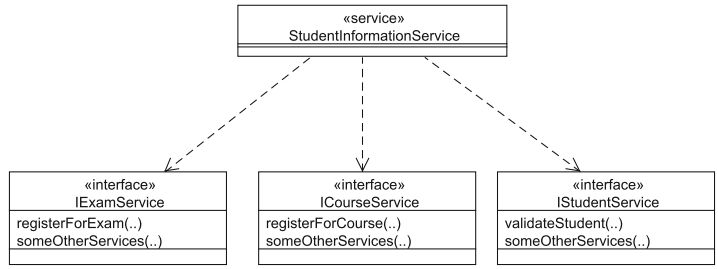


Abbildung 9.7 *Beispiel für eine Dependency (Abhängigkeit).*

takt, sondern wird erst über ein Zwischensystem aufgelöst und der Anrufer bekommt vom Zwischensystem die entsprechende Verbindung hergestellt. Die eigentliche „Verdrahtung" findet beim Telefonie-Anbieter statt, der das System entsprechend konfiguriert dass die Nummern richtig geroutet werden. Dies kann man sich als einfaches Beispiel von Dependency-Injection vorstellen.

DI im Software-Kontext

Wie verhält sich Dependency-Injection nun im Kontext von Software-Komponenten? Software besteht aus einer Menge unterschiedlicher Komponenten. Die Architektur beschreibt wie diese Komponenten zueinander in Verbindung stehen, deren Abhängigkeiten und wie Komponenten zur Laufzeit miteinander kommunizieren. Bei der Ausführung müssen die Komponenten wissen, welche anderen Komponenten benötigt werden, wo diese aufzufinden sind und wie mit diesen zu kommunizieren ist (welche, wo, wie). In Abbildung 9.7 benötigt z. B. das `StudentInformationService` die Komponenten `IExamService`, `IExamCourseService` und `IStudentService`. An dieser Stelle sei nochmals erwähnt, dass gerade bei Depedency-Injection das *Interface Pattern* (siehe Abschnitt 8.2.1) fundamental ist. Das `StudentInformationService` kommuniziert nur über die *öffentliche Schnittstelle* der abhängigen Komponente (im obigen Beispiel die Telefonnummer). Die konkrete Implementation (im Beispiel der tatsächliche Anschluss, zu dem verbunden werden soll) wird später bestimmt.

Konventionelle Ansätze

Für die Strukturierung und Auflösung von Abhängigkeiten zwischen Komponenten können grundsätzlich zwei Wege gewählt werden. Der erste Weg ist, dass sich eine Komponente selbst um Auflösung, Initialisierung und Kommunikation zu deren Abhängigkeiten kümmert. Dabei kann die Komponente ein Objekt direkt instanziieren oder das Objekt wird über eine Factory (siehe Abschnitt 8.3.2) erstellt. Beim zweiten Weg definiert der Client lediglich seine Abhängigkeiten und ein Framework regelt die Auflösung, die Initialisierung und Kommunikation. Eine Änderung der Kommunikation hat beim ersten Ansatz in der Regel wesentlich mehr Auswirkung in anderen Codeteilen als beim zweiten Ansatz. Nicht selten endet dies bei schlechter Organisation in einem endlosen Refactoring.

Eine Factory verbirgt zwar die Komplexität der Instanziierung, die Lokalisierung des Services ist jedoch weiterhin an den Client gebunden. Auch

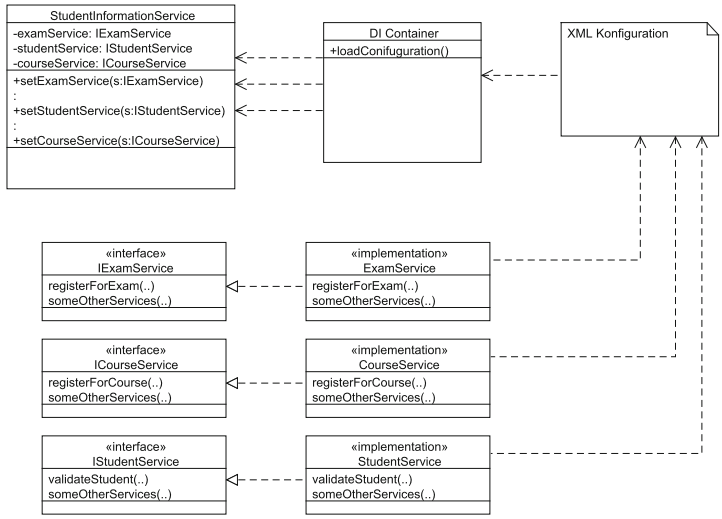


Abbildung 9.8
Dependency-Injection Container injiziert die konkrete Implementierung der Komponente.

lassen sich die Objekte der Factory nicht leicht durch Mock-Objekte für das Testen ersetzen. Hierfür müsste die Factory entsprechend erweitert werden, damit sie sich im Testbetrieb anders verhält als im Produktivbetrieb[7]. Zudem müssen Factories für verschiedene Anwendungsfälle immer neu entwickelt werden.

Dependency-Injection Ansatz

Der Dependency-Injection Ansatz geht den Weg, dass der Client seine Abhängigkeiten definiert und einem Framework die Arbeit der Lokalisierung, Instanziierung, der Konfiguration der Objekte und der Kommunikation überlässt. Der Client kennt nur die öffentliche Schnittstelle und nicht die konkrete Implementierung. Die konkreten Implementierungen werden an einer zentralen Stelle definiert und durch das Framework injiziert (Inversion of Control). Abhängig vom eingesetzten Framework wird die Konfiguration der Komponenten in Java Code, Annotations oder über XML beschrieben. Die Konfigurationen werden zur Laufzeit vom Komponenten-Framework geladen und das Framework injiziert die Abhängigkeiten in die Komponenten. Die Komponenten sind somit von der Implementierung der Abhängigkeiten vollständig losgelöst.

DI-Frameworks

Da Dependency-Injection ein Muster ist, kann dieses Muster auf unterschiedliche Art und Weise implementiert werden. Es ist möglich, ein eigenes DI-Framework zu implementieren. Das macht in der Praxis meist

[7] Directory Services wie das Java Naming and Directory Interface (JNDI) ist auch ein häufig aufzufindender Ansatz, um Abhängigkeiten zwischen Komponenten aufzulösen. JNDI ist ebenfalls eine Factory, die jedoch in einer Infrastruktur (z. B. Applikationsserver) konfiguriert werden muss. Dies ist auch gleichzeitig der Nachteil von JNDI. Der Umstieg auf eine leichtgewichtige Umgebung ohne Servereinsatz ist oft nicht möglich.

Tabelle 9.1 *Dependency-Injection Frameworks im Java-Umfeld.*

Framework	Beschreibung
Spring-Framework (`www.springframework.org`)	Die Konfiguration der Abhängigkeiten wird wahlweise in XML oder über Annotations definiert. Die Komponenten werden mit einer ID versehen, die zur Laufzeit für die Injection verwendet werden.
Pico-Container (`http://www.picocontainer.org/`)	Der Pico Container ist ein sehr leichtgewichtiger DI Container, der über Javacode konfiguriert wird. Es wird nur die Constructor Injection unterstützt. Die Abhängigkeiten einer Komponente werden zur Laufzeit über den Konstruktur aufgelöst.
Hivemind (`http://hivemind.apache.org/`)	Ein weiterer Vertreter eines DI Frameworks, das die Abhängigkeiten über Setter-Methoden auflöst.
Google Guice (`http://code.google.com/p/google-guice/`)	Ist ein DI Framework von Google, das von den Java Annotations für die Auflösung von Abhängigkeiten Gebrauch macht.

wenig Sinn da bereits sehr viele Lösungen am Markt vorhanden sind. Tabelle 9.1 listet einige Vertreter von DI-Frameworks auf und beschreibt, wie diese das Dependency-Injection Muster umsetzen. Bei Dependency-Injection gibt es zwei grundsätzliche Arten wie eine Abhängigkeit an eine Komponente übergeben werden kann. Die erste Möglichkeit ist die Abhängigkeit über den Konstruktor zu übergeben *(Constructor Injection)*. Diese Variante hat bei komplexen Abhängigkeiten jedoch den Nachteil, dass der Konstruktor mit der Zeit sehr unübersichtlich wird. Bei der *Setter Injection* werden die Abhängigkeiten über die angebotenen Setter-Methoden der Komponente von außen zugewiesen. Diese Methode wird in der Praxis meistens bevorzugt, sofern es das eingesetzte DI Framework unterstützt.

Es wird nun kurz (mit Bezug auf das eingangs erwähnte Beispiel des Studenten-Services in Abbildung 9.7) der Gebrauch eines DI Frameworks anhand des Spring-Frameworks erläutert. Die Implementierung des Services wird als *Plain Old Java Object* (POJO)[8] realisiert:

```
1  public class StudentInformationService
2              implements IStudentInformationService {
3    private IExamService examService;
4    private IStudentService studentService;
5    private ICourseService courseService;
6
7    // Setter methods for injection
8    public void setExamService(IExamService
         injectedService){
9      examService = injectedService;
10   }
```

[8] Als POJO wird ein ganz normales Objekt in der Java Plattform bezeichnet, ohne spezielle Abhängigkeiten von Infrastruktur Schnittstellen, wie z. B. `HttpServlet`

```
11    public void setCourseService(ICourseService
          injectedService){
12      courseService = injectedService;
13    }
14    public void setStudentService(IStudentService
          injectedService){
15      studentService = injectedService;
16    }
17
18    public StudentInformationService() {
19    }
20
21    public void registerForExam(long examId,
22                  String matrNr,
23                  String comment) {
24      ..
25      //do some pre work
26      ...
27      //call service for exam registration
28      examService.registerForExam(...);
29      //do some post work
30      ...
31    }
32  }
```

Konfiguration in Spring

Das Spring-Framework bietet verschiedene Möglichkeiten an die Komponenten und deren Abhängigkeiten zu konfigurieren: XML, Autowiring oder Annotationen.

Eine recht komfortable Art und Weise den Container zu konfigurieren[9] ist die XML-Variante, bei der eine oder mehrere XML-Dateien verwendet werden können. Im vorgestellten Beispiel wird die XML-Variante verwendet. Spring bietet zudem das *Autowiring*-Prinzip an. Dabei entscheidet der DI-Container anhand von Namenskonventionen oder Typ-Kompatibilität, welche Objekte zueinander in Beziehung zu setzen sind. Dies erscheint für einfache Fälle eine elegante Möglichkeit zu sein. Für komplexe Software-Systeme ist dieses Vorgehen nicht zu empfehlen, da die Abhängigkeiten nicht mehr explizit bestimmt werden können. Die Übersicht kann dann leicht verloren gehen. Auch das Testen wird durch Autowiring erschwert.

Möchte man nicht mit Konfigurationsdateien arbeiten, so kann man auf Java-Annotationen zurückgreifen. Dabei kann der Entwickler die volle Java-IDE-Unterstützung beim Refactoring ausnutzen. Dies hat jedoch den Nachteil, dass die Informationen der Abhängigkeiten wieder im Sourcecode gehalten werden. Änderungen an der Konfiguration haben daher Änderungen im Sourcecode zur Folge. In der Praxis sind sowohl die XML-Va-

[9] Das Eclipse-Plugin bietet eine komfortable Möglichkeit die Konfigurationen zu verwalten.

riante als auch die Variante der Annotationen weitverbreitet. Sie können auch gemischt eingesetzt werden.

Beispiel: XML-Konfiguration

Alle Komponenten und Abhängigkeiten werden in einer XML-Datei konfiguriert. Komponenten werden in der XML-Datei als `<bean>`-Element mit eindeutiger `id` definiert. Diese Id verwendet Spring für die Referenzierung im gesamten Container. Das `class`-Attribut definiert die Klasse der Implementierung. Die Klasse `ExamService` ist eine Implementierung des Interface `IExamService`[10]. Neben der `id`- und `class`-Attribute können noch weitere Eigenschaften festgelegt werden, die jedoch für die DI nicht relevant sind. Das `StudentInformationService` wird ebenfalls als eigenständige Bean definiert und erhält zusätzlich noch `property`-Elemente. Diese Daten verwendet Spring für die Dependency Injection: Die Property-Elememte repräsentieren Setter-Methoden in der `StudentInformationService`-Klasse, wie im vorigen Listing ersichtlich ist. Dabei werden die Namenskonventionen von *Java Beans* verwendet, d. h., der erste Buchstabe der Setter-Methode in der XML-Konfiguration beginnt mit einem Kleinbuchstaben.

Auflösen der Abhängigkeiten

Fordert man im Java-Sourcecode (wie weiter unten beschrieben wird) eine Instanz mit der Bean-ID `studentInformationService` an, so instanziiert Spring die Klasse `StudentInformationService`. Auch erkennt Spring anhand der `property`-Elemente und dem `ref`-Attribut, dass dieses Objekt von anderen Objekten abhängig (*dependent*) ist. Daher werden auch Instanzen der abhängigen Klassen (`ExamService`, `CourseService`, `StudentService`) erstellt. Diese Instanzen werden in die Instanz der `studentInformationService` durch Aufruf der entsprechenden Setter-Methoden *injiziert*.

```
1  <beans ...
2    <bean id="examService"
3          class="ExamService"/>
4    <bean id="courseService"
5          class="CourseService"/>
6    <bean id="studentService"
7          class="StudentService"/>
8    <bean id="studentInformationService"
9          class="StudentInformationService">
10     <property name="examService"
11               ref="examService"/>
12     <property name="courseService"
13               ref="courseService"/>
14     <property name="studentService"
15               ref="studentService"/>
16   </bean>
17 </beans>
```

[10] Aus Gründen der Lesbarkeit wird hier auf die Angabe von Packages verzichtet.

Konfiguration von Komponenten

Auch statische Konfigurationen können an dieser Stelle vorgenommen werden. Wird anstelle des `ref`-Attributes ein `value`-Attribut verwendet, so wird der angegebene Wert mittels Setter-Methode gesetzt:

```
<property name="encoding" value="utf-8" />
```

Verwendung des Containers

Nachdem die Komponenten und deren Abhängigkeiten in der XML-Datei konfiguriert wurden, kann der Container initialisiert werden. Dieser Container wird mithilfe eines `ApplicationContext` erstellt, der durch die XML-Datei beschrieben wird. Beim Starten werden alle Beans in der XML-Datei geladen und die Abhängigkeiten aufgelöst. Mit der `getBean()`-Funktion können die Beans über die definierte Bean-ID vom Container geladen werden. Spring verwendet dabei das *Lazy-Loading*-Prinzip. Von einer Bean wird erst dann eine Instanz erstellt, wenn diese das erste Mal benötigt wird[11]. Wird nun das `StudentInformationService` aus dem Container geladen, so erhält man (wie oben beschrieben) eine bereits voll funktionsfähige und konfigurierte Komponente, die auch schon die konkreten Service-Implementierungen der abhängigen Komponenten beinhaltet:

```
ApplicationContext ctx =
  new ClassPathXmlApplicationContext("config.xml");
//Receive service from container
IStudentInformationService service =
  (IStudentInformationService) ctx.
  getBean("studentInformationService");
//do some work with the service
```

Struktur im Code

Dieses Beispiel verdeutlicht, dass durch den Einsatz von DI-Frameworks der Code besser strukturiert werden kann. Die Komplexität der Lokalisierung, Instanziierung und der Kommunikation wird ausgelagert, und es können auch weitere Features (wie z. B. entfernte Service-Aufrufe, AOP etc.) des eingesetzten Frameworks genutzt werden. Das Testen von Komponenten wird durch den DI-Ansatz ebenfalls einfacher, da der Austausch von Objekten ohne großen Aufwand und vor allem ohne Eingriff in den Sourcecode möglich ist.

Vorteile

Zusammenfassend können folgende Vorteile für den Einsatz von DI angeführt werden:

- Durch den Einsatz von DI-Frameworks werden die Komponenten gut strukturiert, und die Abhängigkeiten zwischen den Komponenten werden auf die Schnittstelle reduziert.

[11] Üblicherweise werden alle Beans im Container als Singleton (siehe Abschnitt 8.3.1) behandelt, d. h., es gibt immer nur eine Instanz der Klasse im Container. Soll jedoch immer eine neue Instanz erstellt werden, wenn eine Bean aus dem Container geladen wird, dann kann das `singleton`-Attribut auf `false` gesetzt werden.

- Die Testbarkeit der einzelnen Komponenten wird erleichtert, und auch Integrationstests sind einfacher durchzuführen. Der Einsatz von Mock-Objekten stellt kein Problem mehr dar.
- Der Möglichkeit der Wiederverwendbarkeit steigt.

9.4 Persistente Datenhaltung in komponentenbasierten Systemen

Persistenz = dauerhaftes Speichern

Für die meisten Anwendungen ist es nicht ausreichend, Daten nur im flüchtigen Speicher (RAM) zu halten, sondern sie benötigen einen Mechanismus der es erlaubt den internen Zustand der Software oder die Daten die Benutzer dauerhaft zu sichern, also zu persistieren. Nun gibt es viele verschiedene Persistenzmechanismen und -strategien, die je nach Problemfall geeignet zu wählen sind. Dieser Abschnitt behandelt nicht alle möglichen Strategien im Detail, sondern fokussiert auf architektonische Aspekte, die bei der Auswahl und Implementierung der verschiedenen Ansätze zu beachten sind.

Moderne Persistenz-Strategien sind häufig komponentenorientiert. Dies bedeutet, dass man versucht Persistenz und Geschäftslogik sauber in getrennten Komponenten zu organisieren. Auch sollen die Persistenz-Komponenten wiederverwendbar und austauschbar sein. In diesem Abschnitt wird ein Überblick über wichtige Aspekte der Modellierung und Planung einer geeigneten Persistenzschicht gegeben. Es werden auch Konzepte, die in anderen Abschnitten schon beschrieben wurden (z. B. Data Access Object Pattern, Schichtenarchitektur, Services), nochmals aufgegriffen, um ein konsistentes Bild der Persistenz-Problematik zu zeichnen.

9.4.1 Anforderungen an eine Persistenzschicht

Rahmenbedingungen

Bevor man sich mit einer bestimmten Strategie näher auseinandersetzt, sollte man sich überlegen, welche Rahmenbedingungen zu beachten sind, die die Entscheidung für eine bestimmte Vorgehensweise beeinflussen. Hier sind unter anderem folgende Aspekte zu betrachten:

- Wird das Projekt „auf der grünen Wiese“ entwickelt, oder ist es in bestehende Systeme wie Legacy Datenbanken, Services usw. einzubetten?
- Wo soll das Projekt installiert werden?
- Wie sind die Daten strukturiert?
- Ist eine strenge Trennung von Daten und Logik gewünscht?

- Wie ist die Zugriffsstrategie?
- Wie treten die Daten auf (z. B. als Multimedia-Stream, nahe am Programmiermodell, unabhängig vom Programmiermodell)?
- Liegt ein Client/Server System vor, eine Web-Anwendung oder sind die Daten lokal beim Benutzer zu speichern?
- Wie ist der erwartete Umfang der Daten, sowie die Anforderungen an Performance und Skalierung?
- Wie herausfordernd ist die notwendige Zuverlässigkeit und Verfügbarkeit der Persistenz-Lösung?
- Wie groß ist der Aufwand sowie die Komplexität der unterschiedlichen Persistenzlösungen und welche ist der Problemstellung adäquat?
- Wer wartet das operative System?
- Sollen „konventionelle“ Persistenz-Strategien eingesetzt werden oder sind Cloud Services eine Option?

Integration in bestehende Systeme

Eine wesentliche Überlegung ist zunächst, ob das Projekt von Grund auf neu entwickelt wird und keine wesentlichen Abhängigkeiten von anderen Systemen hat, oder ob es z. B. innerhalb einer Firma entwickelt wird und mit bestehenden Daten arbeiten muss. Im letzteren Fall sind die Freiheitsgrade natürlich stark eingeschränkt. Soll beispielsweise auf eine große relationale Datenbank und mit bestehender Infrastruktur gearbeitet werden, so muss man sich an die Gegebenheiten anpassen. Im kommerziellen Umfeld gibt es oft auch schon Richtlinien, welche Persistenz-Strategien in welchem Fall einzusetzen ist bzw. wer überhaupt Datenbankabfragen erstellen darf. Dennoch, auch bei der Interaktion mit bestehenden relationalen Datenbanken gibt es verschiedene Möglichkeiten, die etwas detaillierter in Abschnitt 9.4.8 besprochen werden. Selbst wenn das Projekt nicht in eine bestehende Infrastruktur eingebettet werden muss, so sollte man sich die Frage stellen wo die Software installiert werden soll. Die eleganteste Datenbanklösung wird zum Problem, wenn man keinen Provider findet, wo diese installiert ist bzw. wenn potenzielle Kunden von einem System, das sie nicht kennen und nicht administrieren, abgeschreckt werden.

Daten, Logik und Zugriffsstrategie

Eine weitere Frage, die man klären muss ist, in welcher Form die Daten überhaupt auftreten: Handelt es sich um stark strukturierte Daten (wie sie z. B. im relationalen Modell gut abgebildet werden können), um hierarchische, Dokument-artige semi-strukturierte Daten oder um (Multimedia-) Streams? Damit in Zusammenhang steht zudem die Frage, ob die Persistenzlösung sehr eng mit der Programmierlogik (z. B. Objektorientierung) verknüpft sein darf, oder ob es z. B. aus Gründen der Wiederverwendung der Daten gefordert ist, dass die Daten unabhängig vom Programmiermodell zu modellieren und zu speichern sind z. B. in einem relationalen Modell. Letzteres wird häufig auch dann gefordert, wenn auf die Daten von anderen Anwendungen z. B. Reporting Werkzeuge, Data Warehousing usw.

zugegriffen werden soll. Dies wird im Detail ab Abschnitt 9.4.3 diskutiert. In manchen Fällen sind auch bestimmte Zugriffsstrategien (z. B. hauptsächlich einfaches lineares lesen; Zugriff hauptsächlich über Primärschlüssel, Volltext-Indizes oder komplexe Abfragen) oder Abfragesprachen (z. B. SQL oder XPath) gefordert oder gewünscht.

Wartung

Bei der Planung der Persistenzlösung sind natürlich nicht nur Aspekte der Programmierung zu berücksichtigen sondern auch zu klären, wer für die Wartung des operativen Systems zuständig ist. Im einfachsten Fall gibt es keine besonderen Wartungsmaßnahmen, bei Client/Server-Lösungen muss aber bedacht werden, dass der Kunde, der das System einsetzt auch die verwendeten Datenbanksysteme und Frameworks im Griff hat. Dies betrifft rein operative Wartungsarbeiten (Software-Updates, Backups, Tuning usw.) aber auch Wartung der Software an sich. Nischenlösungen können hier unter Umständen ein gewisses Akzeptanzproblem verursachen.

Client/Server?

Weiterhin macht es einen Unterschied, ob die Anwendung eine Client/Server-Anwendung ist, wo viele verschiedene Benutzer gleichzeitig mit der Datenbank interagieren wollen, oder ob das Datenmodell nur Daten eines Benutzers lokal persistieren soll.

Verfügbarkeit, Performance, Skalierung

Wird eine Client/Server- oder Web-Anwendung entwickelt, so ist zudem zu klären, welche Zuverlässigkeit, Performance und Skalierung vom System erwartet wird. Es macht einen Unterschied ob es z. B. in Ordnung ist, wenn das System fallweise für Wartungsarbeiten offline sein darf oder ob wirklich eine Verfügbarkeit nahe 100% erwünscht ist. Dies betrifft auch Fragen des Backups (kann dies im laufenden Betrieb durchgeführt werden), sowie welche Failover Szenarien zu bedenken sind. D. h. wie lange darf es z. B. im Fehlerfall dauern, um ein System wiederherzustellen, muss parallel ein synchronisiertes Backupsystem mitlaufen usw.

Ähnliche Überlegungen sind für die Parameter Performance uns Skalierung anzustellen: mit wie vielen (gleichzeitigen) Benutzern rechnet man, wieviele Transaktionen pro Zeiteinheit muss das System beantworten können, auf welches Wachstum soll man sich vorbereiten und nicht zuletzt: von welchem Datenvolumen ist auszugehen? Von der Beantwortung dieser Fragen hängt sehr viel in der Entscheidung einer geeigneten Persistenzstrategie ab.

Aufwand und Komplexität der Frameworks

Auch sollte man keinesfalls den Aufwand und die Komplexität verschiedener Persistenzstrategien und Frameworks als Kriterium außer acht lassen, sowie damit verbunden die Zeit sich in die Systeme einzuarbeiten. Objektrelationale Mapping Frameworks (O/R Mapper) beispielsweise bieten dem Programmierer sehr viele Möglichkeiten bei der Abbildung von Objekten auf das relationale Modell; dabei tritt aber im Detail ein hohes Maß an Komplexität auf, das man auch verstehen sollte, um das Laufzeitverhalten der Anwendung abschätzen zu können. Andernfalls ist es im Problemfall äußerst schwierig zu verstehen, an welcher Stelle das Problem zu beheben ist. Ähnliches ist bei der Verwendung der verschiedenen XML-Persis-

tenz-Strategien zu bedenken. Hier kann fallweise weniger mehr sein: ein einfacheres Framework, das man gut im Griff hat kann die bessere Wahl sein als ein komplexes Framework (das grundsätzlich viel mehr Möglichkeiten bietet), das man nicht richtig versteht oder Funktionen, deren Effekte man nicht richtig abschätzen kann (siehe auch Abschnitt 9.4.8).

Cloud Services

Zuletzt sollte man auch die Möglichkeit nicht außer Acht lassen gar keine klassische Datenbank zu betreiben, sondern Webservices für die Persistenz zu nutzen. Dies können z. B. sogenannte Storage (Cloud) Services sein. Dabei mietet man sich bei einem Dienstleister die benötigte Datenkapazität und verwendet die vom Service angebotene API. Im Prinzip ist dies eine Abwandlung der oben beschriebenen Client/Server-Strategie. Einige Details hierzu werden in Abschnitt 9.4.11 besprochen.

Fazit

Zusammengefasst kann man sagen, dass es mit Sicherheit nicht eine „beste Lösung“ für die meisten Problemstellungen gibt, sondern dass je nach Kategorie des Projekts und der Rahmenbedingungen sehr viele verschiedene Einflussgrößen zu berücksichtigen sind, die letztlich die Wahl einer geeigneten Strategie bestimmen.

9.4.2 Transaktionen

Ein weiteres Problemfeld, mit dem man in Verbindung mit komponentenorientierten Systemen sehr schnell in Kontakt kommt, sind *Transaktionen*. Eine Transaktion lässt sich als logische Klammer über eine oder mehrere Aktionen sehen. Dabei wird das System von einem konsistenten Zustand in einen neuen konsistenten Zustand überführt. Dies ist besonders dann wichtig, wenn mehrere Benutzer oder Prozesse parallel mit dem System und seinen Komponenten arbeiten. Einerseits kann es vorkommen, dass ein Anwender in mehreren Komponenten oder Datensätzen Änderungen durchführt, die nur als Einheit Sinn machen, andererseits können mehrere Benutzer gleichzeitig lesend und schreibend auf dieselben Daten zugreifen.

Beispiele

Angenommen, man schreibt ein verteiltes Online-Spiel in dem sich Spielfiguren in verschiedenen Räumen aufhalten können: Wechselt eine Figur von Raum A nach Raum B so muss sichergestellt sein dass: (a) die Figur zu Raum B hinzugefügt wird (b) die Figur aus Raum A entfernt wird. Der Zustand ist nur dann konsistent wenn sich die Person in *genau einem* der beiden Räume aufhält. Bricht z. B. nach Schritt (a) die Netzwerkverbindung ab, könnte es passieren, dass die die Figur nun in beiden Räumen befindet. Ein weiterer Aspekt ist, ob andere Spieler die Figur bereits nach Schritt (a) in Raum B sehen oder erst nach vollendeter Transaktion, also nach Schritt (b). Ersteres könnte durchaus problematische Seiteneffekte haben, da sich die Figur dann für einen bestimmten Zeitraum scheinbar in beiden Räumen aufhalten würde.

Auch parallele Änderungen können zu problematischen Zuständen führen: Wenn zum Beispiel zwei Anwender in einem ERP System gleichzeitig den

gleichen Kundendatensatz bearbeiten, welche Änderungen werden dann übernommen?

Das Transaktionsmanagement einer Datenbank (oder eines Komponentenframeworks) muss nun sicherstellen, dass die einzelnen Transaktionen einander nicht in die Quere kommen. In diesem Zusammenhang betrachten wir die ACID Eigenschaften einer Transaktion:

ACID

- Atomicity: Entweder werden alle Schritte der Transaktion ausgeführt oder keiner.
- Consistency: Die Daten werden von einem konsistenten Zustand in einen neuen konsistenten Zustand überführt
- Isolation: Sicherstellen, dass die Transaktionen einander nicht in die Quere kommen.
- Durability: Sicherstellen, dass die vorgenommenen Änderungen der Transaktion durchgeführt werden.

Zum Thema Isolation gibt es mehrere Ansätze, die von den Datenbankherstellern unterschiedlich implementiert werden. Martin Fowler beschreibt in seinem Buch [29] mehrere Ansätze wie der parallele Zugriff auf Datenbanken gelöst werden kann.

Um mit Transaktionen arbeiten zu können, stellt jede Datenbank einschlägige Befehle zur Verfügung. Zu den wichtigsten zählen:

- `Begin Transaction.` Öffnet eine neue Transaktion. Alle Schritte nach dem Öffnen der Transaktion werden in dieser Transaktion durchgeführt.
- `Commit.` Alle Schritte nach Begin Transaction werden dauerhaft in der Datenbank gespeichert. Nach dem Commit ist die Datenbank wieder in einem konsistenten Zustand.
- `Rollback.` Alle Schritte nach Begin Transaction werden verworfen. In der Datenbank wird nichts gespeichert.

Wann eine Transaktion gestartet oder beendet wird, muss die Applikation selbst entscheiden. Wie bereits erwähnt, kann eine Transaktion auch mehrere Aktionen beinhalten. Diese Entscheidung ist Aufgabe der Applikation bzw. des Transaktionsmanagements. In Kapitel 9.5.4 wird beschrieben wie das Thema Transaktionsmanagement sehr elegant mit aspektorientierter Programmierung (AOP) gelöst werden kann.

Persistenzlösungen

Die Wahl einer geeigneten Persistenzlösung kann folglich auch von der Frage abhängig sein, wie wesentlich Transaktionen für die konkrete Anwendung sind. Relationale Datenbanken bieten typischerweise sehr weitreichende Unterstützung für Transaktions-Szenarien an, während andere Persistenzmechanismen wie z. B. XML-Datenbanken oder proprietäre Systeme hier häufig Schwächen zeigen.

9.4.3 Architektur und Modellierung

Modellierung der Persistenz

Neben der Wahl der Persistenzstrategie sind auch Aspekte der Modellierung zu beachten. Heute dominierende Programmiersprachen wie Java sind zumeist objektorientiert. Entsprechend wird die Modellierung der Domänenobjekte mit Modellierungssprachen wie UML Klassendiagrammen durchgeführt (siehe Abschnitt 6.1). Objektorientierte Programmierung verwendet Konzepte wie Vererbung und Delegation (siehe Kapitel 8). Objektorientierte Datenbanken verwenden dieselbe Logik, daher besteht kein konzeptioneller Bruch zwischen der internen Repräsentation der Daten und der Repräsentation in der Datenbank. Auch die Abfragesprache integriert sich meist sehr elegant in die objektorientierte Entwicklung (mehr dazu in Abschnitt 9.4.6).

Die heute dominierende Persistenztechnologie ist die relationale Datenbank, die auf dem relationalen Modell beruht [21]. Dieses Modell deckt sich aber nicht vollständig mit dem objektorientierten Modell. Für die Modellierung von relationalen Datenbanken werden daher auch andere Diagramme, z. B. Entity-Relationship Diagramme verwendet (siehe Abschnitt 6.2). Die Definition der Datenstrukturen erfolgt über eine *Data Definition Language* (DDL). Verwendet man hierarchische Persistenzmechanismen wie XML, so unterscheiden sich diese ebenfalls von den Konzepten, die in objektorientierten Sprachen vorherrschen. Im Fall von XML wird die Definition der Datenstrukturen meist über eine XML-Schemasprache wie die W3C Schemas durchgeführt. Um diese konzeptionell unterschiedlichen „Welten“ zu verbinden, gibt es verschiedene Strategien. Persistenz-Frameworks wie Objekt-Relationen-Mapping oder XML-Mapping-Werkzeuge unterstützen den Entwickler dabei, Objektstrukturen in relationale Datenbanken oder in hierarchische XML-Strukturen abzubilden.

Architektur mit Persistenz

Aus Architektursicht ist es häufig nicht wünschenswert, die Zugriffslogik auf eine bestimmte Persistenztechnologie bis in die Geschäftslogik hineinzuziehen. Es ist unter anderem aus Gründen der Wartbarkeit und Erweiterbarkeit vorteilhaft, die Anwendung in Schichten aufzubauen, wie in Kapitel 7 im Detail erklärt wurde. Die Geschäftslogik sollte dann nur mit Interfaces der Persistenzschicht interagieren. Die Persistenzschicht sollte möglichst keine starke Abhängigkeit zu einer bestimmten Technologie aufweisen. Das *Data Access Object* Pattern, das in Abschnitt 8.4.5 im Detail erklärt wurde, dient genau diesem Zweck. Abbildung 6.19 illustriert eine solche Architektur: Mit der Datenbank oder Persistenztechnologie interagieren nur die Data Access Objects. Diese verwenden Transfer-/Domänenobjekte zum Austausch der Daten der Geschäftslogik mit der Persistenzschicht. Die Geschäftslogik verwendet nur Domänenobjekte sowie die abstrakten Interfaces der DAO-Schicht. Bei komplexeren Anwendungen wird häufig noch eine Service-Schicht eingezogen, die komplexere Anfragen in einfache Service-Aufrufe integriert. Die „Verdrahtung“ erfolgt häufig mit einem Komponenten-Framework, wie in Abschnitt 9.2 erklärt wurde.

„Richtung“ der Modellierung

Selbst wenn man architektonisch versucht, die Schichten sauber zu trennen, so gibt es dennoch wechselseitige Einflüsse, die zu beachten sind. Daher betrifft eine weitere Überlegung die Frage des *Startpunkts* der Modellierung eines neuen Systems. In manchen Fällen muss das neue Software-System mit bestehenden Daten(banken) integriert werden. Diese Datenbanken können häufig (wenn überhaupt) nur minimal verändert werden. In diesen Fällen ist es klar, dass man bei der Planung von den Datenbankmodellen ausgeht und versucht, diese möglichst gut auf Domänenobjekte abzubilden.

In anderen Fällen, in denen noch keine Datenbank vorhanden ist, kann man natürlich auch von der Domänenmodellierung starten. In diesem Fall beschäftigt man sich hauptsächlich mit der Modellierung der Domänenobjekte und versucht diese schließlich zu persistieren, d. h., das Datenmodell wird nachgezogen. Dies wird von den gerade modernen Ansätzen des Objekt-Relationen-Mappings (ORM) über Annotation relativ gut unterstützt.

Auch wenn man alle Freiheitsgrade bei der Datenmodellierung hat, ist es doch empfehlenswert, sich nicht alleine auf die Domänenmodellierung zu beschränken. In den meisten Fällen wird es empfehlenswert sein, beide Seiten zu betrachten und in Iterationen die Effekte des einen Modells auf das andere zu überlegen. Wenn der „neutralen“ Datenhaltung eine besondere Bedeutung zukommt, so kann man ein stärkeres Gewicht auf die Datenmodellierung legen.

9.4.4 Abhängigkeiten zwischen Domänenmodell und Persistenzschicht

Modellierung der Domänenobjekte

Mit der Entkopplung der Persistenzschicht von der Geschäftslogik über eine Schichtenarchitektur ist ein wichtiger Schritt passiert, allerdings können sich in der Praxis noch weitere Probleme ergeben, wenn man Objektstrukturen persistieren möchte, die schon bei der Modellierung der Domänenobjekte bedacht werden sollten. Dies kann anhand eines einfachen Beispiels illustriert werden:

Beispiel: Projektmanagement-Werkzeug

Abbildung 9.9 zeigt eine mögliche Objekthierarchie, die man modellieren könnte, um ein Projektmanagement-Werkzeug zu schreiben. In diesem Beispiel gibt es:

- *Projects*, die in *Tasks* strukturiert sind, d. h., jedes Project kann mehrere Tasks haben.
- Ein *Task* ist eine bestimmte Aufgabe, die zu erledigen ist. Sie und kann weiter in eine Reihe von *Todos* aufgegliedert werden
- Außerdem nehmen wir an, dass man einem *Task* Resourcen zuweisen kann, z. B. *URLs* oder *Dateien*.

- Bei *Dateien* möchte man vielleicht verschiedene Typen unterscheiden, um sie getrennt behandeln zu könne (z. B. zum Indizieren).
- Zuletzt gibt es die Entität bzw. das Objekt *Person*, denn Personen arbeiten an Projekten, an Tasks, an Todos, laden Files hoch usw.

Tatsächlich wäre dies vermutlich nur ein Ausschnitt aus dem kompletten Datenmodell, aber für eine erste Überlegung sollte dies reichen. Das Klassendiagramm in Abbildung 9.9 zeigt alle Abhängigkeiten zwischen den Objekten an, d. h. es wird z. B. eine Beziehung vom Objekt *Person* zu allen anderen Objekten hergestellt, zumal es für alle anderen Objekte durchaus wichtig sein kann, die Person, die das Objekt erstellt, besitzt und bearbeitet, zu kennen. Dies bedeutet noch nicht, dass dies das *endgültige* Modell ist, das implementiert werden soll.

Kritische Diskussion

Es soll an dieser Stelle kein konkretes Modell oder ein konkreter Vorschlag erarbeitet, sondern kritisch die Konsequenzen verschiedener Architekturen diskutiert werden. In einem realen Projekt sind diese im konkreten Kontext gegeneinander abzuwägen. Möchte man dieses Problem nun modellieren und letztlich implementieren, so sollte man über drei Aspekte nachdenken:

1. Sollen wirklich alle prinzipiell vorhandenen Abhängigkeiten zwischen Objekten auch modelliert und vor allem implementiert werden? Welche Konsequenzen hätte dies in der Praxis?
2. Wie sieht das dazugehörige Datenmodell aus? Welche Art von Datenbank wird verwendet?
3. Welchen Einfluss haben Datenmodell und Objektmodell auf die Implementierung und Konfiguration (!) der Persistenzschicht sowie auf die Arbeit mit Objekten und Objekt-Hierarchien?

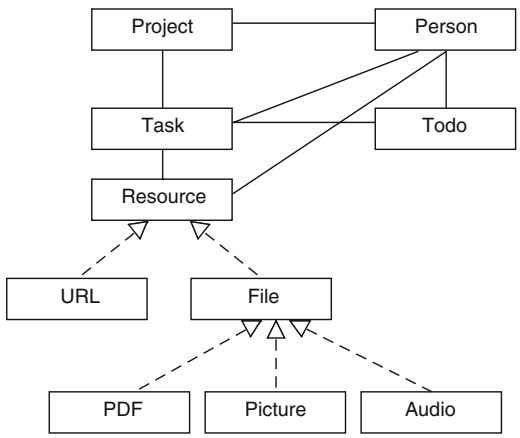


Abbildung 9.9 *Beispiel einer Objekthierarchie, die persistiert werden soll.*

Diskussion des Objektmodells

Betrachten wir zunächst das Objektmodell: Hier gibt es zwei „Extrempositionen“: (1) Es werden wie in der Abbildung alle logischen Verbindungen zwischen den Objekten auch tatsächlich modelliert und implementiert oder (2) es werden gar keine Verbindungen modelliert. Der Zusammenhang zwischen Objekten, z. B. einem Projekt und seinen Tasks, muss dann auf einem anderen Weg ermittelt werden. Modelliert man alle Verbindungen, so ergibt sich nun die Frage, ob dies uni- oder bidirektional erfolgen soll, z.B:

```
public Project {
  private long id;
  private List<Person> projectmembers;
  private List<Task> tasks;
  private String projectname;
  public String getProjectname() {...}
  public void setProjectname (String n) {...}
  public void addMember (Person p) {
    projectmembers.add(p);
  }
  public void removeMember (Person p) {...}
  public List<Person> members() {...}
  public void addTask (Task t) {...}
  public List<Task> tasks {...}
}
```

Dieser Code deutet die Struktur des `Project`-Objekts an. Dieses Objekt hat Stammdaten wie `projectname` aber auch Referenzen zu anderen Objekten, in diesem Fall zu `Person` und `Task` in Form von Listen, denn einem Projekt können ja mehrere Personen und Tasks zugewiesen sein. Das bedeutet, dass man von einer Projektinstanz leicht zu den Mitarbeitern (Tasks) in dem Projekt navigieren kann, z. B. so:

```
...
Project myProject = projectDAO.getProject("Test");
List<Person> members = myProject.members();
Person p1 = members.get(1);
p1.setEmail ("...");
...
```

In diesem Beispiel würde man sich z. B. eine Projektinstanz vom Data Access Object holen, danach folgt man der Referenz, holt sich die Liste der Projektmitarbeiter und ändert die E-mail-Adresse des Mitarbeiters mit dem Index 1. Ist die Abhängigkeit bidirektional implementiert, so muss es auch möglich sein, von der Person zurück zum Projekt und zu anderen Abhängigkeiten zu kommen, dies würde also etwa wie folgt aussehen:

```
public Person {
  private long id;
  % Abhängigkeiten
  private List<Project> projects;
  private List<Task> tasks;
  private List<Todo> todos;
  private List<Resource> resources;
  % "Stammdaten"
```

```
  private String name;
  private String email;
  % Zugriff auf Abhängigkeiten
  public void addProject (Project p) {...}
  public void addTask (Task t) {...}
  public void addTodo (Todo t) {...}
  public void addResource (Resource r) {...}
  public List<Project> projects() {...};
  public List<Task> tasks() {...};
  public List<Todo> todos() {...};
  public List<Resource> resources() {...};
  % Zugriff auf "Stammdaten"
  public void setName  (String n) {...}
  public String getName  () {...}
  public void setEmail (String e) {...}
  public String getEmail  () {...}
}
```

Man erkennt schon an diesem recht einfachen Beispiel, dass eine volle Implementierung aller Abhängigkeiten eventuell sogar bidirektional vergleichsweise aufwendig wird. Dazu kommt, dass die Handhabung in der Praxis etwas mühsam wird, da man bei Änderungen von Beziehungen immer gut darauf achten muss, diese in allen Referenzen korrekt zu aktualisieren.

Keine Referenzen?

Das andere „Extrem“ wäre es, gar keine Referenzen abzubilden. Das würde bedeuten, dass alle Objekte nur die Stammdaten halten, aber keine Bezüge zu anderen Objekten. Dann stellt sich natürlich die Frage, woher dann die wichtige Information bezogen wird, welche Tasks zu einem Projekt gehören, welche Todos zu welchem Task und welche Person eine bestimmte Datei angelegt oder verändert hat? Diese Information könnte im Datenmodell (z. B. in der relationalen Datenbank) abgebildet werden und entsprechen über einen Aufruf an ein Data Access Object oder ein Service-Objekt bezogen werden. Die Klassen könnten etwa so kodiert werden:

```
public Project {
  private long id;
  private String projectname;
  public void setProjectname (String n) {...}
  public String getProjectname() {...}
}
...
public Person {
  private long id;
  private String name;
  private String email;
  public void setName  (String n) {...}
  public String getName  () {...}
  public void setEmail (String e) {...}
  public String getEmail  () {...}
}
```

Verwendet man dann eine Instanz der `Project`-Klasse und möchte die zugehörigen Mitglieder des Projekts wissen, so definiert man eine entsprechende Methode im DAO; die Verwendung könnte etwa wie folgt aussehen:

```
Project p = projectDAO.getProject("Test");
List<Person> pmembers = projectDAO.getMembers(p);
```

Die Implementierung der Objekte ist in diesem Fall sehr einfach, dafür ist mehr Logik im Data Access Object notwendig; außerdem entspricht diese Vorgehensweise auch nicht unbedingt der Idee objektorientierter Programmierung. Dazu später mehr.

Diskussion Persistenz

Aber weiter zum letzten Aspekt, der Frage welche Auswirkung das Objekt-Design auf die Persistenz hat. An dieser Stelle ist es zunächst unerheblich welche Art von Daten-Persistenz verwendet wird. Wir gehen von zwei Annahmen aus: (1) es gibt eine saubere Trennung zwischen Geschäftslogik und Persistenz-Schicht mithilfe des Data Access Object Patterns (siehe Abschnitt 8.4.5) und (2) die Abhängigkeiten zwischen den Entitäten wird in der Datenbank abgebildet (Details dazu werden in den nächsten Abschnitten diskutiert.). Dennoch hat die Art der Modellierung der Objekte einen Einfluss auf die Persistenz-Schicht. Betrachten wir wieder die beiden oben beschriebenen „Extremfälle“, diesmal aber zunächst den einfacheren: Es wird keine Abhängigkeit in den Objekten modelliert. In diesem Fall geht die Implementierung der Data Access Objects recht leicht von der Hand, auch das Speichern von Objekten bereitet keine Probleme. Am Beispiel von `Person` könnte das DAO Interface etwa so aussehen:

```
public interface PersonDAO {
  public Person savePerson(Person p);
  public Person getPerson(String email);
  public void deletePerson(Person p);
  public List<Project> getProjects (Person p);
  public List<Task> getTasks (Person p);
  ...
}
```

Auf diese Weise können die Beziehungen zwischen Objekten über entsprechende DAO-Methodenaufrufe aufgelöst werden. Auch das Speichern einer neuen Person oder das Speichern von Änderungen ist logisch eindeutig: Ruft man die `savePerson()`-Methode auf, wird das entsprechende Objekt persistiert; Abhängigkeiten müssen explizit gespeichert werden. Um ein konkretes Beispiel zu geben: Angenommen, es wird ein neues Projekt angelegt und eine Person diesem Projekt zugewiesen, dann sollen diese beiden Objekte sowie die Abhängigkeit zwischen diesen beiden Objekten gespeichert werden. Dies könnte etwa wie folgt aussehen:

```
Project pr = new Project ("Mein Projekt");
Person pe = new Person ("Hans", "Huber");
projectDAO.saveProject (pr);
personDAO.savePerson (pe);
projectDAO.addPersonToProject (pr, pe);
```

In diesem Beispiel würde es eine entsprechende Methode im `ProjectDAO` geben, die erlaubt, Abhängigkeiten zu speichern.

Abhängigkeiten

Für den Fall, dass Abhängigkeiten modelliert werden, sieht die Sache etwas anders aus. Nehmen wir wieder an, es wären alle Abhängigkeiten wie in Abbildung 9.9 modelliert. Hier muss man sich einige grundlegende Fragen stellen: Was bedeutet es beispielsweise, wenn ein Objekt geladen wird, z. B. ein Projekt?

```
Project pr = projectDAO.getProject("Test");
```

Bedeutet dies, dass *nur* dieses eine Objekt geladen wird; also die Stammdaten? Andererseits hat das Objekt aber Referenzen auf andere Objekte, d. h., nichts würde den Programmierer davon abhalten, durch die Objektstruktur den Referenzen folgend zu navigieren, also etwa folgendes zu tun:

```
Project pr = projectDAO.getProject("Test");
List<Task> tasks = pr.tasks();
Task task = tasks.get (0);
List<Todo> todos = task.todos();
...
```

Dies funktioniert aber nur, wenn das ProjectDAO die Abhängigkeiten auflöst und auch die zugehörigen Tasks sowie alle Todos der Tasks geladen hat. Man erkennt schnell, dass dies in der Praxis nicht möglich ist bzw. nicht wünschenswert sein kann. Über mehrere Ecken (z. B. über `Person`-Objekte) kann es dann leicht passieren, dass man eigentlich nur die Stammdaten eines Projekts laden wollte, tatsächlich aber die gesamte Datenbank aller Projekte, aller Tasks, aller Todos, aller Resourcen usw. geladen hat, da diese prinzipiell über ein Fortschreiten entlang der Referenzen erreichbar wären. Dies ist offensichtlich keine gute Idee.

Diesem Problem kann man auf verschiedenen Wegen begegnen. Manchmal sieht man, dass bei einfachen Lade-Vorgängen nur die Stammdaten geladen und die Referenzen auf `null` gesetzt werden. Dies ist keine wirklich gute Idee, denn wie soll der Entwickler in diesem Fall wissen, ob die `null`-Referenz bei den Tasks bedeutet, dass das Projekt keine Tasks hat oder dass sie einfach nur nicht geladen wurden. Dies ist keine schöne Lösung. Genauso gut könnte man gleich beim einfachen ersten Ansatz bleiben und gar keine Abhängigkeiten modellieren.

Lazy Loading

Ein guter Ansatz wird von verschiedenen Frameworks angeboten. Dabei wird mit dem Proxy-Pattern gearbeitet. Im obigen Beispiel könnte das bedeuten, dass mit dem Aufruf `getProject()` tatsächlich nur das eine Objekt geladen wird. Die Referenz zu Tasks, Person usw. wird aber nicht auf `null` gesetzt, sondern das Framework setzt an diese Stelle ein Proxy-Objekt. Greift man nicht auf die Tasks zu, so werden auch keine Tasks geladen. Greift man aber auf die Tasks zu, so erkennt das der Proxy und lädt die Tasks nach. Diese Vorgehensweise wird auch *Lazy Loading* genannt.

Allerdings hat auch Lazy Loading in der Praxis seine Tücken: Denn einerseits muss das Proxy-Objekt immer über eine Verbindung zur Datenbank

verfügen (was nicht in jedem Einsatzbereich der Fall ist), andererseits kann es bedeuten, dass unnötig viele Zugriffe auf die Datenbank gemacht werden. Möchte man z. B. alle Todos eines Projekts in einer Tabelle anzeigen, so muss man folgende Schritte tun:

- Laden des Projekts,
- Laden aller Tasks dieses Projekts (z. B. 30 Tasks),
- Laden aller Todos dieser Projekts (z. B. in Summe 150 Todos).

Lazy Loading und Abfrage-Effizienz

Ist das Lazy Loading so konfiguriert, dass wirklich erst beim Zugriff auf das jeweilige Objekt die Daten geladen werden so wäre das Ergebnis, dass 181 Queries an die Datenbank abgesetzt werden. Diese Daten könnten aber mit einer einzigen (SQL) Query leicht geladen werden, wenn man das Zugriffsschema vorher kennt. Das bedeutet also, dass die Umsetzung komplexer Objektmodelle auf vernünftige Persistenzstrategien nicht ganz einfach ist und auch gute Hilfsmittel wie Lazy Loading entsprechend der Zugriffsmuster korrekt konfiguriert und verwendet werden müssen.

Fazit

In der Praxis läuft es meist darauf hinaus, dass bestimmte Abhängigkeiten modelliert werden, aber in den meisten Fällen werden nicht alle Abhängigkeiten im Domänenmodell (vielleicht sogar bidirektional) abgebildet. Dies sieht zwar auf den ersten Blick elegant aus, führt aber mit ziemlicher Sicherheit zu Problemen in der Handhabung, besonders wenn man die Persistenz mit relationalen Datenbanken betrachtet.

Es bleibt die Frage offen, welche Abhängigkeiten sich nun im Objektmodell widerspiegeln sollten. Hier gibt es die Faustregel, dass Abhängigkeiten zwischen Objekten modelliert werden sollten, wo die Objekte alleine wenig Sinn machen würden. Ein Beispiel: Hat man eine Klasse, die Rechnungen repräsentiert, und eine Klasse, die Posten einer Rechnung abbildet, so ist es offensichtlich, dass die Posten einer Rechnung in unmittelbarem Zusammenhang mit der Rechnung stehen und umgekehrt. Eine Rechnung ohne die Posten der Rechnung ist unvollständig. Diese Abhängigkeit würde man sicher modellieren.

Im oberen Beispiel ist dies bei Weitem nicht mehr so eindeutig. Man könnte den Standpunkt vertreten, dass eigentlich alle Entitäten durchaus für sich alleine stehen können: Ein Projekt hat bestimmte Stammdaten, die für sich genommen Sinn machen, auch ohne dazugehörige Tasks usw. Andererseits ist sicher auch richtig, dass Projekt und Tasks sowie Todos das „Rückgrat“ des Projektmanagement-Werkzeugs bilden und stark voneinander abhängig sind und entsprechend modelliert werden. Ziemlich sicher ist jedoch, dass die Abhängigkeiten von der `Person`-Klasse weg nicht abgebildet werden. Auch mit bidirektionalen Abhängigkeiten geht man meist vorsichtig um, da sonst der Code und die Persistenz recht umständlich zu implementieren werden. Grundsätzlich sollte man, selbst wenn architektonisch die Persistenz-Schicht von der Geschäftslogik getrennt ist, bei der Modellierung sowohl das Datenmodell als auch das Domänenmodell im Auge haben.

9.4.5 Proprietäre binäre Datenformate

Spezielle Behandlung

Für dokumentorientierte Daten wie Texte, Bilder, Musikstücke, Videos haben sich eine Vielzahl an Binärformaten herausgebildet wie JPEG, MPEG, DOC usw. Daneben gibt es proprietäre Datei- und Datenbankformate, die Entwickler für ihre Anwendungen „maßschneidern“. Auf diese Art der Persistenz wird hier nicht näher eingegangen. Hat man die Anforderung, mit multimedialen Inhalten bzw. mit proprietären Formaten zu arbeiten, so bedient man sich entsprechender Bibliotheken bzw. wird um ein Detailstudium der jeweiligen Formate nicht umhin kommen.

Für eigene Anwendungen proprietäre (binäre) Dateiformate zu definieren, ist natürlich möglich. Für die überwiegende Zahl der Anwendungsfälle ist davon aber eher abzuraten. Im Einzelfall kann dies Sinn machen, meist ist es günstiger, für den Fall von dokumentorientierten Daten das XML-Format zu wählen, für andere Daten aber auf eine der verfügbaren Datenbanktechnologien (objektorientiert, relational, XML) zurückzugreifen.

9.4.6 Objektorientierte Datenbanken

Enge Integration

Objektorientierte Datenbankenmanagementsysteme (OODBMS) sind in der Lage, direkt mit Objekten einer objektorientierten Sprache zu arbeiten. Damit gibt es in dem Sinne keine eigene Datenmodellierung, sondern das Domänenmodell ist gleichzeitig das Datenmodell. Daraus folgt, dass die Verwendung von OODBMS meist recht einfach ist, da man sich nicht überlegen muss, wie man das Objektmodell auf das Datenmodell abbildet. Eine Persistenzschicht mit einer OODBMS ist daher in der Regel sehr schnell geschrieben. Die Integration zwischen Datenbank und objektorientierter Programmiersprache ist sehr eng.

Beispiel: db4o

Ein Beispiel (teilweise Pseudocode, Fehlerbehandlung wird ignoriert), das an die API der objektorientierten Datenbank db4o[12] angelehnt ist:

```
1  % Das Domänenobjekt
2  public Person {
3    private String name;
4    private String email;
5    public Person (...) {...}
6    public void setName  (String n) {...}
7    public String getName  () {...}
8    public void setEmail (String e) {...}
9    public String getEmail  () {...}
10 }
```

[12] http://www.db4o.com

```
1  public Test {
2    public void test() {
3      % Speichern
4      ObjectContainer db = DB4o.openFile("databasename
           ");
5      Person p = new Person ("Alex", "Schatten", "
           email");
6      db.set (p);
7      % Laden
8      Person prot = New Person (null, "Schatten", null
           );
9      ObjectSet result = db.get(proto);
10     Person found = (Person)result.next();
11     % Entfernen
12     db.delete (found);
13   }
14 }
```

Dieses Beispiel ist zwar sehr einfach, aber man erkennt doch sofort, wie elegant der Umgang mit der Datenbank ist. In diesem Beispiel wird die Abfrage über „Prototypen“ gemacht. Dabei erstellt man ein Objekt, das in einigen Eigenschaften mit dem gesuchten Objekt übereinstimmt. Je nach OODBMS gibt es aber dann meist noch eine Vielzahl von anderen mächtigeren Abfragemöglichkeiten.

Einschränkungen

Dies hört sich im Prinzip sehr schön an, und dennoch gibt es einige Aspekte zu bedenken, bevor man ein OODBMS in Betracht zieht. Wieder ist die Tatsache zu nennen, dass OODBM-Systeme meist sehr eng mit der Programmiersprache verbunden sind. Im Falle von db4o ist dies Java oder .net. Dies macht einerseits die Entwicklung einfach, andererseits gibt es kein vom konkreten Programm abstrahiertes Datenmodell. Daten wird aber häufig ein höherer Wert beigemessen als einer bestimmten Anwendung, da die Daten „länger leben“. Außerdem kommt es häufig vor, dass auf bestimmte Daten von verschiedenen Systemen heraus zugegriffen werden soll, z. B. von verschiedenen Programmiersprachen, lokal und remote, von Reporting-Systemen, Modellierungs-, Administrations-, Backup-Werkzeugen oder Data Warehouses usw. Für solche Anwendungsfälle ist ein Datenmodell, das sehr eng mit einer Anwendung verknüpft ist, meist recht ungünstig.

Eng damit verknüpft ist das Problem, dass es bei OODBMS bisher keine standardisierten Zugriffsprotokolle und Abfragesprachen gibt. Dies macht den Zugriff verschiedener Systeme auf eine OODB eher schwierig. Dies führt auch dazu (wie das obige Beispiel zeigt), dass man mangels Standards meist eng mit einer bestimmten API einer DB arbeiten muss, im Beispiel db4o, und man sich damit sehr eng an dieses System bindet.

Leider gibt es auch nur sehr wenige OODBM-Systeme die den „Hype“ der 1990er Jahre überlebt haben, und es gibt praktisch keine OODBM-Systeme im Open-Source-Umfeld. Db4o ist zwar auch unter einer Open-Source-Lizenz verfügbar, aber nur unter GPL. Berücksichtigt man die enge Bindung,

die eine Anwendung mit db4o eingeht, bedeutet dies, dass auch db4o nur für Projekte verwendet werden darf, die selbst unter GPL stehen, oder man greift auf die kommerzielle Version zurück.

Darüber hinaus sollte man sich die Anforderungen an die Datenbank genau überlegen (wie in Abschnitt 9.4 diskutiert). Denn OODBMS verfügen oft nicht über „Enterprise Features“, wie sie für die meisten relationalen Datenbanken selbstverständlich sind.

Fazit

Es stehen also Einschränkungen von OODBM-Systemen der sehr einfachen und eleganten Verwendung gegenüber. Abstrakte Datenmodelle und Abfragesprachen, die mit Objektorientierung wenig gemein haben (wie SQL), dafür aber sehr vielfältig eingesetzt werden können, stehen sehr plattform-angepassten Abfragemöglichkeiten bei OODBM-Systemen gegenüber. Die Wahl einer geeigneten Datenbank sollte also gründlich überlegt werden. Für einige Anwendungsfälle kann man mit OODBM-Systemen sicherlich schneller und effizienter zu Lösungen kommen als beispielsweise mit relationalen Datenbanken, in den meisten Anwendungsfällen werden aber wohl die Einschränkungen der OODBM-Systeme eher gegen einen Einsatz sprechen.

9.4.7 Relationale Datenbanken und objektorientierte Programmierung

Das relationale Modell

Relationale Datenbanken sind mit Sicherheit die meistverbreitete Art von Datenbanksystemen. Das relationale Datenbankmodell wurde von Edgar F. Codd schon 1970 vorgeschlagen [20]. In relationalen Datenbanken werden Daten in Tabellen gespeichert (*Relationen* genannt), die in Spalten (*Attribute*) und Zeilen (*Tupel*) strukturiert sind. Jedes Tupel hat eindeutig zu sein, sich also in wenigstens einem Attribut von den anderen zu unterscheiden. Ein Datensatz entspricht also einer Zeile der Tabelle. Das Relationenschema legt dabei die Struktur der Tabelle fest, also über welche Anzahl und Art von Attributen sie verfügt. Beziehungen zwischen Tabellen können ausgedrückt werden.

Entwurf von Datenbanken

Im Entwurf wird jede Datenbanktabelle genau beschrieben: (a) Welche Felder beinhaltet die Tabelle, (b) Welchen Datentyp haben diese Felder (c) Welche Information wird in diesem Feld gespeichert, (d) Angabe der Schlüssel (Foreign Key, Primary Key) sowie (e) Mit welcher Option werden diese Felder angelegt.

Beispiel

Abbildung 9.10 zeigt eine Beispiel-Datenbankstruktur. Man erkennt sieben Entitäten wie `Professor` oder `Exam`. Zwischen verschiedenen Tabellen gibt es Beziehungen. Beispielsweise muss eine Prüfung einem Kurs zugeordnet sein, daher gibt es in der Tabelle `Exam` das Feld `courseid`, das den Bezug zu einem bestimmten Kurs herstellt.

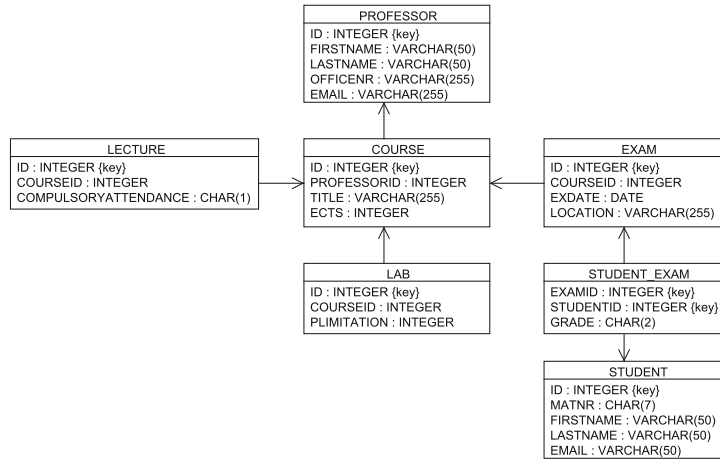


Abbildung 9.10 *Beispiel eines relationalen Datenbankschemas.*

Vorteile

Was man schon an diesem Beispiel erkennen kann, ist, dass die Daten „neutral“ unabhängig von einem bestimmten Programmiermodell oder einer Programmiersprache abgebildet werden. Auf diese Daten kann von verschiedenen Systemen über standardisierte Schnittstellen wie JDBC oder ODBC zugegriffen werden. Die Sprache SQL stellt dabei die notwendige Logik zur Verfügung, um Daten einzufügen, zu ändern, zu löschen oder abzufragen. Dies ist ein großer Vorteil relationaler Datenbanken. Ein weiterer Vorteil relationaler Systeme ist, dass es eine wirklich große Anzahl an verschiedenen Produkten für unterschiedliche Anforderungen und Plattformen gibt. Dies beginnt mit sehr teuren kommerziellen Systemen, über eine große Anzahl an Open-Source-Produkten, bis zu sehr kleinen und schlanken Systemen (wie das in den Beispielen verwendete hsqldb), die nur sehr geringe Resourcen benötigen und auch embedded z. B. auf Mobiltelefonen eingesetzt werden können. Zudem verfügen die „größeren“ RDBM-Systeme über ausgefeilte Transaktionsmechanismen, Clustering-Fähigkeit, Backup im laufenden Betrieb, feingranulare Zugriffsrechte auf Tabellen oder sogar Attribute. Stand der Technik sind auch Funktionen, die die Datenkonsistenz gewährleisten sollen (*constraints*) sowie Prozeduren und Funktionen, die auf der Datenbank ausgeführt werden können und die z. B. von Datenänderungen ausgelöst werden.

Impedance Mismatch

Allerdings erkennt man im obigen Beispiel auch schon den „Bruch“ zwischen dem Datenmodell, der Datenbank und der Programmiersprache. Die damit verbundenen Probleme werden oft auch *Impedance Mismatch* (am besten mit „Unverträglichkeit“ übersetzt) genannt. Einige Detailprobleme sollen hier kurz analysiert werden: Ein wesentlicher Unterschied folgt aus dem Verständnis von *Identität*. In einer relationalen Datenbank gelten zwei Tupel als identisch, wenn sie in allen Attributen übereinstimmen. Bei Objekten trifft das nicht zu: zwei Objekte gelten *nicht* als identisch, selbst dann nicht, wenn sie in allen Eigenschaften übereinstimmen.

Vererbung und Assoziationen

Das relationale Modell kennt keine Vererbung; möchte man Objekthierarchien in einer relationalen Datenbank persistieren, muss man sich Workarounds überlegen. Schließlich werden Assoziationen anders abgebildet. Dies trifft besonders auf 1:n- sowie m:n-Assoziationen zu. In Abbildung 9.10 gibt es beispielsweise die Entität `PROFESSOR`, und ein Professor kann mehrere Kurse abhalten (`COURSE`). Würde man diesen Kontext in Objekten abbilden, würde dies wohl etwa so aussehen:

```
public class Professor {
  private String firstname;
  private String lastname;
  ...
  private List<Course> courses;
}
...
public class Course {
  private String title;
  private int ects;
}
```

Im Relationenmodell wird eine solche 1:n-Beziehung aber genau umgekehrt abgebildet: hier gibt es in der Tabelle `COURSE` ein Feld `PROFESSORID`, das einen Bezug zu einem bestimmten Professor herstellt. Die Abbildung von m:n-Assoziationen kann man am Beispiel in Abbildung 9.9 zeigen. Eine Person kann in mehreren Projekten arbeiten, und einem Projekt können mehrere Personen zugewiesen werde. Dies könnte man wie folgt kodieren:

```
public class Person {
  private String firstname;
  private String lastname;
  ...
  private List<Project> projects;
}
...
public class Project {
  private String projectname;
  ...
  private List<Person> projectmembers;
}
```

In diesem Codebeispiel wird die Beziehung bidirektional abgebildet. In jeder Klasse gibt es eine Liste, deren Elemente Referenzen auf Instanzen der anderen Klasse haben. In der relationalen Datenbank muss dies mit einer Zwischentabelle ausgedrückt werden; die Tabellen `PERSON` und `PROJECT` halten die Stammdaten von Personen und Projekten, eine Tabelle `personproject` stellt z. B. den Bezug zwischen Studenten und Prüfung her (siehe Abbildung 9.11). Allerdings können in der Zwischentabelle neben den Referenzen auch noch weitere Attribute vorhanden sein, wie dies z. B. in Abbildung 9.10 zu sehen ist: Die Tabelle `STUDENT_EXAM` drückt einerseits eine m:n-Beziehung zwischen `STUDENT` und `EXAM` aus, speichert aber auch das Ergebnis der Prüfung (`GRADE`). In diesem Fall könnte

man dies mit einer eigenen Klasse abbilden, zumal ja jeder Student in einer Prüfung eine eigene Note bekommt. Aber auch hier gibt es grundsätzlich verschiedene Möglichkeiten.

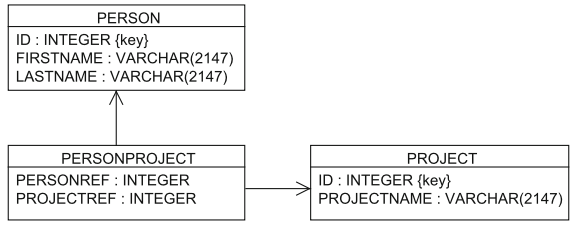


Abbildung 9.11 *Eine m:n-Abbildung in der relationalen Datenbank über eine Zwischentabelle.*

Man erkennt schon an diesen sehr einfachen Beispielen, dass die Abbildung von Objekten auf relationale Tabellen nicht immer ganz einfach ist. Man bezeichnet diesen Vorgang auch als O/R Mapping. Lösungsmöglichkeiten sowie die Konzepte moderner Frameworks, die den Entwickler dabei unterstützen sollen, sind in Abschnitt 9.4.8 thematisiert.

Fazit

Zusammengefasst kann man sagen, dass relationale Datenbanken heute nach wie vor die am meisten verwendete Persistenzlösung darstellen. Es gibt eine Vielzahl an Systemen, die sehr ausgereift sind und die auch über „Enterprise Features" verfügen. Die Daten werden in einer von programmiersprachenunabhängigen „neutralen" Form gespeichert, was gleichzeitig Vor- und Nachteil sein kann. O/R Mapping kann in der Praxis einige Probleme verursachen, die man bei objektorientierten Datenbanken in dieser Form nicht kennt. Daher muss auch hier wieder gesagt werden: Man sollte die Anforderungen im Projekt genau analysieren, um eine geeignete Lösung zu wählen.

9.4.8 Objektrelationales Mapping

Das Problem

In Abschnitt 9.4.7 wurde anhand einiger Beispiele die Problematik dargelegt, dass es erhebliche konzeptionelle Unterschiede zwischen relationalen Datenbanken und objektorientierten Sprachen gibt. Die Abbildung von Daten, die in Objekten gespeichert werden, auf Tabellen wird als objektrelationales (O/R) Mapping bezeichnet. Im Wesentlichen stößt man auf folgende Problembereiche:

- Die Zustände von Objekten müssen in Tabellen gespeichert werden.
- Assoziationen zwischen Objekten müssen gespeichert werden (Collections, 1:1, 1:m, m:n).
- Vererbung muss abgebildet werden (dieses Konzept existiert im relationalen Modell nicht).

- Die Zugriffsstrategien müssen abgestimmt werden: „Navigation“ in Objektbäumen auf der einen Seite stehen SQL Queries auf der anderen gegenüber.
- Zugriffe auf die Datenbank sollten optimiert werden, da Zugriffe (über das Netzwerk) üblicherweise sehr teuer sind; daraus folgen komplexe Patterns wie Lazy Loading, Proxies, Caches usw.
- Modell- vs. Schema-Migration ist zu bedenken, jede Änderung in einem Modell muss im anderen nachgezogen werden; dabei müssen vorhandene Daten (z. B. beim Kunden) berücksichtigt werden!

Mögliche Lösungen

Um das O/R Mapping durchzuführen, gibt es verschiedene Varianten. Keine der hier vorgestellten Möglichkeiten ist den anderen grundsätzlich überlegen, sondern es hängt vom Anwendungsfall ab, welcher Weg besonders geeignet ist:

„Manuelles Mapping“ und Low-level-Interfaces

Zunächst gibt es natürlich die Möglichkeit, nur Low-level-Datenbank-APIs zu verwenden, wie z. B. JDBC, ado.net oder auch spezifische Datenbank-APIs wie `sqlite3` in Python. SQL-Anweisungen werden dann im eigenen Programm zusammengebaut bzw. aus Konfigurationsdateien geladen und über die API an die Datenbank gesendet. Das Mapping zwischen Tabellen und Objekten wird explizit ausprogrammiert. Diese Variante ist nur in den seltensten Fällen zu empfehlen, z. B. für Batch-Prozesse. Die Gefahr bei diesem Ansatz besteht darin, dass man über kurz oder lang auf viele Probleme stößt und letztlich ein eigenes O/R Mapping Framework programmiert. Falls es für spezielle Problemfälle erforderlich sein sollte, sehr low-level mit der Datenbank zu interagieren, sollte man sich nach JDCB-Helper-Bibliotheken oder Template-Mechanismen (z. B. Spring JDBC Template[13], FreeMarker[14], und Velocity[15]) umsehen, die wenigstens das Zusammenbauen von SQL-Anweisungen erleichtern. Denn SQL-Anweisungen mit String-Funktionen „zusammenzuschrauben“ ist eine äußerst lästige und fehleranfällige Angelegenheit.

Data Mapper

Die nächste Stufe stellen Frameworks wie iBatis[16] dar. Dabei handelt es sich gar nicht um den Versuch, Objekte *direkt* auf Tabellen abzubilden, sondern es werden Objekte auf SQL-Anweisungen abgebildet. Die Grundidee am Beispiel von iBatis ist folgende:

- Man externalisiert SQL-Anweisungen in sogenannte SQLMaps. Damit wird eine saubere Trennung zwischen Java und SQL-Code ermöglicht.

[13] `http://www.springframework.org/`
[14] `http://freemarker.org`
[15] `http://velocity.apache.org/`
[16] `http://ibatis.apache.org/`

- Hier können auch Fragmente (z. B. für variable Abfragen) kombiniert, parametrisiert usw. werden.
- Weiterhin definiert man, wie Parameter und Abfrageergebnisse zwischen Objekten und SQL-Anweisungen ausgetauscht werden.
- Das iBatis Framework erledigt dann die Kommunikation mit der Datenbank, baut die SQL-Anweisungen zusammen und tauscht die Daten wie angegeben mit den Java-Objekten aus.
- Lazy Loading und Caching werden ebenfalls unterstützt.

Das bedeutet, dass iBatis gar nicht den Versuch unternimmt, SQL-Anweisungen zu verbergen, sondern stellt diese vielmehr ins Zentrum der Interaktion mit der Datenbank. Über die SQLMaps werden diese SQL-Anweisungen für den Entwickler viel leichter handhabbar. Dabei können in einem Programm die Stärken handgeschriebener SQL-Anweisungen bei gleichzeitig einfacher Verwendung vom Java-, Ruby- oder .net-Programm aus kombiniert werden. Dazu kommt, dass iBatis recht einfach zu erlernen und zu verstehen ist, was auf viele der komplexeren Frameworks nur bedingt zutrifft.

Full O/R Mapping

Die heute am meisten verwendete Kategorie sind O/R Mapping Frameworks, die von Objekten auf Tabellen abbilden und dabei versuchen, SQL-Anweisungen im Hintergrund zu generieren, zu verwenden und dabei vor dem Programmierer „zu verstecken“. Dies hat gleichtzeitig den Vorteil und Nachteil, dass versucht wird von einer bestimmten relationalen Datenbank zu abstrahieren. Der Vorteil liegt darin, dass man die Datenbank im Hintergrund relativ leicht austauschen kann, man sich also nicht an ein bestimmtes System bindet. Man hat also idealerweise eine saubere Trennung zwischen „der Welt“ der Objekte und der Datenbank. Der mögliche Nachteil ist darin zu sehen, dass man dann natürlich die spezifischen Stärken einer Datenbank (z. B. optimierte SQL-Anweisungen) nicht ausnutzen kann. Man kann bei den meisten O/R Mappern optional auch die generierten SQL-Anweisungen verändern bzw. eigene SQL-Anweisungen hinterlegen (ähnlich wie bei iBatis), womit man klarerweise die Abstraktion der Datenbank verliert.

Für O/R-Mapping-Werkzeuge gibt es in verschiedenen Programmiersprachen bereits Standards (z. B. JPA in Java), die in alle wesentlichen Enterprise Frameworks integriert sind (dies schließt z. B. Transaktions-Management und Caching ein). Weiter gibt es eine große Anzahl an Frameworks, unter denen man als Entwickler auswählen kann. Für Java und .net kann man Frameworks wie JBoss Hibernate[17], für Java EclipseLink[18] oder Apa-

[17] http://www.hibernate.org/
[18] http://www.eclipse.org/eclipselink/

che Cayenne[19] nennen. In Ruby/Rails gibt es ActiveRecord[20], für Python SqlAlchemy[21].

Das Mapping

Um das Mapping von Objekten auf Tabellen zu beschreiben, gibt es meist verschiedene Wege. Eine Möglichkeit, die die meisten Frameworks bieten, ist eine Definition über eine (XML-)Konfigurationsdatei. In dieser Datei wird beschrieben, wie ein bestimmtes Objekt auf eine oder mehrere Tabellen abzubilden ist, welche Variablen in welche Spalten in der Tabelle zu lesen oder schreiben sind. In manchen Fällen ist auch zu beschreiben, wie mit inkompatiblen Datentypen umzugehen ist. In Java gibt es beispielsweise einen Boolean-Typ, der nicht in allen relationalen Datenbanken vorhanden ist. Außerdem sind dort Assoziationen zu anderen Objekten zu definieren und festzuhalten, wie mit Vererbung umzugehen ist. Bei Assoziationen gibt es meist umfangreiche Konfigurationsmöglichkeiten für 1:n-, m:n-, 1:1-Assoziationen; außerdem werden meist auch Collections wie Listen, Sets, Maps unterstützt. Auch der konkrete Umgang mit Primärschlüsseln ist zu definieren; welches Feld oder welche Felder sind Primärschlüssel, sollen Auto-increment-Generatoren verwendet werden usw.

Vererbung

Vererbung ist ein verhältnismäßig komplexes Problem, da relationale Datenbanken dieses Konzept nicht kennen. O/R Mapper bieten typischerweise drei verschiedene Möglichkeiten, Vererbung auf die Datenbank abzubilden:

- Eine Tabelle pro Klasse.
- Eine Tabelle pro konkrete Klasse.
- Eine Tabelle pro „Klassenfamilie“/Hierarchie.

Dies kann an einem einfachen Beispiel illustriert werden: Angenommen, man hat die abstrakte Klasse `Fahrzeug` und zusätzlich zwei Klassen `Auto` und `LKW`, die beide von `Fahrzeug` abgeleitet sind. Nun haben wir es mit Instanzen der beiden konkreten Klassen `Auto` und `LKW` zu tun.

Tabelle pro Klasse

Im ersten Fall wird pro Klasse eine Tabelle angelegt, also eine für `Fahrzeug`, eine für `Auto` und eine für `LKW` (siehe Abbildung 9.12). Diese Variante ist eine an sich recht schöne Modellierung und leicht zu verstehen. Sie hat aber einen Performance-Nachteil, da vor allem bei komplexeren Vererbungen ziemlich viele SQL-Joins gemacht werden müssen, um die Daten, die in den einzelnen Tabellen liegen, wieder zusammenzuführen.

Tabelle pro konkreter Klasse

Wählt man die Variante *Tabelle pro konkreter Klasse*, so wird nur pro *konkrete* Klasse eine Tabelle angelegt, in diesem Beispiel nur für `Auto`

[19] `http://cayenne.apache.org`
[20] `http://rubyonrails.org`
[21] `http://www.sqlalchemy.org`

```
Fahrzeug
+------------+-----------+-------+
|Kennzeichen | Marke     | Farbe |
+------------+-----------+-------+
| W12345     | Audi      | rot   |
| W43210     | Mercedes  | grün  |
+------------+-----------+-------+

Auto                        LKW
+------------+-------+      +------------+----------+
|FahrzeugRef | Türen |      |Kennzeichen | Nutzlast |
+------------+-------+      +------------+----------+
| W12345     |   5   |      | W43210     |   6500   |
+------------+-------+      +------------+----------+
```

Abbildung 9.12
OR-Mapping-Beispiel: Tabelle pro Klasse.

und `LKW` (siehe Abbildung 9.13). Diese Variante bietet Performance-Vorteile, weil keine Joins gemacht werden müssen. Auch Zugriff von Reporting-Werkzeugen ist einfach, weil alle zusammengehörigen Daten in einer Tabelle stehen. Der Nachteil ist offensichtlich: Diese Variante ist nicht mehr in Normalform. Außerdem können verschiedene Situationen problematisch werden: Was ist, wenn sich Rollen ändern; also z. B. ein Auto zum LKW wird? Dann müssen Daten umkopiert werden. Problematisch ist auch der Fall, wenn ein Fahrzeug in beide Kategorien gehört; wo wird es dann gespeichert?

Eine Tabelle pro Hierarchie

Zuletzt gibt es noch die Variante *eine Tabelle pro Hierarchie*. Diese Version ist sehr einfach umzusetzen: Für die gesamte Hierarchie wird nur eine Tabelle angelegt (siehe Abbildung 9.14). Es werden dann die Daten aller Objekte in dieser Tabelle gespeichert. Damit das O/R Framework erkennen kann, um welchen Objekttyp es sich bei einem bestimmten Datensatz handelt, muss eine Diskriminator-Spalte eingeführt werden. Im Beispiel wäre

```
Auto
+-------------+----------+-------+----------+
| Kennzeichen |  Marke   | Farbe |  Türen   |
+-------------+----------+-------+----------+
|   W12345    |   Audi   |  rot  |    5     |
+-------------+----------+-------+----------+

LKW
+-------------+----------+-------+----------+
| Kennzeichen |  Marke   | Farbe | Nutzlast |
+-------------+----------+-------+----------+
|   W43210    | Mercedes | grün  |   6500   |
+-------------+----------+-------+----------+
```

Abbildung 9.13
OR-Mapping-Beispiel: Tabelle pro konkreter Klasse.

Typ	Kennzeichen	Marke	Farbe	Türen	Nutzlast
A	W12345	Audi	rot	5	
L	W43210	Mercedes	grün		6500

Abbildung 9.14
OR-Mapping-Beispiel: Tabelle pro Hierarchie.

`A` für Objekte vom Typ `Auto` und `L` für `LKW`. Auch von der Performance her ist diese Variante (zumindest bei einfachen Hierarchien) sehr gut.

Trotz aller Vorteile entspricht diese Variante natürlich alles anderem als einem schönen Datenbank-Design. Nicht nur ist ein Diskriminatorfeld notwendig, es finden sich meist auch Felder, die für bestimmte Objekte keinen Sinn machen.

Annotationen

Neben der Konfiguration mit externen Dateien erlauben die meisten O/R Mapper auch die Verwendung von Java Annotationen (siehe auch Abschnitt 8.2.6). Beide Vorgehensweisen haben Vor- und Nachteile: Die Annotationen sind „näher" am Code und damit oft leichter wartbar. Auf der anderen Seite werden hier zwei Aspekte vermischt, denn in Domänenobjekten werden Details der Persistenz eingebettet, was eigentlich einer sauberen Trennung widerspricht. Dies spricht eher für externe Konfigurationsdateien.

Abfrage

Um Daten abzufragen, werden verschiedene Strategien angeboten. Meist gibt es die Abfragemöglichkeit über eine API oder *by example*, womit sich relativ einfach Abfragen erstellen lassen. Man stellt aber schnell fest, dass die Query APIs für viele Probleme zu eingeschränkt sind, daher bieten die Mapper oft eigene Abfragesprachen. Hibernate verwendet z. B. hql. Dabei handelt es sich um eine an SQL angelehnte Sprache, die wesentlich mehr Möglichkeiten als die API bietet, und auch eine genauere Definition erlaubt, welche Objekte konkret geladen werden sollen und welche erst per Lazy Loading gegebenenfalls nachzuladen sind. Außerdem ist es meist auch möglich, SQL zu verwenden, beispielsweise um besondere Optimierungen vorzunehmen. Damit bricht man aber die Abstraktion von der Datenbank.

Fazit

Moderne O/R Mapping Frameworks bieten einen enormen Funktionsumfang. Wenn man sie im Griff hat, erlauben sie in vielen Projekten eine elegante Objektpersistenz. Was aber manchmal gerne unterschlagen wird, ist die Tatsache, dass O/R Mapping an und für sich über viele kleinere und größere Probleme verfügt, die auch von O/R Mappern nicht beseitigt werden können. Der hohe Grad an Abstraktion bietet in vielen Projekten Vorteile, ist aber nicht für alle Einsatzbereiche geeignet. Hat man es beispielsweise mit einer bestimmten Datenbank zu tun, gegen die zu Programmieren ist, ist es oft von Vorteil, wenn man direkt mit SQL arbeitet und die Stärken der jeweiligen Datenbank ausnutzt. Die Frameworks sind meist gut dokumentiert, aber sehr umfangreich und komplex. Das Problem ist oftmals, dass der erste Einsteig recht locker von der Hand geht, dann aber viele Details

nicht richtig verstanden werden, und man im schlimmsten Fall das Gesamtverhalten der Anwendung nicht mehr im Griff hat.

Man sollte sich also der Vor- und Nachteile dieser Frameworks bewusst sein. Wenn man sich auf Hibernate & Co einlässt, muss man sich auf eine erhebliche Einarbeitungszeit gefasst machen, um die Tiefen des Frameworks ausreichend gut zu verstehen und dieses effizient nutzen zu können. Für viele Anwendungsfälle können aber durchaus auch einfachere Werkzeuge wie iBatis eine gute und viel einfacher zu verstehende Alternative darstellen, oder es kann auch die Verwendung einer objektorientierten Datenbank in Erwägung gezogen werden.

9.4.9 XML-Persistenz

XML

Die Extensible Markup Language (XML) hat sich zu einem Standard für den Datenaustausch über Systemgrenzen etabliert und wird daher auch sehr gerne als Datenformat für dokumentorientierte und semi-strukturierte Daten verwendet. XML ist ein textbasiertes Datenformat, das eine strukturierte und hierarchische Erfassung von Daten erlaubt.

```
<project id="p1" name="CRM Introduction">
  <members>
    <person id="pe1" status="manager">
      <firstname>Petra</firstname>
      <lastname>Huber</lastname>
      ...
    </person>
    ...
  </members>
  <tasks>
    <task id="t1"
      name="Evaluate Options"
      priority="1"
    >
      <description>...</description>
    </taks>
    ...
  </tasks>
</project>
<project id="p2">
...
</project>
```

Man erkennt Elemente (Tags) wie `<project>` und `<task>`; diese strukturieren die Daten und dienen als Meta-Information. Elemente können auch Träger von Attributen wie `name` oder `id` sein. Um Elemente und Attribute auch über Anwendungen hinweg (global) eindeutig zu benennen, können Namensräume definiert werden. Zur Definition von XML-Da-

XML-Standards

tenformaten gibt es Standards wie *Document Type Definitions* DTD oder W3C-Schemas. Viele andere Standards bieten weitere Möglichkeiten wie beispielsweise XLink zur Definition von Links zwischen Daten sowie XSL, um Transformationen zwischen verschiedenen XML-Formaten oder zu anderen Formaten wie PDF durchzuführen. Abfragesprachen wie XPath und XQuery erlauben es, Teile von XML-Dokumenten zu extrahieren bzw. Abfragen auf XML-Datenbanken durchzuführen.

Die meisten XML-Standards sind im Rahmen des World-Wide-Web-Konsortiums[22] offen spezifiziert, entsprechende Standards sowie Links zu weiterführender Information können dort gefunden werden. Es gibt auch eine große Auswahl an Tutorials im Web, z. B. W3Schools[23] oder SelfHTML[24] sowie eine große Zahl an Büchern.

XML-Persistenz

Das XML-Format bietet sich daher auch an, um Daten aus Anwendungen zu persistieren, wobei hier im Wesentlichen drei Szenarien Verwendung finden:

- Dateibasierte Persistenz, d. h., Daten werden von der Anwendung in eine oder mehrere Dateien geschrieben und von dort gelesen.
- Verwendung von XML-Datenbanken und Kommunikation über Netzwerk-Protokolle; in diesem Fall findet meist feingranularer Zugriff statt, außerdem gibt es die Möglichkeit, XML-basierte Abfragesprachen zu verwenden.
- Kommunikation mit Services, z. B. auf REST-Basis, d. h., XML-Daten werden an Services gesendet bzw. Abfragen an diese Services gestellt.

XML-Parsing

Die am meisten verwendete Variante ist wohl die dateibasierte Persistenz. Da XML-Dateien textbasiert sind, ist es prinzipiell möglich, eigene Parser zu schreiben, um XML-Daten zu verarbeiten. In der Praxis ist davon aber dringend abzuraten, da es viele Details gibt, die zu beachten sind. Außerdem gibt es für alle Programmiersprachen eine breite Auswahl an Bibliotheken, die die Arbeit mit XML-Daten recht einfach gestalten. Es haben sich im Wesentlichen drei unterschiedliche Parsing-Strategien durchgesetzt, aus denen man je nach Problemfall die geeignete wählt:

Simple API for XML-Parsing

Die Simple API for XML-Parsing (SAX) ist eine Low-level-Technik, die auch oft als Basis für andere Parser dient. SAX-Parser arbeiten ereignisbasiert und nutzen Callback-Funktionen. Ein SAX-Parser liest ein XML-Dokument linear von Anfang bis Ende und erkennt dabei Elemente,

[22] `http://www.w3.org`
[23] `http://www.w3schools.com/xml`
[24] `http://de.selfhtml.org/xml/`

Kommentare, Element-Inhalte usw. und erzeugt entsprechende Ereignisse, z. B. `StartDocument`, `StartElement` und `EndElement`. Als Entwickler registriert man eine Klasse beim Parser, die für die Ereignisse, die man verarbeiten möchte, entsprechende Methoden, also z. B. eine `StartElement`-Methode, zur Verfügung stellt. Wenn der Parser beim Lesen des Dokuments auf einen neuen Tag stößt, wird die entsprechende `StartElement`-Methode aufgerufen und der Name des Tags, die Attribute usw. übergeben.

SAX-Parser werden selten verwendet, da man relativ viel selbst implementieren muss, um die XML-Daten auszuwerten. Zudem können SAX-Parser meist nur lesend und nicht schreibend arbeiten. SAX-Parser haben allerdings den großen Vorteil, dass sie sehr wenig Speicherbedarf haben und sehr schnell arbeiten. Muss man also beispielsweise sehr große XML-Dokumente importieren oder benötigt man nur bestimmte Informationen eines großen Dokuments, die leicht aus dem SAX-Event-Strom extrahierbar sind, kann die direkte Verwendung eines SAX-Parsers dennoch Sinn machen.

StAX

Ein „jüngerer Bruder“ von SAX ist der relativ neue StAX-Parser-Ansatz. Dieser arbeitet im Prinzip ähnlich wie SAX-Parser, nur mit dem Unterschied, dass nicht der Parser die Kontrolle übernimmt, das Dokument durchläuft und die gefundenen Elemente des XML-Dokuments an die eigene Klasse schickt. StAX-Bibliotheken arbeiten nach dem Pull-Prinzip, d. h., man initialisiert den Parser und fordert dann aktiv ein Element nach dem anderen aus dem XML-Datenstrom an.

Document Object Model

Sehr häufig werden DOM-Parser (Document Object Model) eingesetzt, die ganze XML-Dateien lesen und diese in einen Objekt-Baum im Speicher halten. Der Entwickler kann dann sehr einfach mit diesem Baum interagieren. Dieser Objektbaum ist allerdings generisch, d. h., es ist z. B. ein Baum von `Element`-Objekten. Jedes dieser Objekte speichert den Namen des Elements, Kindelemente usw. Einige DOM-Parser sind auch in der Lage, mit Abfragesprachen wie XPath oder XQuery umzugehen. Damit ist es möglich, bestimmte Informationen wie einzelne Elemente, Sub-Hierarchien oder Listen aus dem kompletten Baum per Abfrage zu extrahieren. Diese DOM-Objekthierarchien können auch verändert werden. Beispielsweise können Daten geändert oder neue Elemente hinzugefügt werden. DOM-Parser bieten dann die Möglichkeit, diese DOM-Bäume wieder zu XML-Dokumenten zu serialisieren. Im Gegensatz zu SAX-Parsern haben DOM-Parser aber den Nachteil, dass gesamte Dokumente im Speicher gehalten werden und damit der Speicherbedarf erheblich sein kann.

Mittlerweile gibt es allerdings auch Parser, die Mischformen der genannten Strategien implementieren und z. B. mit Objekthierarchien arbeiten, diese aber nicht komplett im Speicher halten müssen.

XML-Binding

DOM-Parser sind sehr einfach zu verwenden und bieten eine große Flexibilität. Sie haben aber den Nachteil, dass die Objektbäume aus generischen

Objekten bestehen. Das vorige XML Beispiel würde eine Hierarchie von Element-Objekten ergeben, die man sich etwa wie folgt vorstellen könnte:

```
[Element name="project" attributes="..."]
  [Element name="members"]
    [Element name="person" attributes="(id=x1,...)"]
  [Element name="tasks"]
...
```

In vielen Fällen würde man sich aber wünschen, dass die XML-Elemente nicht auf generische `Element`-Objekte abgebildet werden, sondern auf eine entsprechende Objektstruktur (Domänenmodell), also z. B. auf `Project`-, `Person`-, `Task`-Objekte. Möchte man dann z. B. die Tasks eines Projekts wissen, so ruft man die Methode `Project.getTasks()` auf und bekommt eine Liste als Ergebnis. Die Grundidee ist prinzipiell dem objektrelationalen Mapping sehr ähnlich. Für diesen Einsatzbereich kann man Objekt/XML Binding Frameworks wie Castor [25] oder XMLBeans[26]einsetzen. Mittlerweile gibt es im Java-Umfeld auch schon einen Standard: JAXB (Java Architecture for XML Binding)[27]; in der Java Enterprise Edition findet sich eine Implementierung dieses Standards.

XML-Datenbanken

Zuletzt sollen auch XML-Datenbanken nicht unerwähnt bleiben. Zwar ist es um diese Art von Datenbanken in den letzten Jahren eher still geworden, aber es gibt immer noch vereinzelte Systeme, die sich einer gewissen Beliebtheit erfreuen. Eines der aktiven Projekte ist beispielsweise eXist[28]. Dabei handelt es sich um eine in Java geschriebene native XML-Datenbank, die einer relationalen Datenbank ähnlich ist. Über eine Vielzahl an verschiedenen APIs kann man sich mit der Datenbank verbinden und XML-Daten speichern und abfragen. Die Datenbank indiziert dabei die XML-Daten und bietet Abfragesprachen wie XQuery und XPath an.

Neben nativen XML-Datenbanken bieten auch verschiedene relationale Datenbanken „XML-Aufsätze“ an. Dabei gibt es im Wesentlichen zwei verschiedene Konzepte: (1) Man definiert eine Abbildung von XML-Strukturen auf relationale Strukturen, dann werden XML-Daten im relationalen Modell gespeichert, bzw. Abfragen geben XML-Ergebnisse zurück. (2) Manche Datenbanken bieten XML-Feld-Typen an, in die beliebige XML-Daten geschrieben werden können. Oft ist es auch hier möglich, XPath- oder XQuery-Abfragen abzusetzen.

[25] `http://www.castor.org`: Castor ist sowohl ein O/R Mapper als auch ein Objekt/XML Binding Framework.

[26] `http://xmlbeans.apache.org`

[27] `https://jaxb.dev.java.net`

[28] `http://exist-db.org`

9.4.10 Dateisysteme mit Indexing-Mechanismen

Ein Spezialfall soll noch kurz beschrieben werden: In manchen Anwendungen hat man es zwar mit großen Datenmengen zu tun, diese sind aber nicht sehr strukturiert, also z. B. eine Mischung verschiedener Dokumente (Office, PDF, Bilder etc.) sowie XML-Dokumente. Weiterhin liegt fallweise der Schwerpunkt nicht auf speziellen Datenbankfunktionen, wie Transaktionen oder Multi-User-Fähigkeit, und es stehen überhaupt schnelle und flexible Suchen (für viele Clients) und weniger schreibende Transaktionen im Vordergrund. Auch in diesen Fällen kann man prinzipiell auf relationale Datenbanken zurückgreifen, es gibt aber auch eine andere Möglichkeit, die in derartigen Projekten gerne verwendet wird:

Apache Lucene

Die Daten werden in einem Dateisystem abgelegt und über entsprechende Dateisystem-APIs geladen und gespeichert. Daneben wird ein (Volltext-)Index-Framework wie z. B. Apache Lucene[29] installiert, das ebenfalls über eine API zu bedienen ist. Neue Daten werden damit indiziert, flexible und sehr schnelle Suchen sind über die API möglich. Der strukturierte Index kann Teile der Daten selbst speichern und verweist auf das entsprechende Dokument oder die XML-Datei. Dies ist zwar ein etwas unorthodoxer Persistenzmechanismus, kann aber in einigen Fällen äußerst effizient und gleichzeitig auch recht einfach zu implementieren sein.

9.4.11 Abstraktion über Services

Eigene Services

Eine letzte Variante, mit dem Problem der Persistierung von Daten umzugehen, ist in der Anwendung gar keinen eigenen Persistenzmechanismus zu implementieren, sondern Services zu verwenden, die andere Systeme anbieten. Auch hier gibt es eine Vielzahl an Möglichkeiten. Einerseits kann man in einem größeren System eigene Services implementieren, die sich um Persistenz kümmern, die aber über Service-Interfaces, z. B. auf SOAP/WSDL-Basis, verfügen. Auch viele Enterprise-Service-Bus-Produkte verfügen über Datenbank-Connectoren. Möchte man Daten persistieren, ist dieser Connector zu konfigurieren, und die Daten sind z. B. als JMS-Nachricht an diesen zu senden.

Enterprise Service Bus

Cloud Services

Immer interessanter werden sogenannte *Cloud Services*. Dabei handelt es sich um Dienstleister, die z. B. Storage Services anbieten. Der Dienstleister kümmert sich um Skalierung, Ausfallssicherheit etc. Als Kunde verwendet man die API und speichert die Daten auf den Servern des Dienstleisters. Die Vorteile liegen auf der Hand:

[29] `http://lucene.apache.org`

- Man muss keine eigene komplexe Hardwareinfrastruktur kaufen und warten.
- Man zahlt für den tatsächlich verbrauchten Speicherplatz und nicht für „Spitzenauslastung“, die de facto fast nie eintritt.
- Die meisten Dienste verteilen die Daten und sind ausfallsicher und redundant.
- Auch komplexe Aspekte wie Skalierung oder weltweite Verteilung von Downloads auf vielen parallelen Servern werden meist übernommen.
- Das Programmiermodell ist meist einfach und leicht zu verstehen.

Natürlich haben solche Systeme unter Umständen auch Nachteile: Man muss sich mit den Bedingungen des Anbieters, z. B. in Hinsicht auf Datenschutzrichtlinien, auseinandersetzen. Die APIs dieser Dienste sind heute auch oftmals proprietär, sodass ein Wechsel zwischen verschiedenen Anbietern nicht immer ganz einfach ist. Außerdem ist die Nutzung eines solchen Dienstes auch von den Datenmengen abhängig, die übertragen werden sollen. Da diese ja meist übers Internet übertragen werden, sind entsprechende Bandbreiten auf Client- und Anbieterseite erforderlich.

Zuletzt sollte man genau prüfen, ob die Möglichkeiten des Cloud Services (z. B. die Abfragesprache) den Projektanforderungen genügen.

9.5 Querschnittsfunktionen in Aspekte auslagern

9.5.1 Cross Cutting Concerns

Querschnittsfunktionen

Unter *Cross Cutting Concerns* versteht man Problemfelder, die in verschiedensten Modulen und in verschiedensten Schichten in ähnlicher Weise auftreten (z. B. Logging, Security, Transaktionssteuerung). Solche *Querschnittsfunktionen* können mit rein objektorientierten Konzepten nicht leicht modularisiert werden. Die *aspektorientierte Programmierung* (AOP) ist ein relativ neues Programmierparadigma, das versucht, die Implementation von Querschnittsfunktionen besser in den Griff zu bekommen. AOP baut dabei auf Objektorientierung auf und erweitert diese um sogenannte Aspekte. Unter einem Aspekt versteht man dabei die Implementation einer Querschnittsfunktion, z. B. einer Logging-Funktion, die dann in verschiedenen Modulen zum Einsatz kommen kann, ohne dass diese Module davon berührt werden.

Beispiel

Folgender Pseudo-Codeausschnitt soll verdeutlichen, wie eine konventionell implementierte Querschnittsfunktion den Code unleserlich und schwer wartbar machen kann. Die Anweisungen für solche Funktionen sind immer

gleich, und viele Entwickler tendieren dazu, diese Teile durch Kopieren zu vervielfältigen. In diesem Code ist die tatsächliche Geschäftslogik auf die Zeilen 7–9 beschränkt; der Rest sind Querschnittsfunktionen.

```
1  Logger logger = Logger.getLogger(...);
2  TransactionManager tm  = tmservice.
       getTrascactionManager();
3  public void addAccount(Account account) {
4    logger.info("Creating (" + account + ") Account");
5    try {
6      tm.startTransaction(...);
7      erp.add(account);
8      db.add(account);
9      crm.add(account);
10     tm.commit();
11   } catch (Exception) {
12     tm.rollback();
13     logger.error("Account creation failed");
14   }
15 }
```

9.5.2 Grundlagen der aspektorientierten Programmierung

Beispiel

In diesem Abschnitt werden zunächst die Grundbegriffe von AOP eingeführt. Die einzelnen Konzepte werden am Beispiel eines einfachen Logging-Aspekts gezeigt. Dieses Beispiel verfügt über ein einfaches Domänenmodell (siehe Abbildung 9.15), bei dem ein Kunde mehrere Konten bei unterschiedlichen Banken haben kann. Die Objekte `Kunde`, `Account` und `Bank` werden als Java-Objekte umgesetzt, die hier nicht angeführt sind.

Statisch und dynamisch

Bei der Einführung der Querschnittsfunktionalität kann zwischen der *statischen* und *dynamischen* Querschneidung unterschieden werden. Beim *dynamischen* Cross Cutting wird der Aspekt erst zur Laufzeit an das Programm angefügt. Dabei werden häufig Patterns wie etwa Proxy (siehe Abschnitt 8.4.4), Interceptor (siehe Abschnitt 8.5.3) oder Decorator (siehe Abschnitt 8.5.2) eingesetzt. Das *statische* Cross Cutting hingegen nimmt bereits bei der Kompilierung des Programmcodes die notwendigen Änderungen vor. Dabei werden die bestehenden Klassen um Attribute und Operationen erweitert, die für die Querschnittsfunktion benötigt werden.

Joinpoints

Joinpoints sind jene Punkte im Programmablauf, die es ermöglichen einen Aspekt einzuführen. In AOP können meist folgende Einsprungspunkte für Aspekte verwendet werden:

- Methodenaufruf,
- Ende einer Methode,
- Exception,

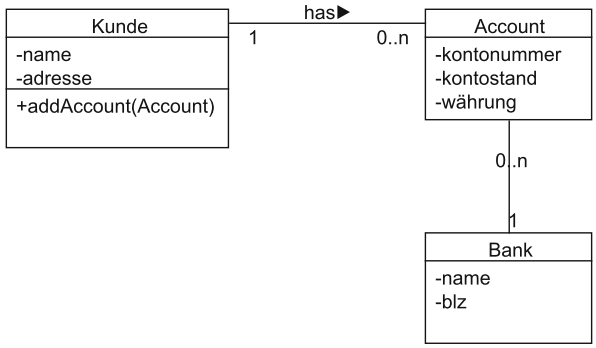


Abbildung 9.15
AOP-Beispiel-Domäne.

- Zugriff auf eine Variable,
- Initialisierung einer Klasse oder eines Objekts.

Muss eine Funktion beispielsweise innerhalb einer Transaktion ausgeführt werden so hat dies folgende Konsequenzen: (a) vor dem Aufruf der Methode muss eine Transaktion gestartet werden,(b) bei erfolgreicher Abarbeitung der Methode ist die Transaktion zu beenden (*commit*), (c) im Fehlerfall ist die Transaktion zurückzusetzen (*rollback*).

Pointcut

Unter einem *Pointcut* versteht man eine Gruppe von *Joinpoints* die bestimmte Eigenschaften (Muster) aufweisen. Die Methode `Bank.addAccount()` in der Klasse `Bank`, könnte mit einem Pointcut definiert werden. Dieser Pointcut fängt alle Aufrufe auf die Methode `addAcount` ab, die ein `Account`-Objekt als Parameter hat und keinen Rückgabewert liefert. Eine Implementierung in AspectJ[30] würde wie folgt umgesetzt werden:

```
1 pointcut addingAccount(Account account):
2   call(void addAccount(Account)) &&
3   args(account);
```

Es ist auch möglich *Wildcards* in einem Pointcut zu verwenden, z. B. `execution(* Account.*(..))`. Dieser Pointcut greift bei allen Operationen der Klasse `Account`, wobei diese in einem beliebigen Package sein können, egal ob diese einen Rückgabewert haben oder nicht. Die Punkte kennzeichnen eine beliebige Reihenfolge von Parametern einer Funktion. Neben den Wildcards können auch Operatoren verwendet werden, um mehrere Filterausdrücke miteinander zu kombinieren. Aufgrund der unterschiedlichen Ausprägungen von Pointcuts kann zwischen zwei Ar-

[30] `http://www.eclipse.org/aspectj/`

ten von Pointcuts unterschieden werden: *Primitive Pointcuts* sind Pointcuts ohne Namen, wie im obigen Beispiel, während *named Pointcuts* wiederverwendbar Pointcuts sind, die über einen eindeutigen Bezeichner verfügen.

Aspekt

In einem *Aspekt* wird die Querschnittsfunktionalität definiert und beinhaltet *Pointcuts* und *Advices* als Java-Methoden oder zusätzliche Feld-Deklarationen. Es könnte beispielsweise Logging-Funktionalität als Aspekt modelliert und in AspectJ implementiert werden:

```
1  public aspect LoggingAspect {
2  // pointcut definitions
3  pointcut addingAccount(Account account):
4    call(void addAccount(Account)) &&
5    args(account);
6
7  pointcut settersToLog():execution(void set*(double))
       ||
8    execution(void set*(String));
9
10 // an advice implementation
11 after(Account account) returning: addingAccount(
       account) {
12       Logger logger = Logger.getLogger("trace");
13       logger.info("New account added: " + account);
14     }
15
16 after() returning: settersToLog(){
17 // ... another advice implementation
18 }
19
20 // ... other pointcuts, advices or introductions
21 }
```

Advice

Der Advice ist die eigentliche Implementierung des Aspekts und wird demnach von einem Pointcut verwendet. Je nachdem, wo ein Advice zum Einsatz kommen soll, können verschiedene Arten unterschieden werden:

- *Before Advice* wird verwendet, bevor der Joinpoint ausgeführt wird. Diese Advices sind beispielsweise für Security-, Transaktions- oder Logging-Funktionen sehr gut geeignet.
- *After Advice* wird nach dem Joinpoint ausgeführt. Dabei kann ein After Advice so konfiguriert werden, dass dieser entweder immer, nur bei erfolgreicher Ausführung oder bei Auftreten einer Exception ausgeführt wird. Solche Advices eigenen sich gut für die Behandlung von Exceptions.
- *Around Advice* ist eine Kombination aus Before und After Advice in einem Advice. Dieser Advice übernimmt die Kontrolle darüber, wann die tatsächliche Operation aufgerufen wird.

Folgender Code stellt ein Beispiel für einen *After Advice* dar, der immer ausgeführt wird, wenn der `addingAccount(Account)` Pointcut zutrifft. Der Advice macht nichts anderes als eine Log-Ausgabe, nachdem die Methode `addAccount()` fertig ist:

```
after(Account account) returning: addingAccount(
    account) {
  Logger logger = Logger.getLogger("trace");
  logger.info("New account added: " + account);
}
```

Ein weiteres Beispiel für einen *After Advice* ist im nächsten Codeteil zu finden, in dem auch der Klassen- oder Methodenname extrahiert und das Caching in Anspruch genommen wird. Dieses Beispiel illustriert sehr gut, wie vielseitig und mächtig AOP ist. Auch der Zugriff auf Informationen der Klasse, Methoden und Parameter ist mit AOP möglich.

```
after() returning: settersToLog() {
  String className = thisJoinPoint.getSignature().
    getDeclaringTypeName();
  String methodName = thisJoinPoint.getSignature().
    getName();
  Object newValue = thisJoinPoint.getArgs()[0];

  Object oldValue = fieldCache.get(thisJoinPoint.
    getSignature());
  if (null == oldValue) {
    fieldCache.put(thisJoinPoint.getSignature(),
      newValue);
    oldValue = new String("N/A");
}
  Logger logger = Logger.getLogger("trace");
  logger.info(className + "." + methodName +
  "() called." + " Previous value: " + oldValue +
  ", new value: " + newValue);
}
```

Introduction

Die oben genannten Konzepte werden ausschließlich für das *dynamische* Cross Cutting benötigt, das in vielen Fällen ausreichend ist. Mithilfe von *Introductions* können Attribute und Operationen einer Klasse hinzugefügt werden. Introductions sind ein Vertreter des statischen AOP, da bei der Kompilierung dieser Aspekt zum Code hinzugefügt wird und auch tatsächlich im Bytecode enthalten ist.

9.5.3 Sicherheit mit AOP

Multi-User-Systeme

Anwendungen für Warenwirtschaft, Buchhaltung, Lagerhaltung oder andere Verwaltungssysteme werden in der Regel von *mehreren Usern* verwendet. Nicht selten werden unterschiedliche Arbeiten von ein und derselben Person durchgeführt. Um die Zugriffsrechte zu strukturieren, werden häufig *Rollen* eingeführt (zum Beispiel Buchhalter, Sachbearbeiter, Control-

ler). Jeder Benutzer des Systems wird dann einer oder mehreren Rollen zugewiesen. Jeder Rolle werden dann bestimmte Berechtigungen zugeteilt. Buchungen darf beispielsweise ein Buchhalter, nicht aber ein Sachbearbeiter durchführen. Dieser hat wiederrum Zugriff auf das Warenwirtschaftssystem. Der Controller könnte Zugriff auf mehrere Systeme haben.

In einem Universitätssystem könnte es Rollen wie Student, Professor, Verwaltung, Finanz usw. geben. Denn Studenten sollen zwar mit dem E-Learning-System interagieren und sich für Prüfungen anmelden können, aber nicht in der Lage sein Zeugnisse auszustellen. Der Zugriff auf Finanzsysteme wiederum sollten bestimmten Verwaltungs-Rollen vorbehalten bleiben.

Rollen und Rechte

Die Rollenzuweisung der einzelnen Mitarbeiter erfolgt meistens durch eigene Systeme, wie etwa ein *LDAP Directory*[31]. Wie AOP helfen kann, um solche Zugriffsrechte umzusetzen, soll in diesem Abschnitt gezeigt werden.

Beispiel

Die Sicherheitsanforderung, die in das Beispiel im ersten Schritt eingebaut werden soll, lautet, dass nur bestimmte Studenten und Professoren bestimmte Service-Aufrufe ausführen dürfen. Diese Zugriffsvorgaben sollen in einem eigenen System gepflegt werden können, und der bestehende Sourcecode soll dabei nicht verändert werden. Um diese Anforderungen mit AOP zu lösen, sind folgende Konstrukte notwendig:

- Das „Pflegesystem“ für die Sicherheit wird als einfache Property-Datei implementiert. In dieser Datei sind die User und die berechtigten Methodenaufrufe angeführt.
- Das Sicherheitsservice, das die Zugriffslogik implementiert und auf das Sicherheitspflegesystem Zugriff hat. Dieses Service erhält bei jedem Methodenaufruf *User* und *Methode* und kann somit auswerten, ob der User zugriffsberechtigt ist.
- Ein Sicherheitsadvice, der den Sicherheitsservice abfragt, ob der User die notwendige Berechtigung für die Ausführung des Services hat.
- Ein Pointcut, der sicherstellt, dass bei allen Methoden der Sicherheitsadvice ausgeführt wird.

Die Property-Datei für die Pflege der Sicherheit hat einen sehr einfachen Aufbau. Für jede User-Id werden die Methoden definiert, für die er berechtigt ist. Ist diese Methode hier nicht angeführt, so hat er keine Berechtigung.

```
1 0201212,getExamsOfCourse
2 0201213,getExamsOfCourse
```

[31] In einem LDAP Directory kann die Organisationsstruktur des Unternehmens abgebildet werden. Ein LDAP-System ist vereinfacht gesagt eine hierarchische Datenbank.

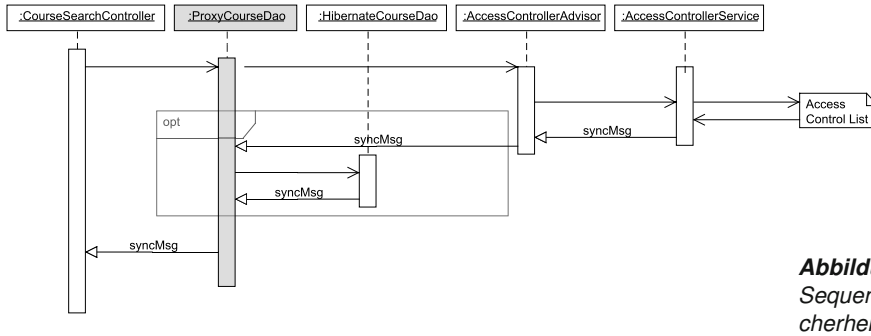


Abbildung 9.16
Sequenzdiagramm Sicherheitsaspekt.

Ablauf

Wird nun eine Programmfunktion vom User aufgerufen, so wird der Aufrufpfad entsprechend der durch die AOP definierten Regeln geleitet. Das Sequenzdiagramm in Abbildung 9.16 illustriert den Ablauf: Ein Student mit der User-Id 0201212 möchte sich gerne alle Prüfungen eines Kurses laden. Im GUI ist dafür ein eigener Menüpunkt vorgesehen, den der User betätigen kann. Bei Aufruf dieser Funktion wird (gesteuert vom AOP-Framework) anstatt des tatsächlichen Service zunächst nur ein *Proxy* aufgerufen, der zuerst den *Sicherheitsadvice* ausführt. Dieser Advice hat Zugriff auf den aktuellen Aufrufkontext und ist somit in der Lage, den aktuellen User und die aktuelle Methode zu ermitteln. Dieser Advice kontaktiert das `AccessControlService` und übergibt dem Service die User-Id und den Methodennamen. Das `AccessControlService` wiederum ist mit der Sicherheit-Pflegedatei verbunden und kann nun abfragen, ob der übergebene User die Methode ausführen darf. Bei positiver Prüfung führt der Advice einen Aufruf der gewünschten Methode durch, andernfalls wird der Aufruf nicht durchgeführt und eine Fehlermeldung generiert.

Im nächsten Schritt könnte man dann das Sicherheitsservice um ein Rollen-Modell erweitern, um sich nicht mit den Rechten einzelner Benutzer herumschlagen zu müssen, sondern diese, wie oben beschrieben, in Gruppen zusammenfassen zu können. An den Prinzipien der AOP-Implementation ändert dies nichts.

Keine Änderung der Business-Logik

Diese Anforderungen konnten ohne Änderungen in der Business-Logik umgesetzt werden. Daraus ergibt sich eine saubere Trennung zweier unterschiedlicher Aspekte einer Anwendung (Separation of Concerns, siehe Abschnitt 7.4). Diese Teile der Anwendung sind besser voneinander entkoppelt und die Entwickler der Business-Logik müssen sich nicht mit Details der Zugriffskontrolle auseinandersetzen. Der Security-Experte kann eine Zugriffskontrolle implementieren, ohne die Geschäftslogik im Detail verstehen zu müssen.

9.5.4 Deklaratives Transaktionsmanagement mit AOP

Beispiel

Ein weiterer Bereich, in dem AOP sinnvoll eingesetzt werden kann, ist das Transaktionsmanagement, das bereits in Abschnitt 9.4.2 ausführlich beschrieben wurde. Transaktionen kommen vor allem in den DAOs und Service-Komponenten einer Anwendung zum Einsatz. Das folgende Listing zeigt ein Beispiel, wie SQL-Updates innerhalb einer Transaktion in Java und JDBC abgebildet werden können:

```
1  Connection con = DriverManager.getConnection(url,
       user, password);
2  // Auto-Commit deaktivieren
3  con.setAutoCommit(false);
4  Statement stmt = con.createStatement();
5  try {
6    stmt.executeUpdate(sqlUpdate1);
7    stmt.executeUpdate(sqlUpdate2);
8    // Änderungen in die DB schreiben
9    con.commit();
10 } catch(SQLException e) {
11   // Bei Fehler Änderungen zurücksetzen
12   con.rollback();
13 }
```

Zunächst wird ein `Connection`-Objekt vom `DriverManager` geladen, um eine Verbindung zur Datenbank aufzubauen. `AutoCommit` wird deaktiviert, um die manuelle Transaktionskontrolle und mehrere Statements in einer Transaktion zu ermöglichen. Nun werden die Statements für die Updates aufbereitet und über das `Statement`-Objekt ausgeführt. Das `Commit` beendet die Transaktion. Tritt allerdings während der Verarbeitung ein Fehler auf, werden alle Statements mit `rollback` zurückgesetzt und der ursprüngliche Zustand wiederhergestellt. Dieses kurze Beispiel hat verdeutlicht, welche Schritte ein Entwickler bei den Service-Komponenten oder DAOs durchführen muss, damit diese transaktionsfähig werden:

- Eine Transaktion muss geöffnet werden.
- Nach der Öffnung müssen die Statements definiert werden, die innerhalb der Transaktion ausgeführt werden sollen.
- Wenn keine Fehler aufgetreten sind, ist ein Commit durchzuführen.
- Wenn ein Fehler auftritt, ist ein Rollback auszulösen.

Redundanter Code

Diese Aktionen müssen bei allen Methoden, die transaktionsfähig sein sollen, angeführt werden. Die immer sehr ähnliche Transaktionslogik verteilt sich somit über viele Klassen der Anwendung. Dies ist lästig in der Implementation und führt dazu, dass der Code für die Transaktion sehr oft kopiert wird und dadurch fehleranfällig ist.

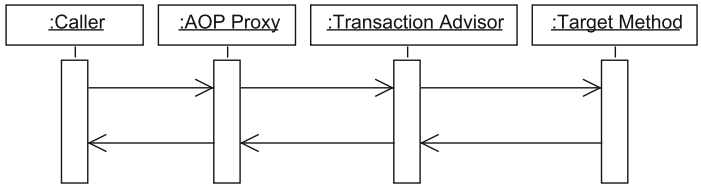


Abbildung 9.17
Sequenzdiagramm: Deklaratives Transaktionsmanagement mit einem Proxy.

Transaktionslogik mit AOP

Mit AOP kann die Transaktionslogik zentral an einer Stelle implementiert und gepflegt werden. Diese Logik kann anschließend dynamisch an Methode geknüpft werden. Die Methoden, die in einer Transaktion ausgeführt werden sollen, müssen auch nicht verändert werden. Dies könnte beispielsweise mit einem Proxy (siehe Abschnitt 8.4.4) und einem *Transaction Advisor* gelöst werden. Diese Variante ist in Abbildung 9.17 als Sequenzdiagramm dargestellt.

Beispiel mit AOP

Als Beispiel soll die Speicherung von Kursdaten, die innerhalb einer Transaktion geschrieben werden sollen, dienen. Für das Speichern wird ein DAO zur Verfügung gestellt. Der Aufrufer kommuniziert allerdings mit einem Proxy anstelle des tatsächlichen DAO-Objekts. Der Proxy verwendet einen Transaction Advisor, der die Steuerung der Transaktion übernimmt. Beim Aufruf wird eine Transaktion geöffnet. Dann wird die tatsächliche Methode aufgerufen. Bei erfolgreicher Ausführung der Methode führt der Transaction Advisor ein Commit, andernfalls ein Rollback durch.

Konfiguration

Folgendes Codebeispiel demonstriert, wie *deklaratives Transaktionsmanagement* mit dem Spring-Framework in XML konfiguriert werden kann.

```
1  <bean id="courseDao" class="at...HibernateCourseDao"
       />
2
3  <!-- Transactional Advice, verwendet
       Transaktionsmanager -->
4  <tx:advice id="txAdvice" transaction-manager="
       txManager">
5    <tx:attributes>
6    <!-- Alle GET Methoden sollen einen Read-only-
         Zugriff auf
7          die DB haben -->
8    <tx:method name="get*" read-only="true"/>
9    <!-- Alle anderen Methoden verwenden die Standard-
         Settings -->
10   <tx:method name="*"/>
11   </tx:attributes>
12 </tx:advice>
13
14 <aop:config>
15   <aop:pointcut
16     id="daoOperations"
17     expression="execution(* at....dao.*.*(..))"
18   />
```

```
19    <aop:advisor
20      advice-ref="txAdvice"
21      pointcut-ref="daoOperations"
22    />
23  </aop:config>
```

Ablauf Zunächst wird ein normales DAO Bean konfiguriert. Anschließend wird der *Transactional Advice* konfiguriert, der einen Transaktionsmanager verwendet. Zusätzlich wird noch definiert, dass alle `GET`-Methoden einen `READ-ONLY`-Zugriff auf die Datenbank haben. Die anderen Methoden werden in einer Standard-Transaktion ausgeführt. Über AOP erfolgt nun die Verknüpfung zwischen den DAO-Objekten und dem Transactional Advice. Für diesen Zweck wird ein Pointcut definiert, der besagt, dass der zugewiesene Transactional Advice bei allen DAO-Objekten, die sich im Package `at...dao` befinden angewendet werden soll. Weder der DAO-Code, noch der Code der aufrufenden Klassen wurde dabei modifiziert! Zur Laufzeit erstellt Spring einen Proxy, der vom Aufrufer verwendet wird. Der Aufrufer merkt jedoch nicht, dass er mit dem Proxy kommuniziert.

Fazit Die beiden Beispiele *Sicherheit mit AOP* und *deklaratives Transaktionsmanagement* haben deutlich gemacht, welche neuen Wege die aspektorientierte Programmierung für die komponentenorientierte Entwicklung öffnet. Durch die Auslagerung von Querschnittsfunktionen in Aspekte wird der Code der Geschäftslogik schlanker und der Grad der Wiederverwendbarkeit steigt. Aspekte können dynamisch, und was vor allem wichtig ist, ohne Anpassung des bestehenden Codes den Komponenten hinzugefügt werden.

9.6 Benutzerschnittstellen in komponentenbasierten Systemen

Die Interaktion zwischen Menschen und Software-Systemen erfolgt üblicherweise über eine grafische Oberfläche (GUI – Graphical User Interface). Die GUIs sind mit einer Vielzahl an GUI-Elementen, wie Eingabefeldern, Tabellen, Buttons und vielen anderen Komponenten ausgestattet, die dem Benutzer entweder Informationen liefern oder vom Benutzer Daten entgegennehmen. Auch die korrekte Formatierung von Daten (zum Beispiel Datum, Zahlen), Validierung oder Mehrsprachigkeit zählt zu den Aufgaben des GUI. Die nächsten Abschnitte liefern einen kurzen Überblick über unterschiedliche GUI-Modelle (siehe Abbildung 9.18), wie die Interaktion von GUI-Komponenten prinzipiell aufgebaut ist und wie letztendlich mit dem Rest der Applikation kommuniziert wird.

9.6.1 Überblick

Command Line Im Zeitalter von *Command-Line*-Applikationen waren die GUI-Entwickler sehr eingeschränkt und das GUI hat sich auf das Command Line Interface

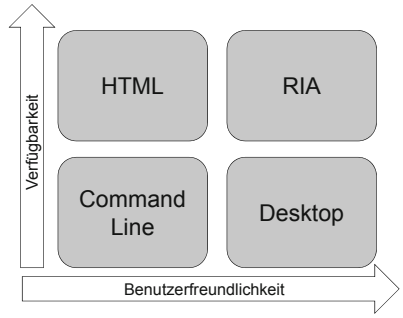


Abbildung 9.18
GUI-Ansätze aus Sicht des Endanwenders.

Fat Cients

beschränkt. Der Benutzer konnte nur über die Tastatur mit dem System kommunizieren. In den 1980er und 1990er Jahren, als Betriebssysteme mit grafischem Interface und Maus-Bedienung entstanden, und vor allem nachdem diese standardisierte APIs für die Benutzerschnittstellen entwickelten kam Bewegung in die Entwicklung von GUI-Anwendungen. Bestehende Kommandozeilen- oder textbasierte Applikationen wurden durch grafische *Desktop*-Applikationen ersetzt. Viele Desktop-Applikationen wurden so konzipiert, dass diese vollständig am Client-Rechner installiert werden mussten. Daher wird diese Architektur auch als *Fat Client* bezeichnet, da sowohl die gesamte Programmlogik als auch das GUI am Client installiert sind. Nach und nach wurde bei vielen Anwendungen Logik aus Fat-Clients gelöst und auf einen Server verlagert. Der Client wurde dadurch schlanker und beinhaltete hauptsächlich die Logik für das GUI. Diese Art von Clients werden als *Thin Client* bezeichnet. Bei dieser Form wird sehr häufig die Drei-Schichten-Architektur (siehe 7.5.2) angewandt.

Thin Client

Web-Applikationen

Mit der steigenden Popularität des Internets ging der Trend seit den 1990er Jahren in Richtung Web-Applikationen, für die Oberflächen mittels Browser-Technologien wie HTML und Javascript erstellt werden. Diese Art der Anwendungen zeichnen sich dadurch aus, dass der Browser als Client verwendet werden kann und der Client keinerlei Installation benötigt. Bei heutigen Software-Anwendungen wird häufig gefordert, dass sie im Web laufen aber die Bedienbarkeit (*Usability*) von Client-Anwendungen bieten: Der aktuelle Trend geht also in Richtung *Rich-Internet-Applikationen* (RIA). Im Grundprinzip handelt es sich dabei um Web-Anwendungen, die jedoch den Komfort einer Desktop-Anwendung haben. Gerade in diesem Bereich haben sich in den letzten Jahren einige interessante Technologien[32] am Markt entwickelt. In Abbildung 9.18 sind die unterschiedlichen Arten von GUI-Anwendungen einander gegenübergestellt.

Rich-Internet-Applikation

Mobile UI

Auch mobile Endgeräte wie Mobiltelefone oder PDAs sind aus dem heutigen Leben von Geschäftsleuten, aber auch im privaten Bereich kaum noch

[32] AJAX, ULC, sind Beispiele für RIA

wegzudenken. Die zugrunde liegenden Betriebssysteme erlauben es auch, benutzerdefinierte Programme, wie Java oder andere, auf dem Handy zu betreiben. Mobile UIs werden in Zukunft eine sehr wichtige Rolle spielen und viele Software-Systeme bieten bereits heute schon *Smart Clients* für das mobile Endgerät an. Die Anforderungen an mobile Oberflächen sind durchaus anspruchsvoll, da der Entwickler mit wenig Platz auf den kleinen Displays auskommen und auch die Bedienung einfach gestaltet werden muss. Bekannte Beispiele für mobile Plattformen sind das iPhone[33] oder Google Android[34], aber auch die auf Java ME[35] basierten Systeme.

9.6.2 Model View Controller

In Kapitel 8 wurden bereits einige Patterns erklärt, die hauptsächlich in Komponenten der unteren Schichten (Daten-Schicht, logische Schicht) verwendet werden. Auch für die GUI-Entwicklung haben sich Patterns etabliert: Das *Model View Controller* Pattern soll in diesem Abschnitt kurz vorgestellt werden.

Problemstellung

Bei der Entwicklung von komplexen GUI-Elementen besteht die Gefahr, Geschäftslogik mit GUI-Logik zu vermischen. Dadurch wird aber die Wiederverwendbarkeit und Wartbarkeit von GUIs und Geschäftslogik erheblich erschwert.

Lösung

Es ist also eine klare Trennung zwischen der Darstellung des GUI, der Logik im GUI und den benötigten Daten wünschenswert. Dieser Zusammenhang ist in Abbildung 9.19 dargestellt. Die durchgezogenen Linien in der Grafik stellen eine direkte Verbindung zwischen den Komponenten dar, während die gestrichelten Linien eine indirekte Kommunikation (sehr oft in Form eines Beobachters, siehe Abschnitt 8.5.1) andeuten.

Modell

Das Modell sind die Daten, die im GUI verarbeitet werden sollen. Im Normalfall handelt es sich hier um die Domänenobjekte, wie zum Beispiel die Daten eines Kunden oder einer Bestellung. Die Objekte enthalten keine Geschäftslogik und keinesfalls GUI-Logik, sondern dienen als reiner Daten-Container. Üblicherweise werden diese Objekte auch in der Logik-Schicht verwendet und über die DAOs aus der Daten-Schicht geladen.

View

Die *View* ist die grafische Repräsentation der Daten. Hier kommen GUI-Elemente wie Eingabefelder, Tabellen, Dropdowns oder ähnliche zum Einsatz. In der View wird auch die Darstellung der einzelnen Elemente konfiguriert (Farbe der Felder, Schriftgröße, Icons etc.). Hier kann auch ein Grafiker involviert sein.

[33] `http://www.apple.com/de/iphone/`
[34] `http://code.google.com/intl/de-DE/android/`
[35] `http://java.sun.com/javame`

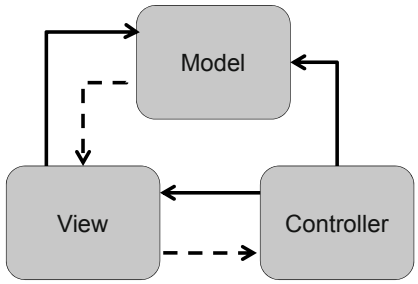


Abbildung 9.19
MVC-Pattern.

Controller

Das Bindeglied zwischen dem Modell und der View ist der *Controller*. Er nimmt Interaktionen vom Anwender entgegen und delegiert in den meisten Fällen an Service-Komponenten in der logischen Schicht weiter, die die tatsächliche Logik ausführen. Änderungen am Modell wiederum reicht der Controller an die *View* weiter.

Implementation

Eine eigene Implementation des MVC Patterns ist in den meisten Fällen nicht erforderlich, da moderne UI-Frameworks, wie beispielsweise Swing, JSF, Wicket oder Android bereits über entsprechende APIs verfügen. Es ist daher wesentlich, sich mit der jeweils eingesetzten Technologie vertraut zu machen und die dort üblichen Best-Practices anzuwenden.

9.6.3 Kommunikation mit dem Backend

Die Präsentationsschicht beinhaltet alle Komponenten, die für die Darstellung der Informationen und die Interaktion mit dem User verantwortlich sind. Die Geschäftslogik wird von entsprechenden Service-Komponenten in der Logik-Schicht implementiert. Wird die Anwendung als Fat Client ausgeliefert, ist die Kommunikation zwischen dem GUI und den Service-Komponenten relativ einfach, da sowohl die GUI-Komponenten als auch die Servicekomponenten im gleichen Anwendungskontext betrieben werden.

Verteilte Anwendungen

Etwas anspruchsvoller wird es bei Thin Clients, bei denen eine Fernkommunikation zwischen dem Client und dem Server stattfinden muss. Denn wie in Abschnitt 7.5.2 beschrieben wurde, können Teile der Logik vom Client auf den Server ausgelagert werden. Nun muss ein Weg gefunden werden, wie der Client mit dem Server kommuniziert. In Abschnitt 8.6 wurden bereits verschiedene Möglichkeiten sowie Muster der Integrationen beschrieben. Ein Großteil dieser Konzepte kann genau für diese Problemstellung herangezogen werden.

Festlegen von Schnittstellen

Unabhängig vom gewählten Ansatz ist es wichtig, dass die Service-Komponenten in der Logik-Schicht über klare Schnittstellen verfügen, denn diese Schnittstellen werden vom GUI verwendet. Zu bedenken ist allerdings, dass es für die GUI-Komponenten bei Verwendung von Schichtenarchitektur oft nicht ersichtlich ist, ob ein Zugriff auf eine Service-Komponente

lokal aufgelöst wird oder über das Netzwerk zu einem Server geht. Dies kann aber in der Performance erhebliche Unterschiede machen und sollte bei der Architektur bedacht werden.

Caching

Um diesem Problem entgegenzuwirken ist bei verteilten Anwendungen ein *clientseitiges Caching* in vielen Fällen unumgänglich. Betrachtet man z. B. ein klassisches ERP-System, so stellt man fest, dass ein Großteil der Daten in vielen Benutzerinteraktionen immer wieder verwendet wird. Gute Beispiele dafür sind Länderkennzeichen, PLZ, Orte, Anreden oder andere Stammdaten im System. Solche Daten ändern sich aber relativ selten. Genau an diesem Punkt kann das GUI profitieren, indem es derartige Daten in einem Cache hält. Man sollte sich bereits beim Design überlegen, welche Lebensdauer bestimmte Daten haben, um in entsprechenden Intervallen beim Server nachzufragen, ob Aktualisierungen vorhanden sind. Manche Cache-Implementationen erlauben auch Benachrichtigung der Clients, wenn sich bestimmte Daten am Server ändern. Caching kann auch sehr elegant mit der aspektorientierten Programmierung (siehe Abschnitt 9.5) umgesetzt werden.

Weitere Punkte

Weitere Punkte, die beim Entwurf und der Kommunikation mit dem Backend berücksichtigt werden müssen:

- *Security:* Der Client ist in vielen Fällen außerhalb der unmittelbaren Kontrolle der Entwickler und Administratoren (z. B. bei Web-Anwendungen). Daher muss prinzipiell damit gerechnet werden, dass versucht wird, den Client oder die Kommunikation zwischen Client und Server zu manipulieren.
- *Validierung am Server:* Aus vorigem Punkt folgt unter anderem, dass Daten, die vom Benutzer eingegeben werden, *immer* am Server validiert werden müssen. Dies gilt unabhängig von einer Validierung auf Client-Seite!
- *Validierung am Client:* Aus Usability-Gründen wird häufig auch am Client validiert (z. B. um dem Benutzer ein schnelles Feedback geben zu können). Wie hoch ist aber der Validierungsgrad auf Client-Seite? Je mehr die Daten am Client gegen Regeln validiert werden, desto geringer ist der Verkehr zwischen Client und Server. Doch je mehr Validierung am Client stattfindet, desto komplexer wird auch die Logik des Clients[36].
- *Exception-Management:* Wer ist für das Exception-Management zuständig? Wie weit werden systemnahe Fehler an die Oberfläche getra-

[36] Für manche UI-Technologien gibt es Frameworks, die es erlauben, client- und serverseitige Regeln an nur einer Stelle zu definieren. Der entsprechende client- und serverseitige Code wird dann vom Framework generiert. Ist dies möglich, reduziert dies natürlich den Implementations- und Wartungsaufwand deutlich.

gen? Man muss sich bewusst sein, dass ein Benutzer mit einer technischen Fehlermeldung nichts anfängt.

> *Request Processing:* Soll der Anwender blockiert werden, während ein Request vom Server verarbeitet wird, oder nicht?

9.7 Lose Koppelung von Systemen

In Abschnitt 8.6 wurde bereits Einiges zum Thema Integration und Kopplung von Systemen gesagt. Serviceorientierte Architekturen sind ein Architekturparadigma, das in Enterprise-Architekturen häufig eingesetzt wird. In Abschnitt 7.6 wurden bereits die Konzepte und Grundlagen einer serviceorientierten Architektur aus der Sicht der *Architektur* erklärt. In diesem Abschnitt wird nun auf die einzelnen Bestandteile einer SOA näher eingegangen und aufgezeigt, wie eine SOA zur losen Koppelung von Systemen beiträgt. Darüber hinaus wird das Zusammenspiel zwischen *SOA* und *Middleware* beschrieben. Die erklärten Konzepte und Muster der Abschnitte 7.6 und 8.6 sind für diese beiden Abschnitte wichtig.

9.7.1 Serviceorientierte Architekturen und deren Bestandteile

Enterprise-Architektur

Eine SOA legt ihren Fokus auf die IT-Infrastruktur eines Unternehmens. Dieses Paradigma wird also nicht auf eine einzelne Applikation alleine angewendet, sondern es ist viel mehr eine Strategie, wie die im Unternehmen befindlichen Systeme organisiert und eingesetzt werden.

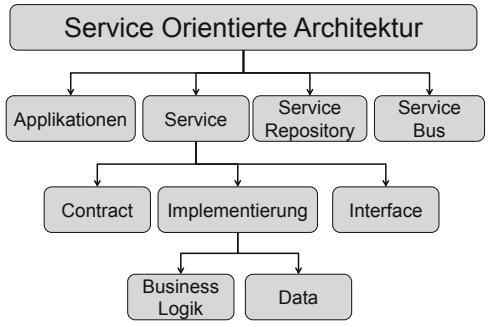


Abbildung 9.20
Komponenten einer serviceorientierten Architektur. [60]

Aus Sicht der Komponenten besteht eine SOA aus mehreren Bestandteilen, die in Abbildung 9.20 dargestellt sind.

Applikationen

Applikationen in einer SOA steuern Services und interagieren mit ihnen. Applikationen können sowohl Business User als auch Drittsysteme sein.

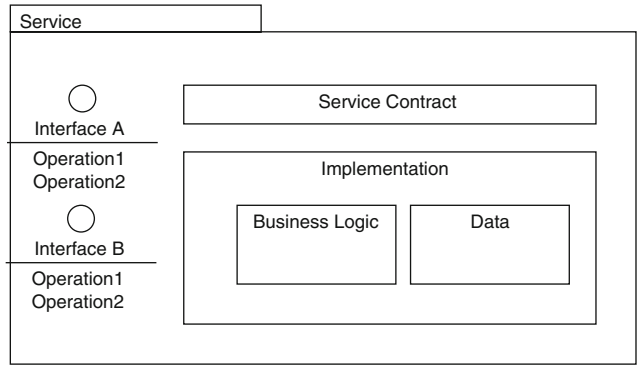


Abbildung 9.21 *Teile eines Services in einer SOA. [60]*

Business User interagieren mit Systemen über zur Verfügung gestellte Anwendungen (meist Web-, Rich- oder mobile Clients). Doch nicht nur Business User, sondern auch andere Systeme, wie Prozesse, Batch-Jobs etc. können Services in Anspruch nehmen.

Service

Kernelement in einer SOA ist das *Service*, das eine bestimmte Art Business-Funktionalität beinhaltet und für die Außenwelt abstrahiert (siehe auch Abschnitt 9.1). Ein Service setzt sich technisch betrachtet aus mehreren Teilen zusammen, die in Abbildung 9.21 dargestellt sind. Der *Service Contract* ist die abstrakte Beschreibung eines Service. Er beschreibt, wie das Service zu verwenden ist. In vielen Fällen ist es eine formale Beschreibung des Interfaces in Sprachen wie WSDL oder IDL. Entscheidend ist, dass der Service Contract eine plattformneutrale Beschreibung des Services ist.

Service Contract

Interface

Das *Interface* ist die funktionale Beschreibung des Services und in den meisten Fällen Teil des Service Contracts. Das Interface beschreibt die zu erwartenden Parameter und Rückgabewert eines Services.

Implementierung

Die Implementierung ist die plattformspezifische Umsetzung des Services zum Beispiel in Java oder .NET. Die Implementierung beinhaltet meist sowohl Business-Logik als auch die Datenzugriffslogik. Ein Großteil aller Services benötigt Zugriff auf Daten, die in verschiedener Form vorliegen.

Service-Repository und Registry

In einem *Service-Repository* werden Services registriert, die von unterschiedlichen Service-Anbietern zur Verfügung gestellt werden. Diese Services können anschließend vom Service-Konsumenten gesucht und verwendet werden. Ein Service-Repository ist ein wesentlicher Bestandteil einer SOA. Ein Service-Repository wird sehr oft in Kombination mit einer *Service-Registry* genutzt. Diese beiden Komponenten werden manchmal synonym verwendet. Es gibt jedoch eine klare Unterscheidung zwischen diesen beiden Konzepten. Eine Service-Registry beinhaltet Meta-Informationen der Services, wie etwa unterschiedliche Arten von Service Level Agreements, Sicherheit und Adressen von Services. Ein Service-Repository hingegen beinhaltet die Services selbst und kann somit auch ohne eine Service-Registry betrieben werden. Service-Registries sind auch sehr wich-

tig, um über einen längeren Zeitraum hinweg den Überblick über vorhandene Services nicht zu verlieren.

Service Bus

Ein Service Bus verbindet alle Teilnehmer in einer SOA. In vielen Fällen kommt an dieser Stelle ein *Enterprise Service Bus* zum Einsatz, der in Abschnitt 9.7.2 näher beschrieben wird.

Web-Services

Wie anfangs erwähnt, kann eine SOA auf unterschiedliche Art und Weise implementiert werden. Eine in der Praxis sehr häufig eingesetzte Variante eine SOA zu realisieren sind Web-Services (siehe Abschnitt 8.6.1). Im Web-Service-Umfeld haben sich in den letzten Jahren viele Standards entwickelt, die sich sehr stark an die Anforderungen an eine SOA angelehnt haben. In Abbildung 9.22 ist der SOA Stack aufgeführt und die dazu passenden Web-Service-Standards.

Transport und Encoding

In der *Transportschicht* wird die eigentliche Übertragung der Daten vorgenommen. Diese Übertragung kann wahlweise mittels HTTP, SMTP oder FTP durchgeführt werden. In vielen Fällen wird das HTTP-Protokoll für den Transport verwendet.

Messaging

Die *Message-Schicht* beschreibt, welches Datenformat für den Austausch der Services verwendet werden soll. Die Nachrichten können in Form von XML-RPC, SOAP oder REST übermittelt werden (siehe Abschnitt 8.6.1).

Description

Die *Beschreibung* eines Services ist in einer neutralen Sprache zu halten, damit dieses in verschiedenen Plattformen verwendet werden kann. Bei Web-Services hat sich die *Web Service Definition Language (WSDL)* als Standard durchgesetzt.

Discovery

In der *Discovery*-Schicht geht es darum, die Services sowohl abzulegen als auch zu suchen. Hier kommt die zuvor beschriebene Service-Registry zum Einsatz. Im Web-Service-Bereich gibt es hierfür den Standard *Universal Description, Discovery and Integration (UDDI)*.

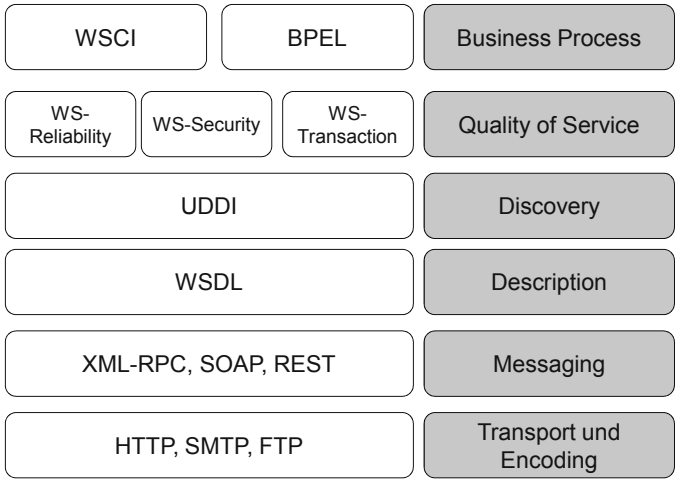


***Abbildung 9.22** Eine SOA mit Web-Services umgesetzt.*

QoS

Quality-of-Service-Aspekte wie Sicherheit, Transaktionsmanagement oder die zuverlässige Zustellung von Nachrichten werden durch mehrere Web-Service-Standards beschrieben, wie im WS-Security-, WS-Reliablity- oder WS-Transaction-Standard.

Business Processes

In der letzten Schicht werden die Geschäftsprozesse abgebildet, wie in den Abschnitten 7.6.3 und 9.7.1 beschrieben. Für die Abbildung der Prozesse wird häufig die *Business Process Execution Language (BPEL)* verwendet, die den Prozess in einer ausführbaren Sprache beschreibt. Prozess-Engines können diesen Prozess interpretieren und ausführen.

SOA- und BPM-Plattformen

Für die Umsetzung von SOA in Unternehmen werden häufig SOA- oder BPM-Plattformen eingesetzt. Eine BPM-Plattform ist für mehrere Rollen in einem Unternehmen ausgelegt, wie etwa Management, Business-Analysten, Integrationsarchitekten und Service-Entwickler. Der Business-Analyst ist der Experte für Prozesse. Er ist in der Lage, mit dem BPM-Werkzeug den Geschäftsprozess grafisch abzubilden. Für die Abbildung von Prozessen existieren unterschiedliche Notationen, wie die *Business Process Management Notation (BPMN)*.

Sichten auf den Prozess

Die grafische Abbildung kann als Grundlage für die IT verwendet werden, die den Prozess mit entsprechenden Tools mit technische Details anreichert. Zum Beispiel gibt es im Prozess Aktivitäten, für die ein Dialog für eine User-Interaktion angezeigt werden muss, dann gibt es Aktivitäten wie Web-Service-Aufrufe, Datenbankzugriffe etc. Die IT erhält damit die technische Sicht auf den Prozess. Es wird somit das gleiche Modell für beide Bereiche verwendet, und es gehen keine Informationen verloren.

9.7.2 Middleware als Lösung für die lose Koppelung

Problemstellung

Heutige Software-Systeme sind mit einer derartigen Komplexität behaftet, dass Themen wie Architektur, Wartung und Flexibilität eine wesentliche Rolle spielen. Die Nutzung von Software-Lösungen über Firmengrenzen hinweg wird durch die Verbreitung von serviceorientierten Architekturen auch immer wichtiger. Dabei nimmt die Integration der IT-Systeme eine fundamentale Rolle ein.

Projekte „auf der grünen Wiese“?

Wenn neue Software-Systeme entstehen, so beginnt dies allerdings in den seltensten Fällen „auf der grünen Wiese“. Vielmehr möchten Unternehmen viele bestehende Applikation weiterhin nutzen und auf Basis der bestehenden Systeme neue Applikationen erstellen. Früher wurde oftmals für jedes anzuknüpfende System ein neuer Adapter entwickelt. Da viele Adapter speziell für ein System entwickelt wurden, stellte sich die Wartung dieser Adapter als sehr aufwendig heraus. Außerdem wird die Kommunikation zwischen vielen Systemen sehr aufwendig, wenn jedes System über spezielle Adapter anzusprechen ist. Man hat also versucht, Alternativen zu finden und somit haben sich in den frühen 1990er Jahren einige interessante Ansätze herauskristallisiert.

Extract, Transform, Load

ETL-Techniken (Extract, Transform and Load) wie FTP-Dateitransfer oder Nightly Build Jobs sind nach wie vor eine beliebte Art der Integration. Nicht selten können diese Lösungen jedoch zu inkonsistenten Daten zwischen den einzelnen Systemen führen. Tritt beispielsweise ein Fehler bei der Synchronisation auf, muss der Job gegebenenfalls neu gestartet werden. *Batch Jobs* werden sehr oft in der Nacht durchgeführt, was natürlich auch eine zeitliche Einschränkung für deren Ausführung mit sich bringt. Bei einer Umgebung mit 24/7-Verfügbarkeit ist diese Technik nicht mehr einsetzbar.

Enterprise Application Integration

EAI Broker bzw. *Hub and Spoke* Integration Brokers haben sich Mitte der 1990er Jahre in der Industrie einen Namen gemacht. Durch deren Hub- and Spoke-Ansatz erfolgt ein zentrales Routing zwischen Applikationen. Man ist somit der Integration und auch der losen Koppelung einen Schritt näher gekommen. Für die regionale Integration reichten EAI Broker aus, doch die Einbindung von externen Partnersystemen war mit einem sehr hohen Aufwand verbunden.

ESB und MOM

Enterprise-Service-Bus-Technologien (ESB) versuchen eine standardisierte Integrationsplattform zur Verfügung zu stellen, damit Applikationen über diese Plattform miteinander verbunden werden können, und bauen dabei auf anderen etablierten Technologien auf. In Abschnitt 8.6 wurden bereits einige Ansätze der Integration dargestellt, wie z. B. Messaging. Eine Message Oriented Middleware (MOM)dient auch beim ESB als Basis für die Kommunikation (siehe Abbildung 9.23). Über diese Middleware werden die Nachrichten zwischen den Applikationen ausgetauscht. Die in Abschnitt 8.6 angeführten Konzepte wie Messaging, Message Channel, Routing usw. sind die Grundlage für eine MOM. Der Bus selbst verwendet ein standardisiertes Format der Nachrichten, meist XML (es können aber auch Objekte sein, wie es zum Beispiel bei Mule[37] der Fall ist). Viele ESB-Hersteller setzen jedoch auf XML, da für diesen Standard viele Werkzeuge vorhanden sind und diese für Schema-Definitionen (XSD), Stylesheets (XLST), XPath- oder XQuery-Abfragen eingesetzt werden können.

Der ESB bildet nun eine Schicht über der MOM, damit sich die Entwickler mit dieser Schicht nur mehr in Spezialfällen beschäftigen müssen. Betrachten wir in den nächsten Absätzen wichtige Basisfunktionen eines ESB.

Router

Router sind ein wesentlicher Bestandteil einer ESB-Infrastruktur. Message Router werden verwendet, um empfangene Nachrichten an einen oder mehrere Empfänger weiterzuleiten. Dieses Routing kann durch eine Logik beeinflusst werden. Soll die Nachricht z. B. abhängig vom Inhalt an eine bestimmte Applikation weitergeleitet werden, spricht man dabei von *Content Based Routing*. Weitere Informationen zu Routern finden sich in Abschnitt 8.6.4.

[37] `www.mulesource.org`

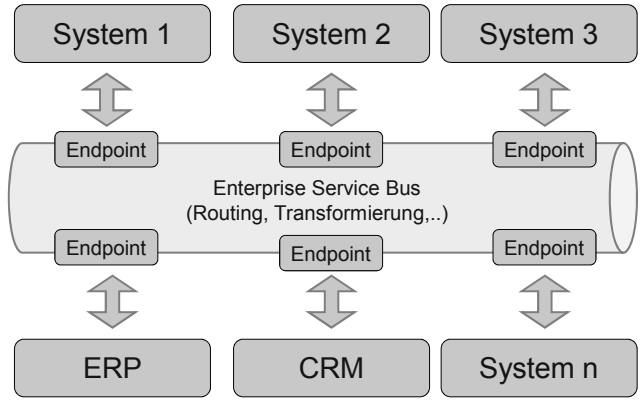


Abbildung 9.23
Enterprise-Service-Bus für die lose Koppelung der Systeme.

Filter

Ein *Filter* ist eine spezielle Ausprägung eines Routers. Er stellt sicher, dass Komponenten auch nur jene Nachrichten erhalten, die tatsächlich von Interesse sind.

Splitter und Aggregator

Um Nachrichten in mehrere Teile zu splitten, kann ein *Message Splitter* verwendet werden. Als Beispiel könnte eine Rechnung in Kopf- und Detaildaten aufgeteilt werden. Diese beiden Teile der Nachricht werden an unterschiedliche Komponenten weitergeleitet. Natürlich muss auch die Möglichkeit bestehen, einzelne Teile wieder zusammenzuführen. Dies geschieht mit einem *Message Aggregator*.

Transformierung

Die *Transformierung* von Nachrichten zählt zu einer wichtigen Fähigkeit des ESB. Häufig unterscheidet sich das Datenformat zwischen den einzelnen Applikationen, obwohl die Bedeutung der Daten die gleiche ist. Ein Transformer stellt sicher, dass die erhaltene Nachricht von der Anwendung verarbeitet werden kann. Zum Beispiel verarbeitet Komponente A die Nachricht in Java-Objekten, während Komponente B die Nachricht in XML benötigt. Auch die Einführung eines generischen Datenformats kann sinnvoll sein. Hier ist der Integrationsarchitekt gefragt. Seine Aufgabe ist es, mit den Verantwortlichen der einzelnen Applikationen zu klären, welche Formate die Applikationen benötigen. Auf Basis dieser Informationen kann ein generisches Datenformat erstellt werden.

Protokolle

Neben der eigentlichen Datentransformation, die in den meisten Fällen vom Anwendungsentwickler gemacht werden muss, ist die *Protokolltransformation* ein entscheidender Aspekt bei einem ESB. Durch die Protokolltransformation wird das Format des Fremdsystems in das Standardformat des ESB konvertiert. Anwendungsentwickler müssen sich daher nur noch in Spezialfällen mit dem Protokoll des Fremdsystems beschäftigen. Wenn man die Produkte der ESB-Hersteller betrachtet, unterstützen die Anbieter ein breite Palette an Protokollen, wie JMS, Web-Services, POP3, SOAP, SMTP und zum Teil auch sehr spezielle Protokolle proprietärer Systeme.

Endpoints

Endpoints (*binding components*) stellen die Verbindung zwischen einer Applikation und einem Kommunikations-Kanal (*channel*) dar. Alle ESB-

Produkte stellen dafür eine API zur Verfügung, die von den zu integrierenden Systemen verwendet werden muss, sofern diese nicht schon über standardisierte Protokolle ansprechbar sind (beispielsweise Web-Services, JMS oder andere Protokolle die angeboten werden). Der Endpoint abstrahiert also den Zugriff auf das Fremdsystem.

Fazit

Ein Enterprise Service Bus öffnet dem Entwickler neue Möglichkeiten der Integration und trägt wesentlich zur losen Koppelung von Systemen bei. Die Entscheidung für einen solchen Ansatz sollte jedoch trotzdem gut überlegt sein, da eine solche Infrastruktur die Komplexität eines Software-Projekts deutlich erhöhen kann.

9.8 Logging: Protokollieren von Systemzuständen

Systemzustände protokollieren

Logging ist eine Technik, die in fast allen Software-Projekten Einsatz findet und die ganz besonders bei Server-Software von großer Bedeutung ist. Während der Laufzeit eines Programms oder einer Gruppe von Programmen kann es erforderlich sein, bestimmte Systemzustände nicht (nur) dem Benutzer mitzuteilen, sondern so zu protokollieren, dass sie einem Administrator oder Entwickler zugänglich sind. Dies betrifft z. B. Fehlerzustände, in die die Anwendung gerät, oder bestimmte Aktivitäten von Benutzern, die für einen Administrator von Bedeutung sein können. Auch Entwickler geben fallweise gerne Debug-Information (z. B. den Inhalt einer Variable) auf der Konsole aus, um die Fehlersuche oder Entwicklung der Software zu erleichtern. Oftmals ist dies ein schnellerer Weg, die korrekte oder fehlerhafte Durchführung eines bestimmten Codeteils zu bestimmen, als die Verwendung eines Debuggers.

System.out.println

Die Verwendung von `System.out.println(...)` (SOP) und andere ähnliche „handgestrickte“ Varianten empfehlen sich allerdings in der Praxis meist nicht: Zunächst einmal ist SOP unflexibel, denn es kann nur auf der Konsole ausgegeben werden. Ein späteres Umleiten der Debug-Meldungen, z. B. in eine Log-Datei oder in eine Datenbank, ist nicht leicht möglich. Zudem kann nicht zwischen unterschiedlichen Zuständen unterschieden werden: Ein Systemfehler wird in derselben Weise ausgegeben wie ein Debug-Statement, das nur fallweise während der Entwicklung benötigt wird. Das Problem verschärft sich, wenn die Software ausgeliefert wird, da man in der Regel die Konsole des Benutzers nicht mit Debug-Statements fluten möchte.

Komponentenorientierte Systeme

Es empfiehlt sich daher schon bei „einfachen“ Anwendungen, die Protokollierung von Systemzuständen mit geeigneten Logging-Frameworks zu erledigen. Besonders wichtig wird es aber bei komplexeren Systemen, die aus mehreren (verteilten) Komponenten bestehen. Für derartige meist serverbasierte Anwendungen (oder solche, die gar keine Konsole haben) ist der oben beschriebene Mechanismus sowieso ungeeignet, weil Adminis-

tratoren oder Entwickler beim laufenden System keinen Zugang zu diesen Protokollen haben. Dazu kommt noch das Problem (vor allem bei Serveranwendungen), dass auf einem Rechner meist eine Vielzahl an Anwendungen und Services läuft, und es kaum wünschenswert ist, dass alle diese Anwendungen über eigene Logging-Mechanismen verfügen. Die Protokolle (*Logs*) erfüllen besonders bei komponentenorientierten Systemen wichtige Funktionen in der Fehlersuche sowie in der Beurteilung des Laufzeitverhaltens der Software.

Anforderungen an Logging-Mechanismen

Für einen Logging-Mechanismus kann man daher einige Anforderungen definieren:

- Logging soll einfach und „unaufdringlich“ sein.
- Logging sollte auf mehreren Ebenen (Log-Level) möglich sein.[38]
- Es sollte möglich sein, je nach Software-Komponente zu definieren, welches Log-Level ausgegeben wird.
- Das Logging-Framework sollte über eine zentrale Konfiguration verfügen.
- Das Ziel des Loggings sollte zentral einstellbar sein, z. B. Konsole, Log-Datei, Datenbank usw.
- Das Ausführen von Log-Statements sollte so wenig Performance wie irgend möglich benötigen und das Programm nicht nennenswert langsamer machen.

Beispiel

In Java gibt es bereits im JDK eine Logging-API, bzw. viele Projekte verwenden die Apache-`log4j`-Bibliothek[39]. In den Beispielen zum Buch wird die log4j-Bibliothek verwendet. Das Integrieren von Logging in ein eigenes Projekt ist grundsätzlich sehr einfach: Zunächst muss die Logging-Konfiguration erstellt werden.

Logger-Konfiguration

Bei log4j kann dies mit der `log4j.properties`-Datei erfolgen, die z.B. folgende Einträge enthält:

```
1 log4j.rootLogger = DEBUG, stdout
2 log4j.logger.org.apache = WARN, stdout
3 log4j.logger.org.springframework = ERROR, stdout
4
5 log4j.appender.stdout = org.apache.log4j.
    ConsoleAppender
```

[38] Bei Log-Levels wird meist mit Hierarchien gearbeitet, z. B. FATAL < ERROR < WARNING < INFO < DEBUG. Das bedeutet: ist der Log-Level auf FATAL gesetzt, werden nur FATAL Meldungen erfasst, alle anderen unterdrückt. Ist der Loglevel auf INFO gesetzt, werden alle Ebenen außer DEBUG ausgegeben usw.

[39] `http://logging.apache.org/log4j/`

```
6 log4j.appender.stdout.layout = org.apache.log4j.
      PatternLayout
7 log4j.appender.stdout.layout.ConversionPattern = %d
      [%t] %-5p %c - %m%n
```

In Zeilen 1–3 werden unterschiedliche Log-Levels für unterschiedliche Komponenten definiert. Dies ist sehr nützlich, weil man z. B. für die eigene Komponenten die `DEBUG`-Statements sehen möchte, aber eigentlich an den `DEBUG`-Meldungen des Spring-Frameworks oder der Apache-Bibliotheken nicht interessiert ist. Daher werden für diese Pakete die Log-Level auf `WARN` oder `ERROR` gesetzt. In Zeile 5–7 wird definiert, in welcher Form die konkrete Ausgabe zu erfolgen hat, sowie wohin protokolliert werden soll (in diesem Beispiel auf die Konsole). Andere Logging-Frameworks (z. B. Linux syslog, OS X Konsole, Windows Fehlerkonsole usw.) sind unterschiedlich zu konfigurieren, die Grundkonzepte sind aber bei fast allen dieselben.

Logging-Statements

In der Anwendung wird dann für jede Klasse eine Instanz des Loggers geholt (Zeile 1) und an beliebiger Stelle (Zeile 4 und 8) können dann Log-Informationen erfasst werden:

```
1 private static Logger log = Logger.getLogger(Basis.
      class);
2
3 public static void main (string args[]) {
4   log.info("Starting Application");
5   MainFrame mf = new MainFrame();
6   ...
7   mf.setVisible(true);
8   log.info("Terminating Start Class");
9 }
```

Log-Ausgabe

In diesem Beispiel wird auf `INFO`-Level geloggt. Wird die Anwendung gestartet, so erhält man etwa folgende Ausgabe auf der Konsole:

```
1 2009-01-27 20:11:25,243 [main] INFO  at...Basis -
      Starting Application
2 ...
3 ...
4 2009-01-27 20:11:28,112 [main] INFO  at....Basis -
      Terminating Start Class
```

Es wird der genaue Zeitpunkt, die Klasse der Log-Level sowie das Log-Statement ausgegeben. In komplexeren Server-Umgebungen können die Log-Statements verschiedener Anwendungen (z. B. Servlets) zusammengefasst werden und mit entsprechenden Werkzeugen in den Log-Datenbanken oder Log-Dateien gesucht werden.

Aspekt-orientierte Entwicklung

Das Logging kann auch sehr elegant mit der aspektorientierte Programmierung umgesetzt werden. Mit AOP kann man in einigen Fällen den Nachteil umgehen, dass der operative Code mit Logging-Statements „verunreinigt“ wird, d. h., dass zwei verschiedene Aspekte (z. B. Geschäftslogik/Logging)

vermischt werden. AOP und Logging mit AOP wird in Abschnitt 9.5 im Detail erklärt.

9.9 Zusammenfassung

Eine gründliche Analyse des eigenen Problems, der Umgebung, in der die Software laufen soll, sowie anderer nicht funktionaler Aspekte wie Performance oder Ausfallsicherheit sind sehr wichtig. Wesentlich ist für die meisten Projekte auch eine saubere komponentenorientierte Architektur, die es erlaubt die verschiedenen Aspekte der Anwendung gut voneinander zu trennen und damit auch wartbar zu halten. Schwerpunkt in diesem Kapitel waren Techniken, die für die komponentenorientierte Entwicklung verwendet werden können. Zu Beginn wurde ein Vorgehensmodell vorgestellt, das die notwendigen Schritte beschrieb, wie man Schritt für Schritt vom Objekt zum Service gelangt. Dabei wurde bewusst gezeigt, dass jeder Schritt in Richtung Service, die Wiederverwendbarkeit und lose Koppelung, aber auch gleichzeitig die Komplexität erhöht. Komponenten und Software-Frameworks unterstützen Entwickler bei der Umsetzung von komponentenorientierten Architekturen und geben dem Entwickler auch Richtlinien und Strukturen vor. Software-Frameworks sind generisch aufgebaut, damit diese in unterschiedlichen Kontexten eingesetzt werden können. Sie stellen bereits eine Basisfunktion zur Verfügung, die an die eigenen Bedürfnisse angepasst werden kann.

Vom Objekt zum Service

Persistenz

Kaum ein Software-Sytsem kommt heute ohne Persistenz aus, und die Abbildung der Persistenzlogik sollte soweit es geht von der Geschäftslogik losgelöst sein. Die Suche nach einem dem Problem angepassten Persistenzmechanismus ist nicht immer einfach. Abschnitt 9.4 lieferte einen Einblick in die Vielfalt der Möglichkeiten sowie die grundsätzlichen der Persistenz. Eines erscheint klar: Es gibt keine „one-size-fits-all“-Lösung, also einen Standard-Mechanismus, auch wenn dies Vertreter mancher Frameworks fallweise gerne behaupten. Wiederverwendbarkeit von Peristenzmodulen ist häufig ebenfalls ein Kriterium.

Für die Abbildung der Objekte auf die Tabellen muss es auch nicht immer das komplexeste Framework sein. O/R-Mapping-Werkzeuge beispielsweise bieten sehr viele Möglichkeiten, haben aber auch ein erhebliches Maß an Komplexität. Hier muss man abwägen, ob nicht ein einfacheres Framework das Problem bei weniger Aufwand ebenso lösen kann.

In manchen Projekten muss es auch keine relationale Datenbank sein. Objektorientierte Systeme können sehr leistungsfähig und elegant in der Implementierung sein, auch XML-Datenbanken oder Dateisystem-Index-Mechanismen können fallweise eine gute Wahl darstellen.

AOP

Wiederkehrende Funktionen im System sollen nicht immer wieder neu implementiert und quer über den gesamten Sourcecode verteilt sein. Diesem

Problem tritt die aspektorientierte Programmierung (AOP) entgegen, indem sie Querschnittsfunktionen, sogenannte Cross Cutting Concerns in einen Aspekt auslagert. Diese Aspekte können an beliebigen Programmstellen statisch oder dynamisch zugewiesen werden, ohne den ursprünglichen Code zu verändern. Als Beispiel wurden etwa Logging, Sicherheit oder das deklarative Transaktionsmanagement vorgestellt.

Benutzerschnittstelle

Die Komponentenorientierung endet nicht am Server, sondern zieht sich durch bis in die grafische Oberfläche. Der GUI wird oft nicht die notwendige Beachtung geschenkt, und die Implementierungen von Oberflächen sind kaum noch wartbar. Auch in der Oberfläche können die Konzepte der Komponentenorientierung eingesetzt werden. Auch wie die Kommunikation mit der logischen Schicht funktioniert, muss genau analysiert werden. Abschnitt 9.6 hat einige Strategien zu diesem Thema vorgestellt.

SOA

Schließlich wurde noch auf die Umsetzung von serviceorientierten Architekturen und was beim Entwurf solcher Systeme beachtet werden muss eingegangen. Bei SOA hat man einen sehr hohen Entkoppelungsgrad zwischen den Komponenten. Entsprechend komplex sind auch die einzusetzenden Techniken und Werkzeuge, die bei einer SOA ins Spiel kommen. Sehr oft wird in Verbindung mit einer SOA eine Message Oriented Middleware eingesetzt, die die Integration der unterschiedlichen Systeme behandelt. Eine solche Middleware trägt wesentlich zu der losen Koppelung der einzelnen Systeme bei.

Middleware

10 | Techniken und Werkzeuge

Software-Entwicklung erfordert den Einsatz von unterschiedlichen Werkzeugen und Methoden, die die verschiedenen Stufen der Software-Entwicklung unterstützen. Im technischen Bereich ist es üblich, eine ausgereifte Entwicklungsumgebung (*Integrated Development Environment*, IDE) mit umfangreichen Funktionen zu verwenden und darüber hinaus Werkzeuge für Versionierung (*Sourcecode Management*, SCM), Issue Tracking, Build Management und Test Reporting sowie Kollaborationswerkzeuge wie Wikis einzusetzen. Diese Werkzeuge unterstützen das strukturierte Arbeiten in einem Team.

In diesem Kapitel werden daher wesentliche konzeptionelle Aspekte sowie Werkzeuge und Praktiken vorgestellt, die den Software-Entwicklungsprozess wesentlich unterstützen, und deren gute Kenntnis heute zum Standard-Repertoire jedes Software-Entwicklers gehören.

Übersicht

10.1 Konvention oder Konfiguration?

Freiheit!

Programmierer sind häufig verspielte Menschen, die gerne an allen möglichen Schrauben, die ein Gerät besitzt, herumdrehen möchten. Neugierde und Spieltrieb sind oft nützliche Eigenschaften, wenn man innovative Anwendungen programmieren bzw. das Letzte aus einem System herauskitzeln möchte. Daher war es lange Zeit üblich, dass Entwicklungssysteme dem Entwickler möglichst wenig Vorgaben machen, wie bestimmte Aufgaben zu erledigen sind. Um beim Beispiel Java zu bleiben: das Basis-JDK von Sun macht eigentlich sehr wenige Annahmen, wie der Entwickler damit umgeht. Es gibt beispielsweise keine Vorgaben, in welchen Verzeichnissen Sourcecode zu liegen hat und wie Konfigurationsfiles zu behandeln sind. Auch Persistenzframeworks wie Hibernate lassen dem Entwickler breiten Spielraum, wie Objekte und Datenbanktabellen zu benennen sind, in welchen Verzeichnissen welche Konfigurationsdaten zu liegen haben usw.

Freiheit?

Diese Freiheitsgrade scheinen durchaus im Sinne des Entwickler(team)s zu sein, denn Freiheit kann ja per se nicht schlecht sein? Spätestens seit Mitte der 1990er Jahre stellt man als Entwickler aber fest, dass schon Standardprojekte über so viele Freiheitsgrade (Schrauben) verfügen, dass man wirklich Mühe hat, deren Funktionen und Auswirkungen auf das ganze Projekt zu verstehen und überall vernünftige Entscheidungen zu treffen. Es wird immer aufwendiger, nur die *Basis-Infrastruktur* für ein Projekt stabil und sauber einzurichten, um dann erst nennenswerte Funktionalität programmieren zu können. David Heinemeier Hansson, einer der Gründer von Ruby on Rails, beschreibt dieses Problem recht pointiert wie folgt:

Ruby on Rails

> „Flexibility is not free; it is a trade-off from something else. [...] How can more flexibility by a bad thing? People like choices a lot better than actually having to choose. This is the paradox of choice“ *David Heinemeier Hansson [37]*

Convention over Configuration...

Man muss also feststellen, dass Freiheit immer etwas kostet, und es ist die Frage, wo Freiheit in einem Software-Projekt sinnvoll ist, und wo sie mehr Zeit kostet und Probleme verursacht als sie Vorteile bringt. Aus diesen Überlegungen entstand das Prinzip *Convention over Configuration*, eine der Kernideen von Ruby on Rails, aber bei Weitem nicht nur dort zu finden. Im Java-Umfeld könnte man Apache Maven nennen, wo diese Idee ganz elementar umgesetzt wurde[1]. Worum geht es nun konkret?

In einem Software-Projekt gibt es zunächst einmal einige Entscheidungen, die man treffen sollte, um ein einheitliches und stabiles Projekt zu bekom-

[1] Maven ist zwar ein kein Web-Framework wie Rails, die *Convention-over-Configuration*-Idee steht aber bei Maven konzeptionell im Zentrum.

men. Für die eigentliche Entwicklungsaufgabe ist es aber irrelevant, wie diese Entscheidungen ausfallen, sie sollten nur konsistent sein. Ein paar Beispiele:

- Wie sieht die Ordnerstruktur des Projekts aus? Ist der Sourcecode im Basisverzeichnis oder im Verzeichnis `src` oder im Verzeichnis `source` oder ...?
- Wohin kopiert man Ressourcen, die ein Programm benötigt (z. B. Konfigurationsdateien, Grafiken, etc.)?
- Wohin packt man Tests? In dieselben Verzeichnisse wie den Programmcode? In ein Test Verzeichnis ...?
- Wie benennt man Klassen, Interfaces, Packages? Soll dies jeder Entwickler nach eigenem Geschmack entscheiden?
- Wie geht man mit Abhängigkeiten (*dependencies*) um? Wohin kopiert man Bibliotheken, wie benennt man sie?
- Wie wird Programmcode dokumentiert?
- Wie wird Logging durchgeführt?
- Wie wird der Build automatisiert?
- Wie wird Persistenz umgesetzt?

Diese und weitere Entscheidungen sollten getroffen werden, und *Convention over Configuration* geht den Weg, dass entweder von Seiten der Sprachentwickler (Ruby on Rails, Java Coding Conventions) oder von Werkzeugen (Java, Maven) bestimmte vernünftige Vorgaben gemacht werden oder dass sich mit der Zeit bestimmte Vorgehensweise etabliert haben (z. B. die Verzeichnisstruktur in Java-Projekten), an die sich der Entwickler halten sollte.

... auf mehreren Ebenen

Betrachtet man die Situation aber etwas genaue, stellt man fest, dass es in der *Convention-over-Configuration*-Philosophie mindestens zwei Ebenen gibt, und hier scheiden sich eventuell auch die Geister. Die erste Ebene ist heute ziemlich unbestritten und könnte so interpretiert werden: *Alle Entscheidungen, die zwar getroffen werden müssen, bei denen es aber ziemlich irrelevant ist, wie sie konkret getroffen werden, sollten einmal erledigt werden (Convention). Dieser Konvention sollten in Zukunft alle folgen, wenn es nicht sehr gute Gründe gibt sie zu brechen.* Dies betrifft die oberen Beispiele der Liste. Es ist z. B. egal, wie man das Verzeichnis nennt, in das der Sourcecode kopiert wird, aber es ist dennoch sehr vernünftig, dass die Dateistruktur verschiedener Projekte in dieser Hinsicht möglichst identisch ist. Denn dies hat klare Vorteile:

- Der erfahrene Entwickler muss sich diese Dinge nicht bei jedem Projekt neu überlegen

- Unerfahrene Entwickler neigen dazu, bei solchen Entscheidungen Fehler zu machen, die sich dann später im Projekt nachteilig auswirken.
- Entwickler-Werkzeuge sind leichter aufeinander abzustimmen, sie liegen also auf der sicheren Seite, den Konventionen zu folgen.
- Neue Entwickler finden sich in Projekten leichter zurecht, weil sie sich nicht erst überlegen müssen, wo sie die Ressourcen und die Unit Tests suchen sollen bzw. in welcher Reihenfolge welche Shellscripts zu starten sind, um den Build zu erledigen.

Dazu kommen Werkzeuge für Entwickler, die für häufig vorkommende Anwendungsfälle Code generieren, z. B. *Scaffolding* in Rails, wo sich Änderungen in der Datenbank werkzeugunterstützt im Sourcecode nachziehen lassen. Ein gutes Beispiel sind auch die Archetypen (das sind im Wesentlichen parametrisierbare Projektvorlagen) in Maven, die im Detail in Abschnitt 10.3.3 beschrieben werden. In Ruby on Rails bietet Rake vergleichbare Funktionalität).

Abweichungen von der Konvention

Bis zu dieser Stelle herrscht wenig Konflikt. Betrachtet man aber die unteren Punkte der ersten Liste (z. B. Logging, Build-Automatisierung, Persistenz) so stellt man fest, dass diese Entscheidungen dann doch nicht mehr so leicht für alle Projekte über einen Kamm zu scheren sind. Build-Automatisierung und Persistenz-Technologie können stark von persönlichen Vorlieben oder Projektnotwendigkeiten bestimmt sein. Bei Rails geht es noch deutlich weiter; um ein Beispiel zu geben: bei Rails ist sogar die Benennung von Domänenobjekten und Tabellen in der *Pluralization Convention* definiert: Heißt das Domänenobjekt z. B. „Employee“ so wird die Tabelle „Employees“, bei einem Domänenobjekt „Child“ wird die Tabelle „Children“ (sic!) genannt. Dies kann aber (s.u.) per Konfiguration deaktiviert werden.

> „Pick the choices when you care! Here is our choice, change it when you don’t like it.“ *David Heinemeier Hansson [37]*

Dies ist wieder das Prinzip von Ruby on Rails. Dort gibt es für alle wesentlichen Aspekte inklusive Persistenz, Build-Automatisierung und Logging Standardvorgaben, mit denen sich viele Entwickler auch gut zurecht finden. Ist man aber der Ansicht, dass z. B. das *ActiveRecord* Pattern als Standard-Persistenztechnik von Rails nicht ins Projekt passt, so tauscht man diese durch eine andere aus.

Und in Java?

In der Java-Welt finden sich ebenfalls seit Längerem ähnliche Überlegungen, aber dort gibt es derartig viele verschiedene Optionen, dass man sich hier nicht guten Gewissens auf ein paar Basistechnologien einigen konnte. Es gibt aber durchaus mit Apache Maven ein Werkzeug, mit dem man sich je nach Projektart entsprechende Archetypen erstellen kann, mit denen

man Basis-Projekte leicht erzeugen lassen kann. Die Maven-Archetypen werden mittlerweile von vielen anderen Projekten (z. B. Apache Wicket oder Cocoon) verwendet, um es deren „Kunden“ zu ermöglichen, schnell Basis-Projekte in der jeweiligen Technologie zu erstellen.

Dennoch, es bleibt die Frage, wie weit man vernünftigerweise die Standardisierung treiben sollte. An dieser Stelle wird es wohl bei grundsätzlicher Übereinstimmung auch in Zukunft unterschiedliche Ansichten geben.

10.2 Sourcecode-Management

10.2.1 Die Entwicklung von Sourcecode-Management-Systemen

Abgleich bei Teamarbeit

Die Verwaltung von Sourcecode sowie anderen Artefakten (Dokumentation, Diagramme etc.) ist eine Herausforderung, ganz besonders, wenn im Team gearbeitet wird. Mehrere Entwickler sollen parallel am selben Projekt arbeiten und der aktuelle Stand möglichst einfach synchronisiert werden können. Manuelle Aktivitäten, wie das Verteilen von Sourcecode per E-mail, FTP oder dergleichen führen mit Sicherheit in ein Chaos. Es empfiehlt sich daher von Projektstart an, ein Sourcecode-Management- (SCM) System einzuführen, das den Sourcecode sowie auch andere Artefakte (wie Dokumentation oder Binaries)[2] die für das Projekt benötigt werde, verwaltet. Auch bei kleinen und Individualprojekten ist der Einsatz von SCM-Systemen empfehlenswert. Schon wegen der sauberen Versionierung oder der parallelen Entwicklungslinien, die im Laufe der Projekte notwendig sind.

Der einsame Entwickler?

Geschichte

SCM-Systeme gibt es schon seit den 1980er Jahren in verschiedener Ausprägung. Eines der ersten Systeme war das Revision-Control-System (RCS), das noch auf Basis einzelner Dateien gearbeitet hat und auch nicht wirklich für verteilte Teamarbeit geeignet war. Später entwickelte sich daraus das Concurrent-Version-System (CVS), ein System mit zentralem (serverbasiertem) Repository. Das bedeutet, der Sourcecode wird zentral an einer Stelle verwaltet, und alle Entwickler gleichen ihre lokalen Arbeitskopien in regelmäßigen Abständen mit dem Server ab. Eines der heute am weitesten verbreiteten und modernsten (Open-Source)-Systeme, das serverbasiert arbeitet ist Subversion (SVN)[3]. Neben Systemen mit zentralisiertem

[2] Dies können Design-Vorlagen, Bilder oder Audio-Dateien für ein Spiel oder Icons für ein Programm sein. Derartige Binärdateien sollten durchaus mit dem Sourcecode mitverwaltet werden. Dateien, die sich (aus dem Sourcecode) generieren lassen, sollten aber keinesfalls mitversioniert werden, also z. B. jars oder PDFs, sowie alles, was sich (wenn man Maven verwendet) im `target`-Verzeichnis findet. Derartige Daten blähen das Repository auf und verursachen unnötige Netzwerk-Last.

[3] `http://subversion.tigris.org`

Ansatz werden seit wenigen Jahren immer häufiger verteilte Systeme wie GIT[4] oder Mercurial[5] eingesetzt. Viele Beobachter sprechen dabei von der dritten Generation der SCM-Systeme, die für die meisten Einsatzbereiche die zentralisierten Systeme ablösen könnten.

10.2.2 Versionierung

Transparenz im Entwicklungsprozess

Eine der wesentlichsten Funktionen eines SCM-Systems ist natürlich die Versionierung. Dadurch ist es jederzeit möglich, auf ältere Versionen des Sourcecodes zurückzugreifen sowie die Entwicklungsgeschichte einzelner Codeteile nachzuvollziehen. Dies kann z. B. sehr nützlich sein, wenn man Dinge „zerstört“ oder gelöscht hat und wieder auf einen früheren stabilen Zustand der Software zurückgreifen möchte oder auch, um zu verstehen, welche Änderungen an der Codebasis einen Fehler eingeführt haben. Die Versionierung kann zudem auch sehr nützlich sein, um Änderungen, die in bestimmten Codeteilen erfolgten, transparent zu machen. Möchte man beispielsweise wissen, welche Änderungen ein bestimmter Programmierer eingeführt hat, vergleicht man einfach die entsprechenden Versionen mit dem SCM-Werkzeug. SCM-Systeme sind auf Versionierung von Text-Dateien spezialisiert und optimieren hier auch den Platzbedarf, indem sie zumeist intern nur Änderungen speichern. Viele Systeme können allerdings auch mit Binärdaten umgehen.

Commit: eine neue Version

Unter einem *commit* versteht man das Erstellen einer neuen Version im Repository. Das bedeutet konkret, dass Änderungen, die seit dem letzten Update erfolgt sind, in das Repository geschrieben werden. Damit wird das Repository wieder auf denselben Stand gebracht wie die Arbeitskopie des Entwicklers. Der Commit ist mit einer *Commit-Message* also einer Beschreibung zu versehen, damit nachvollziehbar ist, was bei dieser Änderung (auch *Changeset* genannt) geändert wurde. Durch den Commit legt das SCM-System meist auch eine neue Versionsnummer oder -kennung (Revision) des Projekts an, um ältere Versionen eindeutig kenntlich zu machen.

Bei modernen Systemen erwartet man sich weiterhin, dass Commits *atomar* erfolgen, also vergleichbar einer Transaktion in Datenbanken. Es müssen entweder alle Änderungen ins Repository übertragen werden oder bei Problemen der ursprüngliche Zustand wiederhergestellt werden. Bei älteren Systemen war dies nicht immer der Fall, was zu sehr schwierig zu bereinigenden, inkonsistenten Zuständen führen konnte.

Markieren von Zuständen

Tagging und *Branching* sind zwei weitere wesentliche Fähigkeiten, die heutige SCM-Systeme bieten. Unter Tagging versteht man im Wesentlichen

[4] `http://git.or.cz`
[5] `http://mercurial.selenic.com`

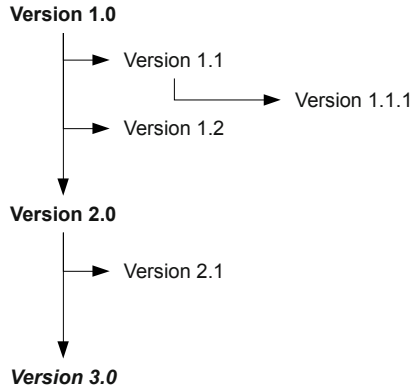


Abbildung 10.1
Beispiel für Branching: Mehrere Versionen müssen gewartet werden.

das Markieren eines bestimmten Zustands des Systems, indem man diesem eine Versionsnummer oder einen Namen zuweist. Hat man mit der eigenen Software z. B. einen Reifegrad erreicht, dass man den aktuellen Zustand als Version 1.0 freigeben möchte, so „tagged" man den aktuellen Zustand des Sourcecodes eben mit dieser Versionsnummer. Damit kann man später immer wieder genau den Zustand der Version 1 rekonstruieren.

10.2.3 Parallele Entwicklungspfade

Entwicklung in parallelen Strängen

Branching geht hier noch einen Schritt weiter und erlaubt das Entwickeln in parallelen Strängen. Beispielsweise können ältere Versionen einer Software parallel zu neueren Versionen weiterentwickelt werden oder spezielle Anpassungen für einzelne Kunden erforderlich sein. Für diese Zwecke kann man Verzweigungen, Varianten (*Branches*) in der Versionsgeschichte einführen. Auch komplexe neue Features werden häufig in eigenen Branches entwickelt, um den Hauptstrang der Entwicklung nicht zu stören.

Branching Beispiel

Ein konkretes Beispiel (siehe auch Abbildung 10.1): In der Praxis kommt es sehr häufig vor, dass man eine bestimmte Version freigibt, z. B. Version 1.0., Kunden kaufen oder laden diese Version aus dem Internet und verwenden sie auf ihren Systemen. Nun entwickelt man weiter und vertreibt Version 2.0, später 2.1. Nun kann aber der Zustand eintreten, dass nicht alle Kunden immer sofort auf die neueste Version updaten, z. B. bei Betriebssystemen, Webservern usw. Dies kann verschiedene Gründe haben: Entweder wollen sie das Update (noch) nicht kaufen, oder sie haben eigene Software, die von der älteren Version abhängig ist, und können nicht sofort nachziehen. Die Folge ist also, dass „in der freien Wildbahn" mehrere Versionen der eigenen Software im Einsatz sind. Dies ist nicht problematisch, solange man alte Versionen nicht warten möchte. In vielen Fällen ist es aber so, dass man nicht nur die neueste, sondern auch ältere Versionen warten muss. Wenn einige Kunden nach wie vor Version 1.0 verwenden, andere schon Version 2, und man z. B. ein Security-Problem in Version 1.0 findet, so muss es möglich sein, eine Version 1.1 herauszubringen (und im

SCM-System zu verwalten) ohne die neueren Versionen (z. B. 2.0, 2.1) damit zu beeinflussen. Damit hat man einen *Branch* erstellt, denn Version 1.1 ist kein direkter Vorgänger mehr von 2.0, sondern bei 1.0 gehen nun zwei „Äste“ (Branches) weg, einer zu 1.1 und ein anderer zu 2.0.

10.2.4 Unterstützung kollaborativer Arbeit

Strategien bei Konflikten

In der Teamarbeit kann es natürlich vorkommen, dass mehrere Personen an ein und derselben Datei arbeiten, und es damit zu Konflikten kommt. Die Lösung dieses Problems ist eine weitere zentrale Funktion eines SCM-Systems. Hier haben sich im Wesentlichen zwei Mechanismen etabliert: Die eine Variante ist es, Konflikte möglichst zu vermeiden. Diese Systeme arbeiten meist mit *Locking*-Mechanismen. Der andere Weg ist es potenzielle Konflikte zunächst zuzulassen, aber das Lösen der Konflikte – das *Merging* – zu unterstützen. Bei Systemen, die mit Locking arbeiten, muss der Entwickler die Dateien, die er ändern möchte, zunächst „auschecken“ oder mit einem bestimmten Kommando sperren. Damit ist für andere erkennbar, dass jemand an diesen Dokumenten zur Zeit arbeitet. Nach der Änderung müssen die Dokumente wieder freigegeben werden.

Sperren: Locking

Merging

In der Praxis der Software-Entwicklung, besonders wenn man mit textbasierten Dokumenten arbeitet, wo verschiedene Versionen zweier Dokumente relativ einfach zusammengeführt werden können[6], hat sich der Locking-Mechanismus kaum durchgesetzt. Systeme wie SVN arbeiten so, dass der Entwickler vor der Arbeit ein *Update* durchführt, dabei wird der lokale Stand auf den aktuellen Stand des Servers gebracht. Dann wird gearbeitet und nach bestimmten Änderungen – jedenfalls aber noch am selben Tag – wird wieder ein Update durchgeführt. Dabei passieren zwei Dinge: (1) Alle Dateien die der Entwickler *nicht* verändert hat, die aber von *anderen* verändert wurden werden wieder lokal aktualisiert (hier gibt es ja keinen Konflikt). (2) Sollte es aber Dateien geben die der Entwickler selbst sowie andere geändert haben, zeigt das System den Konflikt an und versucht nach Möglichkeit auch Vorschläge für die Behebung zu geben: Sind die Änderungen z. B. an verschiedenen Teilen der Datei (z. B. in unterschiedlichen Methoden) erfolgt, so versucht das System die verschiedenen Versionen zusammenzuführen. Handelt es sich um textbasierte Daten, so machen die Systeme meist automatisch Vorschläge, wie zusammengeführt

[6] Hier empfiehlt es sich aber, einige Details zu beachten: Man sollte darauf achten, dass bestimmte Konventionen von allen Entwicklern eingehalten werden, z. B. dass man sich einigt, ob Einrückungen mit Tabulator oder Spaces gemacht werden. Andernfalls kann dies die unangenehme Konsequenz haben, dass ein Entwickler mit Tabs arbeitet, ein anderer das Dokument öffnet und dessen Editor alle Tabs auf Spaces ändert: Das Versionssystem erkennt nun scheinbare Änderungen oder Konflikte in jeder Zeile! Derartige Konventionen kann man auch automatisiert beim Commit prüfen lassen.

werden kann, bei Binär-Dateien werden beide Versionen im Verzeichnis zur Verfügung gestellt, und der Konflikt muss manuell behoben werden[7]. Auch im Fall von Text-Dateien wird der Entwickler natürlich den Konflikt, respektive die automatische Zusammenführung prüfen und gegebenenfalls manuell korrigieren. Danach sind die Tests durchzuführen. Wenn diese korrekt durchlaufen sind, kann die fertige Version mit einem Commit auf den Server gespielt werden.

Merging bei verteilten Systemen

Bei verteilten Systemen wird häufig so vorgegangen, dass vor Änderungen an der Software das Haupt-Repository (das sich z. B. auf einem Server oder auf dem Rechner des Projektleiters befindet) geklont wird, d. h., es wird eine lokale Kopie erstellt. Dann wird auf der lokalen Kopie des Repositories gearbeitet und auch das Commit geht auf diese lokale Version. Am Ende eines bestimmten Entwicklungsschrittes können dann die Versionen wieder zusammengeführt werden, z. B. indem man ein *push* vom lokalen zum Haupt-Repository macht oder indem der Entwicklungsleiter ein *pull* von der Kopie macht. Dabei werden dann mögliche Konflikte aufgelöst.

Man sollte sich vor Merges nicht zu sehr fürchten, denn in der Praxis ist es gar nicht so häufig, dass zwischen zwei Updates mehrere Personen an ein und derselben Datei gearbeitet haben; ein schwierig zu lösender Konflikt ist also in vielen Projektszenarien eher die Ausnahme als die Regel.

Konsistentes Projekt am Server

Arbeitet man mit zentralen SCM-Systemen wie SVN, ist aber dennoch sehr wichtig, den Stand des Codes am SCM-Server konsistent und korrekt zu halten. Andernfalls kann es zu der unschönen Situation kommen, dass Entwickler Code auschecken oder updaten, der nicht einmal funktioniert oder ungelöste Konflikte enthält. Diese Entwickler müssen dann unter Umständen in fremdem Code nach Fehlern suchen, um die Anwendung kompilieren und an den eigenen Teilen weiterentwickeln zu können. Daher müssen einerseits Änderungen am Projekt *regelmäßig* ins Repository committet werden, zumindest einmal pro Arbeitstag. Damit sollten keine zu großen Konflikte entstehen. Außerdem darf ausschließlich Code eingecheckt/committet werden, der fehlerfrei kompiliert; idealerweise auch alle Tests korrekt ausführt! Außerdem ist es empfehlenswert, vor dem Commit immer ein Update zu machen, um mögliche Konflikte frühzeitig zu entdecken. Bei verteilten Systemen trifft dies beim Commit nicht zu, da Commits nur auf das lokale Repository gehen und daher andere Entwickler nicht beeinträchtigen. Irgendwann werden jedoch die Versionen der verschiedenen Entwickler entweder zentral oder verteilt wieder zusammengeführt (meist mit *Push*- oder *Pull*-Kommandos), und für diese gilt natürlich dasselbe wie

Korrektes Commit

[7] Arbeitet man z. B. in der Spieleentwicklung sehr viel mit binären Dokumenten, so erhält man wenig Unterstützung vom SCM-System, um diese bei Konflikten zu mergen. In solchen Fällen kann ein SCM-System mit Locking-Mechanismen eine geeignete Wahl darstellen. Damit kommen aber nur mehr zentralisierte Systeme in Frage, da verteilte SCM-Systeme Locking prinzipbedingt nicht unterstützen können.

oben geschrieben. Nur korrekter und getesteter Code darf hier weitergegeben werden!

10.2.5 Refactoring und Sourcecode-Management

Refactoring

Fallweise ist es erforderlich, die Ordnerstruktur eines Projekts zu ändern, bzw. Dateien von einem Ordner in einen anderen zu verschieben oder Dateien oder Ordner umzubenennen. Viele SCM-Systeme erkennen derartige Umstrukturierungen nicht automatisch. Daher ist es in diesem Fall sehr wichtig, dass diese Operationen *nicht* über Mechanismen des Betriebssystems (Kommandozeile, Mac Finder, File Explorer usw.) gemacht werden. Denn verschiebt man z. B. eine Datei von einem Ordner in einen anderen und führt danach ein Update oder Commit durch, so ist das SCM-System oft nicht in der Lage festzustellen, dass es sich um *dieselbe* Datei handelt, die nur verschoben wurde. Es wird dann ein *Delete* der alten Datei und eine *Add* der neuen durchgeführt. Damit ist aber die gesamte Versionsgeschichte der Datei verlorengegangen, bzw. nur noch sehr schwierig zu rekonstruieren. Um dies zu vermeiden, bieten viele SCM-Systeme eigene *move* Befehle an, um die Datei unter Beibehaltung der Versionsgeschichte zu verschieben.

Entwicklungsumgebungen

Dieses Verhalten vieler SCM-Systeme ist auch besonders zu beachten, wenn man mit integrierten Entwicklungsumgebungen (IDEs) wie Eclipse arbeitet. Diese IDEs bieten häufig *Refactoring*-Unterstützung an, z. B. das Umbenennen von Java-Paketen. Dabei werden dann natürlich auch im Dateisystem Umbenennungen durchgeführt. Ist das verwendete SCM-System von der IDE nicht gut unterstützt (z. B. durch ein entsprechendes Plugin), so kann man mit einem Refactoring das SCM-System gründlich durcheinanderbringen und auch die Versionsgeschichte der umbenannten Dateien verlieren.

10.2.6 Integration in den Arbeitsalltag

Dokumenten-Verwaltung

SCM-Systeme bieten sich aber nicht nur für Team-Software-Projekte an. Hat man sich einmal mit einem System vertraut gemacht, so wird man dieses meist auch für einfachere Projekte verwenden, an denen man nur alleine arbeitet. Da viele SCM-Systeme wie erwähnt auch das Verwalten von Binärdaten wie PDFs, Bildern, Diagrammen, Office-Dokumenten und dergleichen erlauben, können sie auch sehr leistungsfähig zur Dokument- und Artefaktverwaltung (nicht nur) in Software-Projekten eingesetzt werden.

GUIs

Natürlich gibt es für die gebräuchlichen Systeme auch entsprechende Integration in Entwicklungsumgebungen wie Eclipse und Netbeans, aber auch in das Betriebssystem, z. B. in der Form von Plugins für den Windows File Explorer oder den Mac OS Finder. Ebenso gibt es meist Standalone Fatclients, die komplexere Aktionen erleichtern. Diese Plugins erlauben auch

eine Verwendung der SCM-Systeme abseits „klassischer“ Software-Entwicklung.

10.2.7 Die Zukunft? Verteilte und zentralisierte Systeme

Verteilte vs. zentrale SCM-Systeme

Bis heute dominieren in der Software-Entwicklung zentralisierte Systeme wie Subversion. In den letzten Jahren haben, wie schon erwähnt, verteilte SCM-Systeme aber stark an Bedeutung gewonnen. Mit diesen kann man bei Bedarf ebenso arbeiten wie mit zentralisierten Systemen. Andererseits erlauben sie aber ein viel flexibleres Arbeiten, da jeder Entwickler immer mit Kopien des ganzen Repositories arbeitet und Commits immer lokal erfolgen. Man hat damit eine feinere Versionsgeschichte, und die Systeme sind in der Regel wesentlich schneller. Auch lassen sich (wie beispielsweise bei der Entwicklung des Linux Kernels zu sehen) auch andere Formen der Kollaboration durchführen, bei denen unter Umständen gar keine dedizierten Server mehr notwendig sind. Bei verteilten Systemen sind alle Repositories technisch gleichberechtigt, bestimmte Repositories werden aber häufig (in der Open-Source-Entwicklung) aus Gründen der „Autorität“ oder per Definition zum zentralen Repository erklärt.

Systeme wie GIT, Mercurial und Subversion können auch parallel verwendet werden, d. h., es ist möglich, dass Entwickler mit GIT arbeiten, aber ein PUSH in ein Commit auf einen SVN-Server übersetzt wird, oder umgekehrt, dass Entwickler per SVN-Protokoll auf verteilte Repositories zugreifen. Auch für die Datenmigration zwischen den verschiedenen Systemen stehen viele Werkzeuge zur Verfügung. Somit ist ein Wechsel im Prinzip auch ohne Verlust der Versionsgeschichte jederzeit möglich.

Datensicherheit

Der Aspekt der Datensicherheit ist ebenfalls zu bedenken: bei verteilten Systemen (besonders bei solchen mit vielen Entwicklern) hat man automatisch viele Kopien des eigenen Projekts. Dies trifft zumindest auf den gemeinsamen Projektstatus zu, nicht auf lokale Commits. Diese können (z. B. bei Verlust des Notebooks oder defekter Hardware) bei mangelhaftem Backup auf Entwicklerseite verloren gehen. Bei zentralisierten Systemen (besonders im Firmenumfeld) kann der Server natürlich gut in Backup-Strategien integriert werden. Geht das zentrale System verloren, ist damit natürlich auch (im Gegensatz zu verteilten Systemen) die Versionsgeschichte verloren. Bei Open-Source-(aber auch kommerziellen) Entwicklungen empfiehlt sich ohnedies die Verwendung einer geeigneten Entwickler-Plattform. Hier gibt es mittlerweile eine große Auswahl: Sourceforge[8] bietet z. B. Subversion, Git, Mercurial, Bazaar und CVS Repo-

[8] `http://sourceforge.net/`

sitories an, Google Code[9] hat Subversion und Mercurial Unterstützung, Bitbucket[10] ist eine Plattform für Mercurial-Projekte und GitHub[11] hostet GIT-Projekte.

Fazit

Bei neuen Projekten empfiehlt sich eine gründliche Analyse der eigenen Anforderungen und der Möglichkeiten der verschiedenen Systeme. In vielen Fällen wird man heute vermutlich feststellen, dass ein verteiltes System das Werkzeug der Wahl ist. Es gibt trotzdem noch etliche Szenarien (s.o.), in denen auch zentralisierte Systeme noch ihre Berechtigung haben. Vielleicht ist sogar ein hybrider Ansatz, also eine Mischung mehrerer Systeme, für das eigene Projekt optimal. Dies könnte eventuell auch bei laufenden Projekten eine interessante Variante darstellen. Der Umstieg eines bestehenden Projekts von einem zentralisierten zu einem verteilten System ist ebenfalls eine Überlegung wert. Wichtig ist aber eines: ein Software-Projekt sollte niemals ohne SCM-System realisiert werden.

10.2.8 Patches und Diffs

Kompakte Kommunikation von Änderungen

Ein letzter Aspekt, der vor allem auch in der Open-Source-Entwicklung häufig eingesetzt wird, nämlich *Patches*, wird noch kurz besprochen: In der Software-Entwicklung stößt man – sollen Änderungen am Sourcecode, wie neue Features oder Fehlerkorrekturen, kommuniziert oder überprüft werden – immer wieder auf derartige Szenarien:

- Entwickler möchten den Unterschied zwischen zwei Software-Versionen prüfen.
- Ein Nutzer der Software (der keinen direkten Zugriff auf das SCM-Repository hat) findet einen Fehler, liest sich in den Sourcecode ein, programmiert eine Fehlerkorrektur und möchte diese an die Entwickler senden.
- Neue Features sollen reviewed werden, bevor sie in den Hauptzweig (*Trunk*) der Entwicklung eingespielt werden.
- Fehlerkorrekturen, die an einem Branch oder einem Repository gemacht wurden, sollen in einen anderen Branch (oder in ein anderes Repository) übertragen werden
- Installierte, sehr umfangreiche Software-Pakete (wie etwa Betriebssysteme) sollen aktualisiert werden, dabei sollen nur die Teile, die sich tatsächlich geändert haben, upgedated werden.

[9] `http://code.google.com`
[10] `http://bitbucket.org`
[11] `http://github.com/`

Binär vs. Text

An diesen Szenarien erkennt man schon zwei grundsätzlich unterschiedliche Anwendungsfelder: Die ersten Szenarien sind unmittelbar mit der Software-Entwicklung verknüpft und basieren zumeist auf Unterschieden an textbasierten Dateien wie Programm-Sourcecode. Das letzte Beispiel beschreibt einen Prozess der Wartung von ausgelieferter Software und basiert auf Binärdaten. Im letzteren Fall geht es im Wesentlichen darum, bei Updates nicht das gesamte (sehr große) Software-Paket ausliefern zu müssen, sondern nur die Teile, die sich geändert haben[12].

`diff` in der Software-Entwicklung

In der Software-Entwicklung hat man es (wenn von Patches die Rede ist) zumeist mit verschiedenen Versionen textbasierter Dateiformate zu tun, die man vergleichen möchte. Eines der ältesten Unix-Werkzeuge, nämlich `diff`, stellt die Basis für Patches dar[13]. `diff` ist ein Werkzeug, das zwei Dateien vergleicht und den Unterschied zwischen diesen Dateien ausgibt oder in eine Datei schreibt. Dieser Unterschied kann dann dazu verwendet werden, um andere Instanzen der alten Datei auf die neue Version zu *patchen*.

Diff Beispiel

Um dies an einem einfachen Beispiel zu illustrieren: Angenommen man hat folgende Klasse, die viele Methoden implementiert, unter anderem auch die `addLineNumbers` Methode:

```
1  public class FileHelper {
2    ...
3    public static String addLineNumbers (String
         filename)
4    throws FileNotFoundException, IOException {
5      BufferedReader br =
6      new BufferedReader (new FileReader(filename));
7      StringBuffer text = new StringBuffer();
8      String l;
9      int lineNr = 0;
10     while ((l = br.readLine()) != null) {
11       text.append("" + ++lineNr + " " + l);
12     }
13     return text.toString();
14   }
15   ...
16 }
```

Diese Klasse liest ein Textfile ein und sollte alle eingelesenen Zeilen mit Zeilennummern versehen. Diese Klasse ist Teil eines größeren Softwa-

[12] Hier spricht man dann gegebenenfalls auch von kumulativen und nicht kumulativen Patches. Im ersten Fall müssen alle Patches der Reihenfolge nach eingespielt werden (ausgehend von der letzten Komplettinstallation), während im zweiten Fall das Einspielen der letzten Version alleine ausreichend ist.

[13] Hier werden die Prinzipien anhand der Unix-Werkeuge `diff` und `patch` erklärt. Heute gibt es eine Vielzahl an Werkzeugen, die sich an diesen Programmen orientieren und die Funktionalität z. B. in IDEs, SCM-Systemen oder Editoren implementieren.

re-Projekts und wurde auch ausgeliefert. Nun hat ein Tester korrekterweise festgestellt, dass in der `addLineNumbers`-Methode ein Fehler steckt: Es wird nämlich der Zeilenumbruch vergessen. Also korrigiert er diese Klasse wie folgt:

```
1  public class FileHelper {
2    ...
3    public static String addLineNumbers (...) ... {
4      ...
5      while (...) {
6        text.append("" + ++lineNr + " " + l + "\n");
7      }
8      ...
9    }
10   ...
11 }
```

Um diesen Fehler an das Entwicklerteam zu kommunizieren, erzeugt er eine Textdatei, die nur die Unterschiede beinhalten. Dafür kann man das Unix `diff`-Kommando verwenden:

```
1  diff -u FileHelper_old.java FileHelper.java
2  --- FileHelper_old.java 2009-08-10
       16:59:13.000000000 +0100
3  +++ FileHelper.java 2009-08-10 16:59:43.000000000
       +0100
4  @@ -12,7 +12,7 @@
5      String l;
6      int lineNr = 0;
7      while ((l = br.readLine()) != null) {
8  -     text.append("" + ++lineNr + " " + l);
9  +     text.append("" + ++lineNr + " " + l + "\n");
10     }
11     return text.toString();
12   }
```

In der ersten Zeile sieht man den `diff`-Befehl, der die alte Version `FileHelper_old.java` mit der neuen, korrigierten Version `FileHelper.java` vergleicht und den Unterschied[14] ausgibt. In diesem Format wird im Header (Zeile 2 und 3) angegeben, auf welche Dateien sich der Diff bezieht sowie der Kontext der Änderung (Zeilen 5–7 und 10–12). In Zeile 8 wird durch das „-“ angezeigt, dass diese Zeile in der alten Version entfernt wurde und durch Zeile 9 („+“) ersetzt wurde. Schreibt man diesen Unterschied nun in eine Datei, so hat man einen sogenannten *Patch*:

Vom Diff zum Patch

```
1  diff -u FileHelper_old.java FileHelper.java >
       cr_bugfix.patch
```

[14] Um Unterschiede zu kommunizieren, haben sich verschiedene Diff-Formate eingebürgert, hier wird das *unified*-Format verwendet.

Diesen Patch kann der Tester nun an einen der Entwickler senden. Der Entwickler kann nun einerseits den Patch begutachten und sieht sofort *wo* in der Klasse *was* geändert wurde[15] und andererseits seine noch fehlerhafte Version korrigieren. Dies kann wieder mit einem Unix-Befehl, nämlich `patch` erfolgen. Der Entwickler wechselt in das Verzeichnis, in dem die alte Klasse zu finden ist, kopiert dort die Patch-Datei hinein und führt den folgenden Befehl aus:

```
patch < cr_bugfix.patch
```

Patches und SCM-Systeme

Damit wurde seine Datei auf den neuen Stand gebracht. Das im obigen Beispiel gezeigt Verfahren soll das Prinzip eines Patches verdeutlichen. In der Praxis der Software-Entwicklung hat man es bei einer Fehlerbehebung oft mit Änderungen an mehreren Dateien zu tun; außerdem wird man nicht alte und neue Versionen mit unterschiedlichen Namen ablegen – zur Versionierung wird ja ein SCM-System verwendet. SCM-Systeme bieten daher meist auch einfache Methoden an, um Patches zu erstellen. In Mercurial[16] beispielsweise reicht dafür ein einzelner Befehl:

```
hg export -o cr_bugfix.patch 56
```

Hier wird ein Patch erzeugt und in die Datei `cr_bugfix.patch` geschrieben. Der Patch ergibt sich im Beispiel aus Changeset 56 (also durch die Änderungen, die dieses Changeset verursacht hat), das dem Bugfix entsprechen würde. Es wurde wie im obigen Beispiel die eine Zeile korrigiert sowie zusätzlich noch eine weitere Testklasse hinzugefügt. Der Patch sieht dann wie folgt aus:

```
  # HG changeset patch
  # User Firstname Lastname <...>
  # Date 1249923295 -3600
  # Node ID f057e144a44259f73c66d87099366cfccd996417
  # Parent  4e132a273057e14cb6f89317817e280bbbd7e47a
  Fixed CR issue in FileHelper plus added Test

  diff -r 4e132a273057 -r f057e144a442 FileHelper.
      java
  --- a/FileHelper.java Mon Aug 10 17:53:03 2009
      +0100
  +++ b/FileHelper.java Mon Aug 10 17:54:55 2009
      +0100
...
  diff -r 4e132a273057 -r f057e144a442
      FileHelperTest.java
```

[15] Moderne Editoren und IDEs unterstützen den Entwickler in der Interpretation der Diff-Formate, z. B. durch entsprechende farbliche Kennzeichnung der eingefügten und gelöschten Teile.

[16] GIT und Mercurial sind sich sehr ähnlich. Die hier beschriebenen Mercurial-Kommandos funktionieren auf ganz ähnliche Weise in GIT.

```
13    --- /dev/null Thu Jan 01 00:00:00 1970 +0000
14    +++ b/FileHelperTest.java Mon Aug 10 17:54:55 2009
          +0100
15    @@ -0,0 +1,20 @@
16  ...
17    +public class FileHelperTest {
18  ...
19    +}
20    \ No newline at end of file
```

Die Zeilen 1–5 beschreiben den Changeset (Autor, Datum, Parent), und in Zeile 6 findet man die Commit-Message des Changesets. Zeilen 8 und 12 geben die Dateinamen an, die geändert sowie die Revisionen (IDs), die verglichen wurden. Nach Zeile 8 finden sich (hier gekürzt) die Änderungen, wie schon im ersten Diff-Beispiel beschrieben der `FileHelper.java`-Klasse. Ab Zeile 13 wird die gesamte neue Datei aufgelistet (ebenfalls hier im Text gekürzt), da diese in der vorigen Version ja noch gar nicht vorhanden war. Dieser Patch kann wie oben erwähnt an andere Entwickler weitergeleitet werden, um dort entsprechende Aktualisierungen nachzuziehen.

Bedeutung von Patches bei verteilten SCM-Systemen?

Die Bedeutung von Patches in der Software-Entwicklung hat mit modernen verteilten Versionssystemen wie GIT oder Mercurial etwas abgenommen, da es heute in einigen Fällen einfacher ist, mittels Push oder Pull (Request) Änderungen zu verteilen.

10.3 Build-Management und Automatisierung

10.3.1 Warum Automatisierung?

Manuelle Ausführung repetitiver Aufgaben

In der Software-Entwicklung gibt es eine Vielzahl an Aufgaben, die immer wieder in identischer oder zumindest ähnlicher Weise zu erledigen sind, z. B. das Kompilieren der Software, Post- und Pre-Processing des Sourcecodes, Erstellen der Dokumentation, Ausführen von Tests usw. Dies sind nur einige Beispiele, die regelmäßig, oftmals sogar täglich oder noch häufiger durchzuführen sind. Eine manuelle Abarbeitung solcher Aufgaben ist aus verschiedenen Gründen nicht empfehlenswert: Der erste und offensichtlichste Grund ist natürlich, dass die manuelle Ausführung dieser Aufgaben oft fehlerträchtig ist. Noch ungünstiger ist aber die Tatsache, dass die Reproduzierbarkeit leidet: Einerseits führen Entwickler unter Umständen Aktivitäten aus, die für sie persönlich offensichtlich sind, die aber für andere unter Umständen nicht klar sind. Beispielsweise könnte es notwendig sein, eine bestimmte Umgebungsvariable oder in einer Konfigurationsdatei einen bestimmten Wert zu setzen. Da dies aber für den einen Entwickler ein logischer Schritt ist, wird dieser nicht explizit dokumentiert. Damit ist der Ablauf für andere nicht mehr reproduzierbar. Andere, seltener notwen-

dige Schritte werden vielleicht vergessen oder fehlerhaft durchgeführt und führen damit zu Fehlern in der Software.

Team-Arbeit, Dokumentation und Qualitäts-Management

Team-Arbeit sowie Qualitätsmanagement ist unter solchen Voraussetzungen kaum möglich. Daher ist eine saubere Automation des Buildprozesses eine wesentliche Voraussetzung für die Arbeit im Team und für die Erstellung qualitativ hochwertiger Software. Die Automatisierung hilft auch bei der Erstellung der Dokumentation (siehe Abschnitt 10.7) und verbessert die Qualitätssicherung durch automatisierte Tests und Code-Quality--Checks (siehe Abschnitt 5.8.1). Auch ist eine konsistente und möglichst einfache Integration aller dieser Schritte mit dem Sourcecode-Management-System sowie anderen Werkzeugen, die die Entwicklung unterstützen, z. B. Bug-/Issue-Tracking-Systeme zu gewährleisten .

Der Fokus dieses Abschnitts liegt aber bei der Definition des Build-Prozesses, der Infrastruktur, die für die Automatisierung notwendig ist und Best-Practices für größere Projekte.

10.3.2 Der Build-Lifecycle: Best-Practices

Portabilität des Entwicklungs-Prozesses

Der Build-Prozess soll – vor allem, wenn in Teams, vielleicht sogar in verteilten Teams gearbeitet wird – portabel und unabhängig von einer bestimmten Entwicklungsumgebung sein (IDE, siehe Abschnitt 10.4). Die Integration von IDEs auch in den Build-Prozess ist natürlich sehr wichtig, um effiziente Arbeit zu gewährleisten. Die Tatsache aber, dass die entwickelte Software sich auf dem Rechner *eines* Entwicklers kompilieren, ausführen und testen lässt, ist erfreulich für *diesen einen* Entwickler, für das gesamte Projekt aber irrelevant. Wichtig ist, dass sich das System auf den Rechnern *aller* Entwickler sowie auch auf einem „neutralen“ Rechner, unabhängig von kompliziert konfigurierten Entwickler-Maschinen, builden und testen lässt, und dass die Erstellung einer Distribution zuverlässig und automatisiert funktioniert. Jeder unnötige manuelle Schritt in dieser Entwicklung ist lästig und bringt potenzielle Fehler und Unsicherheiten ein.

Was kann automatisiert werden?

Welche Schritte im Entwicklungsprozess (*build-lifecycle*) können automatisiert werden? Einige Beispiele sind:

- Validierung von Sourcecode.
- Code-Generierung (z. B. Deployment-Deskriptoren oder Mapping-Dokumente).
- Automatisierte Code-Quality-Checks (siehe Abschnitt 5.8).
- Kompilieren der Software.
- Dependency-Management: Welche Bibliotheken werden benötigt? Welche werden nur für die Kompilierung benötigt? Welche müssen ausgeliefert werden etc.?

- Post-Processing von Binaries (z. B. Optimierung oder Anwendung spezieller AOP Frameworks).
- Ausführen von Tests und Erstellen von Test-Reports.
- Generieren der Dokumentation (z. B. javadoc, Webseiten, Projekt-Info, Ergebnisse von Tests und Qualitätsprüfungen).
- Zusammenstellen der Software für die Auslieferung.
- „Verpacken“ der Software in Installations-Dateien oder ZIP-Archiven usw.
- Hochladen der aktuellen Webseite auf den Webserver.

Eigene Skripte zur Automatisierung

Prinzipiell könnte man Teile der Automatisierung auch mit (Unix) Shell Scripts oder (Windows-) Batch-Dateien oder in irgendeiner anderen Skriptsprache durchführen. Viele Projekte wurden auch so begonnen. Ein derartiger generischer Ansatz hat in der Praxis aber einige Nachteile:

Einerseits muss man im Endeffekt relativ viel händisch implementieren. Ein Beispiel zur Illustration: In einem Java-Projekt liegen meist Abhängigkeiten von Bibliotheken vor. Diese werden oft als `.jar`-Datei in ein entsprechendes `lib`-Verzeichnis im Projekt kopiert. Um das Programm zu kompilieren, muss der Klassenpfad entsprechend zusammengesetzt werden. Schreibt man *Shell-Scripts*, so muss man diese Variable entweder manuell erstellen oder Code schreiben, der die entsprechenden Dateien aus dem Verzeichnis herausliest und den Pfad dynamisch zusammenbaut. Dieses einfache Klassenpfad-Management benötigt man aber eigentlich in allen Java-Projekten. Also wird man versuchen, entsprechende Skripte wiederzuverwenden. Zudem möchte man Javadoc-Dokumente generieren sowie Unit-Tests automatisiert ausführen, die Test-Ergebnisse in HTML konvertieren und in die Distribution kopieren usw. Dann möchte man Projektdateien für IDEs wie Eclipse schreiben, die z. B. den Klassenpfad übernehmen, damit nicht die Entwicklung mit der IDE und die Build-Automatisierung asynchron werden. Vielleicht möchte man später ein komplexeres *Dependency-Management* abbilden, um z. B. besser mit Versionskonflikten umgehen zu können, und damit wird es wieder einen Schritt komplizierter. Man wird sich also über kurz oder lang selbst dabei finden, entweder ein Build-System, das in verschiedenen Projekten und in verschiedenen Kontexten verwendet werden kann, zu schreiben oder für jedes neue Projekt viele Konfigurationen oder Skripte umfangreich händisch anzupassen. Dazu kommt weiter das Problem, dass Shell oder Batch Scripts nicht plattformunabhängig und damit schlecht portabel sind.

Build-Werkzeuge für verschiedene Plattformen

Maven

Zum Glück sind solche Eigenentwicklungen in den meisten Fällen nicht notwendig, weil jede moderne Entwicklungsplattform bereits über derartige Werkzeuge verfügt. Als Beispiele kann man make für C/C++ unter Unix, Maven und Ant für Java, Rake für Ruby (on Rails) oder nAnt für .net nennen. Maven ist eines der konzeptionell interessantesten Werkzeuge in

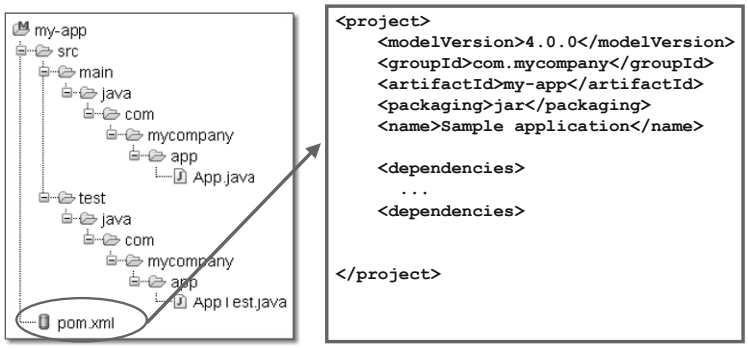


Abbildung 10.2 *Maven: Konfiguration und Standard-Dateistruktur.*

diesem Kontext, denn Maven ist nicht nur ein Werkzeug, sondern eigentlich eine Verkörperung einer Reihe von *Best-Practices*. Die Maven-Entwickler haben zunächst typische Build-Lifecycles untersucht und versucht, daraus ein möglichst allgemeingültiges Modell abzuleiten.

Die Idee der „Convention over Configuration“ (siehe Abschnitt 10.1) ist ebenfalls ein bestimmendes Motiv in modernen Build-Werkzeuge wie Maven oder Rake. In diesem Abschnitt werden daher am Beispiel von Maven wesentliche Aspekte des Build-Lifecycles sowie der Automatisierung erklärt. Diese Ideen lassen sich aber grundsätzlich auf die meisten anderen Systeme übertragen, auch wenn diese im Detail anders arbeiten.

Build-Lifecycle

Die Maven Entwickler definieren zunächst den Build-Lifecycle[17] und teilen diesen in etwa 20 einzelne Schritte, wie zum Beispiel: `validate`, `compile`, `test`, `install` oder `deploy` und noch etliche andere. Wird ein Maven Build durchgeführt, so arbeitet Maven alle 20 Lifecylce-Schritte ab und führt die dafür definierten konkreten Aktivitäten aus, z. B. wird in `compile` der Sourcecode kompiliert und die Binaries in das dafür vorgesehene Verzeichnis kopiert. In `test` wird etwa nach Unit-Test-Klassen gesucht, diese ausgeführt und die Ergebnisse der Tests in das entsprechende Verzeichnis geschrieben.

Darüber hinaus gibt es eine Datei, das *Project Object Model* (`pom.xml`), in der wesentliche Daten des Projekts definiert werden, wie z. B. Name und Version des Projekts, Namen der Entwickler, welches SCM-System verwendet wird, welche speziellen Maven Plugins für den Build benötigt werden sowie welche Abhängigkeiten von anderen Bibliotheken bestehen. Da „Convention over Configuration“ die Basis des Systems ist, werden für viele Vorgaben vernünftige Annahmen, die sich in vielen Projekten bewährt haben, gemacht, z. B. dass die Sourcen im `src/main`-Verzeichnis stehen. Abbildung 10.2 zeigt einen kleinen Ausschnitt aus der Maven-Pro-

[17] `http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`

jekt-Konfiguration sowie die Basis-Ordnerstruktur für ein einfaches Projekt. All diese Vorgaben können bei Bedarf verändert werden. Für die meisten „üblichen“ Lifecylcle-Aktionen wie Kompilieren, Dokumentation generieren, Testen usw. existieren bereits Maven Plugins, die ausgeführt werden. Eigene Aktionen kann man als Maven Plugin definieren und an die entsprechende Stelle im Lifecycle einhängen.

10.3.3 Scaffolding und Archetypen

Archetypen

Viele moderne Build-Werkzeuge bieten außerdem Werkzeuge an, um die Struktur von Standardprojekten zu generieren bzw. Funktionalität zu einem Projekt hinzuzufügen. Diese haben zwar nicht unmittelbar mit der Build-Automatisierung zu tun, sollen aber dennoch kurz erwähnt werden, weil sie die Verwendung dieser Werkzeuge stark vereinfachen. Maven bietet sogenannte *Archetypen* an:

```
mvn archetype:create \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DgroupId=com.mycompany.app \
  -DartifactId=my-app
```

Dieser Maven-Befehl erzeugt beispielsweise die in Abbildung 10.2 gezeigte Projektstruktur für ein einfaches Maven-Projekt. Es gibt mittlerweile eine Vielzahl an anderen Archetypen, z. B. solche, die ein Wicket-Web-Projekt erzeugen. Damit wird dem Entwickler unter die Arme gegriffen, da das Erstellen einer vernünftigen Basis-Konfiguration, z. B. bei Web-Frameworks, nicht immer ganz einfach ist.

Scaffolding

Ruby on Rails oder Java Grails bieten z. B. sogenannte Scaffolding-Mechanismen an, die den Archetypen ähnlich sind, aber noch feingranularere Möglichkeiten bieten. Ein *Scaffold* ist ein „Gerüst“; man kann sich diese Mechanismen so vorstellen, dass dem Entwickler bestimmte Funktionalität von dem Gerüst zur Verfügung gestellt wird, das damit die konkrete Implementation von Funktionalität leitet und vereinfacht. Es können z. B. Controller, Views, Datenbankanbindungen rudimentär erstellt, sodass sie nur noch an die konkrete Problematik angepasst werden müssen.

10.3.4 Dependency-Management

Dependency-Management

Ein weiteres sehr wichtiges Problem, das bei der Build-Automatisierung und Definition eines Projekts oft auftritt, ist der Umgang mit *Abhängigkeiten*. Heutige Projekte bauen in der Regel auf einer Vielzahl von Bibliotheken auf, die bestimmte Funktionalität bereitstellen, z. B. Web-Frameworks, XML Parser oder Persistenz-Lösungen. Nun entwickelt sich einerseits das

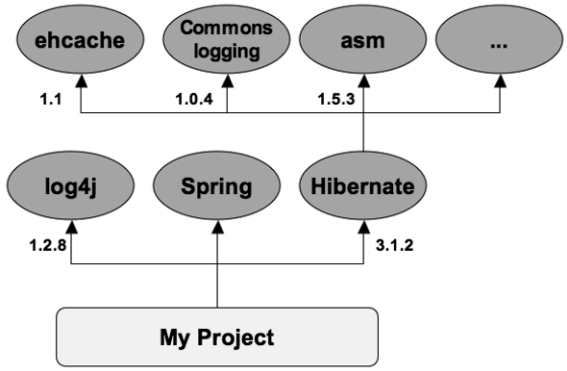


Abbildung 10.3 *Beispiel für transitive Abhängigkeiten in Projekten.*

eigene Projekt weiter, andererseits werden auch Bibliotheken regelmäßig aktualisiert. Um trotz dieser parallelen Entwicklungen einen konsistenten Zustand zu halten, sollte man daher ein Versionsmanagement verfolgen. Dies bedeutet, dass jeder Entwickler dieselben Versionen aller Bibliotheken verwenden und die operative Software ebenfalls mit diesen Versionen der Bibliothek ausgeliefert werden muss.

Irgendwann wird man aber auf neuere Versionen bestimmter Bibliotheken aktualisieren wollen; nun ist die Situation komplizierter geworden: Ältere Versionen der eigenen Software wurden mit älteren Versionen bestimmter Bibliotheken getestet und sollten daher auch nur mit diesen ausgeliefert werden, um Inkompatibilitäten und andere Seiteneffekte auszuschließen. Auch zwischen bestimmten Bibliotheken gibt es leider immer wieder Seiteneffekte. Man muss also für jede Version der eigenen Software immer auch die passenden Versionen der Bibliotheken mitführen. Andernfalls wird die Entwicklung instabil.

Transitive Abhängigkeiten und Konflikte

Das Ganze wird leider noch komplizierter, wie Abbildung 10.3 zeigt: In diesem Beispiel ist die letzte Version der eigenen Software von log4j Version 1.2.8 von Spring in einer nicht näher spezifizierten Version sowie von Hibernate in Version 3.1.2 abhängig. Hibernate selbst ist aber wieder von anderen Bibliotheken abhängig, z. B. ehcache, commons logging (ebenfalls in speziellen Versionsn) usw. Dies nennt man *transitive Abhängigkeiten*. Dadurch wird die Verwaltung von Bibliotheken leider nicht einfacher, denn möchte man (um beim Beispiel zu bleiben) Hibernate verwenden, muss nicht nur die Hibernate-Bibliothek selbst in der richtigen Version vorliegen, sondern auch die Bibliotheken, von denen Hibernate abhängig ist. Dazu kommt noch, dass es unter Umständen auch zu Versions-Konflikten kommen kann; z. B. wenn Hibernate eine bestimmte Version von *commons-logging* benötigt, aber die eigene Software eine andere.

In manchen Software-Projekten war es daher üblich, die Bibliotheken einfach in ein `lib`-Verzeichnis zu kopieren. Unter den gerade geschilderten Voraussetzungen ist dieser Weg leider als nicht sehr strukturiert zu bezeichnen. Selbst wenn man einmal einen konsistenten Zustand hergestellt hat,

muss dieser zwischen den Entwicklern synchronisiert werden. Was passiert aber, wenn man eine Bibliothek updaten oder einfach einmal testen möchte, ob die eigene Software auch mit einer anderen Version einer bestimmten Bibliothek korrekt arbeitet, oder wenn man sie durch eine andere Bibliothek austauschen möchte? Dann steht man vor der Situation, analysieren zu müssen, welche Abhängigkeiten vorliegen, welche Bibliotheken man löschen, welche man updaten darf, welche nicht verändert werden dürfen usw. Verliert man in der Vielzahl der Bibliothken den Überblick, kann man sich unerwartete Seiteneffekte einhandeln oder Bibliotheken behalten, die man eigentlich gar nicht mehr benötigt.

Deklaratives Dependency-Management am Beispiel von Maven

Maven und andere Werkzeuge bieten heute die Möglichkeit, sich um die Verwaltung dieser Abhängigkeiten zu kümmern. Der Weg, den Maven einschlägt, ist folgender:

- Man deklariert die *direkten* Abhängigkeiten des eigenen Projekts in der Projekt-Definition (in der `pom.xml`-Datei).
- Dabei gibt man die Bibliothek sowie die benötigte Version der Bibliothek an.
- Transitive Abhängigkeiten müssen in der Regel nicht deklariert werden, d. h., wenn eine Bibliothek selbst Abhängigkeiten hat, so steht dies in der `pom.xml`-Datei der Bibliothek, und Maven löst dies rekursiv auf[18].
- Die `pom.xml`-Datei wird im Sourcecode-Management-System gespeichert; Änderungen der Abhängigkeiten über verschiedene Versionen sind also einfach nachvollziehbar.
- Maven speichert alle verwendeten Bibliotheken aller Projekte in einem zentralen Verzeichnis auf dem Computer des Entwicklers.

Auflösen der Abhängigkeiten

Stößt man einen Build mit Maven an, so macht Maven Folgendes:

1. Zunächst werden die deklarierten Abhängigkeiten gelesen und geprüft, ob die benötigten Bibliotheken schon in der geforderten Version im lokalen Repository vorhanden sind.
2. Ist dies nicht der Fall, wird die entsprechende Bibliothek von einem zentralen Repository (*Server*) geladen[19] und im lokalen gespeichert.

[18] Zum Beispiel: In Abbildung 10.3 würde man nur `log4j`, `Spring` und `Hibernate` als Abhängigkeit angeben. Dass Hibernate weitere Abhängigkeiten hat, würde Maven herausfinden. Damit dies funktioniert, müssen natürlich für die Bibliotheken entsprechende `pom.xml`-Dateien vorliegen. Für die meisten gängigen Bibliotheken ist dies allerdings schon der Fall.

[19] Das Maven Projekt verweist auf eigene Server, auf denen fast alle wesentlichen Open-Source-Bibliotheken schon vorhanden sind; möchte man dies nicht, bzw. braucht

3. Maven prüft nun im weiteren Verlauf, ob die angegebene Bibliothek eigene Abhängigkeiten hat und ob diese schon lokal vorhanden sind usw.

4. Liegen letztlich alle benötigten Bibliotheken in der richtigen Version lokal vor, so erstellt Maven einen entsprechenden Klassenpfad für den Compile- und Test-Lauf oder andere Operationen[20].

Hat man diese Build-Umgebung einmal richtig eingerichtet, so wird das Management von Abhängigkeiten auch im Team deutlich erleichtert. Außerdem können auch über das `pom.xml` bzw. automatisch generierte Dokumente sauber über Versionswechsel hinweg dokumentiert werden. Von den häufig verwendeten Bibliotheken hat man immer nur eine Version lokal gespeichert. Möchte man Eclipse oder eine andere IDE verwenden, so lässt man sich von Maven die Projekt-Dateien erstellen. Diese verweisen für den Klassenpfad auf das lokale Repository. Änderungen an den Abhängigkeiten sind aber immer in der Maven-Definition festzuhalten und dann ist gegebenenfalls die Eclipse-Projektdatei neu zu generieren[21].

Integration mit IDEs?

10.3.5 Reporting und Dokumentation

Reporting und Dokumentation

Ein sehr wesentlicher Teil der Software-Entwicklung besteht aus Dokumentation (siehe auch Abschnitt. 10.7). Viele Dokumente können automatisch erstellt werden, dazu zählen z. B. Webseiten oder Benutzerhandbücher. Gute Build-Automatisierungs-Werkzeuge bieten daher auch die Möglichkeit an, aus Quell-Dateien, die im Sourcecode-Management-System abgelegt sind Dokumentation zu generieren. Der Vorteil liegt auf der Hand: Die Generierung der Dokumentation erfolgt mit jedem Build automatisch, und die Dokumentation wird mit dem Sourcecode versioniert.

Auch Ergebnisse von Tests oder von Code-Analysen können z. B. für die Darstellung im Web aufbereitet werden. Auch hier kommt der Build-Automatisierung eine wichtige Rolle zu: Sie kann nicht nur zum automatischen Ausführen von Code-Quality-Checks und -Tests dienen, sondern auch, um die entsprechenden Ergebnisse in die restliche Dokumentation zu integrieren.

---

man ein eigenes zentrales Repository für Firmenprojekte, so legt man sich einfach einen oder mehrere eigene Server an.

[20] Bei Bedarf können für unterschiedliche Operationen auch unterschiedliche Abhängigkeiten angegeben werden; z. B. wird `junit.jar` normalerweise nur für die Tests, nicht aber für den operativen Betrieb benötigt.

[21] Details finden sich in der BPSE-Webseite: `http://bpse.ifs.tuwien.ac.at` Details zu Maven findet man auch in dem sehr guten und kostenlosen Buch „Maven, The Definitive Guide“ [84].

10.3.6 Continuous Integration

Gewinn durch Build-Automatisierung

Aus den vorigen Abschnitten ergeben sich zusammengefasst folgende Vorteile einer gut aufgesetzten Build-Automatisierung:

- Ein neues Projekt kann mithilfe von Archetypen oder Scaffolding-Mechanismen schnell (Best-Practices folgend) aufgesetzt werden.
- Das Projekt ist portabel und nicht von konkreten Konfigurationen oder IDE-Einstellungen auf einem Rechner abhängig.
- Der Build-Lifecycle ist übersichtlich und nachvollziehbar.
- Abhängigkeiten des Projekts von anderen Bibliotheken (auch transitiven) sind leichter darstellbar und werden möglichst automatisiert aufgelöst.
- Die Projekt-Qualität kann durch die Integration automatisierter Tests und Code-Quality-Checks verbessert werden.
- Reporting im Projekt kann automatisiert werden (z. B. für Testergebnisse).
- Verschiedene andere, für den Build-Prozess wichtige Aufgaben können leicht in die Automatisierung integriert werden.
- Ein gutes Zusammenspiel zwischen der Build-Automatisierung und der integrierten Entwicklungsumgebung ist gegeben.

Die bisherige Beschreibung hat zwar eine hochgradige Automatisierung des Build-Prozesses zum Ziel; Werkzeuge wie Maven oder Ant werden häufig in Kombination mit IDEs wie Eclipse auf Entwicklungs-Rechnern eingesetzt und erfordern immer noch Handarbeit, um den Build anzustoßen, Werkzeuge zu integrieren, Testläufe auf Probleme hin zu untersuchen und Distributionen zu erstellen.

Continuous Integration

Der letzte „Puzzlestein“ in der Build-Automatisierung wird auch *Continuous Integration* (CI) genannt. Dabei handelt es sich um eine Reihe von Best-Practices, die empfohlen werden [68]. Martin Fowler nennt im Wesentlichen folgende Aspekte [31]:

- Ein Projekt sollte nur über ein Sourcecode-Repository verfügen, mit dem alle Entwickler arbeiten.
- Der Build sollte so weit wie möglich automatisiert ablaufen (wie oben beschrieben).
- Die Automatisierung sollte soweit wie möglich auch automatisierte Tests beinhalten.

- Jeder Entwickler, der am Projekt arbeitet, sollte wenigstens einmal täglich seine Änderungen in das Sourcecode-Repository committen.
- Der Build sollte schnell ablaufen (um häufige Builds möglich zu machen).
- Das Deployment sollte ebenfalls automatisiert sein.
- Der Build selbst sollte wenigstens einmal täglich auf einem *neutralen* Rechner automatisiert ausgeführt werden.
- Die Transparenz des Entwicklungsprozesses sollte gewährleistet sein.

Der Herzschlag des Projekts

Die ersten Punkte wurden schon in anderen Abschnitten besprochen. Eine wesentliche neue Eigenschaft einer Continuous-Integration-Strategie besteht darin, dass ein eigener „neutraler" Rechner[22] als CI-Server bereitgestellt wird. Auf diesem Server wird automatisiert mindestens einmal täglich (oder jedesmal, wenn Änderungen in das SCM-System eingecheckt werden) der Build inkl. Tests angestoßen. Das Ergebnis des Builds und der Tests wird z. B. auf einer Intranet-Webseite zusammengefasst dargestellt und sollte allen am Entwicklungsprozess beteiligten Personen zugänglich sein. Eventuell werden auch Entwickler z. B. per E-Mail über Probleme benachrichtigt. Zudem sollte der Buildserver auch automatisiert Installationspakete (*daily builds*) erstellen, die ebenfalls im Inter- oder Intranet zugänglich sind. Der tägliche Build wird manchmal mit dem „Herzschlag" eines Projekts verglichen. Schlägt dieses nicht mehr, ist das Projekt tot.

> „The most fundamental part of the daily build is the *daily* part. Treat the daily build as the heartbeat of the project. If there's no heartbeat, the project is dead." (James McCarthy)

CI-Werkzeuge

Für das Aufsetzen eines CI-Servers gilt Ähnliches, wie für das Aufsetzen einer Build-Automatisierung. Auch hier kann man natürlich wieder selbst Hand anlegen und sich entsprechende Skripte schreiben. Es gibt aber schon eigene CI-Server, z. B. Apache Continuum[23] oder Hudson[24], die auch diese Aufgabe wesentlich erleichtern. Diese Web-Anwendungen erlauben das Definieren mehrerer Software-Projekte, die mit Maven, Ant, Shellscript usw. automatisiert sind, und deren Sourcecode in einem Sourcecode-Management-System (SCM) abgelegt ist. Die CI-Server aktualisieren dann in definierten Intervallen alle Projekte aus dem SCM und exekutieren die

[22] „Neutral" bedeutet, dass dieser Rechner dem Zielsystem entsprechen sollte. Ein erfolgreicher Build auf einem Entwicklungssystem könnte immer auch mit speziellen Konfigurationen auf diesem zu tun haben, aber auf anderen fehlschlagen.

[23] `http://continuum.apache.org/`

[24] `https://hudson.dev.java.net/`

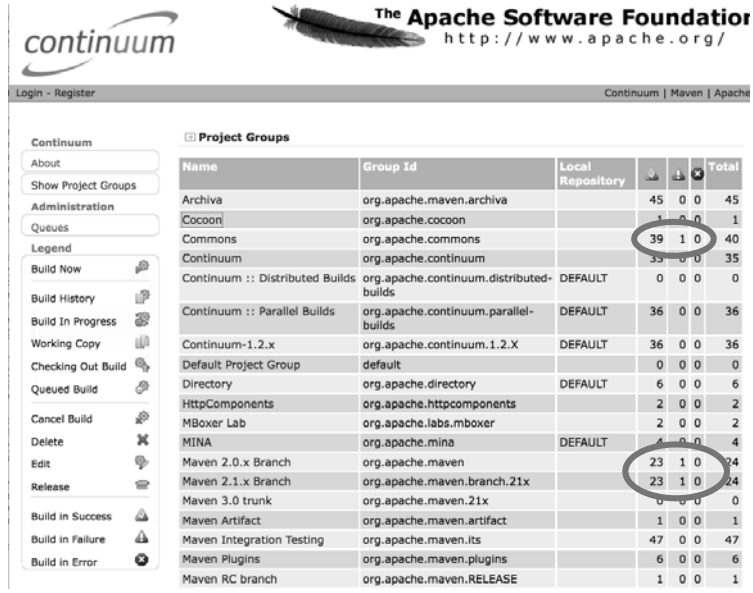


Abbildung 10.4
Screenshot der Apache-Continuum-Webanwendung: Man erkennt in der Übersicht alle Projekte und auf einem Blick, bei welchen der letzte Build fehlgeschlagen ist.

Build-Automatisierung. Der Screenshot in Abbildung 10.4 zeigt den Apache-Continuum-Server, in dem eine Reihe von Apache-Projekten definiert sind, deren Build täglich ausgeführt wird. Man erkennt auf einen Blick in welchem Projekt ein Build fehlgeschlagen ist und kann sich dann zu Details „weiterklicken". Die Ergebnisse der Builds werden also im Web leicht zugänglich dargestellt, Entwickler werden benachrichtigt und Daily Builds können ebenso automatisiert erstellt werden. Manche CI-Werkzeuge sind auch in der Lage, zu erkennen, wenn es Änderungen im SCM gibt und werden dann aktiv.

10.4 Die integrierte Entwicklungsumgebung

Definition

Der Ausdruck IDE kommt aus dem Englischen und bezeichnet üblicherweise *Integrated Development Environment* oder *Integrated Debugging Environment*, wobei mit dem Aufkommen von objektorientierten Sprachen und umfassenden Werkzeugen, wie visuellen Editoren für grafische Benutzeroberflächen sowie die UML-Modellierung, heutzutage auch der Gebrauch von *Integrated Design Environment* für die meisten IDE's durchaus angebracht ist.

Zweck & Bestandteile

Im Deutschen wird die IDE als *Integrierte Entwicklungsumgebung* bezeichnet, also eine Software, die eine oder mehrere Programmiersprachen unterstützt und dem Entwickler bei häufig anfallenden Aufgaben zur Hand geht oder diese automatisiert. Entwicklungsumgebungen sollen es dem Entwickler erlauben, sich von einer zentralen Stelle aus auf die eigentliche Programmiertätigkeit zu konzentrieren, statt sich mit nebensächlichen Tä-

tigkeiten oder einer Vielzahl an Hilfsprogrammen beschäftigen zu müssen. Grundsätzlich besteht eine Entwicklungsumgebung aus einem Sourcecode-Editor, Compiler, Linker und Debugger sowie einer Projekt-Verwaltung. Weitere Funktionalität kann meist über Plugins ergänzt werden.

Feature-Overkill?

Moderne Entwicklungsumgebungen bestehen allerdings aus so vielen Werkzeugen und Hilfsprogrammen, dass oft der ursprüngliche Sinn und Zweck der Entwicklungsumgebung aus den Augen verloren wird. Für unerfahrene Entwickler kann dadurch die Funktion der IDE undurchschaubar und verwirrend werden; besonders wenn Probleme auftreten, kann es dann schwierig werden, die Abläufe zu durchblicken.

Eclipse IDE

Besonders der unerfahrene Entwickler sollte daher zunächst im Wesentlichen die oben erwähnten elementaren Funktionen einer IDE im Zusammenspiel mit einer IDE-unabhängigen Build-Automatisierung nutzen. Im Java-Umfeld ist die IDE Eclipse-dominierend. Eclipse ist im Kern ein erweiterbares Framework. Nach und nach können weitere, tatsächlich benötigte Plugins nachinstalliert und damit der Funktionsumfang schrittweise erweitert werden. Die Plugin-Architektur und der Open-Source-Charakter von Eclipse führen dazu, dass es heute zwar für alle möglichen Werkzeuge, von SCM-Systemen über Issue-Tracking bis zu UML-Werkzeugen Eclipse Plugins gibt, diese jedoch nicht zwingend verwendet werden müssen. Eine der wesentlichen Grundsätze von Eclipse ist es, *non-intrusive* bezüglich der Projekt-Struktur zu sein, d. h., Eclipse geht davon aus, dass es nicht die einzige IDE ist, welche in einem Projekt verwendet wird.

Gerade im Umfeld von IDEs und Editoren ist es schwierig, allgemeingültige Best-Practices anzugeben, weil hier die persönliche Arbeitsweise der Entwickler häufig sehr unterschiedlich ist, und diese auch meist, ohne das Projekt zu gefährden, respektiert werden kann.

10.5 Virtualisierung von Hard- und Software

Virtualisierung und Software-Engineering?

Unter Virtualisierung soll an dieser Stelle die *Nachbildung von Hardware in Software* verstanden werden, und zwar zum konkreten Zweck, virtuelle Rechner auf Desktop- und Server-Gastsystemen zu starten. Virtualisierung gibt es zwar schon seit Jahrzehnten, aber erst in den letzten Jahren haben Systeme wie VMWare[25] oder Virtual Box[26] für eine entsprechende Verbreitung auch für Desktop-Anwendungen und leichte Zugänglichkeit auch abseits von komplexen Server-Virtualisierungslösungen (z. B. auf IBM Mainframes) gesorgt.

[25] `http://www.vmware.com`
[26] `http://www.virtualbox.org/`

Systeme wie Virtual Box sind Software-Produkte (in diesem Fall sogar unter einer Open-Source-Lizenz), die auf einem Gastrechner installiert werden. Startet man die Virtualisierungslösung, so simuliert diese zumeist einen „nackten“ PC, auf dem man ein beliebiges anderes Betriebssystem installieren kann. Damit können z. B. auf einem Linux-Rechner eine oder mehrere virtuelle Maschinen gestartet werden, die wieder unter Linux, Windows oder BSD laufen. Eine Virtualisierungs-Umgebung erstellt und verwendet ein sogenanntes Image (also eine Datei), das alle Daten der virtuellen Maschine enthält und das leicht weitergegeben werden kann. Bei entsprechender Konfiguration können die virtuellen Maschinen über das Gastsystem auf vorhandene Ressourcen wie Netzwerk oder Hardware zugreifen und mit dem Gastsystem oder untereinander interagieren (z. B. über das Netzwerk). Virtualisierungslösungen können Software-Entwicklung auf verschiedenen Ebenen unterstützen; als Beispiele könnte man folgende Szenarien nennen:

Einfache Demonstrations-Umgebung

Manche Software-Produkte erfordern eine komplexe Installation und Konfiguration sowie verschiedene andere Systeme, die entsprechend eingerichtet werden müssen, um zu funktionieren. Als Beispiel kann man sich eine Webshop-Anwendung vorstellen, die folgende Komponenten benötigt:

- Webserver,
- J2EE Applikationsserver,
- WebDAV-Server,
- Relationale Datenbank,
- Fat-Clients zur Administration und ein
- ERP-System.

All diese Komponenten wie Webserver und WebDAV-Server müssen korrekt installiert und konfiguriert werden. Die Datenbank muss eingerichtet und der Fat-Client installiert und konfiguriert sein. Um das System demonstrieren zu können, müssen nun Testdaten in der Datenbank und dem WebDAV-Server vorliegen sowie das ERP-System installiert, konfiguriert und ebenfalls mit Testdaten gefüllt sein.

Es ist unschwer zu erkennen, dass es wenigstens einen Experten benötigt, um dieses System zum Laufen zu bringen. Nun möchte man aber das eigene System potenziellen Kunden zeigen, gegebenenfalls auch Test-Versionen zum Download oder auf DVD anbieten. Ein potenzieller Kunde verliert aber vermutlich schnell das Interesse, wenn dem „Rumspielen“ mit dem System Stunden an Installation und Konfiguration vorangehen. Für solche Zwecke bieten sich Virtualisierungsumgebungen an. Hier kann das Entwicklerteam selbst auf der virtuellen Maschine die Software mit allen benötigten Systemen korrekt installieren, konfigurieren und mit Test-Daten

befüllen. Das erstellte Image kann dann einfach an Kunden weitergegeben werden, die dann nur mehr die Virtualisierungs-Software benötigen, um diesen virtuellen Rechner komplett konfiguriert zu starten. Nicht selten werden solche Demostrations-Umgebungen auch von Mitarbeitern im Vertrieb für Präsentationen bei Kunden, auf Messen oder diversen Veranstaltungen verwendet.

Test-Umgebungen für komplexe Test-Szenarien

Ein weiterer Einsatzbereich für Virtualisierungslösungen im Software-Engineering besteht in der Vereinfachung und Automatisierung auch komplexer Integrationstests. Für diese gilt ähnliches wie beim oben beschriebenen Szenario. Oft müssen Fremdsysteme installiert, konfiguriert und in einen bestimmten Zustand gebracht werden. Auch hier können Images für bestimmte Test-Szenarien angelegt werden, die dann in der Virtualisierungslösung gestartet werden. Die Fremdsysteme, Datenbanken etc. sind damit immer in einem klar definierten Zustand, und es muss nur noch die aktuelle Version der Software eingespielt und die Tests gestartet werden. Eventuell kann man auch diesen Prozess automatisieren, indem man die Virtualisierunglösung von der Continuous-Integration-Lösung her fernsteuert.

10.6 Projektplanung und Steuerung

Projektstrukturplan (PSP, engl. WBS)

Zum initialen Erfassen und Top-down-Verfeinern der Arbeitspakete ist eine einfache Tabellenkalkulation bzw. Textverarbeitung ausreichend, hier geht es nur um Arbeitspakete, also z. B. Zeilen in einer Tabelle, denen durch Einrückungen oder Nummerierung eine hierarchische Struktur verliehen wird. Anschließend wird jede „Zeile“ um zusätzliche Informationen, wie z. B. Aufwand oder Zuständigkeit, ergänzt. Das Ziel ist es, Arbeitspakete in kleinere Einheiten aufzuteilen.

GANTT-Project

Für die grobe Planung von Arbeiten bzw. Arbeitspaketen in einem Projekt ist es sinnvoll, ein Werkzeug wie etwa GANTT-Project[27] einzusetzen. Dies bietet sich vor allem am Projektanfang an. Die initiale Top-down-Verfeinerung der Arbeit innerhalb der Projektlaufzeit lässt sich mittels GANTT elegant realisieren, d. h. Projektanfang, Projektende sowie Zeiten, zu denen nicht gearbeitet wird, werden definiert, und in der resultierenden, verfügbaren Arbeitszeit werden Tätigkeiten eingeplant. So lässt sich von erfahrenen Projektmanagern früh feststellen, ob ein bestimmtes Projekt in einer bestimmten Zeitspanne mit definierten Ressourcen und Personal machbar ist. Die Aktualisierung des Diagramms bzw. der Arbeitspakete muss bei jedem Meeting manuell durch Statusabfragen des Projektmanagers durchgeführt werden.

[27] `http://www.ganttproject.biz`

dotProject

Bei kleinen Projekten sind GANTT-Project bzw. vergleichbare andere Werkzeuge ausreichend. Sobald jedoch viele Personen am Projekt beteiligt sind (bzw. mit fortschreitenden Projektverlauf bei mittelgroßen Projekten), kann es von Vorteil sein, die Informationen der Projektplanung automatisiert allen Teammitgliedern zur Verfügung zu stellen. Ebenso kann es wünschenswert sein, dass Teammitglieder selbst Änderungen an ihren Arbeitspaketen vornehmen bzw. ihre geleisteten Stunden selbst eintragen, um damit das Projektmanagement zu entlasten. Ein Beispiel für ein Open-Source-Software-Paket, das Projekt-Controlling bietet, ist dotProject[28]. Verwendet man diese Software, wird nicht direkt im GANTT-Diagramm gearbeitet, sondern Arbeitspakete mit ihren Metadaten in einer hierarchischen Struktur eingetragen und z. B. mit Stundenaufwand durch die Teammitglieder bewertet. Das resultierende GANTT wird aus diesen Informationen ausgerechnet und mittels eines Plugins grafisch dargestellt.

Weitere nützliche Werkzeuge für Projektplanung und Steuerung sind sogenannte *Ticketing-Systeme* wie etwa Trac oder Bugzilla (siehe Abschnitt 10.8). Die Vorgehensweise in der oben erwähnten Planung mittels eines Projektstrukturplans bzw. GANTT ist ein hierarchischer Top-down-Ansatz, während die Planung über Roadmaps und Tickets eher als Bottom-up zu sehen sind und z. B. Open-Source-Projekten sehr entgegenkommen. Anhand eines Projekt-Controlling-Werkzeugs wie Trac oder Bugzilla ist der „Ist-Stand“ gut ersichtlich, sowie die weitere Planung durch Auswertungen des vergangenen Projektverlaufs leicht möglich. Es macht auch Sinn, diese Werkzeuge während Besprechungen für Statusabfragen einzusetzen und auch direkt während Besprechungen Tickets zu modifizieren bzw. neue zu erstellen.

10.7 Dokumentation

Dokumentation gehört für die meisten Entwickler zu den am wenigsten beliebten Aufgaben. Dennoch sollte man sich im Klaren darüber sein, dass viele technisch gute Projekte an mangelhafter Dokumentation gescheitert sind oder wesentlich an Marktanteil verloren haben. Kann ein Anwender die Software nicht installieren, konfigurieren oder bedienen; versteht ein Software-Entwickler nicht, wie das Framework, das man entwickelt hat, funktioniert, oder findet die potenzielle Community bei einem Open-Source-Projekt eine chaotische Webseite, mangelhafte Sourcecode-Dokumentation und fehlende Architekturdokumente, so kann die Software technisch perfekt sein, die Kunden werden vermutlich ein anderes Produkt vorziehen. Auch aus Sicht des Managements und der Qualitätssicherung eines Projekts ist Dokumentation ebenfalls ein wesentlicher Faktor. Mangelnde

Management und Qualitätssicherung

[28] `http://www.dotproject.net/`

oder nicht aktuelle Dokumentation kann das Einführen neuer Mitarbeiter erheblich erschweren, sowie die Arbeit anderer Abteilungen wie Support und Marketing negativ beeinflussen. Ist die Webseite und die Handbücher (oder auch das Ticketing-System) nicht auf dem letzten Stand, weiß vielleicht die Marketingabteilung nichts davon, dass ein bestimmtes Feature, das Kunden immer wieder anfragen, schon längst implementiert ist, und die Support Abteilung muss direkt mit den Entwicklern interagieren, weil bestimmte Informationen in der Dokumentation fehlen.

Artefakte

Folgende Dokumentations-Artefakte liegen in einem Software-Projekt typischerweise vor und sollten auch immer parallel mit der laufenden Software-Entwicklung gewartet werden:

- Anforderungs- und Design-Dokumente,
- Sourcecode-Dokumentation,
- Projekt-Metadaten, z. B. URLs von SCM Repositories, Bug-Tracker, Projektdatenbanken, Mailinglisten, Wikis usw. (siehe z. B. Abschnitt 10.8),
- Protokolle von Besprechungen (siehe Abschnitt 10.8),
- Build-Report (siehe Abschnitt 10.3),
- Ergebnisse von automatisierten Tests (siehe Abschnitt 5.8.1),
- Berichte von manuellen Tests, Code-Inspektionen usw.,
- Ergebnisse von automatisierten Code-Quality-Checks und dergleichen,
- Handbücher oder Online-Dokumentationen für Installation, Konfiguration, Wartung,
- Handbücher oder Online-Dokumentationen für Entwickler, die auf dem eigenen System aufbauen,
- Handbücher oder Online-Dokumentationen für Benutzer und
- Projekt- und Produkt-Webseiten.

Kontext der Artefakte

Details sind natürlich vom konkreten Projekt abhängig. Ein Web-Framework wird vermutlich kaum „Benutzer“ im üblichen Sinne haben, denn das Framework ist ja ein Hilfssystem für andere Software-Entwickler. Außerdem ist zu beachten, dass die verschiedene Artefakte sehr unterschiedliche Zielgruppen haben. Anforderungs-Dokumente oder Test-Ergebnisse sind z. B. für die Software-Entwickler, Qualitätssicherer oder das Management wesentlich; Handbücher für Kunden oder Anwender.

Anforderungs-Dokumente

Alle Anwendungsfälle eines Software-Systems zusammen spiegeln die Gesamtfunktionalität und alle funktionalen Anforderungen des Systems wieder. Durch die Anwendungsfallbeschreibungen können detailliertere Informationen und auch nicht funktionale Beschränkungen (*nonfunctional constraints*) dokumentiert werden. Zur Dokumentation von Anforderungen gibt es Alternativen zum einfachen „Prosatext“ mithilfe von UML, wobei bei Artefakten der Analysephase sogenannte User Stories sehr hilfreich sind, um das Verhalten des fertiggestellten Systems bei Inbetriebnahme zu vermitteln und bereits frühzeitig mögliche Abnahme-Tests zu erstellen.

User Stories

UML-Dokumentation

Schon in kleinen bis mittelgroßen Projekten wird der Umfang von Dokumentation in der UML sehr groß und es gibt zahlreiche Werkzeuge für die Modellierung. Bei einer Unterteilung der Features und damit dem Einsatz in der Entwicklung fallen diese Werkzeuge grundsätzlich in drei Kategorien. Um die Diagramme der UML zu zeichnen, ohne gegen das UML-Metamodell zu verifizieren, ist ein gutes Zeichenprogramm mit vorgefertigten UML-Paletten vorteilhaft (UMLet[29], OmniGraffle, Visio). Die einzelnen Modellelemente können jedoch nur manuell in anderen Diagrammen wiederverwendet werden, und der Aufwand, Änderungen nachzuziehen, wird sehr schnell unhandhabbar.

Zeichenprogramme

Ein Modell, viele Sichten

Aushilfe bieten Werkzeuge, die zwischen Modell und Diagram unterscheiden bzw. das Modell in XMI[30] speichern und eine Benutzerschnittstelle zum Zeichnen der Diagramme anbieten. Im Open-Source-Umfeld könnte man *BOUML*[31] oder *ArgoUML* nennen, kommerziell *Enterprise Architect*, *Visual Paradigm* oder *Rational Rose*.

Die Daten der UML-Werkzeuge sollten im SCM-System abgelegt und damit versioniert sein; beim Export bzw. bei der Einbindung in die Dokumentation der UML-Diagramme ist es am sinnvollsten, die Diagramme als PDF, SVG bzw. Vektor Grafik zu exportieren. Diese Abbildungen können in die Projektdokumentation eingebunden werden. Die Dokumentation in Textform, etwa Beschreibungen der Anwendungsfälle, können im jeweiligen Dokumentationswerkzeug festgehalten werden, in welchem auch die Diagramme zur Präsentation gesammelt werden. Der Austausch von Daten zwischen den verschiedenen Werkzeugen ist leider meist nicht auf einfachem Wege möglich. Fortgeschrittene Werkzeuge bieten die Eingabe von implementationsspezifischen Informationen an, etwa die JavaDoc-Beschreibung von Klassen.

Model-Driven-Architecture

Unter dem Prinzip von Model-Driven-Architecture (MDA, siehe Abschnitt 7.2) können auch werkzeugunterstützt Codeteile bzw. Vorlagen

[29] `http://www.umlet.com/`

[30] XML Metadata Interchange; eine XML-Sprache, die dem Austausch von UML-Modellen und anderen Metadaten zwischen Software-Entwicklungswerkzeugen dient.

[31] `http://bouml.free.fr`

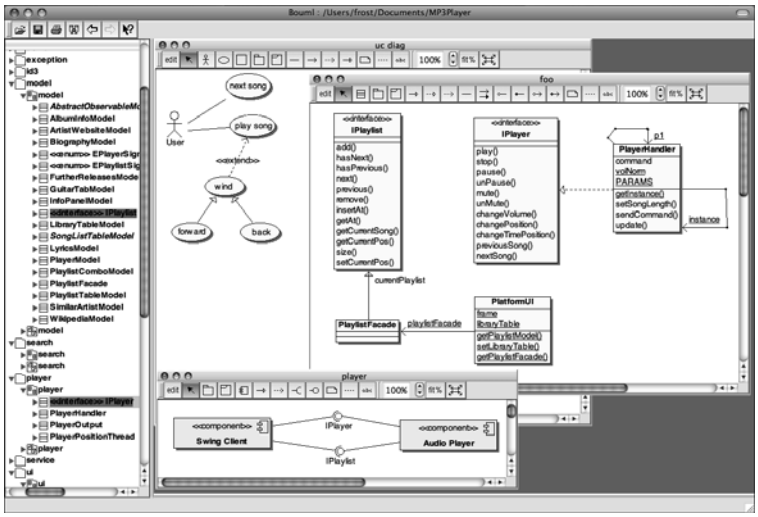


Abbildung 10.5 *Beispiel BOUML-UML-Werkzeug: die Modelelemente werden links als Liste dargestellt und können per drag&drop in das Klassendiagramm eingefügt werden.*

(*templates*) in verschiedenen Sprachen generiert werden. Gute Werkzeuge unterstützen meist auch die Darstellung von bestehendem Code als UML-Diagramm (*reverse-engineering*).

Synchronisation von Modell und Implementierung

Es kann nicht immer nach dem MDA-Paradigma entwickelt werden; viel wahrscheinlicher ist die Dokumentation von bestehendem Sourcecode oder besser die parallele Entwicklung von Modell und Implementation in einem agilen Prozess. Daher werden ausgereifte, meist kostenpflichtige Werkzeuge eingesetzt, um Dokumentation und Implementation regelmäßig abzustimmen bzw. im Nachhinein zu dokumentieren. Dabei sollte jedoch vorsichtig umgegangen werden, denn die Reverse-Engineering-Funktionalität ist zwar meist sehr effektiv, jedoch muss der Entwickler noch immer entscheiden, welche Sichten (*views*) auf die Software dargestellt werden. Während also das Modell einer bestehenden Software bereits ziemlich gut aus Sourcecode heraus geparst werden kann, müssen die Diagramme bzw. Sichten auf das System jedoch noch immer manuell erstellt bzw. nachbearbeitet werden, um übersichtlich zu sein. Das anschaulichste Beispiel hierfür sind UML-Klassendiagramme, die sehr rasch ziemlich unübersichtlich werden können, insbesondere wenn alle Attribute und Methoden eingezeichnet werden – etwas, was in der Regel nicht notwendig sein sollte.

Modelle in verschiedenen Diagrammen

Da also, egal in welcher Domäne, viele verschiedene Modellelemente in mehreren Diagrammen und Diagramm-Typen verwendet werden, ist es in erster Linie wichtig, ein Werkzeug zu verwenden, welches vorgefertigte Paletten für gängige UML-Diagramm-Typen unterstützt. Bei der Vorgehensweise sollte es jedoch nur ein Modell geben, die Diagramme sollten Abbildungen (Sichten) auf das Software-System sein. Dies bedeutet, dass ein Modellelement mit gleichem Namen in zwei verschiedenen Diagrammen auf dasselbe Element im UML-Modell verweisen muss. Dadurch wächst der Aufwand beim Modellieren nicht übermäßig an, da aus einer Liste, dem „Modell“, Elemente wiederverwendet werden können. Die meisten Werk-

zeuge unterstützen dies per Drag&Drop. Um die Übersicht zu behalten, wird also empfohlen ein UML-Werkzeug zu verwenden, das es erlaubt, dieselben Modell-Objekte in mehren Diagrammtypen zu verwenden (wie in Abbildung 10.5 am Beispiel des BOUML Open-Source-Werkzeugs, zu sehen ist, z. B. „IPlaylist“ und „IPlayer“).

Sourcecode-Dokumentation

Die Sourcecode-Dokumentation richtet sich an Entwickler und dokumentiert die Klassen, Methoden, Interfaces sowie komplexere Algorithmen. Für jede Programmiersprache gibt es hier eigene Dokumentations-Richtlinien, die man einhalten sollte. In Java ist dies der Javadoc-Standard[32]. Eine Klasse und eine Methode werden beispielsweise wie folgt dokumentiert:

```
1  /**
2   * This class can be used for importing and
        exporting xml-files..
3   * @author Max Meier
4   * @version 2.3
5   * @see Export
6   * @see Import
7   * @see <a href="http://dom4j.org" target="newWindow
        ">dom4j-website</a>
8   */
9  public class XmlExportImport implements Export,
       Import {
10   ...
11   /**
12    * This method reads a List of Students from a xml
          -file...
13    * @param filename The name of the file to read.
14    * @return list of students
15    * @see #readXml(String)
16    */
17   public List<Student> read(String filename) throws
         IOException { ...
18   }
19 }
```

Im Beispiel erkennt man, dass die Funktion der Klasse kurz beschrieben, sowie die Autoren genannt werden. Gegebenenfalls können Querverweise zu anderen Klassen oder Resourcen angegeben werden. Die Methode in Zeile 19 wird ebenfalls kurz beschrieben. Anzugeben sind hier auch die Übergabeparameter sowie der Rückgabewert der Methode. Verwendet man einen Standard wie Javadoc für die Dokumentation des Sourcecode kann man sich im Rahmen der Build-Automatisierung (siehe Abschnitt 10.3) auch automatisch eine HTML-Version der API-Dokumentation erstellen lassen. Abbildung 10.6 zeigt die zum obigen Beispiel passende Klassen-Dokumentation, Abbildung 10.7 die der Methode. Mit dieser Vorgehens-

[32] Details zu Javdoc findet man unter: `http://java.sun.com/j2se/javadoc`. Andere Sprachen haben vergleichbare Dokumentationssysteme.

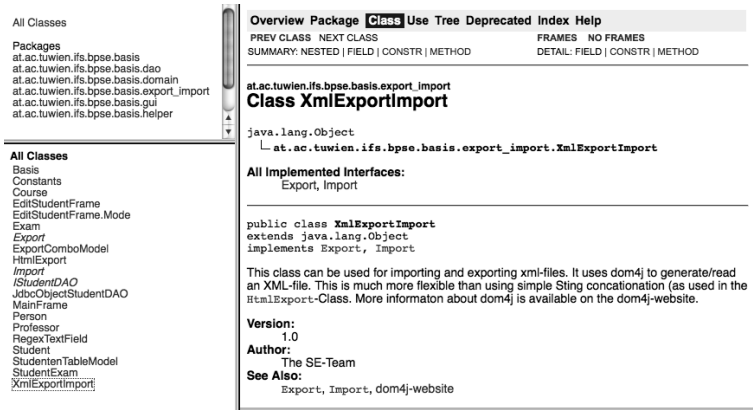


Abbildung 10.6
Javadoc HTML-Dokumentation einer Java-Klasse.

weise ist es viel einfacher, die API-Dokumentation immer auf dem Laufenden zu halten, weil Änderungen am Code auch sofort an derselben Stelle im Code dokumentiert werden können, und nicht erst in einer externen Dokumentation nachgezogen werden müssen.

Projekt-Metadaten

Um die Übersicht über die vielen an einer Software-Entwicklung beteiligten Personen und Resourcen nicht zu verlieren, sollten an zentraler Stelle auch „Projekt-Metadaten“ gesammelt werden: Dazu zählen Projektname, -version und -webseite, Entwickler mit Kontaktinformation (bei Open-Source-Projekten besonders wichtig), URLs zu verschiedenen Informationssystemen wie SCM, Issue Tracker, Datenbanken usw. Build-Systeme wie Maven bieten die Möglichkeit, derartige Informationen in der Projekt-Datei abzulegen. Bei der Dokumentations-Generierung werden entsprechende HTML-Dokumente erstellt.

Berichte von Tests, Build usw.

Während der Software-Entwicklung fallen zumeist eine Vielzahl an Berichten an, häufig angestoßen durch Build-Automatisierung oder das Continuous-Integration-System (siehe Abschnitt 10.3). Dazu zählen unter anderem der Build-Report (z. B. Fehlerberichte des Compilers), Ergebnisse von automatisierten Tests und Ergebnisse von automatisierten Code-Quali-

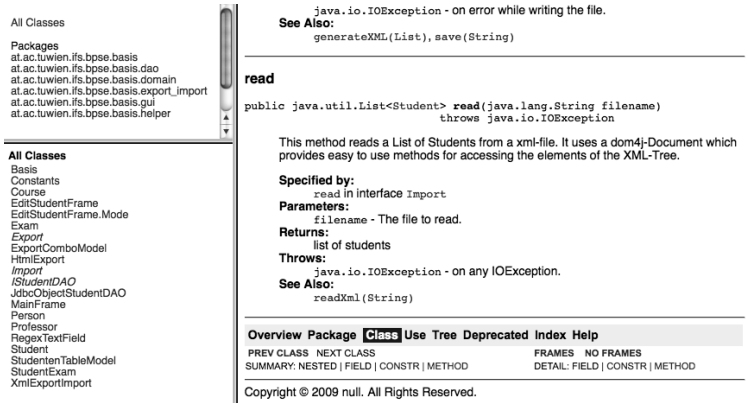


Abbildung 10.7
Javadoc HTML-Dokumentation einer Java-Methode.

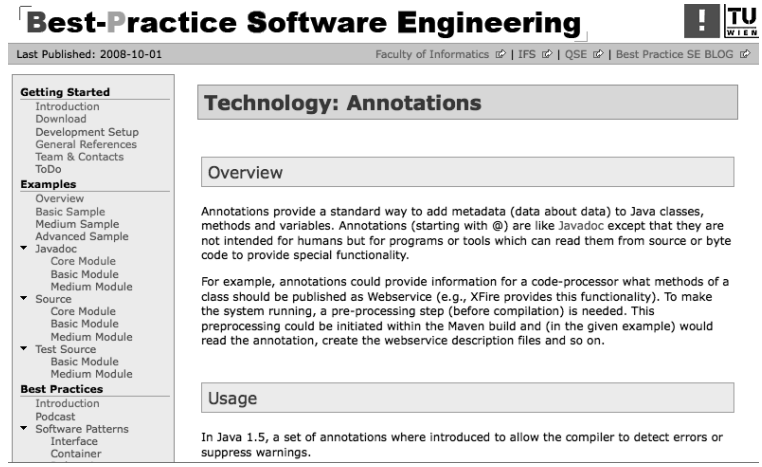


Abbildung 10.8 *Beispiel einer Dokumentation erstellt mit Maven Site. Verschiedene Artefakte wie API-Dokumentation, Test-Ergebnisse und Benutzer-Dokumentation werden automatisch in einer Webseite integriert.*

ty-Checks. Die einzelnen erzeugten Reports sollten für die Projekt-Dokumentation noch zusammengeführt, in ein „menschenlesbares“ Format (etwa HTML oder PDF) gebracht und zur Projektdokumentation hinzugefügt werden.

Besprechungs-Protokolle

Protokolle von Besprechungen sollten an einer zentralen Stelle, z. B. im SCM-System jedem zugänglich abgelegt werden, da sie wichtige Entscheidungen und Beschlüsse bezüglich des Projekts enthalten. Protokolle sollten in einem plattformunabhängigen Format, also z. B. als Plain-Text oder PDF abgelegt werden. Mehr zur Vor- und Nachbereitung von Besprechungen und zum Verfassen von Protokollen findet sich in Abschnitt 10.8.

Versionierung

Weiterhin ist zu beachten, dass für die genannten Dokumentations-Artefakte natürlich dieselben Standards, z. B. hinsichtlich Versionierung, gelten wie für den Sourcecode selbst (siehe auch Abschnitt 10.2). Daher macht es Sinn, Dokumentation (natürlich außer Reports von Tests, Builds usw.) ebenfalls im SCM-System zu verwalten und damit demselben Versionsmanagement wie für dem Sourcecode zu unterwerfen. Damit hat man zur jeweiligen Software-Version oder zum jeweiligen Branch auch die passende Dokumentation.

Benutzer-dokumentation

Einige der bisherigen Dokumentations-Artefakte werden automatisch erzeugt, z. B. als Ergebnisse von Tests, bzw. „halb-automatisch“ wie die API-Dokumentation, oder es war spezielle Software für die Erstellung erforderlich (z. B. für die Anforderungs- und Design-Dokumente). Zuletzt sollte aber auch die Benutzer-Dokumentation („Handbücher“) sowie die Webseiten nicht vergessen werden. Diese können natürlich mit gängiger Office-Software erstellt werden. Dies bewährt sich in der Praxis aber selten. Office-Dokumente sind Binär-Formate und können damit nicht leicht die volle Leistungsfähigkeit des Sourcecode-Management-Systems nutzen. Auch die Teamarbeit an größeren Dokumenten gepaart mit der Fehleranfälligkeit vieler Office Pakete (vor allen Dingen bei Mengesatz) machen die Dokumentations-Erstellung problematisch und unzuverlässig. Außer-

dem integrieren sich Office-Pakete meist nicht gut in Build-Systeme und sind schlecht portabel.

Daher bieten sich in vielen Fällen besser in die Build-Automatisierung integrierte Systeme an. Apache Maven beispielweise bietet ein eigenes Dokumentations-System zum Erstellen einer Web-Dokumentation an[33]. Beliebt sind in einigen Projekten auch Systeme wie Docbook[34]. Dabei handelt es sich um einen offenen Standard, in dem die Dokumente in XML (oder SGML) geschrieben werden und sich mit entsprechenden Prozessoren in Formate wie HTML oder PDF umsetzen lassen. Auch LaTeXist sehr gut geeignet, vor allem dann, wenn gedruckte Dokumente zu erstellen sind. Diese Systeme sind gut geeignet, um auch im Team große Dokumentationen zu erstellen, zu versionieren und zu verwalten[35].

Integration in CI

Die Generierung, Zusammenfassung und Integration der verschiedenen Dokumentationsartefakte erfolgt am besten automatisiert, z. B. im Rahmen der Continuous-Integration. Damit hat man nicht nur einen regelmäßigen (täglichen) Build der Software, sondern auch automatisch die dazugehörige und integrierte Dokumentation, z. B. im HTML-Format. Verwendet man für die Benutzerdokumente Systeme wie Maven oder Docbook, kann man auch die Erstellung dieser Dokumentation recht einfach in die Build-Automatisierung integrieren. Das Ergebnis des Builds sind dann nicht nur ein Software-Paket, sondern auch die zugehörige Webseite, PDF-Handbücher, Test-Ergebnisse usw. Abbildung 10.8 zeigt die Homepage der Best-Practice-SE-Beispiele: Mithilfe von Maven wurden hier verschiedene Dokumentations-Artefakte automatisch in eine Webseite integriert, und diese kann automatisch bei jedem Build erstellt werden.

10.8 Kommunikation im (global verteilten) Team

Von Anfang bis Ende der gesamten Projektdauer findet Offline- und Online-Kommunikation statt. Bereits gebräuchliche elektronische Wege der Kommunikation per E-Mail, Chat und Telefon werden von einer Seite durch Projektmanagement-Werkzeuge, wie Online-Dokumentation per

[33] `http://maven.apache.org/maven-1.x/plugins/site`

[34] `http://www.docbook.org`

[35] Alternativ werden auch gerne Content-Management-Systeme (CMS) vor allem für Projekt-Webseiten eingesetzt. Diese haben allerdings gegenüber den vorher erwähnten Systemen den Nachteil, dass sie zumeist einen Server voraussetzen. Die Dokumentationen, die mit Maven oder Docbook erstellt werden, erzeugen statische Webseiten, die sowohl online als auch offline leicht verwendet werden können. Auch die Dokumentation für verschiedene Software-Versionen ist mit CMS-Systemen nicht immer ganz einfach zu verwalten.

Wiki, und technische Entwicklungswerkzeuge wie Bug-Issuetracking ergänzt. Obwohl die Anzahl der Hilfsmittel groß ist und dies die Kommunikation und Dokumentation fördert, sollte auch bei der Auswahl der Werkzeuge strukturiert vorgegangen und das gesamte Projekt betrachtet werden. Nach Möglichkeit sollten die Informationen von einem Werkzeug als Input-Daten für ein anderes dienen. Beispielsweise können TODO-Items die bei einer Team-Besprechung ermittelt werden, direkt in die jeweilige Stundenliste des dafür zuständigen Teammitglieds überfließen, wenn Sie elektronisch erfasst werden.

Skype, Chat

Ad-hoc-Kommunikation per Internet sollte manuell in einem Besprechungsprotokoll bzw. Wikieintrag zusammengefasst werden, sofern technische Design- bzw. Projektmanagement-Entscheidungen getroffen wurden. Das Protokoll sollte jedem Teammitglied zugänglich sein, und es sollte aktiv darauf hingewiesen werden, z. B. per Mailingliste. Das Einrichten von E-Mail bzw. Mailinglisten sollte so früh wie möglich im Projektverlauf passieren.

Wiki

Ein Wiki ist für die Sammlung und Präsentation von Ideen sehr hilfreich, insbesondere in den Anfangsphasen ist dies ein sehr hilfreiches Werkzeug für Teamkommunikation. Auf einem Wiki ist es nur schwer möglich, umfangreichere Projektmanagement-Aufwände zu betreiben, der Hauptvorteil liegt darin, dass jedes Teammitglied unstrukturiert Änderungen machen kann.

Dokumentation

Die TODO-Items, die bei Besprechungen erarbeitet werden, sollten, sofern möglich, sofort zu Aufgaben im Entwicklungsprozess überfließen, dazu bieten System wie Bugzilla und Trac bewährte Möglichkeiten, um Change Requests im System zu dokumentieren und priorisieren. Das Erstellen eines Bugs oder Issues in solchen Systemen bedeutet das Zuteilen von Arbeit im Entwicklungsprozess.

Mylyn

Die Einbindung der Issues von einem Bugtrackingsystem in die technische Entwicklung sollte, sofern möglich, automatisch erfolgen. Als Input dienen hier nicht nur die Issues von Besprechungen bzw. Bugs, sondern auch die Tasks des Projektplans, also verfeinerte Arbeitspakete der Work Breakdown Structure. Der Output, eine aufbereitete Version der abgehandelten Tasks, sollte in das SCM einfließen. Davon sind vor allem Versionierungssysteme und Stundenlisten betroffen. Manche Entwicklungsumgebungen, wie z. B. Eclipse mit Mylyn, bieten durch ihre Pluginstruktur die Integration von mehreren externen Quellen an einer zentralen Stelle in der IDE an. Mit Mylyn kann also Taskverwaltung, Entwicklung und Dokumentation an einer Stelle aufbereitet werden. Die Issues aus Trac werden in der TODO-Liste dargestellt und durch den Entwickler mit Klassen verknüpft. Beim Einchecken mit SVN werden die Metainformationen des Tasks automatisch als Commit-Message übernommen.

In diesem Kapitel wurden bereits verschiedene Werkzeuge wie Sourcecode-Management-Systeme, Continuous-Integration-Systeme oder Doku-

mentations-Werkzeuge vorgestellt, die zum Teil auch die Arbeit und Kommunikation im Team erleichtern oder überhaupt erst ermöglichen. Daneben gibt es aber noch eine Vielzahl an weiteren Werkzeugen und Techniken, die sich zum überwiegenden Teil aus der Open-Source-Community heraus entwickelt haben. Dort hat sich die Notwendigkeit nach leistungsfähigen Mechanismen, um auch in großen Projekten über die ganze Welt verteilt arbeiten zu können, sehr früh herauskristallisiert. Mittlerweile haben viele der in der Open-Source-Entwicklung verwendeten Werkzeuge aber auch Eingang in die kommerzielle Software-Entwicklung gefunden. In diesem Abschnitt werden einige Werkzeuge und Vorgehensweisen, die die Kommunikation in (global) verteilten Teams, wie z. B. in Open-Source-Projekten ermöglichen, die aber richtig eingesetzt auch lokale Arbeitsgruppen gut in der Arbeit unterstützen können. Es ist aber sehr wichtig zu verstehen, dass alle genannten Werkzeuge nur Hilfsmittel sind. Verteilte Entwicklung erfordert ein hohes Maß an Anpassung an den Prozess und an sorgfältige und nachvollziehbare Arbeitsweise aller beteiligten Entwickler.

Open-Source-Community

Zunächst muss man zwischen Werkzeugen unterscheiden, die asynchron arbeiten, und solchen, die synchrone Kommunikation erlauben. Synchrone Kommunikation ist beispielsweise Telefonie, Chat, Voice over IP (VoIP) und natürlich persönliche Treffen und Konferenzen. Synchrone Kommunikation wird natürlich in der Software-Entwicklung eingesetzt: Persönliche Treffen oder Konferenzen sind fallweise wichtig und auch in Open-Source-Projekten üblich, aber dennoch stellt synchrone Kommunikation bei verteilten Projekten natürlich eher die Ausnahme dar. Dafür gibt es verschiedene Gründe: Reisen sind teuer und kosten allen Beteiligten viel Zeit, werden also nur fallweise stattfinden. Telefonie, Konferenzgespräche, VoIP und Videokonferenzen können eine gute Ergänzung darstellen, haben aber den Nachteil, dass sie schwer dokumentierbar und archivierbar sind. Manchmal werden für jede derartige Kommunikation Protokolle verfasst, um alle anderen auf dem Laufenden zu halten. Erfahrungsgemäß funktioniert dies aber nicht sehr gut. Oftmals werden die Protokolle kaum je gelesen. Auch ist eine spätere Interaktion oder Diskussion, die auf derartigen Gesprächen aufbauen, schwer möglich.

Synchrone Kommunikation

Konferenzgespräche

Text-basierte Chats empfinden manche als mühsames Kommunikationsmittel. Tatsächlich haben Chats aber einige Vorteile: Sie sind einerseits nicht ganz so synchron wie ein Telefongespräch, denn ein Chat kann sich eventuell auch über Stunden ziehen, neue Gesprächspartner können auch später einsteigen (und auch die bisherige Diskussion nachverfolgen). Zudem sind Chats technisch nicht aufwendig und daher leicht umzusetzen. Sie bieten nicht zuletzt die Möglichkeit an, das Chatprotokoll leicht zu archivieren und zu verteilen. Dennoch sind auch Chats eher die Ausnahme und wie andere synchrone Verfahren auch besser geeignet, wenn *wenige* Beteiligte konkrete Probleme, die nicht unbedingt die Allgemeinheit betreffen, diskutieren wollen. Chats werden manchmal auch im Support, also mit Kunden oder der Community verwendet.

Chat

Vor- und Nachbereitung von Besprechungen

Im Zusammenhang mit persönlichen Treffen, aber auch mit Diskussionen via VoIP, Chat usw. ist es sehr wichtig, diese vorzubereiten sowie während des Treffens ein kurzes Protokoll zu schreiben. Einladung und Protokoll sind über Mailingliste, E-Mail, Kalender o.ä. zu verbreiten. Das Verfassen von Protokollen klingt vielleicht für manche Ohren etwas „verstaubt“, erweist sich aber in der Praxis bei richtiger Handhabung als sehr wesentlich, um greifbare Ergebnisse von Besprechungen zu erhalten und eine gemeinsame Sicht auf das besprochene Thema zu entwickeln. Schlecht vor- und nachbereitete Besprechungen werden schnell zur Zeitverschwendung und zur Qual für alle Beteiligten.

Vorbereitung und Einladung

Lädt man zu einer Besprechung ein, und dies gilt besonders, wenn mehrere Personen eingeladen werden, so ist diese inhaltlich vorzubereiten. In der Einladung sollten sich wenigstens folgende Informationen finden:

- *Wann* findet das Treffen statt?
- *Wo* bzw. *wie* wird das Treffen abgehalten (persönlich, Telefonkonferenz, ...)?
- *Wer* ist eingeladen?
- Liste der zu besprechenden *Themen*, eventuell ergänzende Information mitsenden.

Protokoll

Während der Besprechung wird zunächst das Protokoll der letzten Besprechung durchgesehen und dort erwähnte offene Punkte behandelt. Dann werden die geplanten Themen besprochen. Ein Protokollführer schreibt das Protokoll. Ein Besprechungsprotokoll soll keinesfalls die gesamte Besprechung „Nacherzählen“, sondern sich auf eine möglichst kurze Zusammenfassung der wesentlichsten Punkte beschränken und im Wesentlichen folgende Fragen beantworten:

- *Wann* hat die Besprechung *wo* bzw. *in welcher Form* stattgefunden?
- Liste der *Teilnehmer*.
- Wer hat das Protokoll verfasst?
- Liste der *Themen*, die besprochen wurden, mit einer Zusammenfassung wichtiger Erkenntnisse oder Neuigkeiten.
- Welche *Entscheidungen* wurden getroffen?
- Welche *Aufgaben* wurden an *wen* verteilt?
- Eventuell: Termin des *nächsten Treffens*.

Aufgabe des Protokollführers

Ein Besprechungsprotokoll sollte also so kurz, prägnant und sachlich wie möglich gehalten werden. Knappe Aufzählungen sind langatmigen Formulierungen und Prosa-Text vorzuziehen; persönliche Kommentare sind nicht gefragt. Typischerweise sollte ein Protokoll nicht viel mehr als 1-3 Seiten umfassen. Die Strukturierung des Protokolls richtet sich nach der obigen Liste, sowie nach den besprochenen Themen. Verteilte Aufgaben sollten getrennt zusammengefasst und nicht über das ganze Protokoll verteilt werden. Verfügt man über ein Issue-Tracking-System, so sollte der Protokollführer die besprochenen Aufgaben („TODOs“) nach der Besprechung auch gleich in dieses System eintragen.

Da es heute oft schwierig ist, gemeinsame Termine für mehrere Personen zu finden, empfiehlt es sich, den Termin des nächsten Treffens noch während des Treffens zu vereinbaren und ebenfalls im Protokoll zu vermerken. Existiert ein gemeinsamer Kalender, so ist dieser Termin vom Protokollführer auch dort einzutragen.

Das Protokoll wird dann an alle, die an einem Projekt oder Thema interessiert oder davon betroffen sind, verteilt. Dafür bieten sich Mailinglisten an. Wichtig ist natürlich, dass verteilte Aufgaben auch entsprechend den Betroffenen kenntlich gemacht werden, auch (oder gerade) wenn diese nicht anwesend waren.

Aufgabe der Teilnehmer

Alle an dem Treffen beteiligten sollten dieses Protokoll lesen und prüfen, ob es ihrem eigenen Verständnis (z. B. in Hinsicht auf getroffene Vereinbarungen und Beschlüsse) entspricht und gegebenenfalls schnell Änderungen anregen. Wird nicht widersprochen, gilt das Protokoll als Vereinbarung aller an der Besprechung beteiligten.

Fallweise bieten sich Protokolle auch als Zusammenfassung von länger dauernden, asynchronen Diskussionen (s.u.) an, vor allem, wenn diese nicht mehr leicht überschaubar oder nachvollziehbar sind.

Protokoll Ablage

Für die Nachvollziehbarkeit sowie auch für neue Mitarbeiter ist sehr wichtig, dass Protokolle in einem zentralen System abgelegt werden und allen Beteiligten leicht zugänglich sind. Protokolle sind Teil der Projektdokumentation (siehe auch Abschnitt 10.7). Zur Speicherung bietet sich daher ein meist schon vorhandenes SCM-System an.

Asynchrone Kommunikation

Die dominierende Form der Kommunikation in (global) verteilten Projekten ist asynchron. Kommunikation über E-Mail ist natürlich eine naheliegende Möglichkeit, die aber einige Probleme bzw. Risiken birgt: Kommunikation per E-Mail zwischen wenigen Partnern ist oft schwer nachzuverfolgen, neue Gruppenmitglieder können auf private Kommunikation zwischen anderen natürlich nicht (mehr) zugreifen, und Projekt-E-Mails stören häufig den restlichen E-Mailverkehr. Daher werden von nahezu allen Open-Source-Projekten Mailinglisten für verschiedene Zwecke angelegt. Meist werden zwei bis drei Listen pro Projekt verwendet: `developer@projektname.domain`, `user@projektname.domain` und manchmal noch `announce@projektname.domain`.

Mailinglisten

Alle projektrelevanten Diskussionen finden in den ersten beiden Mailinglisten statt. In der Developer-Liste werden Themen diskutiert, die die konkrete Entwicklung des Projekts betreffen. In der User-Liste wird mit Anwendern des Systems diskutiert; sie hat also zum Teil eine Support-Funktion. Die Announce-Liste wird manchmal für Ankündigungen verwendet, z. B. bei neuen Software-Versionen. Manchmal postet auch das Continuous-Integration-System die Ergebnisse des Daily-builds auf dieser Mailingliste. Der Vorteil von Mailinglisten für die Kommunikation ist recht offensichtlich:

- Mit einem E-Mail an die entsprechende Adresse erreicht man automatisch alle Interessenten.
- Der Server für die Mailingliste kümmert sich um An- und Abmeldungen, eventuell auch ein entsprechender Administrator.
- Mails der Mailingliste lassen sich mit Filterregeln im E-Mail-Client leicht in eigene Ordner filtern.
- Diskussionen können über Threading-Funktionalität im E-Mail-Client ebenfalls leicht verfolgt werden.
- Über Mailinglist-Archive können auch alte Diskussionen über Web-basierte Suchen zugänglich gemacht werden.
- Auch neue Mitglieder im Team oder Interessenten am Projekt können sich somit leicht einen Überblick über den aktuellen Status verschaffen.

Problemlösung und Nettiquette

An dieser Stelle sollte vielleicht ein kleiner Hinweis zum „korrekten“ Verhalten für neue Benutzer eines bestimmten Systems gegeben werden, die ein Problem lösen möchten: Viele Probleme treten öfter auf, daher ist es aus verständlichen Gründen für die Community nicht sehr angenehm, wenn in die Mailinglisten immer wieder dieselben Fragen gepostet werden. Man sollte daher bei der Informationssuche als Benutzer folgende Schritte wählen:

1. Abonnieren der User-Mailingliste.
2. Gründliche Suche auf der Webseite des Projekts sowie der offiziellen Dokumentation.
3. Suche mit Internet-Suchmaschinen, häufig werden Probleme auch in anderen Internet-Foren diskutiert.
4. Suche in der User-, eventuell auch Developer-Mailingliste des Projekts.
5. Erst wenn sich durch diese Schritte das Problem nicht lösen lässt, sollte man in die User-Mailingliste posten.

6. Liest man Fragen in der User-Liste, die man selbst beantworten kann, so sollte man dies auch tun.

Als Benutzer sollte man aber keinesfalls in die Developer-Liste posten, denn die Entwickler lesen meist auch die User-Liste mit. Es ist auch wichtig, immer vollständige Problembeschreibungen, eventuell mit Beispiel zu liefern, ebenso den Kontext zu beschreiben (Betriebssystem, Software-Version usw.) und gegebenenfalls Fehlermeldungen mitzuschicken. Keine Binaries posten! Screenshots und dergleichen lädt man auf einen Server hoch und sendet den Link im E-Mail mit.

E-Mails?

In diesem Zusammenhang sollte vielleicht noch kurz auf E-Mail bzw. das korrekte Verfassen von Mails an Mailinglisten eingegangen werden. Das Schreiben von E-Mails ist heute so selbstverständlich, dass es kaum mehr einer weiteren Betrachtung wert zu sein scheint. Das E-Mail-Volumen hat aber ein solches Ausmaß erreicht, dass jeder (Entwickler) dieses an sich sehr mächtige Werkzeug mit großer Sorgfalt einsetzen sollte. Zunächst sollte man sich die Frage stellen, was man mit einem E-Mail erreichen möchte: Soll der Adressat nur über einen Sachverhalt *informiert* werden? Erwartet man sich *konkrete Aktivitäten* vom anderen, oder ist es eine *Antwort* in einer längeren Diskussion (z. B. auf einer Mailingliste)? Erwartet man sich konkrete Aktivitäten von einem der Adressaten, so ist dies entsprechend klar herauszustreichen, besonders dann, wenn das E-Mail an mehrere Personen geht! Soll eine Person etwas aktiv tun, andere nur informiert werden, so sollte man einerseits die Adressierung korrekt durchführen (TO: und CC: verwenden) sowie dies im E-Mail kenntlich machen. Ist das E-Mail eine Antwort auf ein längeres E-Mail eines anderen oder Teil einer längeren Diskussion der Mailingliste, so sollte man nur den Teil der vorigen Kommunikation als Zitat im E-Mail belassen, der für das unmittelbare Verständnis der eigenen Antwort wesentlich ist und nicht etwa immer die gesamte Kommunikation zitieren (dasselbe gilt natürlich für Kommunikation in Diskussionsforen, s.u.).

Diskussions-Foren

Eine Alternative zu Mailinglisten ist die Verwendung von Diskussions-Foren. Im Prinzip ist dies mit Mailinglisten vergleichbar, wenn nicht gar identisch. Die meisten Foren-Systeme bieten die Möglichkeit an, per E-Mail-Beiträge zu senden, neue Beiträge per E-Mail zu erhalten sowie sich um An- und Abmeldung von neuen Nutzern zu kümmern. Ob man Foren-Software oder Mailinglisten bevorzugt ist eher Geschmacksache. Große Open-Source Communities wie beispielsweise Sourceforge[36] bieten Projekten daher auch beide Möglichkeiten an. Man sollte sich aber frühzeitig für eine Strategie entscheiden, denn parallele Diskussionen in zwei Systemen sind natürlich kontraproduktiv!

[36] `http://www.sourceforge.net`

Wiki

Seit einigen Jahren werden in der Software-Entwicklung auch gerne Wikis eingesetzt. Bei allen Informationssystemen, die man einführt – und das gilt besonders für Wikis – sollte man das Einsatzgebiet klar definieren. Wikis passen recht gut, wenn es darum geht, kollaborativ an inhaltlich relativ eng abgegrenzten Dokumenten zu arbeiten (z. B. für FAQs oder Entwürfe von Projektaufträgen und dergleichen). Die Erstellung von größeren zusammenhängenden Texten (z. B. Handbüchern) funktioniert meist nicht sehr gut. Hier sollte man klare Abgrenzungen ziehen, welche Artefakte mit welchem System erstellt werden, um Doppelgleisigkeiten bzw. verweiste Wikis zu vermeiden.

Issue-Tracking

Ein weiteres, für die Software-Entwicklung zentrales Informationssystem ist der Bug- oder Issue-Tracker. Die Aufgabe eines solchen Systems ist es, Fehler im Programm, Änderungswünsche aber auch Roadmaps in einer nachvollziehbaren Weise zu verwalten. Mailinglisten, Foren usw. sind gut geeignet, um Diskussionen abzuhalten, aber wenig geeignet, um Aufgaben zu verteilen oder deren Erledigung zu überwachen.

Kunden und Benutzer-Interaktion

Ein Issue-Tracker sollte allen an einem Projekt beteiligten – aber idealerweise auch den Nutzern und Kunden offen stehen. Probleme, Bug-Reports, aber auch Änderungswünsche können dann im System erfasst und verantwortlichen Personen (z. B. Entwicklern) zugewiesen werden. Der Issue-Tracker erlaubt dabei Diskussionen um Issues, Statusänderungen (öffnen, schließen, wiedereröffnen, löschen usw.) sowie die Nachvollziehbarkeit, wann welches Issue oder welcher Bug von wem eingetragen, gefixed, wieder geöffnet oder endgültig geschlossen wurde.

Trac und Bugzilla

Ticketing-Werkzeuge, die Projektplanung und Steuerung in einem kollaborativen Stil ermöglichen und auch größeren Bezug zum Entwicklungsprozess haben, sind beispielsweise die Open-Source-Werkzeuge Trac[37] und Bugzilla[38].

Diese Werkzeuge haben sich in der Entwicklung bei mittelgroßen Software-Projekten bewährt, müssen jedoch systematisch von allen Teammitgliedern verwendet werden, damit daraus wirklich Nutzen gezogen werden kann. In Bezug auf die Projektplanung bieten diese Werkzeuge keine hierarchische Struktur der Arbeitspakete bzw. Tickets an. Tickets haben den Charakter einer „endlosen ToDo-Liste“, die allerdings nach Projekten, Modulen, Verantwortlichkeiten sowie in Roadmaps organisiert werden können.

Integration mit SCM

Moderne Systeme wie Trac oder Jira[39] können auch mit dem SCM-System integriert werden. Dies bietet im Wesentlichen zwei Vorteile: (1) Ein Entwickler, der einen Fehler behebt, kann beim *Commit* im SCM die Fehler-

[37] `http://trac.edgewall.org`
[38] `http://www.bugzilla.org/`
[39] `http://www.atlassian.com/software/jira`

Abbildung 10.9 *Beispiel aus der Trac Entwicklung: Ein Changeset wird mit allen wesentlichen Informationen angezeigt. Darunter wird über Verknüpfung zum SVN-System angezeigt, was konkret im Sourcecode geändert wurde.*

nummer angeben; damit wird der Fehler auch automatisch im Bug-Tracker geschlossen (2) Möchte jemand im Bug-Tracker nachvollziehen, wie ein bestimmter Fehler behoben wurde, greift der Bug-Tracker auf das SCM-System zu und zeigt die Code-Teile, die im Rahmen der Fehlerbehebung geändert wurden. Abbildung 10.9 zeigt ein Beispiel aus der Entwicklung des Trac-Systems selbst.

Die meisten Bug-Issue-Tracker sind Software-Pakete, deren Funktionalität aber deutlich über reines Issue-Tracking hinausgeht. Meist bieten die Systeme Wikis an, Editoren um Issues zu Roadmaps zu organisieren, Integration des Build-Systems, das Erstellen komplexer Suchen, Reports und Statistiken, Anpassung des Prozesses usw.

Integrierte Plattformen

Eine Frage, die sich bei Projekten immer wieder stellt ist, ob eigene Server betrieben werden und die genannten Werkzeuge selbst installiert und gewartet werden sollen. Dies bietet natürlich die größten Freiheitsgrade, macht aber auch die meiste Arbeit. Mittlerweile gibt es aber auch kommerzielle Anbieter (z. B. Assembla) sowie Plattformen für Open-Source-Entwicklungen (z. B. Sourceforge oder Google Code), die verschiedene in diesem Abschnitt beschriebene Werkzeuge unter einer Oberfläche integrieren.

Kommunikation über Tests

Zuletzt soll noch auf eine Art der Kommunikation hingewiesen werden, die sehr wichtig, vielen aber nicht bewusst ist. In Kapitel 5 wird im Rahmen von Qualitätssicherung auch die *testgetriebene Entwicklung* vorgestellt. Die Bedeutung von (automatisierten) Tests im Sinne der Qualitäts-

sicherung sollte sich aus mehreren Kapiteln dieses Buches heraus klar ergeben. Die Bedeutung von (automatisierten) Tests als *Form der Kommunikation* ist aber vielleicht nicht sofort offensichtlich, lässt sich aber an einem Beispiel gut erklären:

Beispiel

In einem Entwicklerteam entwickelt Programmierer A eine bestimmte Bibliothek `a.jar`, Entwickler B schreibt Geschäftslogik und verwendet dabei diese Bibliothek `a.jar` von Entwickler A. Angenommen B findet einen Fehler in dieser Bibliothek oder jedenfalls ein Verhalten, das sich nicht mit seinen Erwartungen der Funktionalität deckt. Wie sollte er vorgehen? Die meisten würden vielleicht versuchen das problematische Verhalten in einem E-Mail oder in persönlicher Kommunikation zu beschreiben und somit A mitzuteilen. In vielen Fällen entstehen dann daraus lange fruchtlose Diskussionen, bei denen A die Probleme von B nicht nachvollziehen kann oder den Fehler behebt, aber nicht prüfen kann, ob er tatsächlich nach Bs Vorstellung behoben ist usw.

Der bessere Weg ist meist folgender: B schreibt einen möglichst einfachen (Unit-) Test, der den Fehler reproduziert und daher natürlich mit der aktuellen Version von `a.jar` fehlschlägt. Diesen Unit-Test checkt er ins SCM-System ein und informiert A darüber. Nun hat A ganz konkreten Code, der das „problematische“ Verhalten zeigt. Nun stehen ihm mehrere Möglichkeiten offen. (1) Handelt es sich tatsächlich um einen Fehler, dann kann A diesen beheben, und es gibt auch schon einen Test, der die Behebung in Bs Sinne dokumentiert und zeigt. Gleichzeitig erhöht dieser Test in Summe die Testabdeckung und Qualität des Systems, oder (2) es handelt sich nicht um einen Fehler, sondern z. B. um ein schlecht dokumentiertes Interface oder ein Missverständnis Bs, so ist aber auch dies offensichtlich und kann entsprechend geklärt werden.

Diese Vorgehensweise bietet sich nicht nur innerhalb eines Programmierteams an, sondern auch, wenn man beispielsweise glaubt, in einem Open-Source-Projekt einen Fehler gefunden zu haben, und diesen den Entwicklern kommunizieren möchte.

10.9 Zusammenfassung

Convention over Configuration

Man kann zusammenfassend sagen, dass die Idee *Convention over Configuration* eine wesentliche Basis der meisten modernen Programmiersprachen und -werkzeuge darstellt. Freiheit macht nicht an jeder Stelle Sinn, sondern belastet an den falschen Stellen den Entwickler nur unnötig, denn es wird immer schwieriger und aufwendiger, ein Basisprojekt korrekt zusammenzustellen, sodass sich alle Konfigurationsdateien an der richtigen Stelle befinden, der Build-Prozess läuft usw. Hier sind Ansätze wie die Maven-Archetypen oder Rails' Scaffolding sehr hilfreich. Idealerweise gibt es einige Archetypen für verschiedenartige Projekte, die den Start vereinfachen. Da-

nach adaptiert und ändert man nur jene Aspekte, die aus technischen oder anderen Überlegungen besonders relevant sind.

Sourcecode-Management

Da Software-Projekte heute fast ausschließlich im Team entwickelt werden, sind Sourcecode-Management-Systeme eine wesentliche Basis der Kollaboration. Sie unterstützen Versionierung und Weiterentwicklung bzw. Wartung verschiedener Versionen eines Software-Produkts sowie die verteilte Arbeit im Team. Man findet im Wesentlichen zwei unterschiedliche Konzepte vor: Bei zentralisierten Systemen gleichen alle Entwickler ihre lokale Arbeitskopie mit einem Server ab; bei den moderneren verteilten Systemen hingegen ist keine zentralisierte Infrastruktur mehr erforderlich. Sie ermöglichen damit eine Vielzahl an unterschiedlichen Arten der Zusammenarbeit.

Build Management und Automatisierung

In der täglichen Arbeit der Software-Entwicklung treten viele repetitive Arbeiten auf (z.B. Kompilieren, Testen, Erstellen von Dokumentation). Die händische Ausführung solcher Tätigkeiten ist lästig, zeitaufwendig, fehlerhaft und kann nicht zuverlässig standardisiert und wiederholt werden. Aus diesem Grund haben sich für verschiedene Plattformen Systeme etabliert, die sich um die Automatisierung des Build- und Test-Prozesses sowie verschiedener anderer Aktivitäten kümmern. Erst die Automatisierung erlaubt es, den Software-Build und eine große Zahl an Tests regelmäßig (täglich) durchzuführen (*continuous integration*).

Dokumentation

Dokumentation ist eine wesentliche, leider häufig vernachlässigte Komponente eines Software-Projekts. Sie hilft unter anderem, den Überblick über ein Projekt zu bewahren, neue Entwickler einzuführen sowie Entscheidungen nachvollziehbar zu machen. Zur Dokumentation zählt man Artefakte wie Benutzerdokumentation, Modelle, Anforderungen und andere Projektmanagement-Dokumente, aber auch Sourcecode-Dokumentation, Testberichte und Besprechungsprotokolle. Die Dokumentation sollte mit der Software im SCM-System verwaltet werden und mit dem Daily-Build erstellt werden, sofern das sinnvoll ist (z. B. Benutzerdokumentation, Webseite, Testberichte). Dies ermöglicht es, für verschiedene Programmversionen die zugehörige Dokumentation zur Verfügung zu stellen.

Kommunikation im Team

Die bereits erwähnten SCM-Systeme sowie eine gründliche Dokumentations-Strategie unterstützen die Arbeit an einem Software-Projekt im (verteilten) Team. Zusätzlich gibt es andere Kommunikationsmittel, die verteilte Arbeit erleichtern oder überhaupt erst ermöglichen. Unabdingbar sind z. B. Ticketing-Systeme (Bug-Tracker), die es erlauben, Fehler, Änderungswünsche oder andere Probleme (auch durch Kunden) erfassbar und nachvollziehbar zu machen. Gesprächsprotokolle dienen der knappen aber klaren Kommunikation von Entscheidungen, Problemen und verteilten Aufgaben und sollten als Ergebnis jeder Besprechung vorliegen. Eine richtige Vorbereitung erlaubt die schnelle und effiziente Abwicklung von Besprechungen. Daneben gibt es noch eine Vielzahl anderer Systeme wie Wikis, Voice-over-IP, Chat, Mail etc., die auch einen wichtigen Beitrag leisten können. Man sollte sich jedoch im Team einigen, welcher Kommunikationskanal für welche Art der Kommunikation verwendet wird.

11 | Epilog

Zusammenfassung

Ziel des Buches war es, nicht einzelne und spezialisierte Aspekte des Software-Engineerings im Detail zu betrachten, sondern ein breiteres Verständnis über den ganzen Lebenszyklus der Software-Entwicklung zu erarbeiten: Zunächst wurden mit der Beschreibung des *Lebenszyklusses eines Software-Produkts* und oft verwendete *Vorgehensmodelle* die Grundlagen für eine Vielzahl von Projekten gelegt. Das Kapitel *Software-Projektmanagement* erklärt allgemeine Aspekte des Projektmanagements aber auch Besonderheiten von Software-Projekten. Die konsequente Verfolgung des Projektfortschrittes ist ein wesentlicher Aspekt eines erfolgreichen Projekts.

Die weiteren Kapitel beziehen sich schon stärker auf die technische Umsetzung. Um die Bedeutung von *Qualitätssicherung* zu betonen schließt dieses Kapitel direkt an die Einführung ins Projektmanagement an. Grafische Notationen werden an vielen Stellen in diesem Buch, aber auch in den meisten Software-Projekten verwendet. Eine Einführung und ein Kapitel zum Nachschlagen ist das Kapitel *Notationen und Methoden der Modellierung*. Anschließend wurden die technische Umsetzung eines Software-Projekts und wesentliche, oft wiederkehrende *Muster und Architektur-Konzepte* vorgestellt. Im Kapitel *komponentenorientierte Software-Entwicklung* werden viele der zuvor eingeführten Aspekte zusammengeführt und die Modularisierung in modernen Software-Projekten beschrieben. Zuletzt haben wir die *Techniken und Werkzeuge*, die in den meisten erfolgreichen Projekten zum Einsatz kommen, behandelt.

Damit wurde von der Konzeption bis zur technischen Umsetzung moderner komponentenorientierter Software-Entwicklung sowie den notwendigen Tools und Praktiken für die Arbeit im Team ein Bogen gespannt. Dabei konnten natürlich nicht alle Themen im Detail behandelt werden. Ergänzendes Material, das regelmäßig aktualisiert wird, finden Sie im Web unter: `http://bpse.ifs.tuwien.ac.at/`. Dort stehen unter anderem Beispiele, technische Details zu den verwendeten Tools und Frameworks und verschiedene weiterführende Materialien zum Download bereit. Außerdem betreiben wir ein Blog unter `http://best-practice-software-engineering.blogspot.com/`. Wir laden alle Leser ein, aktuelle Themen dort mit uns zu diskutieren!

An dieser Stelle soll noch die weiterführende Literatur empfohlen werden um sich je nach eigenem Interesse und Projektart entsprechend weiterentwickeln zu können:

(Interaktions) Design

Erstellt man Anwendungen, die direkt mit Benutzern interagieren, ist es für ein erfolgreiches Projekt sehr wesentlich, sich mit der Interaktion zwischen Benutzern und Software auseinanderzusetzen. *Designing Interactions* von Bill Moggridge ist eine praxisorientierte Einführung, die mit vielen Fall-

beispielen und Interviews arbeitet. Das Buch *Sketching User Experiences* von Bill Buxton, der jahrelang im Umfeld von Xerox Parc gearbeitet hat, beschäftigt sich eigentlich mit Produktdesign. Buxton bezieht sich auf verschiedene erfolgreiche und weniger erfolgreiche Produkte und deren spezifische Interaktion (meist mithilfe von Software) mit dem Benutzer. Zuletzt soll *Designing Interfaces* von Jenifer Tidwell empfohlen werden. Dieses Buch beschreibt Best-Practices und Patterns des User Interface Designs in einer sehr praxisorientierten Weise.

Security

In den meisten heutigen Anwendungen spielt Security eine wichtige Rolle. Fehler in der Implementierung und im Design können beispielsweise zum Verlust wertvoller Daten führen oder Systeme des Kunden angreifbar machen. *Software Security: Building Security In* von Gary McGraw ist eine gute allgemeine Einführung in verschiedenste Security-Aspekte in Software-Produkten. In *Threat Modeling* beschreiben Frank Swiderski und Window Snyder einen strukturierte Vorgehensweise, um mögliche Bedrohungen zu beschreiben und mögliche Attacken vorherzusehen. Dies ist ein wesentlicher erster Schritt um Bedrohungsszenarien frühzeitig zu erkennen. *Writing Secure Code* von Michael Howard und David LeBlanc ist eine praxisorientierte Einführung. Die Autoren beschreiben wesentliche Angriffsmuster und stellen Checklisten als Anleitung für eigene Projekte zur Verfügung.

Verteilte Systeme

Verteilte Software-Systeme treten in der Praxis immer häufiger auf. Durch ihre Komplexität und die Verteilung der Software-Komponenten müssen bestimmte Design-Richtlinien und Vorgehensweisen bei der Implementierung beachtet werden. *Verteilte Systeme: Prinzipien und Paradigmen* von Andrew S. Tanenbaum ist die ideale Einführungsliteratur zu diesem Thema. Im Zuge von verteilten Architekturen kommen Software-Leute auch immer wieder mit serviceorientierten Architekturen in Kontakt. Zu diesem Thema gibt es ein breites Angebot an guter Literatur. *Enterprise SOA* war eines der ersten SOA-Bücher und verdeutlicht sehr gut und anhand konkreter Praxisbeispiele die Grundlagen und Gedanken einer SOA. Nicht zuletzt soll auch das Buch *Enterprise Service Bus* von David A. Chappell erwähnt werden, wo der Enterprise-Service-Bus in seine Einzelbestandteile zerlegt wird. Interessenten erfahren in diesem Buch ganz genau, wie es in einer derartigen Middleware aussieht.

Produkt- und Prozessverbesserung

Software-Produkt- und -Prozessverbesserungen sind zentrale Anforderungen in Unternehmen und Organisationen, um bessere Produkte schneller auf dem Markt bringen zu können. Neben der Anwendung der in diesem Buch beschriebenen Best-Practices ist auch die Einbettung in die Unternehmensprozesse notwendig, um wiederholbar gute Produkte zu erstellen. Qualitätsmanagementstandards ermöglichen die Bereitstellung dieses geeigneten Rahmens. Zu diesem Themenbereich existiert ein breites Spektrum an guter Literatur. Ernest Wallmüller gibt etwa in seinem Buch *Software-Qualitätsmanagement in der Praxis* einen guten Überblick über die Integration qualitätssichernder Maßnahmen. Ein Überblick über

Qualitätsmanagementsysteme, wie etwa CMMI oder SPICE (ISO 15504) mit Schwerpunkt auf der Software-Entwicklung findet sich etwa in dem Buch *Software Process Improvement mit CMMI, PSP/TSP und ISO 15504* von Ernest Wallmüller. Dem Schwerpunktthema *CMMI: Verbesserung von Software-Prozessen mit Capability Maturity Model Integration* widmet sich Ralph Kneuper, der in dem gleichnamigen Buch diesen Standard im Detail beschreibt. Eine umfassende Referenz auf SPICE wird von Han van Loon in seinem Buch *Process Assessment and ISO/IEC 15540* beschrieben.

Empirische Software-Technik

Die Sammlung geeigneter Daten und die Anwendung konkreter Metriken zur Bestimmung des Produkt- und Prozessreifegrads ist die Voraussetzung für Prozessverbesserungsinitiativen. Umfassende Informationen zu Metriken finden sich beispielsweise in *Software Measurement: Establish – Extract – Evaluate – Execute* von Christof Ebert und Reiner Dumke sowie in *Metrics and Models in Software Quality Engineering* von Stephen H. Kan. Eine Sammlung von guten Artikeln im Bereich der empirischen Software-Technik findet sich etwa in dem Buch *Guide to Advanced Empirical Software Engineering*, das von Forrest Shull, Janice Singer und Dag I.K.Sjøberg herausgegeben wurde.

Literaturverzeichnis

[1] AMBE, MILDRED N. und FREDERICK VIZEACOUMAR: *Evaluation of two Architectures using the Architecture Tradeoff Analysis Method (ATAM)*. Technischer Bericht, University of Alberta, 2002.

[2] *Atom Standard, Description and RFC*. `http://www.atomenabled.org`, 2008.

[3] BASILI, VICTOR R., SCOTT GREEN, OLIVER LAITENBERGER, FILIPPO LANUBILE, FORREST SHULL, SIVERT SORUMARD und MARVIN V. ZELKOVITZ: *The Empirical Investigation of Perspective-Based Reading*. Journal on Empirical Software Engineering, 1(2):133–164, 1996.

[4] BASS, LEN, PAUL CLEMENTS und RICK KAZMAN: *Software Architecture in Practice*. Addison-Wesley, 2. Auflage, 2003.

[5] BECK, KENT: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2005.

[6] BECK, KENT, MIKE BEEDLE, ARIE VAN BENNEKUM, ALISTAIR COCKBURN, WARD CUNNINGHAM, MARTIN FOWLER, JAMES GRENNING, JIM HIGHSMITH, ANDREW HUNT, RON JEFFRIES, JON KERN, BRIAN MARICK, ROBERT C. MARTIN, STEVE MELLOR, KEN SCHWABER, JEFF SUTHERLAND und DAVE THOMAS: *Agile Manifesto for Agile Software Development*. `http://agilemanifesto.org`, 2001.

[7] BIEN, ADAM: *Enterprise Architekturen – Leitfaden für effiziente Software-Entwicklung*. Entwickler.Press, 2006.

[8] BIFFL, STEFAN: *Software Inspection Techniques to support Project and Quality Management*. Shaker Verlag, 2001.

[9] BIFFL, STEFAN, AYBÜKE AURUM und BARRY BOEHM: *Value-Based Software Engineering*. Springer, 2005.

[10] BIFFL, STEFAN, DIETMAR WINKLER und DENIS FRAST: *Qualitätssicherung, Qualitätsmanagement und Testen in der Softwareentwicklung*. Skriptum zur Lehrveranstaltung, Technische Universität Wien, `http://qse.ifs.tuwien.ac.at/courses/skriptum/script.htm`, 2004.

[11] BISANT, DAVID B. und JAMES R. LYLE: *A Two-Person Inspection Method to Improve Programming Productivity*. IEEE Transactions on Software Engineering, 15(10):1294–1304, 1989.

[12] BOEHM, BARRY: *A Spiral Model of Software Development and Enhancement*. IEEE Computer, 21(5):61–72, 1988.

[13] BOEHM, BARRY: *Software Engineering is a Value-based Contact Sport*. IEEE Software, 19(5):95–96, 2002.

[14] BOEHM, BARRY, CHRIS ABTS, A. WINSOR BROWN, SUNITA CHULANI und BRADFORD K. CLARK: *Software Cost Estimation With Cocomo II*. Prentice Hall, 2009.

[15] BROOKS, FRED: *The Mythical Man-Month*. Addison Wesley, 1995.

[16] BRÖHL, ADOLF-PETER und WOLFGANG DRÖSCHEL: *Das V-Modell: Der Standard für die Softwareentwicklung mit Praxisleitfaden*. Oldenburg, 1993.

[17] *Standish Group: The Chaos Report*. `http://net.educause.edu/ir/library/pdf/NCP08083B.pdf`, 1995.

[18] CHEN, PETER PIN-SHAN: *The Entity-Relationship Model – toward a Unified View of Data*. ACM Transactions on Database Systems, 1(1):9–36, 1976.

[19] COCKBURN, ALISTAIR: *Crystal-Clear a Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004.

[20] CODD, EDGAR F.: *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM, 13(6):377–387, 1970.

[21] DATE, CHRIS: *An Introduction to Database Systems*. Addison Wesley, 6. Auflage, 2003.

[22] DEMOLSKY, MARKUS: *Generic Data Access Objects with Java Generics*. http://best-practice-software-engineering.blogspot.com/2008/01/java-generic-dao.html, 2008.

[23] DIJKSTRA, EDGSER: *On the Role of Scientific Thought*. In: *Selected Writings on Computing: A Personal Perspective*, Springer, 1974.

[24] *DIN 69901: Projektmanagement, Begriffe*, 1987.

[25] DREWS, GÜNTER und NORBERT HILLEBRANDT: *Lexikon der Projektmanagement-Methoden: Die wichtigsten Methoden im Projektmanagement-Life-Cycle*. Haufe, 2007.

[26] FAGAN, M.E.: *Design and Code Inspections to Reduce Errors in Program Development*. IBM System Journal, 3:182–211, 1976.

[27] FEYHL, ACHIM W.: *Management und Controlling von Softwareprojekten*. Dr. Th. Gabler Verlag, 2004.

[28] FOWLER, MARTIN: *Analysis Patterns – Reusable Object Models*. Addison Wesley, 1996.

[29] FOWLER, MARTIN: *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[30] FOWLER, MARTIN: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 3. Auflage, 2003.

[31] FOWLER, MARTIN: *Continuous Integration*. http://martinfowler.com/articles/continuousIntegration.html, 2006.

[32] GAMMA, ERICH, RICHARD HELM und RALPH E. JOHNSON: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1. Auflage, 2004.

[33] *The Gantt-Project Team: Gantt-Project ... Project Management as free as Bee(r).* http://ganttproject.biz, 2008.

[34] GILB, TOM und DOROTHY GRAHAM: *Software Inspection*. Addison Wesley, 1993.

[35] GRÜNBACHER, PAUL: *Collaborative Requirements Negotiation with EasyWinWin*. 11th International Workshop on Database and Expert Systems, Seiten 954–960, 2000.

[36] HAMILL, PAUL: *Unit Test Frameworks*. O'Reilly, 2004.

[37] HEINEMEIER-HANSSON, DAVID: *The Great Surplus*. O'Reilly Media RailsConf, http://itc.conversationsnetwork.org/shows/detail3762.html, 2008.

[38] HEINRICH, LUTZ J. und FRANZ LEHNER: *Informationsmanagement*. Oldenbourg, 8. Auflage, 2005.

[39] HINDEL, BERND, KLAUS HÖRMANN und MARKUS MÜLLER: *Basiswissen Software-Projektmanagement*. dpunkt, 2006.

[40] HOHPE, GREGOR: *Programming Without a Call Stack – Event Driven Architectures*. http://eaipatterns.com/docs/EDA.pdf, 2006.

[41] HOHPE, GREGOR und BOBBY WOOLF: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[42] HOOF, JACK VAN: *How EDA extends SOA and why it is important*. http://jack.vanhoof.soa.eda.googlepages.com/How_EDA_extends_SOA_and_why_it_is_important_-_Jack_van_Hoof_-_v6.0_-_2006.pdf, 2006.

[43] HÖHN, REINHARD und STEPHAN HÖPPNER: *Das V-Modell XT*. Springer, eXamen Press, 2008.

[44] *IEEE Std 1028: IEEE Standard for Software Reviews*, 1997.

[45] *IEEE Std 1219: IEEE Standard for Software Maintenance*, 1998.

[46] *IEEE Std 610.12: Standard Glossary of Software Engineering Terminology*, 1990.

[47] *IEEE Std 12207: Standard for Information Technology – Software Life Cycle Processes*, 1996.

[48] ING, DAVID: *Dude, where's my SOA?* http://www.from9till2.com/PermaLink.aspx?guid=bdf07eaf-02dd-4f65-bb57-83e00e914e45, 2005.

[49] *ISO 9000: Quality Management Systems – Fundamentals and Vocabulary*, 2005.

[50] *ISO/IEC 9126-1: Software Engineering – Product Quality – Part 1: Quality Model*, 2001.

[51] *JDK 5.0 Documentation*. `http://java.sun.com/j2se/1.5.0/docs/index.html`, 2008.

[52] JECKLE, MARIO, CHRIS RUPP, JÜRGEN HAHN, BARBARA ZENGLER und STEFAN QUEINS: *UML 2 Glasklar*. Hanser, 2004.

[53] JOHNSON, RALPH E. und BRIAN FOOTE: *Designing Reusable Classes*. The Journal of Object-Oriented Programming, Seiten 22–35, 1998.

[54] *Java Script Object Notation Standard*. `http://www.json.org/`, 2008.

[55] KANER, CEM, JACK L. FALK und HUNG-QUOC NGUYEN: *Testing Computer Software*. John Wiley & Sons, 2. Auflage, 1999.

[56] KARNOVSKY, HANS: *Grundlagen des Projektmanagements: ein Leitfaden für die Projektpraxis*. Paul Bernecker Verlag, 2002.

[57] KAZMAN, RICK, MARIO BARBACCI, MARK KLEIN, S. JEROME CARRIERE und STEVEN G. WOODS: *Experience with Performing Architecture Tradeoff Analysis*. Proceeding of the 21th International Conference on Software Engineering, ACM Press, 1999.

[58] KAZMAN, RICK, MARK KLEIN und PAUL CLEMENTS: *ATAM: Method for Architecture Evaluation*. Technischer Bericht CMU/SEI-2000-TR-004, ESC-TR-2000-004, Carnegie Mellon University, Software Engineering Institute, 2000.

[59] KLEINSCHMIDT, PETER und CHRISTIAN RANK: *Relationale Datenbanksysteme: Eine praktische Einführung*. Springer, 2004.

[60] KRAFZIG, DIRK, KARL BANKE und DIRK SLAMA: *Enterprise SOA, Service Orientied Architecture Best Practices*. Prentice Hall, 1. Auflage, 2005.

[61] KRUCHTEN, PHILIPPE: *The Rational Unified Process: An Introduction*. Addison-Wesley, 2004.

[62] KRÜGER, GUIDE und THOMAS STARK: *Handbuch der Java Programmierung*. Addison-Wesley, 5. Auflage, 2008.

[63] LAITENBERGER, OLIVER: *Studying the Effects of Code Inspection and Structural Testing on Software Quality*. Technical Report IESE-No 024.98/E, ISERN-98-10, Fraunhofer Institute for Experimental Software Engineering, 1998.

[64] LAITENBERGER, OLIVER, COLIN ATKINSON, MAUD SCHLICH und KHALED EL EMAM: *An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents*. Technischer Bericht IESE 080.99/E, ISERN 2000-01, Fraunhofer Institute for Experimental Software Engineering, 1999.

[65] LARMAN, CRAIG: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 3. Auflage, 2004.

[66] LIU, LING, PAUL SORENSON, GARY FRÖHLICH und H. JAMES HOOVER: *Designing Object-Oriented Frameworks*. ACM, 1998.

[67] MARTIN, JOHNNY und W.T TSAI: *N-Fold Inspection: A Requirements Analysis Technique*. Communications of the ACM, 33(2):225–323, 1990.

[68] MATYAS, STEPHEN M., NICHOLAS SCHNEIDER, MARK VOIT und PAUL DUVALL: *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[69] MESZAROS, GERARD: *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[70] MYERS, GLENFORD J.: *Art of Software Testing*. John Wiley & Sons, 1979.

[71] PFLEEGER, SHARI LAWRENCE und JOANNE M. ATLEE: *Software Engineering – Theory and Practice*. Pearson Education, 2006.

[72] POENSGEN, BENJAMIN und BERTRAM BOCK (Herausgeber): *Die Function-Point Analyse: Ein Praxishandbuch*. Dpunkt Verlag, 2005.

[73] POL, MARTIN, TIM KOOMEN und ANDREAS SPILLNER: *Management und Optimierung des Testprozesses: ein praktischer Leitfaden für erfolgreiches Testen von Software, mit TPI und TMap*. dpunkt, 2000.

[74] RAUSCH, ANDREAS: *IT Projekte erfolgreich ... mit dem neuen V-Modell XT*. `http://www.v-modell-xt.de`, 2006. Präsentation.

[75] REUSSNER, RALF und WILHELM HASSELBRING: *Handbuch der Software-Architektur*. dpunkt, 2006.

[76] ROYCE, W.W.: *Managing the Development of Large Software Systems*. IEEE WESCON, Seiten 1–9, 1970.

[77] RUMBAUGH, JAMES, IVAR JACOBSON und GRADY BOOCH: *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2. Auflage, 2004.

[78] RUMBAUGH, JAMES, IVAR JACOBSON und GRADY BOOCH: *The Unified Modeling Language User Guide*. Addison-Wesley, 2. Auflage, 2005.

[79] SCHLICH, MAUD: *Inspektion des Systemlastenhefts*. Technischer Bericht IESE-Nr. 36-02, Fraunhofer Institute for Experimental Software Engineering, 2002.

[80] SCHWABER, KEN und THOMAS IRLBECK: *Agiles Projektmanagement mit Scrum*. Microsoft Press Deutschland, 2007.

[81] *Simple Object Access Protocol (SOAP) Standard, W3C*. `http://www.w3.org/TR/soap/`, 2008.

[82] *Software Architecture – A Pattern Language for Building Sustainable Software Architectures*. `http://www.voelter.de/data/pub/ArchitecturePatterns.pdf`, 2005.

[83] SOMMERVILLE, IAN: *Software Engineering*. Addison-Wesley, 2007.

[84] SONATYPE: *Maven, the Definitive Guide*. `http://www.sonatype.com/book/lifecycle.html`, 2008.

[85] SPILLNER, ANDREAS und TILO LINZ: *Basiswissen Software-Test*. dpunkt, 2003.

[86] STARKE, GERNOT: *Effektive Software Architkturen - Ein praktischer Leitfaden*. Hanser, 2. Auflage, 2005.

[87] STROOPER, PAUL, LEESA MURRAY und DAVID CARRINGTON: *An Approach to Specifying Software Frameworks*. ACM International Conference Proceeding Services, 56:185–192, 2004.

[88] *Software Engineering Body of Knowledge (SWEBOK)*. `http://www.swebok.org`, 2004.

[89] THALLER, GEORG ERWIN: *Software Qualität*. VDE Verlag, 2000.

[90] TOWNSEND, MARK: *Exploring the Singleton Design Pattern*. Technischer Bericht, Microsoft Developer Network, 1998.

[91] VLIET, HANS VAN: *Software Engineering – Principles and Practice*. Wiley & Sons, 2008.

[92] WINKLER, DIETMAR: *Improvement of Defect Detection with Software Inspection Variants*. VDM Verlag, 2008.

[93] *Webservice Description Language (WSDL) Standard, W3C*. `http://www.w3.org/TR/wsdl`, 2008.

[94] ZIMMER, WALTER: *Relationships between Design Patterns*. In: *Pattern Languages of Program Design*, Seiten 345–364. Addison-Wesley, 1994.

[95] ÖSTERREICH, BERND und CHRISTIAN WEISS: *APM - Agiles Projektmanagement: Erfolgreiches Timeboxing für IT-Projekte*. dpunkt, 2007.

Index