Förderkreis der
Angewandten
Informatik

an der
Westfälischen
Wilhelms-Universität Münster e.V.

Working Paper No. 1

Best Practices in der Softwareentwicklung

Christian Arndt, Christian Hermanns,
Herbert Kuchen, Michael Poldner

16. Februar 2009

Westfälische
Wilhelms-Universität
Münster

Best Practices in der Softwareentwicklung

Christian Arndt, Christian Hermanns, Herbert Kuchen, Michael Poldner

16. Februar 2009

Christian Arndt, Christian Hermanns, Prof. Dr. Herbert Kuchen, Michael Poldner
Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität Münster
Leonardo Campus 3
48149 Münster
kuchen@uni-muenster.de

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

ISSN 1868-0801

Printed in Germany

Vorwort

Die Entwicklung leistungsfähiger Software ist ein wesentlicher Kern von Informationstechnologie. Zahlreiche Unternehmen entwickeln Software, entweder als Softwareanbieter im Rahmen eines eigenständigen Geschäftsmodells, oder als Nutzer selbstentwickelter Anwendungen zur Unterstützung der Geschäftsprozesse.

Auch im IHK-Bezirk Nord Westfalen gibt es sehr viele Firmen, die Software produzieren - allein rund 1500 Anbieter. Sie stärken die wirtschaftliche Leistungskraft der Region. Und sie sorgen natürlich in hohem Maße dafür, dass die unverzichtbare IT „rund läuft“. Für diese Unternehmen ist es wichtig, stets über den aktuellen Stand der Softwareentwicklung informiert zu sein und fortschrittliche Methoden anzuwenden. Nur dann sind Leistungs- und Wettbewerbsfähigkeit auf Dauer möglich. Das gilt vor allem auch zunehmend in globalisierten Märkten. Heimische Anbieter konkurrieren oft mehr und mehr mit günstigen Software-Produzenten etwa in Indien oder Osteuropa.

Die vorliegende Broschüre „Best Practices in der Softwareentwicklung“ soll insbesondere kleinen und mittelgroßen Betrieben eine Arbeitshilfe zur rationellen Softwareentwicklung geben. Sie soll in überschaubarer Form Methoden vorstellen, die die Entwicklung effizienter gestalten können.

Der vorliegende Leitfaden ist durch den Förderkreis für Angewandte Informatik an der Westfälischen Universität Münster angeregt und finanziert worden. Der Förderkreis wird von rund 30 Unternehmen aus der Region und der Industrie- und Handelskammer Nord Westfalen getragen. Hauptziel ist die Förderung der praxisorientierten Forschung und Lehre - sowie der schnelle Wissenstransfer.

Besonderer Dank gebührt dem Direktor des Instituts für Angewandte Informatik, Herrn Professor Dr. Herbert Kuchen, und seinen Mitarbeitern für die außerordentlich engagierte und praxisnahe Umsetzung des Projektes.

Ein weiteres großes Dankeschön geht an die Mitglieder einer projektbegleitenden Arbeitsgruppe aus Unternehmen: Frau Cornelia Gaebert (INDAL, Münster), Herrn Hans-Hermann Göcke (mdi-nora, Ibbenbüren) und Herrn Johannes Schlattmann (LVM, Münster). Ihre Mitarbeit hat zu weiteren wertvollen praxisbezogenen Hinweisen geführt.

Martin Kittner
Vorsitzender des Förderkreises
für Angewandte Informatik

Dr. Christoph Asmacher

IHK Nord Westfalen

Inhaltsverzeichnis

1 Einleitung **1**

2 Vorgehensmodelle **5**
2.1 Das Wasserfall-Modell 7
2.2 Der Rational Unified Process 8
2.3 Extreme-Programming 11

3 Management-Aspekte **17**
3.1 Aufwandsschätzung 17
3.2 Risiko-Management 19
3.2.1 Risiko-Identifikation 20
3.2.2 Risiko-Analyse 21
3.2.3 Risiko-Prioritätenbildung 21
3.2.4 Risikomanagement-Planung 21
3.2.5 Risiko-Überwindung 21
3.2.6 Risiko-Überwachung 22

4 Objektorientierung und UML **25**
4.1 Grundkonzepte der Objektorientierung 27
4.1.1 Objekte und Klassen 27
4.1.2 Attribute und Operationen 27
4.1.3 Vererbung 27
4.2 Unified Modeling Language 28
4.2.1 Klassendiagramme 28
4.2.2 Darstellung einer Klasse im Klassendiagramm 29
4.2.3 Anwendungsfalldiagramme 33
4.2.4 Sequenzdiagramme 35
4.2.5 Aktivitätsdiagramme 36
4.2.6 Weitere UML-Diagramme 38

5 Softwarearchitekturen **39**
5.1 Schichtenarchitekturen . . . 39
5.1.1 Das Client-Server-Modell . . . 39
5.1.2 Aufbau von Schichtenarchitekturen . . . 40
5.1.3 Vergleich von Schichten-Architekturen . . . 42
5.2 Komponentenbasierte Architekturen . . . 45
5.2.1 Komponenten . . . 46
5.2.2 CORBA . . . 47
5.2.3 Java EE . . . 49
5.2.4 .NET . . . 56
5.2.5 Vergleich Java EE und .NET . . . 60

6 Entwurfsmuster **65**
6.1 Erzeugungsmuster . . . 66
6.1.1 Abstrakte Fabrik . . . 66
6.1.2 Weitere Erzeugungsmuster . . . 68
6.2 Strukturmuster . . . 68
6.2.1 Adapter . . . 69
6.2.2 Weitere Strukturmuster . . . 71
6.3 Verhaltensmuster . . . 72
6.3.1 Befehl . . . 72
6.3.2 Beobachter . . . 73
6.3.3 Weitere Verhaltensmuster . . . 75
6.4 Weitere Entwurfsmuster . . . 76

7 Testen **77**
7.1 Isolation zu testender Einheiten . . . 77
7.2 Automatisierung des Testens . . . 79
7.3 Black-Box-Testen . . . 80
7.4 Glass-Box-Testen . . . 81

8 Tools **83**
8.1 Versionsverwaltung . . . 83
8.1.1 Aufgaben . . . 83
8.1.2 Funktionsweise . . . 84
8.2 CASE-Tools . . . 86
8.2.1 Funktionsweise . . . 86
8.2.2 Auswahl von CASE-Plattformen . . . 86
8.2.3 Test-Werkzeuge . . . 88
8.2.4 Werkzeuge zur Architekturüberwachung . . . 88

9 Enterprise Application Integration **89**
9.1 Heterogene Systemlandschaften 90
9.2 Motivationen für Integrationsmaßnahmen 90
9.3 EAI-Architekturkonzepte 92
9.3.1 Punkt-zu-Punkt-Integration 92
9.3.2 Hub-and-Spokes-Integration 93
9.3.3 Bus-Integration 93
9.4 Integrationsebenen 94
9.4.1 Integration auf Datenebene 94
9.4.2 Integration auf Funktionsebene 96
9.4.3 Integration über die Benutzerschnittstelle 105
9.5 Zusammenfassende Bewertung 107

10 Modell-getriebene Software-Entwicklung **109**

Abbildungsverzeichnis

2.1 Wasserfallmodell 7
2.2 Rational Unified Process 10
2.3 Extreme Programming 11

3.1 Die sechs Schritte des Risikomanagements. 20

4.1 Beispiel: Klasse Rechteck im Klassendiagramm. 28
4.2 Beispiel: Klassenhierarchie und zugehöriger Java-Code. 30
4.3 Klassendiagramm mit Assoziation. 31
4.4 Klassendiagramm mit Aggregation und Komposition. 31
4.5 Klassendiagramm für ein Krankenhaus-Softwaresystem. 32
4.6 Klassendiagramm für arithmetische Ausdrücke. 33
4.7 Beispiel: Anwendungsfall-Diagramm. 34
4.8 Sequenzdiagramm für die Auswertung eines arithmetischen Ausdrucks. 35
4.9 Aktivitätsdiagramm für die Frühstückszubereitung. 37

5.1 Das Client/Server-Modell 40
5.2 Strikte, lineare und baumartige Schichtenstruktur. 41
5.3 Schichtenarchitektur für Desktop-Anwendungen. 43
5.4 Schichtenarchitekturen für Web-Anwendungen. 44
5.5 Multi-Channel-Architektur. 46
5.6 Architektur eines CORBA-Systems. 48
5.7 Entfernter Methodenaufruf mit CORBA. 48
5.8 Architektur einer Java EE-Anwendung. 50
5.9 Aufbau einer EJB-Komponente. 51
5.10 Common Language Infrastructure 57
5.11 Architektur des .NET Frameworks 57
5.12 Aufbau einer COM-Komponente 58
5.13 Vergleich Java EE- und .NET-Architektur 60

6.1 Entwurfsmuster Abstrakte Fabrik am Beispiel. 67
6.2 Entwurfsmuster Adapter am Beispiel. 69
6.3 Klassen-Adapter. 70

6.4 Objekt-Adapter. 70
6.5 Entwurfsmuster Befehl am Beispiel. 72
6.6 Entwurfsmuster Befehl. 73
6.7 Objekt mit unterschiedlichen Visualisierungen. 74
6.8 Entwurfsmuster Beobachter. 75

7.1 Teststümpfe 78
7.2 Mock-Klassen 78
7.3 Kontrollflussüberdeckung 82

8.1 Speichermodelle für die Versionskontrolle 85
8.2 Kategorisierung von CASE-Tools 86

9.1 Vollständige Punkt-zu-Punkt-Integration von sechs Systemen. 92
9.2 Hub-and-Spokes-Integration. 93
9.3 Bus-Integration. 94
9.4 Integration auf Datenebene. 95
9.5 Integration auf Funktionsebene. 96
9.6 Übersicht Integrationstechnologien. 97
9.7 Entfernter Methodenaufruf über einen Objekt Request Broker. 99
9.8 Interaktionen zwischen Web-Services. 104
9.9 Integration auf Benutzerschnittstellenebene. 106

10.1 Klassendiagramm Buch und Exemplar 110
10.2 Generierter EJB-Code für die Klasse Exemplar im Buch-Beispiel. . . 110
10.3 xPand-Transformationsregeln im Buch-Beispiel. 111

Tabellenverzeichnis

1.1 Eignung von Technologien, Maßnahmen und Ansätzen. 4

3.1 FP-Bewertung von Eingaben, Ausgaben und Abfragen 18
3.2 FP-Bewertung von internen Datenbeständen und Referenzdaten . . . 18
3.3 Beispiel: Ermittlung von Function Points. 19
3.4 Typische Risiken einer Software-Entwicklung. 22

5.1 Vor- und Nachteile von Schichtenarchitekturen. 42
5.2 Vergleich von Schichtenarchitekturen. 45
5.3 Vor- und Nachteile des Struts-Frameworks. 53
5.4 Vor- und Nachteile des JSF-Frameworks. 54
5.5 Vor- und Nachteile des Spring MVC-Frameworks. 55
5.6 Vor- und Nachteile des Tapestry-Frameworks. 56

Kapitel 1

Einleitung

Was das Vorgehen und die eingesetzten Methoden und Werkzeuge bei der Softwareentwicklung angeht, gibt es zwischen verschiedenen Firmen deutliche Unterschiede. Während manche Firmen die Vorschläge aus Lehrbüchern zur Softwaretechnik weitgehend umsetzen, gibt es andere, die hinter dem aktuellen Stand der Technik zurückbleiben. Bei größeren Firmen kann man feststellen, dass solche Unterschiede sogar zwischen verschiedenen Abteilungen bestehen. Die vorliegende Broschüre hat zum Ziel, Methoden aufzuzeigen, über deren Einsatz zumindest nachgedacht werden kann und die die Softwareentwicklung effizienter und qualitativ hochwertiger gestalten können.

Im Folgenden wird davon ausgegangen, dass Sie als Leser zumindest über Grundkenntnisse der Softwareentwicklung verfügen. Einige oder vielleicht sogar die meisten der angesprochenen Themen werden Ihnen bekannt sein. Es wird aber zumindest ein paar Themen und Aspekte geben, bei denen sie auf Ansätze stoßen, die Ihnen weniger vertraut sind und die in Ihrer Firma vielleicht sinnvoll eingesetzt werden können. Wenn dem so sein sollte, hat diese Broschüre ihr Ziel erreicht. Wenn Sie einige Themen bereits kennen sollten, ist es nicht unbedingt zwingend, die Broschüre vollständig im Detail zu lesen. Sie können dann gezielt die Kapitel durcharbeiten, in denen Sie neue Aspekte vorfinden, und die anderen eher diagonal lesen.

Zu jedem der hier behandelten Themen gibt es eigene, umgangreiche Bücher. Eine umfassende, detaillierte Darstellung hätte daher mehrere Tausend Seiten erfordert. Ziel dieser Broschüre ist es daher nicht, alle Aspekte der Softwaretechnik im Detail darzustellen, sondern interessante Methoden und deren Vorzüge kurz vorzustellen und Ihnen durch Literaturhinweise die Möglichkeit zu geben, die für Sie relevanten Aspekte in den entsprechenden umfassenderen Werk nochmal ausführlicher nachzuschlagen.

Die folgenden Themen werden in dieser Broschüre behandelt. In Kapitel 2 werden zunächst in der aktuellen Diskussion vielbeachtete Vorgehensmodelle für die Softwareentwicklung vorgestellt, und zwar der Unified Process und Extreme Programming. Viele Firmen setzen, was das Vorgehensmodell angeht, noch das klassi-

sche Wasserfallmodell ein. In einigen Projekten mag das sinnvoll sein. Zumindest aber in Projekten, bei denen es bedeutsame Risiken zu bewältigen gilt, sollte zur Reduktion der Risisken und der Kosten der Behebung spät erkannter Fehler eher ein iteratives Vorgehensmodell, wie eines der beiden genannten Modelle, eingesetzt werden. Auch wenn man keinen der beiden Ansätze komplett übernimmt, so kann man sich hieran zumindest einzelne Aspekte abschauen und sie in der eigenen Firma sinnvoll umsetzen.

In Kapitel 3 wird dann zunächst ein Ansatz zur Schätzung des Aufwands eines Projekts vorgestellt, die Function-Point-Analyse. Sie liefert typischerweise mit geringem Aufwand genauere Ergebnisse als ein simpler Vergleich mit abgeschlossenen Projekten oder eine rein erfahrungsbasierte Schätzung, wie man sie häufig vorfindet. Weiterhin wird in diesem Kapitel ein systematisches Vorgehen zur Identifikation von Risiken bei einer Softwareentwicklung sowie zu deren Bewertung und Beherrschung vorgeschlagen. Vor dem Hintergrund, dass auch heute noch ein erheblicher Prozentsatz der Softwareprojekte scheitert, ist eine systematische Behandlung der Risiken empfehlenswert.

In Kapitel 4 werden die Vorteile der objektorientierten Softwareentwicklung herausgearbeitet. Letztere wird zwar in der Praxis vielfach eingesetzt. Aber es ist vermutlich nicht jedem klar, worin die Vorteile im Einzelnen liegen. Während bei eingebetteten System wie Waschmaschinen, Supermarktkassen und Aufzügen auch klassiche imperative Sprachen wie C nach wie vor ihre Vorzüge haben, so können objektorientierte Sprachen wie Java, C++ und C# gerade bei größeren Softwaresystemen und Informationssystemen ihre besseren Strukturierungsmöglichkeiten aus Sicht der so genannten Programmierung im Großen ausspielen. Hinzu kommt, dass die objektorientierte Softwareentwicklung durch das Vorhandensein einer standardisierten Modellierungssprache, der Unified Modeling Language (UML), und hierauf aufbauender Werkzeuge erleichtert wird. Die UML wird ebenfalls in diesem Kapitel kurz vorgestellt. Jeder Softwareentwickler sollte sie heute kennen und beherrschen.

In Kapitel 5 werden dann einige Entwurfsaspekte von Softwaresystemen behandelt. Inbesondere die Vorteile von Schichtenarchitekturen und komponentenbasierten Architekturen, bei denen Middleware zur Reduktion des Aufwands und der Fertigungstiefe eingesetzt wird, sollte jeder Softwareentwickler kennen.

In Kapitel 6 werden so genannte Entwurfsmuster vorgestellt. Hierbei handelt es sich um Systeme von Klassen, die sich zur Lösung gewisser, immer wiederkehrender Entwurfsprobleme bewährt haben und die in vielen Fällen die angestrebte Flexibilität der Lösung gewährleisten können. Diese Muster gehören zum Handwerkszeug jedes objektorientierten Entwicklers. Wer sie nicht kennt, kann kaum hochwertige objektorientierte Software entwickeln.

Kapitel 7 beschäftigt sich dann mit dem Thema Testen. Die diesbezüglichen Grundlagen werden erläutert. Weiterhin wird hier auf Automatiserungsmöglichkeiten eingegangen. Es ist wegen der Kosten und des Zeitbedarfs heute nur noch in weni-

gen Fällen sinnvoll, Testeingaben per Hand bei jedem Test erneut in den Rechner einzutippen. Viele Firmen haben, was die Automatisierung von Tests angeht, noch erhebliches Potenzial.

Bei der Softwareentwicklung sollte heute zur Steigerung der Produktivität auf eine Vielzahl von Werkzeugen zurückgegriffen werden, um u.a. Aspekte wie Versionsverwaltung, Modellierung, Implementierung und Testen zeitgemäß handhaben zu können. Hierauf wird in Kapitel 8 eingegangen.

Ein Thema, mit dem sich z.Z. viele Firmen beschäftigen, ist die so genannte Enterprise Application Integration, d.h. die Integration der verschiedenen, in einem Unternehmen vorhandenen Softwaresysteme. Ziel ist die Unterstützung neuer Geschäftsprozesse oder die Beschleunigung vorhandener Prozesse. Welche Ansätze hierfür in Frage kommen und welche Vor- und Nachteile diese haben, wird in Kapitel 9 ausgeführt.

Kapitel 10 beschäftigt sich mit der Modell-getriebenen Softwareentwicklung. Hierbei wird Code zum großen Teil oder manchmal sogar vollständig aus Modellen, meist UML-Modellen, generiert. Dies hat nicht nur den Vorteil, dass Modelle und Code konsistent bleiben, sondern ermöglicht auch eine erhebliche Steigerung der Produktivität in Situationen, in denen häufig neuen Versionen oder kundenspezisfische oder plattformspezifische Varianten eines Softwaresystems erstellt werden müssen. Auch wenn bereits einige Firmen, auch im Münsterland, die Modell-getriebene Softwareentwicklung produktiv einsetzen, so handelt es sich hier um eine Technologie, die ihren Zenit noch vor sich hat. Es ist davon auszugehen, dass sie in den nächsten Jahren an Bedeutung gewinnen wird. Ihr Nachteil liegt darin, dass sie mit einem erheblichen Einarbeitungsaufwand verbunden ist.

Tabelle 1.1 fasst die behandelten Technologien, Maßnahmen und Ansätze zusammen und gibt an, wo diese besonders geeignet sind. Die Darstellung in der Tabelle ist aus Platzgründen etwas pauschal. Eine genauere Erläuterung finden Sie in den jeweiligen Kapiteln.

Die Autoren dieser Broschüre sind an Ihren Anregungen und Anmerkungen sehr interessiert. Falls Sie solche haben, schicken Sie sie bitte per Email an:
kuchen@uni-muenster.de

Technologie / Ansatz	**Wo / wann geeignet?**
Prozessmodelle für die Software-Entwicklung	
Wasserfallmodell	bei kleineren und ggf. mittleren Projekten mit wenig innovativem Charakter
Extreme Programming	bei kleinen und mittleren Projekten mit innovativem Anteil und sich während der Entwicklung wandelnden Kundenwünschen
Unified Process	bei mittleren und größeren Projekten mit innovativem Anteil; inbesondere das iterative Vorgehen ist zur Risikoreduktion empfehlenswert
Aufwandsabschätzung und Risiko-Management	
Function-Point-Methode	falls Umrechnung von Function-Points in Mitarbeitermonate aus abgeschlossenen Projekten bekannt
Risiko-Management	bei Projekten mit innovativem Charakter und / oder neuen Technologien oder Plattformen
Software-Entwicklungsparadigmen und Modellierung	
Objekt-Orientierung	bei Informationssystemen; weniger geeignet bei manchen eingebetteten Systemen
UML	immer bei OO; zumindest Klassendiagramme und oft auch Aktivitätsdiagramme und Use-Case-Diagramme sowie manchmal weitere Diagramme wie State Charts sollten erstellt und gepflegt werden
Werkzeuge	
CASE-Tool	zumindest bei Verwendung von OO und UML
IDE	zur Steigerung der Produktivität immer zu empfehlen
CVS	insbesondere bei Teamarbeit immer zu empfehlen
Testautomatisierung	sollte zur Steigerung der Produktivität soweit wie möglich genutzt werden
Software-Entwurf	
Schichtenarchitektur	zur Steigerung der Übersichtlichkeit, Wartbarkeit und Anpassbarkeit (fast) immer zu empfehlen
Middleware	zur Senkung des Entwicklungsaufwands bei Informationssystemen zu empfehlen, sofern keine extremen Effizienzanforderungen bestehen
Entwurfsmuster	sollte jeder OO-Entwickler kennen, da sie u.a. die Wartbarkeit und Flexibilität erhöhen
Sonstiges	
EAI	bei großen Unternehmen mit bisher nicht oder schlecht verbundenen Einzelsystemen
MDA /MDSD	bei vielen Versionen / Varianten eines Softwaresystems zur Steigerung der Produktivität nach allerdings erheblichem Einarbeitungsaufwand

Tabelle 1.1: Eignung von Technologien, Maßnahmen und Ansätzen.

Kapitel 2

Vorgehensmodelle in der Software-Entwicklung

Jedes Softwareprojekt einer nennenswerten Komplexität sollte in einem mehr oder weniger festgelegten organisatorischen Rahmen verlaufen. Obwohl diese Erkenntnis nicht neu ist, verlaufen zahlreiche Projekte noch immer nach dem anarchischen Code- and Fix-Ansatz oder einer Variante des Wasserfall-Modells. In Literatur und Praxis hat sich eine verwirrende Vielfalt von Prozessmodellen und Entwicklungsmethodologien herausgebildet. Dennoch oder gerade deswegen scheitern immer noch zahlreiche Projekte aufgrund von Fehlsteuerungen im Rahmen des Projektmanagements. Laut einer Untersuchung der Standish Group wurden 2004 lediglich 29% aller IT Projekte erfolgreich abgeschlossen. Als endgültig gescheitert galten 18% aller Vorhaben. Mehr als die Hälfte aller Projekte (53%) konnten nur nach zum Teil erheblichen Termin– oder Budgetüberschreitungen bzw. einer Reduzierung des Funktionsumfangs abgeschlossen werden. Neben einer fehlenden Disziplin bei der Durchführung von Tests und einer unklaren oder lückenhaften Erfassung der Anforderungen werden nach wie vor Projektmanagementprobleme aufgrund eines fehlenden oder ungeeigneten Vorgehensmodells für die Misere verantwortlich gemacht. Der vollständige Verzicht auf ein Vorgehensmodell stellt für Projekte, die eine gewisse Größe überschreiten, die denkbar schlechteste Alternative dar. Trotzdem kann auch die Verwendung eines Vorgehensmodells, dessen Eigenschaften das Projektmanagement in einem konkreten Fall nur unzureichend unterstützen oder sich sogar kontraproduktiv auswirken, zu erheblichen Problemen führen. Die verfügbaren Vorgehensmodelle unterscheiden sich zum Teil erheblich in ihren Voraussetzungen, Zusicherungen und Rahmenbedingungen. Daher kann sich ein Vorgehensmodell, welches sich in einem Projekt bewährt hat, in einem nur geringfügig anders gearteten Projekt als fataler Fehlgriff erweisen. Dennoch neigen viele Projektleiter dazu, aus Bequemlichkeit, Routine oder mangelnden Kenntnissen, für unterschiedliche Projekte stets die gleichen Vorgehensmodelle einzusetzen. Insbesondere Varianten des

Wasserfall-Modells werden aufgrund ihrer Einfachheit und ihrer hohen Verbreitung in der Praxis vielmals als eine Art Standardmethodologie eingesetzt.

Ein Vorgehensmodell oder Prozessmodell sollte Vorgaben für folgende Bereiche der Projektplanung, –überwachung und –steuerung geben.

- sachlogische Strukturierung des Arbeitsablaufs (Gliederung in Entwicklungsschritte, Phasen oder Iterationen)
- Festlegung der jeweils durchzuführenden Aktivitäten
- Fixierung der in den einzelnen Gliederungsabschnitten zu erstellenden Leistungen
- Definition von Kriterien zur Überwachung des Projektfortschrittes (Meilensteine, Artefakte)
- Ermittlung der benötigten sowie verfügbaren personellen und materiellen Ressourcen
- Festlegung sowie Abgrenzung von Rollen, Kompetenzen und Verantwortlichkeiten
- Ausarbeitung der anzuwendenden Standards, Richtlinien, Methoden und Werkzeuge

Die zentralen Aspekte, für die ein Vorgehensmodell Handlungsempfehlungen bieten sollte, lassen sich kurz und prägnant mit folgender Frage zusammenfassen:

Wer (welche Akteure) macht wann (zeitliche und sachlogische Abfolge) wie (welche Standards, Richtlinien, Werkzeuge und Methoden) was (welche Artefakte)?

Vorgehensmodelle für die Softwareentwicklung lassen sich zwei grundlegenden Kategorien zuordnen, den phasenorientierten und den agilen Vorgehensmodellen. Als Orientierungshilfe werden im Folgenden mit dem Rational Unified Process (RUP) und Extreme-Programming (XP) zwei Vertreter dieser unterschiedlichen Denkrichtungen vorgestellt. Zuvor soll jedoch zum Vergleich das Wasserfallmodell skizziert werden, welches eines der einfachsten phasenbasierten Vorgehensmodelle darstellt. Aufgrund der großen Verbreitung des Wasserfallmodells in der Praxis, soll auf eine ausführliche Darstellung jedoch verzichtet und vielmehr auf die Schwachstellen dieses Ansatzes eingegangen werden.

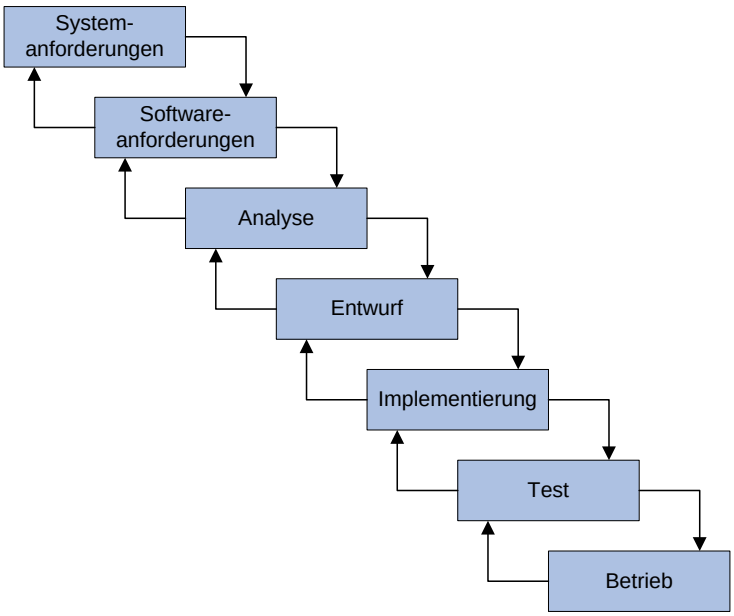


Abbildung 2.1: Das Wasserfallmodell.

2.1 Das Wasserfall-Modell

Das Wasserfall-Modell [Bal98, S. 99 f] gliedert den Entwicklungsprozess in mehrere Phasen, die sukzessive abgearbeitet werden und an deren Ende das fertige Softwareprodukt steht. In der ursprünglichen Form werden die Phasen Anforderungsdefinition, Analyse, Entwurf, Implementierung, Test und Inbetriebnahme unterschieden. Der Entwicklungsablauf verläuft sequentiell, d.h. eine Aktivität muss vollständig beendet werden, bevor die nächste gestartet werden kann (vgl. Abbildung 2.1). Um auf Änderungen eingehen oder nachträglich Korrekturen vornehmen zu können, sind in jeder Phase Rücksprünge auf vorangegangene und bereits abgearbeitete Phasen möglich. Aufgrund seiner Einfachheit und des geringen Managementaufwands wird das Wasserfall-Modell in der Praxis verbreitet eingesetzt. Allerdings gibt es auch eine Reihe von Kritikpunkten:

- Das Wasserfall-Modell verleitet aufgrund seines sequentiellen Charakters dazu, die Anforderungsdefinition zu Beginn des Projektes vorzunehmen und anschließend für beendet zu erklären. Ergeben sich im weiteren Verlauf des Projektes Änderungen an den ursprünglich erhobenen Anforderungen oder Ergänzungen, so können diese in einer späten Projektphase nicht mehr oder nur mit enormen Aufwand berücksichtigt werden.

- Test- und Integrationsaktivitäten sieht das Wasserfall-Modell erst nach dem Abschluss der Implementierungsphase vor. Fehler werden oftmals erst dann erkannt, wenn das Projekt weit fortgeschritten und ein Grossteil des verfügbaren

Budgets aufgebraucht ist. Zudem steigt der Aufwand für die Behebung von Fehlern mit zunehmendem Fortschritt des Projektes überproportional an. Insbesondere die Beseitigung von Problemen, die auf Fehlentscheidungen in der Definitions– oder Entwurfsphase zurückgehen, muss zu diesem späten Zeitpunkt teuer erkauft werden.

- Ein Projekt, welches dem Wasserfall-Modell folgt, liefert in der Regel erst sehr spät eine lauffähige Version der zu entwickelnden Software an den Kunden aus. Stellt dieser fest, dass seine Anforderungen nicht vollständig oder nicht in der gewünschten Form umgesetzt wurden, so ist eine Anpassung in den allermeisten Fällen nicht mehr möglich. Eine frühe Auslieferung von Teilfunktionalitäten ermöglicht es, frühzeitig ein Feedback durch den Kunden zu erhalten. Mögliche Fehlsteuerungen, die sich aus einer unklaren oder unvollständigen Erhebung der Anforderungen ergeben können, lassen sich auf diese Weise rechtzeitig korrigieren.

2.2 Der Rational Unified Process

Der Rational Unified Process (RUP) [Rat07] ist ein iteratives Vorgehensmodell zur Softwareentwicklung, welches von der Firma Rational Software (heute IBM Rational Software) auf der Grundlage des Unified Process entwickelt wurde [Rat07]. Der Unified Process wurde parallel zur Unified Modelling Language (UML) [Oes05] von Ivar Jacobson, Grady Booch und James Rumbaugh entwickelt und kann als Metamodell für Vorgehensmodelle zur (objektorientierten) Softwareentwicklung aufgefasst werden. Der RUP stellt eine konkrete Implementierung des Unified Process dar und basiert ebenfalls auf der UML als Modellierungssprache. Als zentrale Prinzipien der Softwareentwicklung werden die Orientierung an *Anwendungsfällen* (use cases) als Ausgangspunkt für die Entwicklung, die *Architekturzentrierung* der Planung sowie ein *inkrementelles und iteratives* Vorgehen in den Mittelpunkt gestellt. Die Orientierung an den Anwendungsfällen bzw. Geschäftsprozessen bewirkt, dass sich die Anwendungsfälle wie ein roter Faden durch die Entwicklung ziehen und jeder Entwicklungsschritt sich letztlich auf einen Anwendungsfall zurückführen lässt. Features, bei denen sich die Entwickler selbst verwirklicht haben, die aber von den Benutzern nicht verlangt wurden, werden dadurch verhindert. Die Architekturzentrierung verlangt, dass bereits in einem frühen Stadium die Grobarchitektur (z.B. 4-Schichten-Webapplikation basierend auf Java EE und einer r5elationalen Datenbank) auf der Grundlage einer kleinen Auswahl der wichtigsten Geschäftsprozesse festgelegt wird, so dass die restlichen Anwendungsfälle auf der Grundlage dieser Grobarchitektur leichter formuliert werden können und Anwendungsfälle, die mit dieser Grobarchitektur nicht verträglich sind, von Anfang an anders formuliert werden. Die iterative, inkrementelle Entwicklung verlangt, dass zunächst ein Kernsystem basie-

rend auf wenigen, ausgewählten Geschäftsprozessen erstellt wird, dass dann in jeder Iteration um weitere Funktionalität ergänzt wird. Diese iterative Vorgehen ist ganz entscheidend für die Verminderung der Risiken bei der Softwareentwicklung und der Kosten für die Fehlerbehebung. Wenn sich eine Entwurfs- oder Implementierungsentscheidung als ungeeignet herausstellt, so ist hier im Gegensatz zum Wasserfallmodell nicht das Gesamtsystem, sondern typischerweise die letzte Iteration betroffen, die dann vergleichsweise kostengünstig korrigiert werden kann. Die drei zentralen Prinzipien des Unified Process sind auch losgelöst hiervon interessant und können auch in eigenen, auf das jeweilige Projekt zugeschnittenen Vorgehensmodellen verwendet umgesetzt werden.

Die Unified Process basiert auch auf den folgenden Konzepten:

- Phasen ermöglichen eine zeitliche und sachlogische Gliederung des Entwicklungsprozesses. Jede Phase dient einem bestimmten Zweck, umfasst bestimmte Aktivitäten und führt zu einem definierten Ergebnis (Meilenstein). Der RUP unterscheidet zwischen Konzeptionsphase, Entwurfsphase, Konstruktionsphase und Übergabephase.

- Iterationen ermöglichen es, komplexe Teilaufgaben einzelner Phasen in überschaubare Arbeitspakete zu zergliedern. Den Ausgangspunkt einer Iteration bildet jeweils eine intersubjektiv nachprüfbare Zieldefinition, welche am Ende der Iteration herangezogen wird, um den Grad der Zielerreichung festzustellen.

- Meilensteine stellen die Verknüpfung von klar definierten Ergebnissen mit einem festgelegten Zeitpunkt dar und ermöglichen es den Fortschritt des Projektes zu überwachen.

- Aktivitäten stellen die einzelnen Aufgaben, Arbeitsschritte oder Arbeitspakete dar, welche im Rahmen des Entwicklungsprozesses zu erledigen sind. Die Granularität der Aktivitäten sollte so gewählt werden, dass sich diesen jeweils ein sinnvolles und abgrenzbares Ergebnis sowie eindeutig formulierte Kompetenzen und Verantwortlichkeiten zuordnen lassen.

- Disziplinen ermöglichen eine inhaltliche Strukturierung der Entwicklungsaktivitäten. Der RUP unterscheidet die sechs Kerndisziplinen Geschäftsprozessmodellierung, Anforderungsanalyse, Analyse und Design, Implementierung sowie Test und Softwareverteilung. Hinzu kommen die drei unterstützenden Disziplinen Konfigurations– und Änderungsmanagement, Projektmanagement sowie Infrastrukturmanagement.

- Akteure ermöglichen die Verknüpfung von Aufgaben und Artefakten mit den dafür verantwortlichen Personen und Rollen.

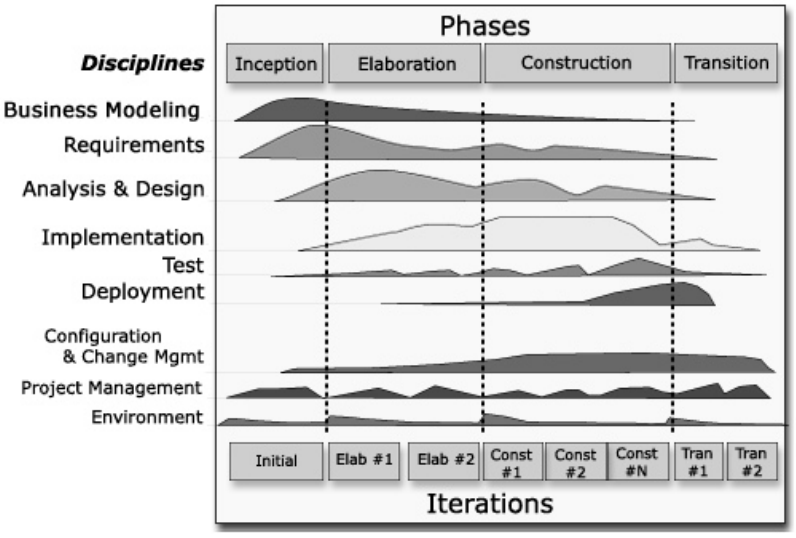


Abbildung 2.2: Der Rational Unified Process.

Abbildung 2.2 verdeutlicht das Zusammenspiel der vorgestellten Grundkonzepte innerhalb des Entwicklungsprozesses. In den vier Projektphasen werden die nach Disziplinen geordneten Aktivitäten durch festgelegte Akteure ausgeführt. Die einzelnen Aktivitäten fallen in den vier Phasen des Projektes in unterschiedlich hohem Maße an, was durch die Darstellung in Form eines Gebirges verdeutlicht wird. Auch in frühen Phasen fallen hier bereits Implementierungs- und Test-Arbeiten an, z.B. wenn es darum geht, anhand eines kleinen Beispiels herauszufinden, ob zwei Plattformen oder Frameworks verträglich sind. Die Phasenziele können in Teilziele zerlegt werden, welche jeweils durch eine Iteration abgearbeitet werden. Der Projektfortschritt wird durch Meilensteine überprüft und dokumentiert, welche den Abschluss einer Phase oder Iteration markieren können. Charakteristisch für den RUP sind sechs so genannte Best Practises, welche bei der Umsetzung der Grundprinzipien helfen sollen. Im Einzelnen sind dies Iterative Entwicklung, Anforderungsmanagement, Architekturzentrierte Entwicklung, Visuelle Modellierung (in der Regel mit UML), Qualitätssicherung und Änderungsmanagement.

Im Rahmen eines RUP-Projektes müssen neben der eigentlichen Software zum Zweck des Projektmanagements eine ganze Reihe zusätzlicher Dokumente erstellt werden. Der hohe Strukturierungsgrad des RUP und seine starke Ausrichtung auf formale Dokumente und Definitionen, macht dieses eher „schwergewichtige“ Vorgehensmodell in erster Linie für Projekte mit mehr als 10 Beteiligten und einer Dauer von mehreren Monaten interessant. Sowohl die Eigenschaften des zu entwickelnden Systems, als auch der Verlauf des Entwicklungsprozesses sind in hohem Grade nachvollziehbar. Aus diesem Grund und aufgrund der besonderen Betonung des Anforderungsmanagements sowie der Qualitätssicherung, eignet sich der Ratio-

nal Unified Process insbesondere für Projekte, die hohen Qualitätsanforderungen gerecht werden müssen. Das umfassende Projektmanagementinstrumentarium erlaubt es, auch große Projekte mit vielen Beteiligten und heterogener Interessenlage koordinieren zu können. In kleineren Teams oder kurzfristigeren Projekten führt der Einsatz des RUP schnell zu einem Verwaltungs-Wasserkopf, der das Projekt mit unnötiger Bürokratie belastet. Ein weiterer Kritikpunkt ist die fehlende Flexibilität, kurzfristig auf neue Anforderungen oder veränderte technologische Rahmenbedingungen reagieren zu können. Um unterschiedlichen Rahmenbedingungen einzelner Projekte gerecht werden zu können, ist der Rational Unified Process in hohem Maße anpassbar. Entsprechend der jeweiligen Anforderungen an das Projektmanagement können einzelne Teile des RUP auf eigene Erfordernisse zugeschnitten oder anstatt des gesamten Prozesses nur eine sinnvolle Teilmenge der Handlungsempfehlungen eingesetzt werden.

2.3 Extreme-Programming

Extreme-Programming (XP) [BA04] ist ein von Kent Beck, Ward Cunningham und Ron Jeffries entwickelte Methodologie für die Entwicklung von Software. Extreme-Programming ist der bedeutsamste Vertreter, der so genannten agilen Vorgehensmodelle. Diese entstanden aus der Kritik an traditionellen phasenorientierten Vorgehensmodellen. Als Hauptkritikpunkte wurden starre und zu restriktive Regelwerke, ausufernde Projektdokumentation, mangelnde Orientierung an den Kundenbedürfnissen, zu lange Integrations– und Releasezyklen sowie eine unzureichende Reaktionsfähigkeit auf Änderungen der Anforderungen oder technologischen Rahmenbedingungen genannt. In Anlehnung an die Schwierigkeiten in durch Bürokratie gekennzeichneten Organisationen, prägten die Autoren hierfür den Begriff der „Software Bürokratie“.

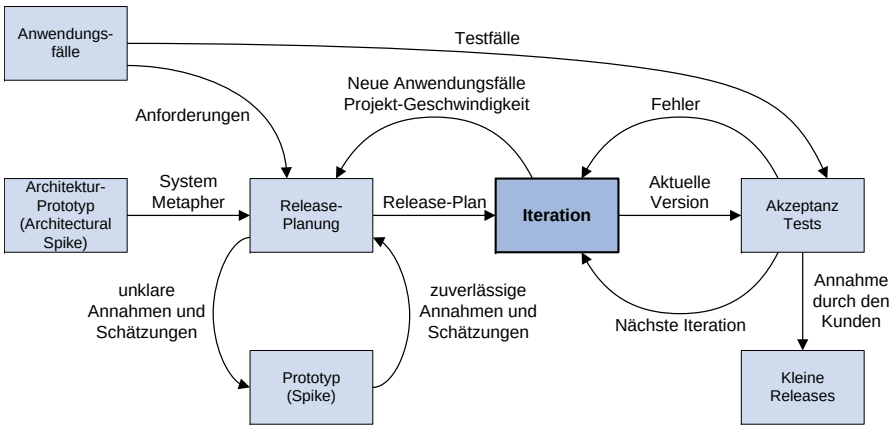


Abbildung 2.3: Der Extreme-Programming-Prozess nach [Wel07].

Extreme-Programming basiert auf den vier grundlegenden Werten Kommunikation, Einfachheit, Feedback und Mut. Ihre Konkretisierung finden sie in 15 Prinzipien, welche aus diesen Werten abgeleitet sind. Zur Umsetzung der Werte und Prinzipien stehen zwölf Praktiken zur Verfügung, die jeweils für sich betrachtet sinnvolle und in der Praxis bewährte Vorgehensweisen der Softwareentwicklung darstellen. Durch kombinierten Einsatz dieser Best Practises soll die Entwicklung qualitativ hochwertiger Software möglich werden, ohne gleichzeitig die Flexibilität aufzugeben, auf neue Anforderungen, Möglichkeiten oder Restriktionen angemessen reagieren zu können. Das Zusammenspiel der verschiedenen XP-Praktiken im Entwicklungsprozess wird durch Abbildung 2.3 veranschaulicht. Vier der wichtigsten XP-Praktiken sollen nun exemplarisch vorgestellt werden.

Testgetriebene Entwicklung

Um eine hohe Qualität der Software sicherzustellen, soll in einem XP-Projekt kein Stück Programmcode ungetestet an den Kunden ausgeliefert werden. Gleichermaßen wird angestrebt, fortwährend Integrationen durchzuführen und in kurzen Zeitabständen lauffähige Versionen an den Kunden auszuliefern. Auf diese Weise sollen Integrationsprobleme frühzeitig erkannt und regelmäßig ein Feedback von Seiten des Kunden eingeholt werden. Um beide Ziele auf wirtschaftliche Weise zu erreichen, ist einerseits eine möglichst vollständige Testabdeckung und andererseits ein hoher Automatisierungsgrad der Tests erforderlich. Die *Testfälle* werden stets auf der Grundlage der Anforderungen *vor der eigentlichen Implementierung* der zu testenden Programmteile erstellt. So lässt sich gewährleisten, dass Testfälle nicht im Hinblick auf eine problemlose Testbarkeit bereits implementierter Funktionalitäten gestaltet werden. Zu Beginn der Entwicklung werden alle Testfälle fehlschlagen, da noch keine Programmfunktionalität implementiert wurde. Im weiteren Verlauf dokumentieren die erfolgreich durchlaufenen Testfälle den Projektfortschritt. Weiterhin sind die Testfälle bei so genannten Regressionstests wichtig, um nach einer Änderung schnell und automatisch festzustellen, ob die bisher realisierte Funktionalität immer noch fehlerfrei verfügbar ist. Nur durch diese Regressionstests lässt sich ein Projektfortschritt sicherstellen.

Refactoring

Unter Refactoring [Fow05] versteht man die schrittweise und systematische Verbesserung der Struktur eines Softwaresystems unter Beibehaltung seiner Funktionalität. Der von Martin Fowler entwickelte Ansatz zur Coderestrukturierung gründet auf der Beobachtung, dass Softwaresysteme im Laufe der Zeit, bedingt durch zahlreiche Änderungen, Anpassungen und Erweiterungen, eine zunehmend unüberschaubare und chaotische Struktur annehmen. Bezeichnend für diesen Zustand ist das Anti-Pattern des „Spaghetti-Codes“[Por07], welches auf plakative Weise ein Softwaresys-

tem mit einer undurchdringlichen Struktur und Kontrollflusslogik beschreibt. Aufgrund der immer schwieriger zu durchschauenden Programmstruktur wächst die Gefahr, dass Änderungen in einem Programmteil unerwünschte Seiteneffekte in anderen Teilen der Software auslösen, die sich nur mit hohem Aufwand aufdecken und beheben lassen. Ansatzpunkt für die Durchführung eines Refactorings sind so genannte „bad smells“, welche auf Schwachstellen im Code hinweisen. Beispiele für bad smells sind duplizierter Code, überlange Klassen, Methoden oder Parameterlisten, switch-Anweisungen oder temporäre Variablen. Durch eine iterative Verbesserung der Codequalität mithilfe von elementaren Refactorings können Schwachstellen und potentielle Fehlerquellen im Code beseitigt werden. Durch die erhöhte Qualität des Quelltextes verbessert sich darüber hinaus die Verständlichkeit, Wartbarkeit und Erweiterbarkeit des Softwaresystems. Beispiele für Refactoring-Schritte sind das Verschieben von Klassen, Methoden oder Feldern, die Zusammenführung logisch zusammen gehöriger Felder in einem Objekt oder die Nutzung von Subklassen und Polymorphie zur Vermeidung von switch Anweisungen. Ausgangspunkt und Ziel jedes Refactoring-Schrittes ist stets eine lauffähige Version des Softwaresystems. Um das Risiko unerwünschter Seiteneffekte möglichst gering zu halten, sind komplexe Restrukturierungen in elementare Refactorings zu zerlegen. Am Ende jedes Refactorings stehen Tests, die die Funktionalität des geänderten Systems überprüfen. Um diese vollständig und effizient durchführen zu können, ist eine hohe Abdeckung mit automatisierten Tests erforderlich. Andernfalls werden möglicherweise nicht alle Fehler aufgedeckt oder der Zeitaufwand für die Durchführung der Refactorings steigt unverhältnismäßig an. Auch für die Überprufung von Refactorings sind die vor der Implementierung erstellten Testfälle wichtig.

Fortwährende Integration

Bei der Verwendung traditioneller Vorgehensmodelle zur Softwareentwicklungen werden Integrationsmaßnahmen zumeist relativ weit ans Ende der Entwicklungsarbeiten gestellt. Werden in dieser späten Phase Integrationsprobleme festgestellt, die nicht selten auf grundsätzliche Schwachpunkte in der Systemarchitektur bzw. bei der Gestaltung der Komponenten und Schnittstellen hinweisen, so ist deren Korrektur mit einem überproportional höheren Aufwand verbunden, als zu einem früheren Zeitpunkt des Projektes. Zudem stehen für die Behebung der Probleme nur noch sehr beschränkte Ressourcen zur Verfügung, so das eine Termin– oder Budgetüberschreitung oder gar ein Abbruch des Projektes häufig unausweichlich ist. Durch eine möglichst frühzeitige und fortwährende Integration sollen Fehler im Zusammenwirken der einzelnen Komponenten frühzeitig aufgedeckt werden. Dies ermöglicht es, strukturelle Defizite zu einem Zeitpunkt zu erkennen, zu dem deren Korrektur noch mit einem verhältnismäßig geringen Aufwand möglich ist. Idealerweise sollten Integrationen täglich durchgeführt werden, wobei jede Integra-

tion eine lauffähige Version des Softwaresystems zum Ergebnis hat. Um die Integrationsfähigkeit einer Änderung und die Funktionalität des Gesamtsystems zu überprüfen, spielen *automatisierte Tests* eine tragende Rolle.

Einbeziehung des Kunden

Viele Entwickler neigen zu einer phasenorientierten Sicht– und Denkweise, bei der die Aufnahme der Kundenanforderungen in einer frühen Phase des Entwicklungsprozesses abgearbeitet und anschließend endgültig für beendet erklärt wird. Kunden verfügen jedoch zu Beginn des Projektes häufig nur über ein sehr unscharfes Bild ihrer Anforderungen und Erwartungen, welches sich im Verlauf des Projektes durch den Kontakt mit den Entwicklern und das Feedback aus der Betrachtung des entstehenden Systems allmählich konkretisiert und vervollständigt. Die Anforderungsanalyse ist demnach vielmehr ein wechselseitiger Lernprozess, bei dem die Entwickler im Projektverlauf immer mehr über die Bedürfnisse des Kunden erfahren, während dieser seine Vorstellungen über die Möglichkeiten und Restriktionen der Technologie erweitert. XP fordert daher eine intensive Einbeziehung des Kunden in den gesamten Entwicklungsprozess. Anforderungen werden dabei nicht vollständig zu Beginn des Projektes erhoben, sondern iterativ und projektbegleitend mithilfe informeller „Story Cards“.

Die größten Vorteile verspricht der Einsatz von XP nur dann, wenn alle 12 Praktiken gemeinsam eingesetzt werden. Dies stellt jedoch hohe Anforderungen an alle Beteiligten und lässt sich mit den Rahmenbedingungen eines Projektes oftmals nicht vereinen. Zum Beispiel ist eine intensive Zusammenarbeit mit dem Kunden nicht immer möglich oder auf wirtschaftliche Weise realisierbar. Kritisch zu beurteilen sind auch die hier nicht näher behandelten Techniken der Programmierung in Paaren und der gemeinsamen Verantwortlichkeit für die Quellcodebasis. Programmieren in Paaren bedeutet, dass bei der Erstellung des Quellcodes jeweils zwei Entwickler gemeinsam an einem Rechner arbeiten. Einer der beiden Entwickler erstellt den Code, während der andere über aktuelle Problemstellungen nachdenkt und den geschriebenen Code kontrolliert. Um einen intensiven Erfahrungsaustausch zu gewährleisten werden die Rollen regelmäßig getauscht und die Programmierteams durch Rotation immer wieder neu zusammengestellt. In einem Unternehmen, dessen Betriebsklima durch Neid, Misstrauen und Angst geprägt ist oder in einem Team dessen Mitarbeiter sehr unterschiedliche Qualifikationen besitzen, werden diese Praktiken schwerlich auf fruchtbaren Boden fallen. Arbeiten die Entwickler räumlich verteilt, so ist der Einsatz spezieller Werkzeuge für das „distributed pair-programming “erforderlich. Diese ermöglichen den Entwicklern eine einheitliche und konsistente Sicht auf den zu bearbeitenden Quellcode. Die Synchronisation der verteilt stattfindenden Benutzerinteraktionen erfolgt dabei automatisiert über

eine Datenverbindung. Jede Änderung wird somit unmittelbar für jeden Teilnehmer der Pair-Programming-Sitzung sichtbar. Entsprechende Erweiterungen sind für zahlreiche Entwicklungsumgebungen und Editoren verfügbar. In Projekten, deren Anforderungen bereits zu Beginn des Vorhabens feststehen und die nur geringfügigen Veränderungen der technologischen Rahmenbedingungen unterworfen sind, bringt die Flexibilität und Agilität von XP keinen nennenswerten Vorteil. Bei der Entwicklung sicherheitskritischer Anwendungen kann eine konsequente Befolgung der Werte von Extreme-Programming gar zu gefährlichen ad hoc Endscheidungen verleiten, die zu einem unberechenbaren und unzuverlässigen Systemverhalten führen können. Da Extreme-Programming dem Projektmanager für seine Arbeit nur wenige konkrete Handlungsempfehlungen mit auf den Weg gibt, hängt der Erfolg des Projektes in hohem Maße von dessen Fähigkeiten ab.

Auch dann wenn für ein konkretes Projekt der gemeinsame Einsatz aller XP-Techniken nicht sinnvoll oder möglich ist, stellen die einzelnen Extreme-Programming Praktiken nützliche und in der Praxis bewährte Techniken zur Erhöhung der Qualität und Produktivität in der Softwareentwicklung dar. Insbesondere die Einführung automatisierter Tests hilft, die Effizienz des Entwicklungsprozesses sowie die Qualität des Softwareproduktes zu steigern, da Fehler zuverlässiger erkannt werden und sich der Zeitaufwand für die Fehlersuche gleichzeitig vermindern lässt. Aufgrund ihres universellen Charakters lassen sich viele der im Rahmen von Extreme-Programming diskutierten Prinzipien und Vorgehensweisen auch innerhalb traditioneller Vorgehensmodelle sinnvoll einsetzen.

Extreme Programming ist nur für kleinere und mittlere Projekte mit maximal 15 Entwicklern geeignet. Größere Projekte erfordern mehr organisatorischen Aufwand.

Kapitel 3

Management-Aspekte

3.1 Aufwandsschätzung

Die Schätzung des Aufwands eines Projekts ist aufgrund der bei Projektanfang unzureichenden Informationen einerseits schwierig, andererseits aber für Planunng von Laufzeit, Kosten und Personaleinsatz unabdingbar. Vollständig überzeugende Lösungen für dieses Problem gibt es nicht. In vielen Fällen hilft ein simpler Vergleich des neuen Projekts mit einem oder mehreren ähnlich gelagerten, abgeschlossenen Projekten. Diesen Ansatz bezeichnet man auch als *Analogiemethode*. Besonders präzise Schätzungen kann sie aber in der Regel nicht liefern.

In empirischen Vergleichen verschiedener Schätzmethoden hat die so genannte *Function-Point-Analyse* (FPA) [PB05] oder Function-Point-Methode am besten abgeschnitten [Bal01]. Sie hat daher in Deutschland und auch weltweit eine beachtliche Benutzergemeinde, die IFPUG [IFP08]. Sie ist recht leicht und mit überschaubarem Aufwand durchführbar und führt zu relativ präzisen Ergebnissen. Selbst wenn man in der eigenen Firma kein eigenes Know-How bezüglich der Function-Point-Analyse aufbauen will, so kann man ggf. auf einen hierfür von der IFPUG zertifizierten Berater zurückgreifen.

Die FPA eignet sich nicht nur zur Schätzung des Aufwands von Software-Projekten, sondern u.a auch zur Bewertung von Altsystemen sowie von Angeboten von Auftragnehmern und bei Make-or-Buy-Entscheidungen.

Im Gegensatz zu älteren Verfahren, die mit unbefriedigendem Erfolg versucht haben, den Aufwand eines Projekts auf der Basis von ebenfalls unbekannten und geschätzten Codezeilen vorherzusagen, basiert die FPA auf den fachlichen Anforderungen, wie sie in einem Lasten- oder Pflichtenheft formuliert sind. Zur quantitativen Bestimmung des fachlichen Funktionsumfangs einer IT-Anwendung zerlegt die FPA diese aus Sicht der Anwender in ihre Elementarprozesse. Ein Elementarprozess ist die aus Anwendersicht kleinste sinnvolle und in sich abgeschlossene Aktivität, die mit dem System durchführbar ist. Unterschieden werden die Elementarprozesse *Einga-*

Eingaben			
$b \setminus e$	1-4	5-15	≥16
0-1	e	e	m
2	e	m	k
≥ 3	m	k	k

Ausgaben und Abfragen			
$b \setminus e$	1-4	6-19	≥20
0-1	e	e	m
2-3	e	m	k
≥ 4	m	k	k

Tabelle 3.1: Zuordnung von Komplexitätsstufen (einfach (e), mittel (m), komplex (k)) für Eingaben, Ausgaben und Abfragen abhängig von der Anzahl e der betroffenen Datenelemente und der Anzahl b der betroffenen Datenbestände.

$f \setminus e$	1-19	20-50	≥51
1	e	e	m
2-5	e	m	k
≥ 6	m	k	k

Tabelle 3.2: Zuordnung von Komplexitätsstufen (einfach (e), mittel (m), komplex (k)) zu internen Datenbeständen und Referenzdaten abhängig von der Anzahl e der betroffenen Datenelemente und der Anzahl f der betroffenen Feldgruppen.

be, Ausgabe und Abfrage (d.h. Ausgabe mit trivialer Verarbeitungslogik). Weiterhin betrachtet werden die vom System zu pflegenden *Datenbestände* und die genutzten, von externen Systemen gepflegten, so genannten *Referenzdaten*. Ein Elementarprozess kann z.B. die Erfassung einer Kundenadresse, der Ausdruck einer Rechnung, die Berechnung eines Tarifs oder die Anzeige eines Kontostands sein. Ein wesentliches Kriterium für die Identifikation eines Elementarprozesses ist seine Einmaligkeit. Ein Elementarprozess gilt dann als einmalig, wenn er durch die ein- oder ausgegebenen Daten oder durch die Verarbeitungslogik unterscheidbar ist. An der Oberfläche ähnliche Elementarprozesse, welche beispielsweise eine gemeinsame Bildschirmmaske für das Anzeigen und Erfassen von Kundendaten verwenden, lassen sich auf diese Weise differenzieren.

Jeder identifizierten Eingabe, Ausgabe und Abfrage wird in Abhängigkeit von der Anzahl der betroffenen Datenelemente (z.B. Name, Kundennummer) und Datenbestände (d.h. z.B. einer Tabelle der Datenbank) eine Komplexität zugeordnet (vgl. Tabellen 3.1). Bei Datenbeständen und Referenzdaten hängt die Komplexität von der Anzahl der betroffenen Datenelemente und der Anzahl der so genannten Feldgruppen ab (vgl. Tabelle 3.2). Eine Feldgruppe besteht aus inhaltlisch zusammengehörigen Datenelementen (z.B. Nachname und Vorname). Diese Bewertung erfolgt pro Elementarprozess genau einmal, unabhängig davon, wie häufig dieser in der Anwendung auftritt und genutzt wird. Die klassifizierten Elementarprozesse werden dann in ein Rechenblatt (Abb. 3.3) eingetragen und mit Punktzahlen bewertet. Durch Addition der so ermittelten Punktewerte erhält man zunächst die so genannten *unbewerteten Function Points* E_1. Diese werden dann unter Berücksichtigung

von insgesamt 14 Einflussgrößen (wie z.B. der Frage, ob das zu erstellende System auf mehreren Rechnern verteilt arbeiten soll oder ob schwer zu erfüllende Anforderungen an die Effizienz bestehen) um maximal 35 % verringert oder erhöht. Als Ergebnis erhält man dann die *bewerteten Function Points* E_3, die mithilfe einer Tabelle dann in Mitarbeiter-Monate umgerechnet werden können.

Kategorie	Anzahl	Klassifizierung	Gewichtung	$\sum$
Eingabedaten		einfach	× 3	
	9	mittel	× 4	36
		komplex	× 6	
Abfragen		einfach	× 3	
		mittel	× 4	
		komplex	× 6	
Ausgaben		einfach	× 4	
	3	mittel	× 5	15
		komplex	× 7	
Datenbestände	3	einfach	× 7	21
		mittel	×10	
		komplex	×15	
Referenzdaten		einfach	× 5	
		mittel	× 7	
		komplex	×10	
Summe (E_1)				72
Einflussfaktoren andern den FP-Wert um +/- 35 %		1 Datenkommunikation Frontend-Backend (0-5)		3
		2 Verteilte Verarbeitung (0-5)		4
		3 Leistungsanforderungen (0-5)		1
		4 Ressourcennutzung (0-5)		0
		5 Transaktionsrate (0-5)		1
		6 Online-Benutzerschnittstelle (0-5)		5
		7 Anwenderfreundlichkeit (0-5)		3
		8 Onlineverarbeitung (0-5)		4
		9 Komplexe Verarbeitung (0-5)		0
		10 Wiederverwendbarkeit (0-5)		0
		11 Migrations- u. Installationshilfen (0-5)		0
		12 Betriebshilfen (0-5)		0
		13 Mehrfachinstallationen (0-5)		0
		14 Änderungsfreundlichkeit (0-5)		1
Summe E_2 der Einfüsse				22
Bewertete Function Points $E_3 = \lceil E_1 \cdot (E_2/100 + 0.65) \rceil$				63

Tabelle 3.3: Beispiel: Ermittlung von Function Points.

3.2 Risiko-Management

Untersuchungen zeigen, dass zur Überarbeitung von fehlerhafter Software ungefähr 80% der Überarbeitungskosten benötigt werden, um lediglich 20% der Fehler zu beseitigen. Bei diesen 20% der Probleme handelt es sich um die risikoreichen Probleme, die im Rahmen des Software-Managements identifiziert und beseitigt werden müssen, bevor sie den Projekterfolg gefährden und solange die entsprechenden Überarbeitungskosten noch relativ gering sind.

Nach Balzert [Bal98] besteht das Risikomanagement aus sechs Schritten (s. Abb.

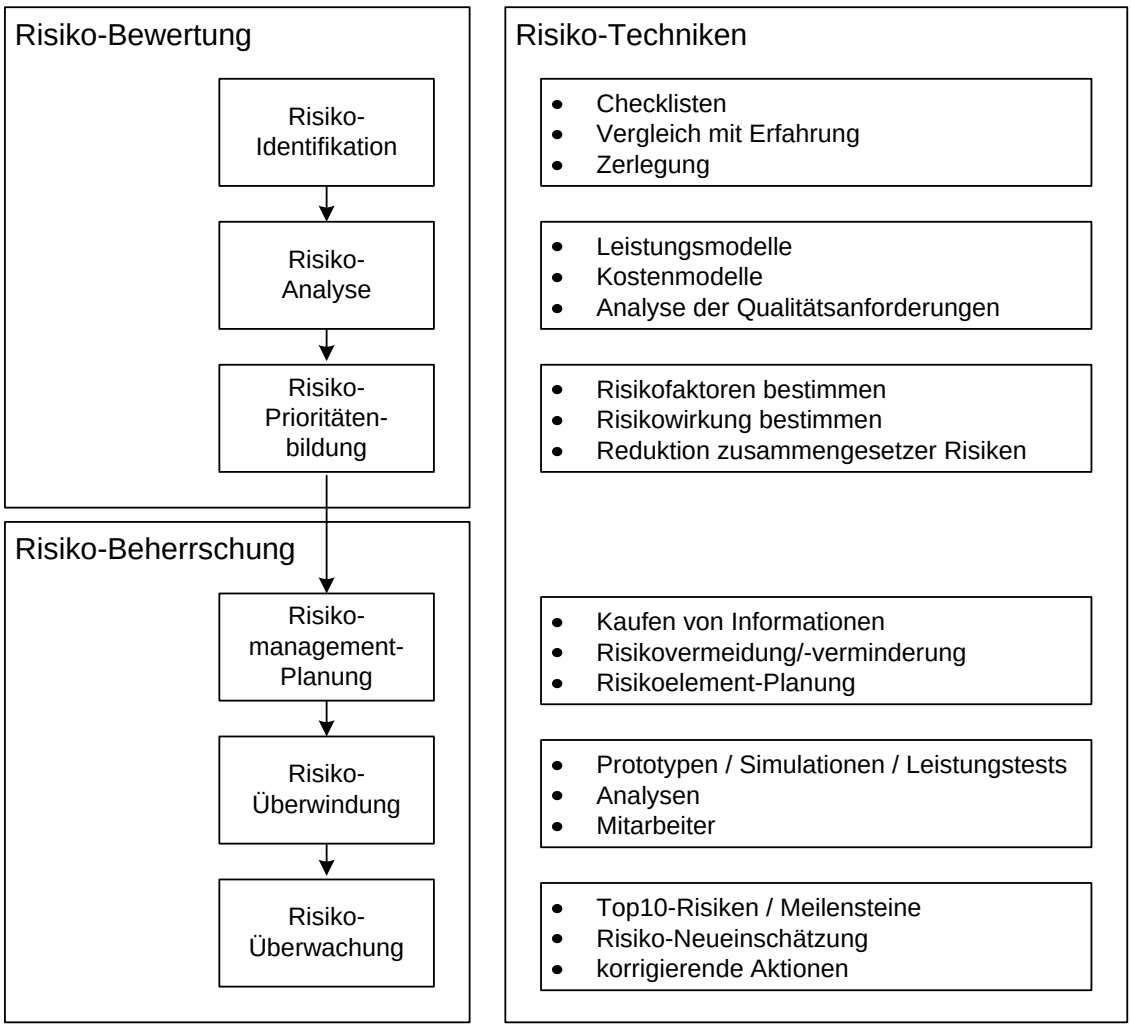


Abbildung 3.1: Die sechs Schritte des Risikomanagements.

3.1), denen jeweils mehrere Techniken zugeordnet werden können. Fallstudien zum Risiko-Management finden sich in z.B. in [Boe91, Boe89, Fai94].

3.2.1 Risiko-Identifikation

Checklisten sind ein hilfreiches Instrument, um im Rahmen der Risikoidentifikation die projektspezifischen Risiken zu finden. Tabelle 3.4 zeigt die zehn wichtigsten Quellen für Risiken in Software-Projekten und die entsprechenden Techniken, mit denen man die Risiken vermeidet oder überwindet.

3.2.2 Risiko-Analyse

Bei der Risiko-Analyse werden die Schadenswahrscheinlichkeit und das Schadensausmaß für jedes identifizierte Risikoelement und für zusammengesetzte Risiken abgeschätzt. Eine quantitative Bewertung eines Risikos erfolgt durch den so genannten Risiko-Faktor, der sich aus dem Produkt aus dem erwarteten Verlust bzw. Schaden und der Schadenswahrscheinlichkeit ergibt.

$$\text{Risikofaktor} = \text{Schadenswahrscheinlichkeit} * \text{Schadensausmaß}$$

Ein unbefriedigendes Ergebnis liegt vor, wenn die Hauptbeteiligten an einem Software-Projekt durch das Ergebnis einen Schaden erleiden:

1. für Kunden und Entwickler: Kosten- und Terminüberschreitungen,
2. für Benutzer: falsche Funktionalität, mit Defiziten der Benutzeroberfläche, Leistung oder Zuverlässigkeit,
3. für Wartungsingenieure: schlechte Qualität.

3.2.3 Risiko-Prioritätenbildung

Um zu verhindern, dass die wirklich relevanten Risiken angemessen beachtet werden, empfiehlt es sich, die Risiken nach ihrem Risikofaktor zu sortieren. Oftmals ist es sehr schwierig, die Eintrittswahrscheinlichkeiten hinreichend genau zu schätzen. Eine vollständige Risiko-Analyse würde jedoch teure und in der Entwicklung zeitaufwändige Prototypen, Leistungsmessungen und Simulationen erfordern, so dass man sich im Allgemeinen mit groben Schätzungen zufrieden gibt.

3.2.4 Risikomanagement-Planung

Nachdem die Hauptrisikoelemente ermittelt wurden, muss für jedes Risikoelement ein Risiko-Plan entwickelt werden, welcher die zur Kontrolle des Risikoelements notwendigen Aktivitäten spezifiziert. Eine Hilfe hierzu gibt Tabelle 3.4, in welcher erfolgreiche Risikomanagement-Techniken für die wichtigsten Risikoelemente aufgelistet sind. Der letzte Planungsschritt besteht darin, die Risikopläne in den übergeordneten Projektplan zu integrieren.

3.2.5 Risiko-Überwindung

Nachdem die Risikopläne aufgestellt wurden, werden die dort festgelegten Aktivitäten zur Risiko-Minimierung ausgeführt. Beispielsweise wird ein Prototyp erstellt oder es werden die Anforderungen gelockert.

Risikoelement	**Risikomanagement-Techniken**
1. Personelle Defizite	- qualifizierte Mitarbeiter einstellen - Teams zusammenstellen
2. Unrealistische Termin- und Kostenvorgaben	- detaillierte Kosten- und Zeitschätzung mit mehreren Methoden - Produkt an Kostenvorgaben orientieren - Inkrementelle Entwicklung - Wiederverwendung von Software - Anforderungen streichen
3. Entwicklung von falschen Funktionen und Eigenschaften	- Benutzerbeteiligung - Prototypen - Frühzeitiges Benutzerhandbuch
4. Entwicklung der falschen Benutzungsschnittstelle	- Prototypen - Aufgabenanalyse - Benutzerbeteiligung
5. Vergolden (Realisierung nicht geforderter Features)	- Anforderungen streichen - Prototypen - Kosten/Nutzen-Analyse - Entwicklung an den Kosten orientieren
6. Kontinuierliche Anforderungsänderungen	- hohe Änderungsschwelle - Inkrementelle Entwicklung (Änderungen auf spätere Erweiterungen verschieben)
7. Defizite bei extern gelieferten Komponenten	- Leistungstest - Inspektionen - Kompatibilitätsanalyse
8. Defizite bei extern erledigten Aufträgen	- Prototypen - Frühzeitige Überprüfung - Verträge auf Erfolgsbasis
9. Defizite in der Echtzeitleistung	- Simulation - Leistungstests - Modellierung - Prototypen - Instrumentierung - Tuning
10. Überfordern der Software-Technik	- Technische Analyse - Kosten/Nutzen-Analyse - Prototypen

Tabelle 3.4: Typische Risiken einer Software-Entwicklung.

3.2.6 Risiko-Überwachung

Die Fortschritte der Risiko-Minimierung müssen ständig überwacht werden, damit bei Abweichungen unmittelbar korrigierende Maßnahmen durchgeführt werden können. Eine bewährte Technik zur Risiko-Überwachung stellt die Verfolgung der *Top-10-Risiken* dar, welche es dem Manager ermöglicht, sich jeweils auf die kritischen Risikoelemente zu konzentrieren. Die Verfolgung der Top-10-Risiken beinhaltet folgende Schritte:

1. Sortieren der Risikoelemente nach ihrer Priorität,
2. Definieren von regelmäßigen Überprüfungsterminen,
3. Erstellen eines Fortschrittsbericht im Vergleich zum vorhergehenden Termin,
4. gezieltes Beseitigen der aktuellen Risikoelemente.

Die Risikoelemente werden regelmäßig neu bewertet, was zur Folge hat, dass einige der Risikoelemente eine niedrigere Priorität erhalten oder ganz verschwinden, während andere höhere priorisiert werden oder neu in die Liste aufgenommen werden.

Kapitel 4

Objektorientierung und Unified Modeling Language

Die Objektorientierung (OO) hat sich heute in der Softwareentwicklung weitgehend durchgesetzt. Einige Firmen werden objektorientiert entwickeln, weil dies im Trend liegt, ohne sich über die Vorteile vollständig im Klaren zu sein. Im Folgenden sollen diese Vorteile herausgearbeitet werden.

Der wohl am häufigsten genannte Grund für den Einsatz von Objektorientierung ist die *Kapselung* von Struktur und Verhalten oder, anders gesagt, von Daten und den auf ihnen arbeitenden Operationen, im OO-Kontext als Methoden bezeichnet. Diese Kapselung in so genannten Klassen ermöglicht zunächst eine bessere Strukturierung von Softwaresystemen, die nun als Systeme von Klassen organisiert werden können. Da nicht-triviale Softwaresysteme üblicherweise in Teamarbeit erstellt werden, ist es weiterhin wichtig, dass der verwendete Programmieransatz hierfür geeignet ist. Die erwähnte Kapselung gewährleistet dies, da sie eine Trennung von Nutzern und Entwicklern einer Klasse ermöglicht. Die Nutzer einer Klasse müssen nur deren Schnittstelle, gegeben durch die Namen der Methoden und die Typen ihrer Parameter und ihrer Ergebnisse, kennen, nicht aber deren Implementierung. Letztere interessiert nur die Entwickler. Diese haben auch die Möglichkeit, die Implementierung z.B. aus Effizienzgründen zu ändern, ohne dass dies auf die Nutzer einen Einfluss hat; vorausgesetzt, dass die Schnittstelle unverändert bleibt. Die erwähnte Kapselung findet sich aber nicht nur in objektorientierten Programmiersprachen, sondern auch in nicht-objektorientierten, modularen Sprachen wie z.B. Visual Basic, Ada und Modula. Die Kapselung allein kann also nicht der Grund für den Siegeszug der Objektorientierung sein.

Ein mindestens ebenso wichtiger Aspekt ist die *Vererbung*, die kennzeichnendes Merkmal objektorientierter Sprachen ist. Sie erlaubt, Spezialisierungen einer Klasse in Form von so genannten Unterklassen zu bilden, welche die Struktur und das Verhalten ihrer Oberklasse erben. Dies ermöglicht zunächst eine Wiederver-

wendung von Code. Methoden der Oberklasse brauchen in den Unterklassen nicht erneut implementiert zu werden. Bei Bedarf dürfen ausgewählte Methoden in einer Unterklasse aber, wie man sagt, *überschrieben*, d.h. neu implementiert, werden. Dies ermöglicht eine Reihe von mächtigen Programmiertechniken. Alle der in Unterkapitel 6 vorgestellten Entwurfsmuster verwenden Vererbung (oft in Kombination mit Überschreiben von Methoden), um die zur Lösung der betrachteten Probleme benötigte Flexibilität zu gewinnen. Vererbung sorgt beispielsweise dafür, dass das objektorientierte Gegenstück zu klassischen Software-Bibliotheken, die so genannten Frameworks, flexibler als konventionelle Bibliotheken sind. Letztere können ganz oder gar nicht genutzt werden. Bei der Verwendung von Frameworks dagegen können die fehlenden oder nicht passenden Methoden durch Überschreiben in selbst erstellten Unterklassen ergänzt bzw. ersetzt werden, so dass auch zunächst nicht vollständig passende Klassen genutzt werden können. Dieser Aspekt wird insbesondere bei dem Adpater-Muster in Unterkapitel 6 besonders deutlich.

Neben der Kapselung und der Vererbung kommen inzwischen aber noch einige weitere Aspekte hinzu, die für die objektorientierte Softwareentwicklung sprechen. Zunächst ist hier die *Unified Modeling Language* (UML) zu nennen. Erfreulicherweise ist es der objektorientierten Community gelungen, sich auf eine allgemein akzeptierte, standardisierte Modellierungssprache, die UML, zu einigen. Als Folge hiervon haben sich auch die Hersteller von Software-Engineering-Tools an dieser einheitlichen Notation orientiert, so dass inzwischen eine vorher nicht gekannte Vielfalt von weitgehend kompatiblen *Werkzeugen* bereit steht, die die objektorientierte Softwareentwicklung weiter vereinfachen.

Weiterhin sind objektorientierte Programmiersprachen wie Java und C# mit einem umfassenden Satz an *vordefinierten Klassen und Frameworks* für nahezu alle Aspekte der Softwareentwicklung ausgestattet, die die Anwendungsentwicklung zusätzlich enorm erleichtern. Im Falle von Java reicht dies von Ansätzen zur Gestaltung der Präsentationsschicht von Webanwendungen wie Servlets, JSP, Struts und JavaServer Faces über Konzepte zur Realisierung der Geschäftslogik und Datenhaltung von Informationssystemen wie Enterprise JavaBeans, Spring und Hibernate bis hin zu Frameworks für mobile Endgeräte.

Demgegenüber tritt der ursprüngliche Vorteil von Java, nämlich die Möglichkeit, spezielle Java-Klassen, so genannte Applets, über das Internet zu laden ud gefahrlos auf dem eigenen Rechner in einem geschützten Bereich, der so genannten Sandbox, ausführen zu können, deutlich in den Hintergrund.

In den nachfolgenden Abschnitten werden die grundlegenden Konzepte der Objektorientierung dargestellt. Eine ausführlichere Einführung in die Objektorientierung bieten [Zep04, Bal98, LR06]. Des Weiteren werden in Anlehnung an [Oes05, Stü05] einige der wichtigsten Diagrammarten der UML sowie deren Semantik beschrieben. Unter [OOS08] findet sich eine Gegenüberstellung von mehr als 100 UML-Werkzeugen und deren Leistungsumfang. Leser, die mit der Objektorientierung ud

der UML vertraut sind, können den Rest des Kapitels überspringen.

4.1 Grundkonzepte der Objektorientierung

4.1.1 Objekte und Klassen

Bei jedem Programmmieransatz müssen Objekte und Konzepte der Realwelt im Rechner repräsentiert werden. In der Objektorientierung geschieht dies durch so genannte Objekte. Jedes *Objekt* lässt sich durch zwei Merkmale charakterisieren: (1) es besitzt einen eindeutigen Zustand und ein wohldefiniertes Verhalten, und (2) unter allen Objekten ist es eindeutig identifizierbar. Nicht jedes relevante Objekt wird individuell konzipiert. Stattdessen beschreibt man Objekte mit gleicher Struktur und gleichem Verhalten durch eine so genannte *Klasse*. Ein einzelnes Objekt gehört immer nur zu einer Klasse und kann diese Zugehörigkeit nicht ändern. Ein Objekt wird auch als *Instanz* seiner Klasse bezeichnet.

4.1.2 Attribute und Operationen

Die Struktur eines Objekts ergibt sich aus seinen Bestandteilen bzw. den in ihm enthaltenen Daten. Letztere werden auch *Attribute* genannt. Die Objekte einer Klasse besitzen alle die gleichen Attribute, unterscheiden sich aber in den Werten, die diese annehmen. Zwei Objekte mit identischen Attributausprägungen können über die Objektidentitäten eindeutig unterschieden werden. Die Werte der Attribute können sich im Programmverlauf ändern. Dazu werden Operationen, auch *Methoden* genannt, definiert, die die Attributwerte manipulieren und das mögliche Verhalten der Objekte festlegen. Operationen werden in der jeweiligen Klasse definiert und sind für alle Objekte einer Klasse gleich. Eine direkte Manipulation der Attribute durch Operationen anderer Klassen sollte unterbleiben. Als spezielle Operationen einer Klasse sind die Konstruktoren sowie die Destruktoren zu nennen. Ein Konstruktor erzeugt ein neues Objekt einer Klasse, ein Destruktor löscht ein Objekt.

4.1.3 Vererbung

Mit dem Konzept der Vererbung lassen sich Spezialisierungsbeziehungen zwischen Klassen herstellen. Oftmals können gewisse Objekte einer Klasse mit weiteren speziellen Attributen ausgestattet werden, so dass diese Objekte eine spezialisierte Variante der allgemeinen Klassendefinition repräsentieren. Die in gleicher Weise spezialisierten Objekte bilden dann eine *Unterklasse* der Ausgangsklasse. Letztere wird dann auch als Oberklasse bezeichnet. Diese Objekte einer Unterklasse erben alle (nicht-privaten) Attribute und Operationen der Oberklasse. Zusätzlich können weitere Attribute und Operationen ergänzt werden. Eine Oberklasse kann hierbei so

allgemein gehalten sein, dass es nicht immer sinnvoll ist, hierfür konkrete Objekte zu bilden. Dies ist genau dann der Fall, wenn eine Klasse nur gemeinsame Attribute und Methoden für alle Unterklassen bereitstellen soll. Solche Klassen werden als *abstrakte Klassen* bezeichnet. Es können keine Objekte von abstrakten Klassen erzeugt werden.

4.2 Unified Modeling Language

Bei der objektorientierten Softwareentwicklung wird das zu erstellende System vor der Implementierung typischerweise in der allgemein akzeptierten, standardisierten Notation Unified Modeling Language (UML) modelliert [Oes05, Stï05]. Die UML stellt hierfür insgesamt sechs Strukturdiagramme zur Beschreibung statischer Strukturen, wie z.B. der Beziehungen zwischen Objekten, Klassen und Paketen, und sieben Verhaltensdiagramme zum Modellierung des dynamischen Verhaltens eines Sytems zur Verfügung. Die wichtigsten dieser Diagramme werden im Folgenden kurz vorgestellt, nämlich das Strukturdiagramm Klassendiagramm sowie die Verhaltensdiagramme Anwendungsfalldiagramm, Sequenzdiagramm und Aktivitätsdiagramm.

4.2.1 Klassendiagramme

Klassen und ihre Beziehungen werden in der UML durch so genannte Klassendiagramme dargestellt.

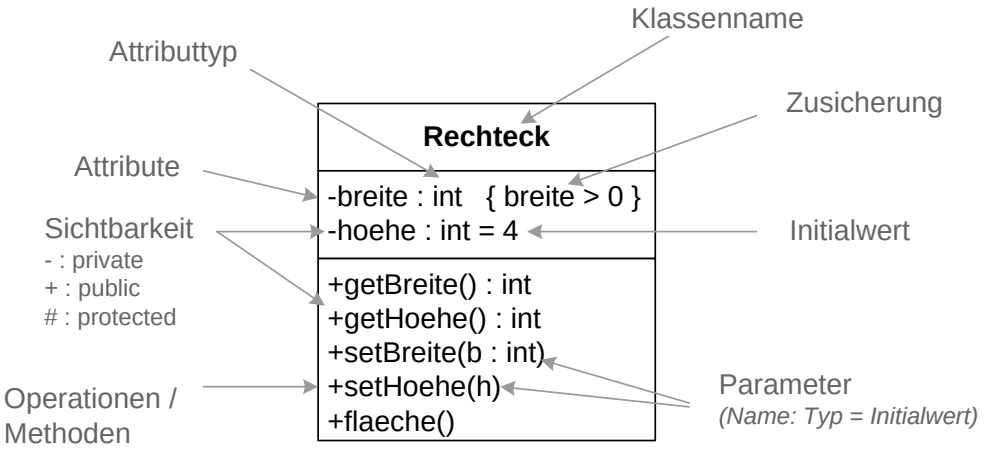


Abbildung 4.1: Beispiel: Klasse Rechteck im Klassendiagramm.

4.2.2 Darstellung einer Klasse im Klassendiagramm

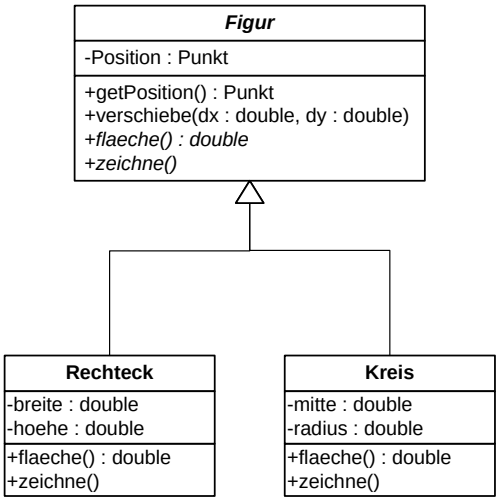
Die grafischen Darstellung einer Klasse in einem so genannten (UML-)*Klassendiagramm* besteht aus einem dreigeteilten Rechteck (s. Abb. 4.1). Im oberen Teil steht der Name der Klasse, im mittleren die Attribute und im unteren die Methoden. Der mittlere und untere Teil können weggelassen werden, wenn diese Details nicht betrachtet werden sollen. Die Attribute und Methoden sowie deren Parameter können durch eine Typangabe (z.B. `int`) näher charakterisiert werden. In der grafischen Darstellung wird der *Typ* vom zugehörigen Bezeichner durch einen Doppelpunkt getrennt (z.B. `breite : int`); *Initialwerte* können hinter einem angefügten „=“ einem Attribut zugeordnet werden (z.B. `hoehe : int = 4`). Zusätzliche Bedingungen, Voraussetzungen oder Regeln, die die Objekte erfüllen müssen, werden Zusicherungen genannt und in geschweiften Klammern hinter das zu spezifizierende Element geschrieben.

Über die *Sichtbarkeit* wird festgelegt, ob und in welcher Weise ein Zugriff auf die Attribute und Methoden erfolgen darf. Mögliche Sichtbarkeiten sind `private` (-), `protected` (#) und `public` (+). Als `private` deklarierte Attribute und Methoden sind nur innerhalb der Klasse selbst und nicht von außen sichtbar. Typischerweise schützt man so die Attribute einer Klasse vor unkontrolliertem Zugriff. Man spricht in diesem Zusammenhang auch von Datenkapselung. Auf die gleiche Weise können Hilfsmethoden verborgen werden, die nur innerhalb der Klasse genutzt werden sollen. Deklariert man Attribute oder Methoden mit `protected`, so sind diese nicht nur innerhalb der Klasse selbst, sondern auch für alle eventuell vorhandenen Unterklassen sichtbar (vgl. Unterkapitel 4.1.3). Mit `public` werden die Methoden ausgezeichnet, die die Schnittstelle einer Klasse nach außen bilden.

Für die Attribute einer Klasse werden typischerweise so genannte Set- und Get-Operationen bereit gestellt, mit denen die Attributwerte gesetzt bzw. ausgelesen werden können. Durch die kontrollierte Änderung der Attributwerte über die Set-Operationen kann ein konsistenter Zustand eines Objekts im Bezug auf die angegebenen Zusicherungen gewährleistet werden. So kann beispielsweise die `setBreite`-Operation der Klasse Rechteck beim Versuch, die Breite des Rechtecks auf einen negativen Wert zu setzen, eine Exception auslösen und auf diese Weise das Erreichen eines inkonsistenten Objektzustands unterbinden. Dieses Verhalten kann nicht sichergestellt werden, wenn das `breite`-Attribut als `public` deklariert ist, da so die ggf. zur Verfügung gestellte Set-Methode umgangen werden kann. Oft muss nicht auf jedes der vorhandenen Attribute lesend oder schreibend zugegriffen werden. Insofern liegt der Gedanke nahe, nur diejenigen Get- und Set-Operationen zu implementieren, die auch tatsächlich benötigt werden. Dennoch ist es sinnvoll, für jedes Attribut einer Klasse eine Set- und Get-Operation zu implementieren, damit die Klasse besser getestet werden kann. Da Get- und Set-Operationen sowie der Konstruktor einer Klasse üblicherweise vorhanden sind, werden sie in der Regel nicht explizit im Operationen-Block einer Klasse im Klassendiagramm aufgelistet.

Darstellung von Vererbung im Klassendiagramm

Im Klassendiagramm werden Vererbungshierarchien durch eine Verbindungslinie zwischen der Ober- und der Unterklasse dargestellt, die auf der Seite der Oberklasse durch ein Dreieck gekennzeichnet wird. Der Name einer abstrakten Klasse wird kursiv geschrieben.




```
public abstract class Figur {
  protected Punkt position;
  public Figur(Punkt position) {
    this.position = position;
  }
  public Punkt getPosition() {
    return position;
  }
  public void verschiebe(double dx,
                         double dy) {
    position.aendere(dx,dy);
  }
  // abstrakte Methoden
  public abstract double flaeche();
  public abstract void zeichne();
}

public class Rechteck extends Figur {
  double breite, hoehe;
  public Rechteck(double breite,
                  double hoehe,
                  Punkt pos) {
    this.breite = breite;
    this.hoehe = hoehe;
    super(pos);
  }
  public double flaeche() {
    return breite*hoehe;
  }
  public void zeichne() {...}
}

Public class Kreis extends Figur {
  // analog zu Rechteck
}
```

Abbildung 4.2: Beispiel: Klassenhierarchie und zugehöriger Java-Code.

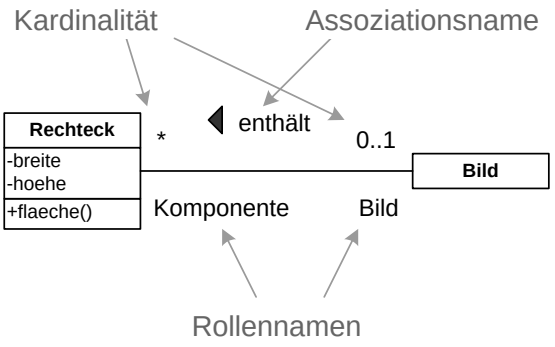
Abbildung 4.2 zeigt eine sehr einfache Vererbungshierarchie am Beispiel von geometrischen Figuren sowie deren Implementierung in Java. Die gemeinsamen Attribute und Methoden von konkreten Figuren wie Rechtecken und Kreisen werden in einer abstrakten Oberklasse `Figur` zusammengefasst, von der keine Instanzen gebildet werden können. Jede Figur ist also entweder ein Rechteck oder ein Kreis. Alle Figuren können gezeichnet werden und besitzen eine Fläche, jedoch unterscheiden sich die Implementierungen dieser Methoden, da sie in den Unterklassen jeweils

geeignet überschrieben werden.

Assoziationen

Verweisen Objekte einer Klasse dauerhaft auf Objekte einer anderen Klasse, so sagt man, dass die beiden Klassen *assoziert* sind. Im Klassendiagramm wird dies dadurch visualisiert, dass die beiden Klassen durch eine Linie verbunden werden. An dieser Linie kann annotiert werden, mit wie vielen Objekten der Nachbarklasse ein Objekt der betrachteten Klasse verbunden ist. * steht hierbei für beliebig viele Objekte; $n..m$ für mindestens n und höchstens m Nachbarobjekte. Weiterhin kann eine Assoziation mit einem Namen und mit Namen für die Rollen jeder Klasse auch Sicht der Nachbarklasse annotiert werden.

Eine Assoziation wird häufig dadurch implementiert, dass jede beteiligte Klasse Attribute vom Typ der Nachbarklasse bekommt. Gibt es nur höchstens ein Nachbarobjekt, so reicht ein einzelnes Attribut vom Typ der Nachbarklasse. Bei mehreren oder beliebig vielen Nachbarobjekten bekommt die Klasse üblicherweise als Attribut eine Kollektion (Array, Liste, Menge, . . .) von Objekten vom Typ der Nachbarklasse.




```
public class Rechteck {
  protected Bild bild;
  …
  public Rechteck(int b,
                  int h,
                  Bild abb) {
    breite=b; hoehe = h;
    bild = abb;
  }
  …
}
```

Abbildung 4.3: Klassendiagramm mit Assoziation.

In Abbildung 4.3 wird dargestellt, dass ein Bild beliebig viele Rechtecke enthält, während ein Rechteck in höchstens einem Bild vorkommt.

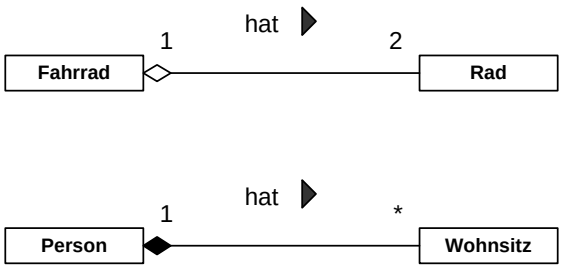


Abbildung 4.4: Klassendiagramm mit Aggregation und Komposition.

Eine spezielle Assoziation, welche eine Ganzes-Teil-Beziehung ausdrückt, nennt man *Aggregation*. Sofern es einem Modellierer wichtig ist, diese spezielle Art der Beziehung herauszustellen, kann er die beteiligten Klassen statt durch eine durchgehende Linie auch durch eine Linie verbinden, die mit einer weißen Raute beginnt. In Abbildung 4.4, oben, wird modelliert, dass ein Fahrrad zwei Reifen enthält.

Ist eine Ganzes-Teil-Beziehung so geartet, dass beim Löschen des Objektes für das Ganze auch die Teil-Objekte ihre Existenzberechtigung verlieren, so kann man dies bei der Modellierung durch eine spezielle Form von Aggregation, eine *Komposition*, zum Ausdruck bringen. Im Klassendiagramm wird statt einer weißen dann eine schwarze Raute verwendet.

In Abbildung 4.4, unten, wird beispielsweise dargestellt, dass eine Person beliebige viele Wohnsitze hat. Stirbt die Person, so hat sie auch keine Wohnsitze mehr (sondern es gibt höchstens noch Wohnungen). Aus Implementierungssicht zeigt eine Komposition an, dass hier ein so genanntes kaskadierendes Löschen realisiert werden muss.

Man beachte, dass in obigem Fahrrad-Beispiel ein Rad auch bei einem anderen Fahrrad montiert werden kann, wenn das ursprüngliche Fahrrad verschrottet wird. Daher wurde hier keine Komposition modelliert.

Beipiel 1: Krankenhausanwendung

Nachdem die Bestandteile eines Klassendiagramms nun erläutert wurden, sollen nun zwei Beipielanwendungen gezeigt werden, in denen diese verwendet werden.

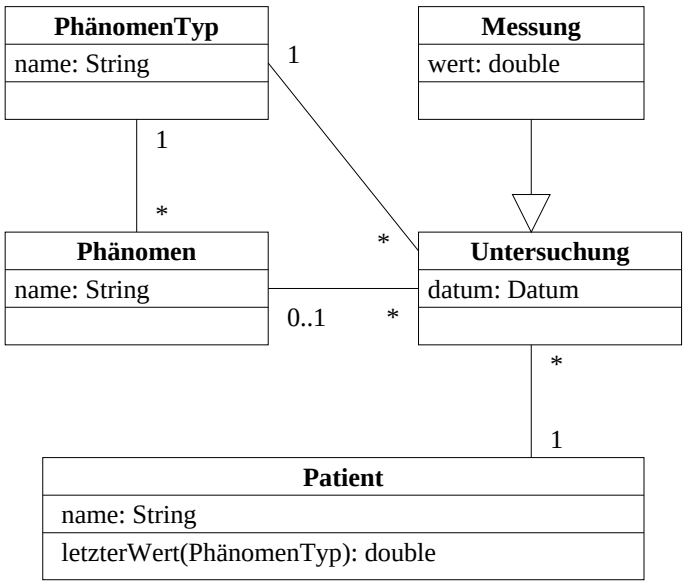


Abbildung 4.5: Klassendiagramm für ein Krankenhaus-Softwaresystem.

Abb. 4.5 zeigt ein Beispiel-Klassendiagramm für eine Krankenhausanwendung. Hier wird modelliert, dass Patienten untersucht werden. Eine Messung ist eine spezielle Untersuchung, daher wird hier Vererbung eingesetzt. Jeder Untersuchung ist ein Phänomentyp (z.B. Blutgruppe) und ggf. ein Phänomen (z.B. A+) zugeordnet. Bei Messungen gibt es manchmal kein zugeordnetes Phänomen, sondern nur einen gemessenen Wert; so z.B. bei Messungen zum Phänomentyp Körpertemperatur,

Beispiel 2: Arithmetischer Ausdruck

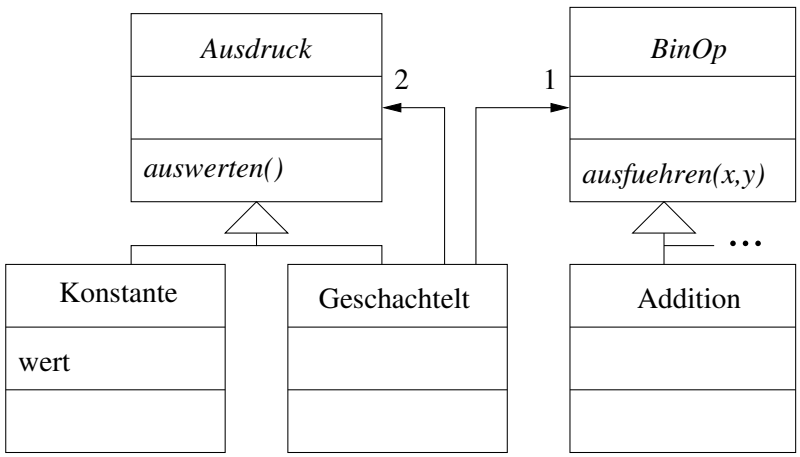


Abbildung 4.6: Klassendiagramm für arithmetische Ausdrücke.

Ein arithmetischer Ausdruck ist entweder einfach eine Konstante oder ein geschachtelter Ausdruck, bei dem zwei Teilausdrücke durch eine binäre Operation, wie z.B. die Addition, verknüpft werden (vgl. Abb. 4.6). Die Pfeilspitzen an den Assoziationen sagen aus, dass der Zugriff auf die jeweiligen Nachbarobjekte nur in Pfeilrichtung möglich ist. Von einem geschachtelten Ausdruck kann also beispielsweise auf die beiden Teilausdrücke zugegriffen werden, nicht aber umgekehrt.

4.2.3 Anwendungsfalldiagramme

Ein *Anwendungsfall* (use case) beschreibt eine konkrete funktionalen Anforderung an ein Anwendungssystem bzw. eine Interaktion zwischen einem *Akteur* und dem System. Hierbei wird nur beschrieben, was das System leisten muss, aber nicht wie es das leisten soll. Ein Anwendungsfall wird durch einen Akteur initiiert. Ein Akteur kann sowohl ein Anwender als auch ein anderes Anwendungssystem sein. Üblicherweise werden nicht die konkreten, beteiligten Personen unterschieden, sondern ihre Rollen, die sie im Kontext eines bestimmten Anwendungsfalls einnehmen (z.B. Händler und Kunde).

Ein *Anwendungsfalldiagramm* zeigt die Zusammenhänge und Abhängigkeiten zwischen Anwendungsfällen und Akteuren. Ein als Strichmännchen dargestellter Akteur ist im Diagramm durch eine Linie mit einem oder mehreren als Ellipse dargestellten Anwendungsfällen verbunden, an denen er beteiligt ist. Ein Beispiel für ein Bibliothekssystem ist in Abbildung 4.7 dargestellt.

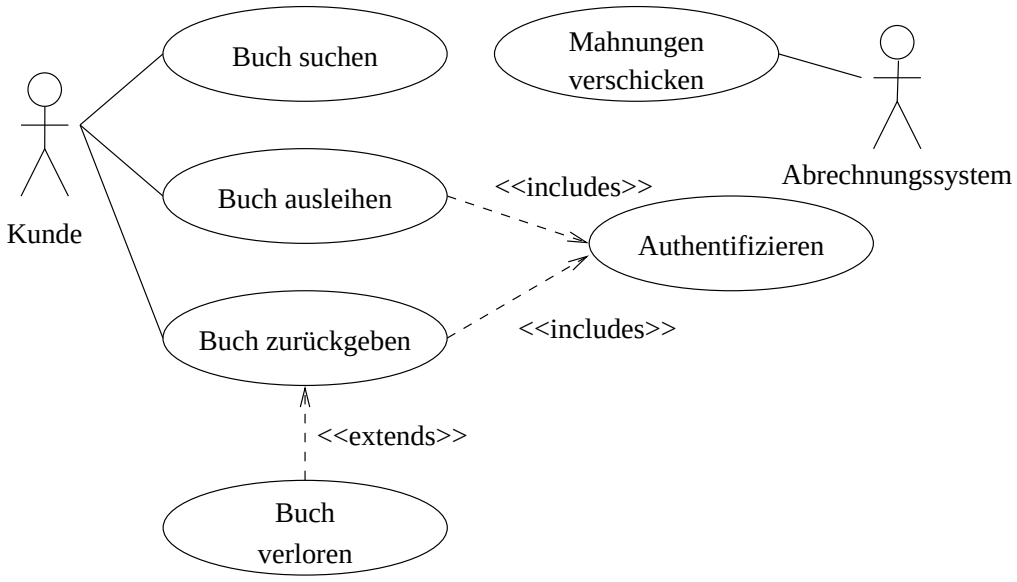


Abbildung 4.7: Beispiel: Anwendungsfall-Diagramm.

Ein Anwendungsfall beschreibt meist nur den Normalfall einer Interaktion zwischen Akteur und System. Die Behandlung von Fehlern und Sonderfällen wird zur Vereinfachung des Anwendungsfalls oft in eigene Anwendungsfälle ausgelagert, die mit dem Normallfall durch eine mit `extends` beschriftete, gestrichelte Linie verbunden sind (vgl. Abb. 4.7).

Enthalten mehrere Anwendungsfälle einen gleichen Anteil, so kann man diesen in einen eigenen Anwendungsfall herausziehen, der dann im Diagramm über eine mit `inncludes` beschriftete, gestrichelte Linie an die Ausgangsanwendungsfälle angebunden wird (vgl. Abb. 4.7).

Anwendungsfalldiagramme beschreiben ein Softwaresystem offensichtlich auf einem sehr groben Niveau. Mindestens so wichtig wie das Diagramm selber ist daher die textuelle Beschreibung jedes Anwendungsfalls, die spezifiziert, was bei dem Anwendungsfall passieren soll und welche Bedingungen vor und nach seiner Ausführung gelten müssen.

Der Ablauf eines Anwendungsfalls wird oft durch ein Verhaltensdiagramm wie z.B. ein Sequenzdiagramm (s.u.) genauer beschrieben.

4.2.4 Sequenzdiagramme

Ein Sequenzdiagramm dient dazu, zu spezifizieren, in welcher Weise Objekte interagieren, um eine vorgegebene Aufgabe zu erfüllen. Jedes der beteiligten Objekte wird hierbei durch eine vertikale Linie repräsentiert, die an den Stellen, an denen das jeweilige Objekt aktiv ist, verdickt wird. Horizontale Pfeile zeigen an, wo eine Nachricht an ein Nachbarobjekt geschickt wird, d.h. wo eine Methode des Nachbarobjekts mit welchen Parametern aufgerufen wird. Eine gestrichelte Kante in Rückrichtung kann verwendet werden, um anzuzeigen, wann welches Ergebnis des Methodenaufrufs zurückgeschickt wird. Ein Pfeil von einem Objekt zu sich selber repräsentiert eine interne Operation des Objekts. Ein Objekt kann nur dann eine Nachricht an ein anderes Objekt schicken, wenn es dieses Objekt „kennt“, d.h. wenn die zugehörigen Klassen durch eine Assoziation verbunden sind, das erste Objekt das zweite erzeugt hat oder eine Referenz auf das zweite Objekt als (Teil-)Ergebnis einer vorherigen Rechnung des ersten Objekts ermittelt wurde. UML-Modellierungswerkzeuge können insbesondere überprüfen, ob die in einem Sequenzdiagramm unterstellte Assoziation zwischen zwei Klassen in einem Klassendiagramm wirklich vorhanden ist und die Modelle insofern konsistent sind.

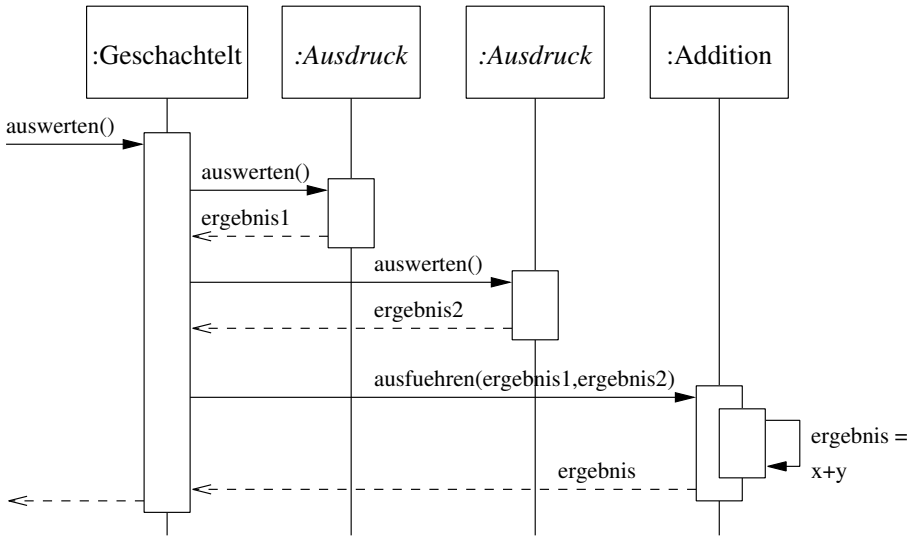


Abbildung 4.8: Sequenzdiagramm für die Auswertung eines arithmetischen Ausdrucks.

Abbildung 4.8 zeigt ein Sequenzdiagramm, das den Ablauf der Auswertung eines arithmetischen Ausdrucks, genauer einer Summe von zwei Teilausdrücken, spezifiziert. Hierbei wird angenommen, dass ein arithmetischer Ausdruck durch das in Abbildung 4.6 dargestellte System von Klassen repräsentiert wird. Zur Auswertung des geschachtelten Ausdrucks, der Summe, werden zunächst die beiden Teilausdrücke ausgewertet. Die Zwischenergebnisse werden als Parameter an die Methode

`ausfuehren` des Additionsobjekts übergeben, auf das das Objekt verweist, das den geschachtelten Ausdruck repräsentiert. Das hierbei berechnete Ergebnis ist der gesuchte Summenwert.

Ab UML 2.0 ist auch möglich, Kontrollstrukturen wie Verzweigungen und Schleifen in Sequenzdiagrammen darzustellen. Hierzu wird ein Sequenzdiagramm vertikal in Bereiche unterteilt, die alternativ, optional, wiederholt oder parallel ausgeführt werden. Details hierzu findet man in [Oes05, Sti05].

Sequenzdiagramme sind eine Variante von Interaktionsdiagrammen; weitere Varianten sind Kommunikationsdiagramme und Zeitverlaufsdiagramme, auf die hier nicht im Detail eingegangen wird. Kommunikationsdiagramme sind semantisch äquivalent zu Sequenzdiagrammen; sie unterscheiden sich lediglich in der Art der Visualisierung. Von üblichen UML-Modellierungswerkzeugen können Sequenzdiagramme auf Knopfdruck in äquivalente Komunikationsdiagramme transformiert werden und umgekehrt. Zeitverlaufsdiagramm sind etwas detaillierter, da sie auch noch den genauen zeitlichen Ablauf auszudrücken gestatten, wie dies bei der Modellierung von reaktiven Systemen häufig erforderlich ist.

4.2.5 Aktivitätsdiagramme

Wie Sequenzdiagramme erlauben auch Aktivitätsdiagramme das dynamische Verhalten eines Systems zu modellieren; allerdings auf einem etwas abstrakteren Niveau. Hier wird ein Ablauf nicht auf die Beiträge beteiligter Objekte heruntergebrochen, sondern losgelöst von Objekten und Nachrichten als Folge von Aktivitäten dargestellt. Im Diagramm wird eine Aktivität durch ein abgerundetes Rechteck repräsentiert. Pfeile zwischen den Aktivitäten geben an, in welcher Reihenfolge die Aktiviäten durchlaufen werden. Schwarze Balken zeigen an, wo der Ablauf in parallel bearbeitete Teilabläufe aufgespalten wird bzw. wo diese Teilabläufe wieder zusammengeführt werden. Rauten stellen dar, wo in Abhängigkeit von gewissen Bedingungen einer von zwei alternativen Abläufen ausgewählt wird bzw. wo die Alternativen wieder zusammengeführt werden.

Ein Pfeil zwischen zwei Aktivitäten kann optional mit einem Rechteck annotiert werden, das anzeigt, welche Daten in Pfeilrichtung fließen. Geht von einer Aktivität ein mit einem Blitz annotierter Pfeil aus, so zeigt dies an, dass im Falle eines Fehlers bei Ausführung der Aktivität die Ausnahme (Exception) ausgelöst wird, auf die der Pfeil zeigt. Geht der Ablauf hinter dem Rechteck mit der Ausnahme weiter, so wird der Fehler durch den entsprechenden Ablauf behandelt.

Aktivitäten mit einem Gabel-ähnlichen Symbol werden durch eigene Aktivitätsdiagramme verfeinert.

Aktivitätsdiagramme können horizontal in Bereiche unterteilt werden, die jeweils einem über dem Bereich angegebenen Bearbeiter zugeordnet werden. Dieser Bearbeiter ist für die Ausführung der Aktivitätsfolge in seinem Bereich verantwortlich.

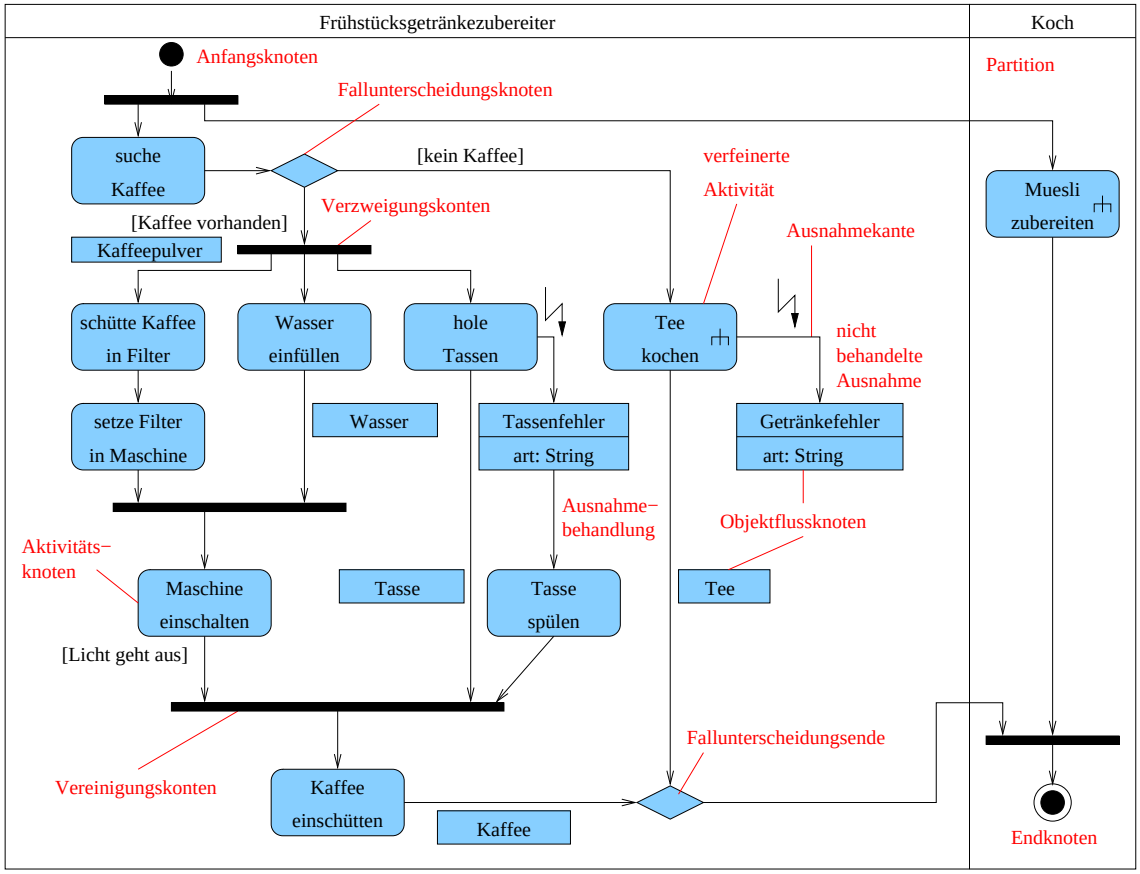


Abbildung 4.9: Aktivitätsdiagramm für die Frühstückszubereitung.

Abb. 4.9 zeigt ein Aktivitätsdiagramm für die Zubereitung eines Frühstücks. Das Diagramm besteht aus zwei Bereichen; der linke Bereich modelliert die Bereitstellung eines Getränks durch einen Frühstücksgetränkezubereiter, während im rechten Bereich die Zubereitung von Muesli durch einen Koch dargestellt wird. Der Ablauf wird gleich am Anfang in die beiden genannten Teilabläufe aufgespalten und am Ende wieder zusammengeführt. Im linken Bereich wird zunächst Kaffee gesucht. Ist kein Kaffe vorhanden, wird Tee gekocht, sofern das möglich ist. Geht auch das nicht, so wird eine Exception ausgelöst, die anzeigt, dass der Ablauf gescheitert ist. Ist Kaffe vorhanden, wird der Ablauf in drei parallele Teilabläufe unterteilt. Im ersten Ast werden Kaffee und Filter in die Kaffemaschine gefüllt. Parallel hierzu werden Wasser in die Maschine gefüllt und Tassen geholt. Sobald Kaffee, Filter und Wasser in der Kaffeemaschine sind, wird diese eingeschaltet. Ist keine Tasse mehr vorhanden, so wird die hierdurch ausgelöste Ausnahme dadurch behandelt, dass eine Tasse gespült wird. Wenn die Machine fertig ist und die Tasse bereit ist, wird der Kaffee eingeschüttet.

4.2.6 Weitere UML-Diagramme

Klassendiagramme, Anwendungsfalldiagramme, Sequenzdiagramme und Aktivitätsdiagramme sind die am häufigsten verwendeten UML-Diagramme. Weitere, hier nicht im Detail erläuterte UML-Diagramme sind die folgenden Strukturdiagramme:

- Montagediagramme: modellieren die Grobstruktur eines Systems,
- Komponentendiagramme: zeigen, aus welchen Komponenten ein System besteht und welche Scnittstellen diese bieten und nutzen,
- Verteilungsdiagramme: stellen die Verteilung von Komponenten auf Rechenknoten dar,
- Objekdiagramme: ähneln Klassendiagrammen; sie beschreiben aber die Verknüpfungen einzelner Objekte statt ganzer Klassen,
- Paketdiagramme: beschreiben die Abhängigkeiten zwischen Paketen (jeweils bestehend aus mehreren Klassen)

und die folgenden Verhaltensdiagramme:

- Zustandsautomaten: modellieren zustandsabhängiges Verhalten; genauer: eine Menge von möglichen Zuständen und die Übergangsmöglichkeiten zwischen diesen,
- Interaktionsübersichtdiagramme: nutzen die Ausdrucksmittel von Aktivitätsdiagrammen, um Interaktionen, die durch Interaktionsdiagramme beschrieben wurden, zu kombinieren

Details hierzu findet man zum Beispiel in [Oes05, Stï05].

Kapitel 5

Softwarearchitekturen

Eine Softwarearchitektur beschreibt den grundlegenden Aufbau eines Softwaresystems, indem sie dessen wesentliche Elemente und deren Beziehungen zueinander darstellt. Im Folgenden werden zwei weit verbreitete Ansätze zur Strukturierung von Softwaresystemen vorgestellt, die sich auch kombinieren lassen, nämlich Schichtenbildung und die Zerlegung in Komponenten.

5.1 Schichtenarchitekturen

Softwarearchitekturen, deren Elemente (Module, Klassen) in übereinander liegenden Schichten angeordnet sind, werden als Schichtenarchitekturen bezeichnet. Sie beschränken die Zugriffe jedes Elements auf Elemente der gleichen oder tieferer Schichten. Schichtenarchitekturen eigenen sich vor allem für die Strukturierung von komplexen Systemen. Sie reduzieren die Abhängigkeiten zwischen den Elementen verschiedener Schichten und erhöhen so u.a. die Übersichtlichkeit, Testbarkeit, Wartbarkeit und Wiederverwendbarkeit.

Im folgenden Abschnitt wird zunächst das Client-Server-Modell dargestellt, welches eine wichtige Grundlage für die Konzeption von Schichtenarchitekturen darstellt. Darauf aufbauend werden die grundlegenden Eigenschaften von Schichtenarchitekturen beschrieben. Abschließend werden zwei häufig vorkommende Typen von Schichtenarchitekturen vorgestellt, die 3-Schichtenarchitektur und die 4-Schichtenarchitektur.

5.1.1 Das Client-Server-Modell

Das Client-Server-Modell beschreibt einen Ansatz, der die Elemente eines Softwaresystem in *Clients* und *Server* unterteilt. Wie in Abbildung 5.1 dargestellt, nimmt ein Client zur Erfüllung seiner Aufgaben Dienste eines Servers in Anspruch, indem er eine Anfrage schickt. Der Server nimmt die Anfrage entgegen und schickt

das Ergebnis der Anfrage an den Client zurück. Von besonderer Bedeutung ist das Client-Server-Modell bei der Umsetzung von verteilten Systemen. Bei einer Client-Server-Architektur können Clients und Server auf unterschiedlichen Rechnern laufen, die über ein geeignetes Netzwerk mit einander verbunden sind. Eine Client-Server-Architektur erhöht also in der Regel die Flexibilität und Skalierbarkeit eines Systems.

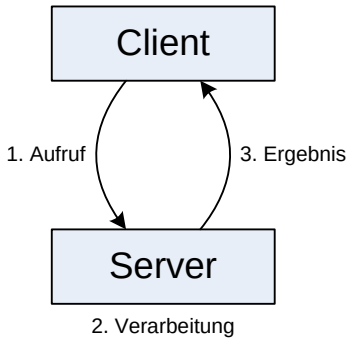


Abbildung 5.1: Das Client/Server-Modell

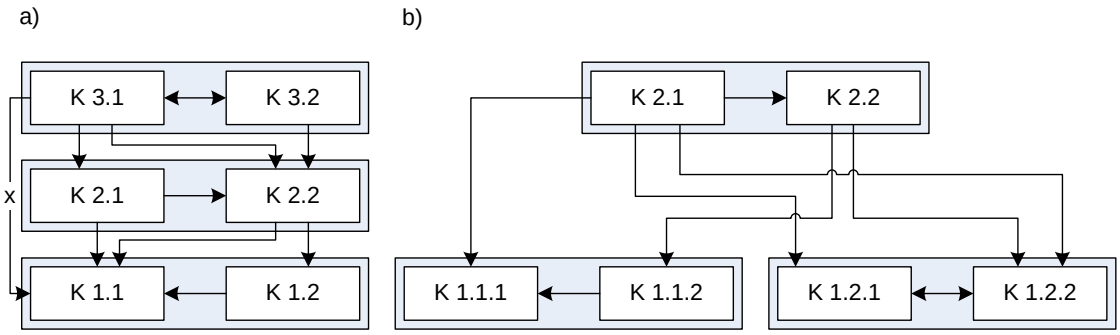
Abbildung 5.1 zeigt die einfachste Form des Client-Server-Modells bestehend aus einem Client und einem Server. Komplexere Client-Server-Architekturen ergeben sich, wenn ein Prozess sowohl die Rolle eines Clients als auch die Rolle eines Servers übernimmt. Ein solcher Prozess stellt als Server Dienste zur Verfügung, während er zur Erbringung dieser Dienste auf die Dienste anderer Server zurückgreift.

5.1.2 Aufbau von Schichtenarchitekturen

Komplexe Softwaresysteme, die aus vielen Komponenten bestehen, werden häufig als Schichtenarchitekturen konzipiert, um eine stärkere Strukturierung zu erreichen. Die Schichten eines Softwaresystems erlauben beliebigen Zugriff zwischen den Komponenten derselben Schicht, schränken jedoch die Zugriffe zwischen den Schichten ein.

Die Anordnung der Schichten in einer Architektur ist hierarchisch in Abhängigkeit von den Abstraktionsniveaus der Schichten. Dabei nehmen die höher liegenden Schichten von untergeordneten Schichten Dienste in Anspruch. Diese Rollenverteilung entspricht dem Client-Server-Modell. Die Schichtenhierarchie kann eine strikte, lineare oder baumartige Struktur annehmen, wie in Abbildung 5.2 dargestellt.

In einem Schichtenmodell mit strikter Ordnung befinden sich alle Schichten auf einem unterschiedlichen Abstraktionsniveau und es gilt die Einschränkung, dass eine Schicht auf alle Schichten mit einem niedrigeren Abstraktionsniveau, jedoch nicht auf Schichten mit einem höheren Abstraktionsniveau, zugreifen darf.



(Vgl. Balzert [Bal01, S. 697ff.])

Abbildung 5.2: Strikte, lineare und baumartige Schichtenstruktur. Abbildung a) zeigt eine strikte Schichtenstruktur (mit Zugriffskante x) und eine lineare Schichtenstruktur (ohne Zugriffskante x). Abbildung b) ist ein Beispiel für eine baumartige Schichtenstruktur. Innerhalb der Schichten befinden sich die mit *K...* bezeichneten Komponenten.

Ein Schichtenmodell mit linearer Ordnung schränkt die Zugriffsmöglichkeiten gegenüber einer strikten Ordnung zusätzlich dadurch ein, dass nur Zugriffe auf die eigene oder die nächstniedrige Schicht erlaubt werden. Ein bekanntes lineares Schichtenmodell ist das ISO/OSI-Modell für die Kommunikation in Rechnernetzen. Hierbei werden die unterschiedlichen Aufgaben der elektronischen Kommunikation je nach ihrem Abstraktionsgrad auf insgesamt sieben Schichten verteilt [CDKM02, S. 103 ff.].

Bei einer baumartigen Schichtenarchitektur ist es im Gegensatz zur strikten oder linearen Architektur möglich, mehrere Schichten auf der gleichen Abstraktionsebene anzuordnen. Da diese Architektur Zugriffe zwischen Schichten derselben Abstraktionsebene verbietet, ergibt sich durch die Kommunikationsbeziehungen zwischen den Schichten eine baumartige Struktur, wie sie in Abbildung 5.2b dargestellt ist.

In einer sinnvoll gestalteten Schichtenarchitektur besitzen alle Dienstleistungen einer Schicht dasselbe Abstraktionsniveau. Darüber hinaus sind die Schichten so zueinander angeordnet, dass eine Schicht ausschließlich die Dienstleistungen der untergeordneten Schichten in Anspruch nimmt, während sie den übergeordneten Schichten Dienstleistungen zur Verfügung stellt.

Die Anzahl der Schichten zählt zu den zentralen Eigenschaften einer Architektur und ist daher entscheidend für einen guten Softwareentwurf. Während zu wenige Schichten die Wiederverwendbarkeit, Anpassbarkeit und Portabilität einer Anwendung einschränken, führen zu viele Schichten dazu, dass die Komplexität des Gesamtsystems durch den erhöhten Aufwand, der durch die Schichtenaufteilung entsteht, steigt.

Die Vor- und Nachteile, die sich durch die Umsetzung einer Schichtenarchitektur ergeben können, sind in der Tabelle 5.1 aufgeführt. Im nachfolgenden Abschnitt

werden mögliche Schichtenarchitekturen für Desktop- und Web-Anwendungen vorgestellt und verglichen.

Vorteile	Nachteile
Strukturierung in Abstraktionsebenen fördert Übersichtlichkeit.	Geringer Effizienzverlust, da Daten oft über mehrere Schichten transportiert werden müssen.
Innerhalb einer Schicht freie Gestaltungsmöglichkeiten, da keine Zugriffsbeschränkung.	Definition von eindeutig abgegrenzten Schichten nicht immer möglich.
Unterstützt Wiederverwendbarkeit, Änderbarkeit, Wartbarkeit, Portabilität und Testbarkeit.	

(Vgl. Balzert [Bal01, S. 698])

Tabelle 5.1: Vor- und Nachteile von Schichtenarchitekturen.

5.1.3 Vergleich von Schichten-Architekturen

Klassische Schichten-Architekturen

Ein Anwendungssystem hat in der Regel drei wichtige Funktionen zu übernehmen: *Präsentation*, *Verarbeitung* und *Datenhaltung*. Die Präsentationsfunktion ist die Schnittstelle zum Anwender in Form einer Benutzeroberfläche, die Verarbeitungsfunktion umfasst die gesamte Geschäftslogik der Anwendung und für das Lesen und Speichern der Anwendungsdaten ist die Datenhaltung zuständig.

Abbildung 5.3 zeigt, wie diese Funktionen auf eine, zwei oder drei Schichten verteilt werden können. Bei einem monolithischen System werden alle Funktionen durch eine Schicht übernommen, während in einer 3-Schichten-Architektur jede Funktion durch eine eigene Schicht übernommen wird. Die 2-Schichten-Architektur stellt eine Zwischenform dar, bei der Präsentation und Verarbeitung durch eine einzelne Schicht realisiert werden.

Die Anzahl der Schichten, in die ein Anwendungssystem unterteilt werden sollte, ist abhängig vom Funktionsumfang. Je umfangreicher ein System ist, desto mehr Schichten sollte die Architektur besitzen. Wie bereits erwähnt, bedingt die Einführung einer weitern Schicht zunächst einen höheren Implementierungsaufwand und meist auch einen erhöhten Kommunikationsaufwand zwischen den Schichten. Im Gegenzug wird jedoch die Strukturierung des Gesamtsystems erhöht, was vor allem bei komplexeren Systemen sinnvoll ist. Zudem wird die Skalierbarkeit der Anwendung verbessert, da sich die Schichten leicht auf unterschiedliche Rechner verteilen lassen. Besonders bei komplexeren Anwendungen empfiehlt es sich im Hinblick auf

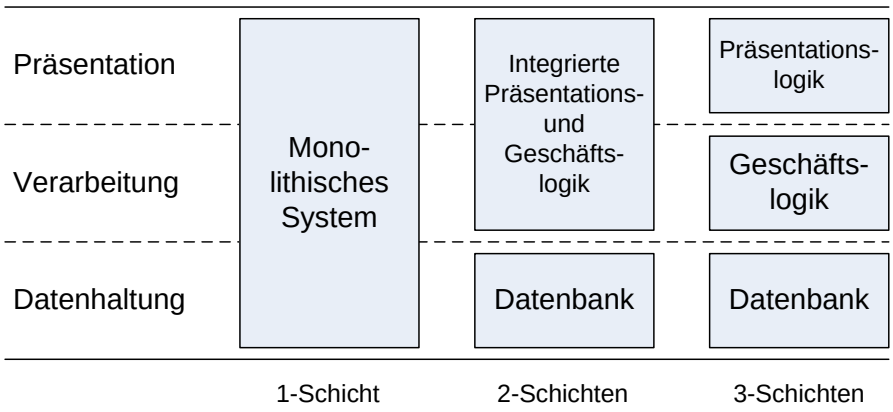


Abbildung 5.3: Schichtenarchitektur für Desktop-Anwendungen.

die Flexibilität des Gesamtsystems eine Architektur mit drei oder mehr Schichten zu verwenden. Ist das System später um weitere Benutzerschnittstellen, wie z.B. eine Web-Schnittstelle oder eine Web-Service-Schnittstelle zu erweitern, so können in der Regel Teile der Verarbeitungs- und Datenhaltungsschicht wiederverwendet werden. Ein solches Szenario wird im Abschnitt *Multi-Channel-Architekturen* beschrieben.

Web-Architekturen

Im Unterschied zu den klassischen Schichten-Architekturen weisen die Web-Architekturen eine verteilte Präsentationsfunktion auf. Diese wird sowohl von einem Web-Browser als auch von einen Web-Server übernommen. Der Web-Server ist dafür zuständig, die anzuzeigende Webseite zu erzeugen und an den Web-Browser zu schicken. Der Web-Browser übernimmt lediglich die Darstellung der empfangenen Seite, ggf. ergänzt um (z.B. in JavaScript realisierte) Präsentationslogik (z.B. zur elementaren Prüfung der Eingabe), und sendet Benutzereingaben an den Web-Server zurück. Mit Ausnahme der Aufteilung der Präsentationsfunktion zwischen Web-Browser und Web-Server sind die Web-Architekturen analog zu den klassischen Schichtenarchitekturen. Wie in Abbildung 5.4 dargestellt, besitzen Web-Architekturen daher üblicherweise zwei, drei oder vier Schichten. Für die Wahl der geeigneten Schichtenanzahl gelten die gleichen Argumente, die auch bei der Betrachtung der klassischen Schichten-Architekturen genannt wurden. Vor allem für komplexere Systeme ist daher eine Architektur mit vier oder mehr Schichten zu bevorzugen. Der Abschnitt *Multi-Channel-Architekturen* zeigt, welche Vorteile die Verteilung der einzelnen Systemfunktionen auf unterschiedliche Schichten mit sich bringt.

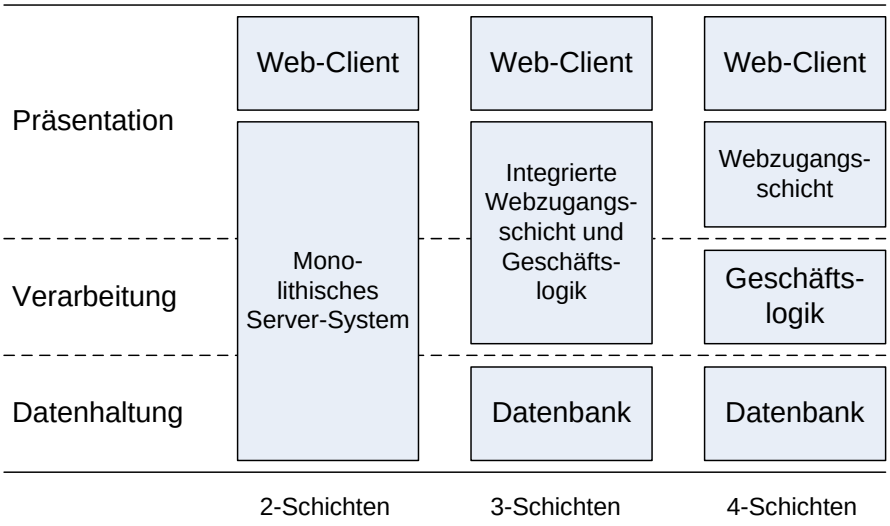


Abbildung 5.4: Schichtenarchitekturen für Web-Anwendungen.

Vergleich

Wie bereits erwähnt, liegt der Unterschied zwischen klassischen Architekturen und den Web-Architekturen in der Umsetzung der Präsentationsfunktion. Während die klassischen Architekturen eine Desktop-Anwendung benutzen, wird bei den Web-Architekturen diese Funktion auf Web-Server und Web-Browser aufgeteilt.

Die Kommunikation zwischen Web-Browser und Web-Server basiert auf dem HTTP-Protokoll, was zur Folge hat, dass die Verbindung zwischen Client und Server nach jedem Request-Response-Zyklus wieder getrennt wird. Im Unterschied zu einem Desktop-Client besteht also keine dauerhafte Verbindung zum Server. Um dennoch eine Zusammenfassung von Zugriffen zu Sitzungen zu ermöglichen, wird auf Techniken wie Cookies zurückgegriffen. Diese werden bei jeder Kommunikation zwischen Web-Client und Web-Serverim Header der betreffenden HTTP-Nachricht mitgeschickt und ermöglichen so die Zuordnung eines Zugriffs zu einer Sitzung.

Aufgrund dieser zentralen Unterschiede ergeben sich unterschiedliche Vor- und Nachteile der beiden Architekturen, welche in der nachfolgenden Tabelle gegenübergestellt werden.

Multi-Channel-Architektur Der Vorteil einer Schichtenarchitektur zeigt sich vor allem dann, wenn der Zugang zum System über verschiedene Kanäle erfolgen soll. Durch die Verteilung der Präsentations-, Verarbeitungs- und Datenhaltungsfunktion auf unterschiedliche Schichten, muss für eine neue Zugangsmöglichkeit nicht das komplette System neu implementiert werden, da es ausreichend ist, die entsprechenden Sichten um weitere Module zu ergänzen. Dabei kann auf die Funktionalitäten der bereits vorhandenen Schichten zurückgegriffen werden.

	Schichtenarchitekturen	
Merkmal	Klassisch	Web
Plattform-unabhängigkeit	Eingeschränkt, da Desktop-Clients oft plattformabhängig sind.	Gegeben. Es müssen u. U. jedoch unterschiedliche Web-Browser unterstützt werden.
Verteilung	Aufwändig, da Client-Software auf dem Client installiert werden muss.	Leicht, keine anwendungsspezifische Software auf dem Web-Client zu installieren.
Skalierbarkeit	Gut.	Gut.
Wartbarkeit	Eingeschränkt, da Client-Software evtl. neu installiert werden muss.	Gut.
Sitzungs-verwaltung	Leicht, da eine permanente Verbindung besteht.	u.a. durch Cookies.

(angelehnt an Balzert [Bal01, S. 703ff.])

Tabelle 5.2: Vergleich von Schichtenarchitekturen.

Abbildung 5.5 zeigt eine Multi-Channel-Architektur, die über die verschiedenen Module der Präsentationsschicht unterschiedliche Zugriffsmöglichkeiten auf das System bietet. Die Dienste des Systems können über einen Desktop-, einen Web-, oder einen Web-Service-Client in Anspruch genommen werden. Aufgrund der Schichtenarchitektur greifen alle Module der Präsentationsschicht auf die gleiche Geschäftslogik zu. Geschäftslogik und Datenhaltung müssen daher nur einmalig implementiert werden, wodurch der Aufwand für die Erweiterung des Systems um neue Zugangsmöglichkeiten minimiert wird.

5.2 Komponentenbasierte Architekturen

In der komponentenbasierten Softwareentwicklung wird ein Softwaresystem als eine Zusammenstellung einzelner Softwarekomponenten beschrieben. Eine *Komponente* ist ein Software-Baustein mit einer spezifizierten Schnittstelle, der konform zu einem *Komponentenmodell* ist, das u.a. die Verknüpfungsmöglichkeiten mit anderen Komponenten bestimmt. Sie benötigt zur Ausführung oft eine *Komponentenplattform*, von der ihr Dienste wie z.B. Zugriffskontrolle, Transaktionsverarbeitung und Synchronisation mit einer Datenbank bereitgestellt werden. Nicht zuletzt durch diese Dienste lässt sich eine Erhöhung der *Produktivität* und *Qualität* erreichen. Weiterhin wird angestrebt, sorgfältig getestete und zuverlässige Komponenten auch in anderen Projekten *wiederzuverwenden*, was wiederum eine Steigerung der Produktivität ermöglicht. Außerdem lässt sich durch eine Zerlegung eines Softwaresystems

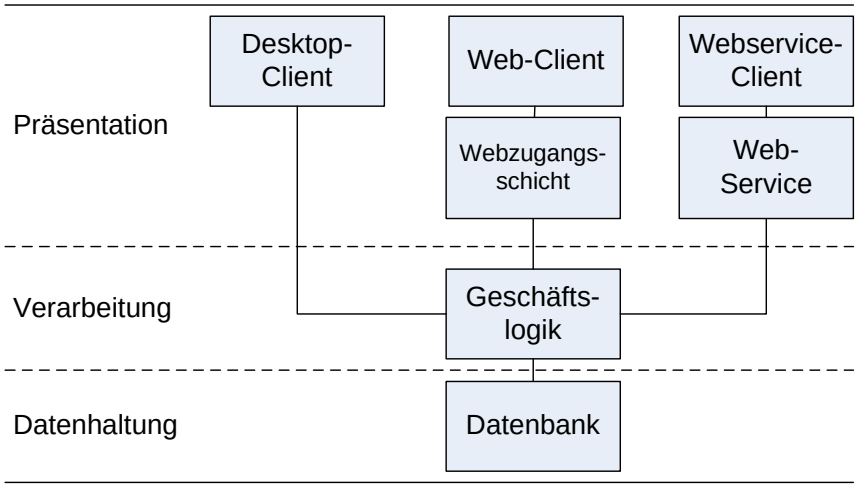


Abbildung 5.5: Multi-Channel-Architektur.

in mehrere, unabhängige Teilkomponenten das *Abstraktionsniveau* für die Softwareentwickler erhöhen.

5.2.1 Komponenten

Grundlage eines komponentenbasierten Softwaresystems ist ein *Komponentenmodell*, welches die Struktur der einzelnen Elemente der Architektur spezifiziert. Ein Komponentenmodell umfasst im Wesentlichen eine genaue Beschreibung des Aufbaus einer Softwarekomponente sowie der Laufzeitumgebung für diese Komponente, der *Komponentenplattform*. Die Spezifikation der Komponentenplattform umfasst sämtliche Dienste und Funktionalitäten, die einer Komponente zur Verfügung gestellt werden und legt somit auch die Interaktions- und Kompositionsmöglichkeiten von Komponenten fest.

Eine *Softwarekomponente* ist ein Softwarebaustein, der über Schnittstellen Operationen, Eigenschaften und Ereignisse zur Verfügung stellt. Zudem ist sie eine unabhängige Auslieferungseinheit, die zur Ausführung eine bestimmte Laufzeitumgebung, die Komponentenplattform, benötigt. Darüber hinaus besitzt sie Metadaten zur Selbstbeschreibung, über die sie ihre Schnittstellen und Kontextabhängigkeiten, d.h. ihre Abhängigkeit von anderen Komponenten und Bibliotheken, offen legt. Eine Komponente sollte zudem in hinreichendem Maße anpassungsfähig sein, um ihre Wiederverwendbarkeit sowie ihre Kompositionsfähigkeit zu gewährleisten.

Eine Komponentenplattform bietet den Komponenten eine Infrastruktur für häufig benötigte Mechanismen wie Verteilung, Nachrichtenaustausch, Persistenz, Sicherheit und Versionierung. Durch die Inanspruchnahme der Dienste der Komponentenplattform müssen diese Funktionen nicht für jede Komponente einzeln implementiert werden. Beispielsweise wird für die Kommunikation zwischen den Softwarekom-

ponenten in der Regel auf die Dienste der Komponentenplattform zurückgegriffen. Die für die Kommunikation verwendeten Netzwerke und Protokolle sind dabei für die Softwarekomponenten transparent, was dazu führt, dass die Entwicklung der Komponenten erheblich vereinfacht wird.

Durch den vorgeschriebenen Aufbau und die vereinheitlichte Interaktion wird eine Entkopplung der einzelnen Komponenten erreicht. Ziel dieser Entkopplung ist es, die bereits erwähnte Wiederverwendbarkeit und Austauschbarkeit der Komponenten zu verbessern. Im Folgenden werden drei weit verbreitete Komponentenmodelle, CORBA, Java EE und .NET näher erläutert.

5.2.2 CORBA

Die *Common Object Request Broker Architecture* (CORBA) [Obj07] ist ein von der Object Management Group entwickelter Standard, der plattform- und programmiersprachenübergreifende Protokolle und Dienste für die Umsetzung objektorientierter und verteilter Anwendungen definiert. Ziel dieser Architektur ist es, die Interaktion von objektorientierten Systemen verschiedener Programmiersprachen und Plattformen zu ermöglichen.

Um die Interoperabilität verschiedener CORBA-Implementierungen zu gewährleisten, basiert die gesamte Architektur ausschließlich auf offenen Standards und Schnittstellen. Der Aufbau eines CORBA-Systems wird in Abbildung 5.6 schematisch dargestellt und setzt sich aus vier zentralen Bestandteilen zusammen: Den Anwendungsobjekten, dem Object Request Broker (ORB), den speziellen Objektdiensten und den allgemeinen Diensten. Die *Anwendungsobjekte*, die in Form von Client- und Server-Objekten die Anwendungsfunktionalität implementieren, greifen über Schnittstellen auf die Dienste des ORB zu. Der *ORB* ermöglicht die transparente Kommunikation zwischen Client- und Server-Objekten über ein Netzwerk. Dafür nutzt der ORB die *speziellen Objektdienste* zur Verwaltung der Objekte im Netzwerk. Zusätzlich stellt der ORB den Anwendungsobjekten weitere *allgemeine Dienste* zur Verfügung. Zu diesen gehören Hilfsdienste, die zum Beispiel Drucken, Datenbankzugriffe und das Verschicken von Emails unterstützen.

Die Kommunikation zwischen Anwendungsobjekten realisiert der ORB in Form von *entfernten Methodenaufrufen*. Wie in Abbildung 5.7 dargestellt, wird dem Client-Objekt dafür über eine entsprechende Schnittstelle ein Objektstumpf zur Verfügung gestellt, der als Stellvertreter das entfernte, aufzurufende Objekt repräsentiert. Der *Stumpf* besitzt dieselbe Schnittstelle wie das durch ihn repräsentierte Objekt. Wird eine Methode des Stumpfes aufgerufen, so leitet diese den Aufruf an den ORB weiter. Der ORB ruft über ein *Skelett* die entsprechende Methode des Server-Objektes auf. Das Skelett übernimmt serverseitig die gleiche Funktion, wie der Stumpf auf der Client-Seite: Es dient dazu dem Server-Objekt einen lokalen Aufruf vorzutäuschen und macht so den entfernten Aufruf transparent. Das Ergebnis des Aufrufs wird über

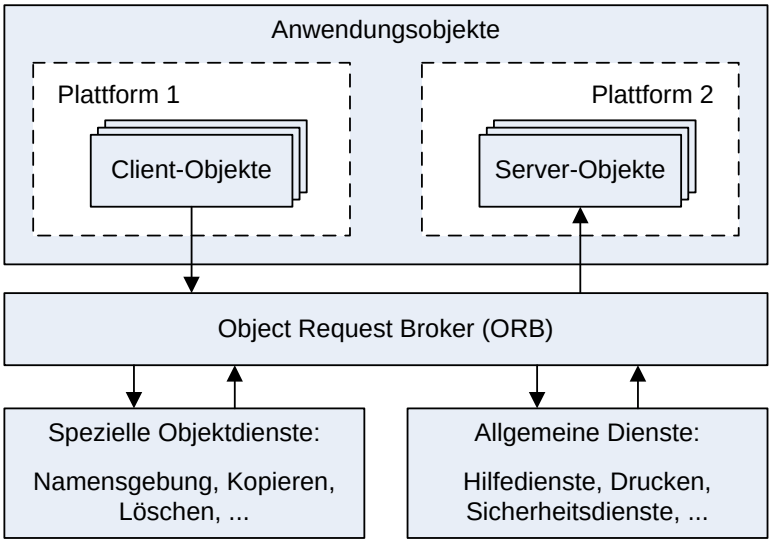


Abbildung 5.6: Architektur eines CORBA-Systems.

den umgekehrten Weg wieder an das Client-Objekt übermittelt. Für den Transport müssen der Methodenaufruf mit seinen Argumenten und das Ergebnis des Aufrufs in einen zu sendenden Datenstrom verwandelt und aus dem empfangenen Datenstrom extrahiert werden. Diese Aufgaben werden als *Marshalling* bzw. *Demarshalling* bezeichnet und werden von den Stumpf- und Skelettobjekten übernommen. Zusammen mit dem ORB, der für die Netzwerkkommunikation verantwortlich ist, sind so bei einem Methodenaufruf der Aufenthaltsort der Anwendungsobjekte sowie die eventuell durchgeführte Netzwerkkommunikation transparent.

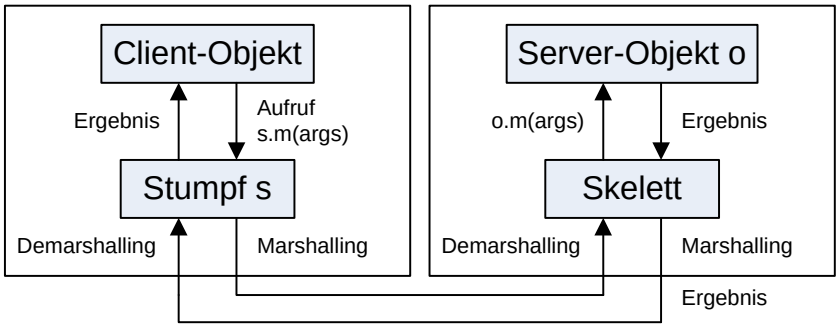


Abbildung 5.7: Entfernter Methodenaufruf mit CORBA.

Um ein Objekt in CORBA nutzbar zu machen, müssen dessen Schnittstellen zur Außenwelt zunächst in der *Interface Definition Language* (IDL) programmiersprachenneutral beschrieben werden. Mithilfe der Metainformationen einer IDL-Objektbeschreibung können sowohl Stumpf als auch Skelett eines Anwendungsobjekts erzeugt werden. Darüber hinaus benötigt der ORB die Metainformationen,

um den entfernten Methodenaufruf durchführen zu können. Eine ausführlichere Beschreibung von CORBA findet sich u.a. in [LP98, SGM02].

Obwohl CORBA, das in der Version 1.0 bereits 1991 veröffentlicht wurde, das älteste hier vorgestellte Komponenten-Framework ist, bleibt seine Verbreitung heute deutlich hinter der von Java EE oder .NET zurück. Gründe hierfür sind u.a. die hohe Komplexität sowie inkompatible, oft unvollständige, Hersteller-spezifische Implementierungen.

5.2.3 Java EE

Grundlagen

Die *Java Platform, Enterprise Edition* (Java EE) [Sun07a] ist eine Menge von Spezifikationen und Benutzerschnittstellen, die auf der *Java Platform, Standard Edition* (Java SE) aufbauen. Während die Java SE für die Entwicklung allein stehender Einzelanwendungen geeignet ist, stellt die Java EE Dienste und Schnittstellen für die Entwicklung serverbasierter Applikationen bereit und ist besonders für die Umsetzung verteilter, mehrschichtiger und komponentenbasierter Anwendungsarchitekturen geeignet.

Der Java EE liegt das Komponentenmodell der *Enterprise JavaBeans* (EJB) zugrunde. Das EJB-Modell fokussiert Systeme bei, denen die folgenden Eigenschaften von besonderer Bedeutung sind: *Skalierbarkeit*, *Verfügbarkeit*, *Sicherheit*, *Transaktionen* und *Ortstranzparenz*.

Der schematische Aufbau einer generischen Java EE-Anwendung ist in Abbildung 5.8 dargestellt. Die Architektur einer klassischen Java EE-Anwendung besteht aus drei oder vier Schichten, je nachdem, ob sie einen Desktop-Client oder einen Web-Client als Frontend besitzt. Es können jedoch auch beide Varianten in einer Anwendung kombiniert werden.

Ein Java EE-Server umfasst einen Web-Server und einen so genannten *Application-Server*. Der Web-Server stellt die Laufzeitumgebung für Servlets und JavaServer Pages (JSPs), die dazu dienen, die Webzugangsschicht zu realisieren; sie nehmen HTTP-Anfragen entgegen und generieren dynamisch HTML-Seiten. Falls die Benutzerschnittstelle der Anwendung ausschließlich ein Desktop-Client ist, entfällt der Einsatz eines Web-Servers. Der Application-Server stellt die Laufzeitumgebung für die Geschäftslogikkomponenten, die *Enterprise JavaBeans* (EJBs), zur Verfügung. Die Geschäftsprozesse werden dabei durch spezielle EJBs, nämlich *Session Beans* und *Message Driven Beans* implementiert, wobei die Session Beans der Umsetzung von synchronen und die Message Driven Beans der Umsetzung von asynchronen Prozessen dienen. Als Datencontainer für die Geschäftsdaten dienen die so geannten *Entities* (früher: Entity Beans), dies sind leichtgewichtige POJOs (Plain Old Java Objects), die durch einen Entity Manager auf die Einträge einer relationalen Datenbank abgebildet werden, d.h. es wird dafür gesorgt, dass jedes Entity-Objekt

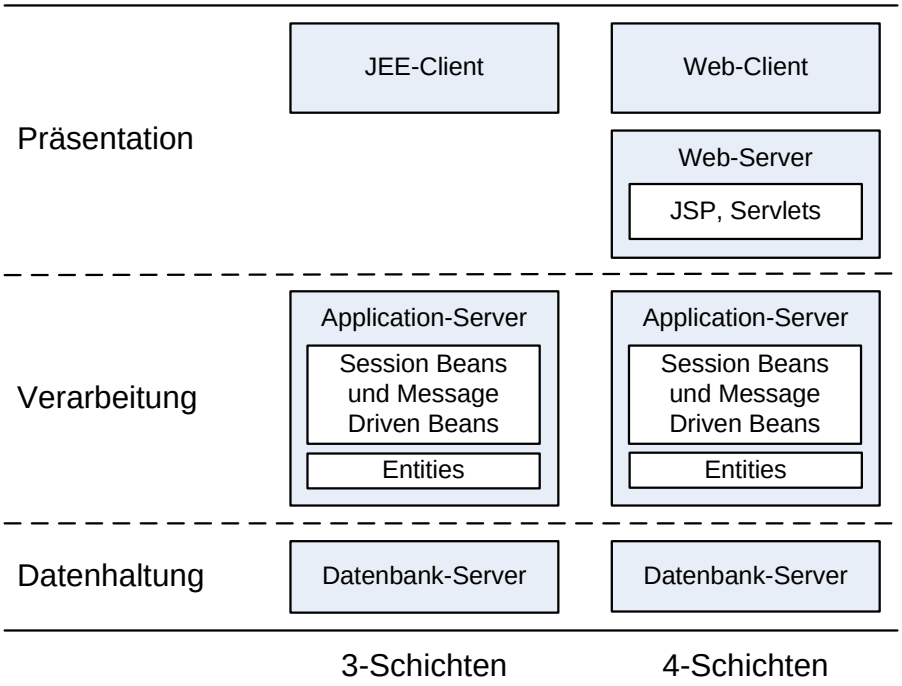


Abbildung 5.8: Architektur einer Java EE-Anwendung.

einem Datenbank-Tupel entspricht und dass beide synchron gehalten werden. Die Bezeichnung POJO soll verdeutlichen, dass es sich dabei um einfache Java-Objekte handelt, die keine externen Abhängigkeiten durch Namenskonventionen oder zu implementierenden Schnittstellen besitzen. Für die Datenhaltung ist in der Regel ein separater Datenbank-Server verantwortlich.

Die *EJB-Komponenten* benötigen zur Ausführung eine spezielle Laufzeitumgebung, den *EJB-Container*, der durch den Application-Server zur Verfügung gestellt wird. Abbildung 5.9 zeigt den Aufbau einer EJB sowie deren Interaktionsmöglichkeiten mit Diensten des EJB-Containers und den Clients, die die Dienste der Komponente in Anspruch nehmen.

Nach der EJB-Spezifikation 3.0 besteht eine EJB-Komponente aus der eigentlichen *EJB-Klasse*, einem *Business-Interface* und einer vom Container erzeugten Verpacker-Klasse (engl. Wrapper). Business-Interface und EJB-Klasse sind durch den Programmierer zu implementieren, während die Wrapper-Klasse automatisch beim so genannten Deployment, d.h. der Bereitstellung der Komponente auf einem Application Server, durch den Application Server erzeugt wird. Die Wrapper-Klasse kapselt Zugriffe auf die EJB-Klasse und ermöglicht es so, spezielle Dienste des Containers vor- bzw. nach einem Methodenaufruf der EJB auszuführen. Durch den Container bereitgestellte Dienste, wie z.B. das Transaktionsmanagement, können so automatisiert und für den Programmierer transparent durchgeführt werden. Das

Business Interface ist die nach außen propagierte Schnittstelle einer EJB, eine Komponente kann über ein Local oder Remote Business Interface verfügen. Diese können entsprechend über einen lokalen bzw. entfernten Client angesprochen werden.

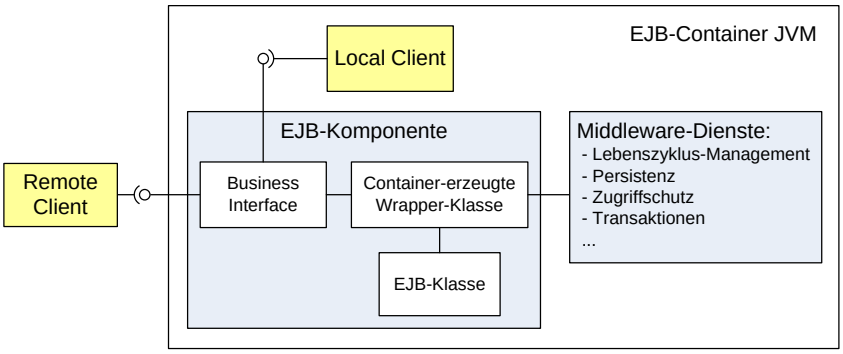


Abbildung 5.9: Aufbau einer EJB-Komponente.

Eine ausführliche Beschreibung des EJB-Komponentenmodells findet sich zum Beispiel bei [SBS06, EL07, SGM02].

Unterschiede zwischen EJB 2.1 und EJB 3.0

Das EJB-Komponentenmodell wurde eingeführt, um die Entwicklung von verteilten, mehrschichtigen und objektorientierten Anwendungen zu vereinfachen. Obwohl das EJB-Modell diesen Anforderungen in weiten Teilen gerecht wurde, haben sich im praktischen Einsatz auch einigen Schwächen gezeigt. Vor allem das Aufkommen einfacherer Alternativen zeigte, dass das EJB-Modell im Hinblick auf die Produktivität bei der Anwendungsentwicklung oft nicht die beste Lösung war. Zu den häufig beanstandeten Mängeln früherer EJB-Spezifikationen zählen:

- **Geringe Flexibilität:** Die Dienste des Frameworks standen nur den EJB-Komponenten zur Verfügung. Es war nicht möglich einzelne Dienste außerhalb des Frameworks, d. h. ohne vollständige Implementierung einer EJB-Lösung, zu nutzen.
- **Aufwändiges Testen:** Das Testen von EJB-Komponenten konnte mitunter recht teuer und zeitaufwändig sein, da sich einzelne Komponenten nicht außerhalb des Frameworks testen ließen. Auch für das Testen kleiner Änderungen musste deshalb jedes Mal ein vollständiges Deployment durchgeführt werden.
- **Komplexität:** Zu einer EJB-Komponente gehörten neben der EJB-Klasse mehrere Schnittstellen für entfernte, lokale und administrative Zugriffe auf die Komponente sowie ein Deployment Descriptor. Zudem mussten Methoden zum Management des Lebenszyklusses implementiert werden, auch wenn diese nicht benötigt wurden.

Mit den in der EJB-Spezifikation 3.0 eingeführten Neuerungen wird versucht, diese Kritikpunkte zu beheben, um die Produktivität bei der Anwendungsentwicklung im Vergleich zu früheren Versionen zu steigern. Zu den wichtigsten Neuerungen gehören:

- **POJOs:** Die EJB-Klassen müssen nun keine bestimmten Klassen oder Schnittstellen des Frameworks mehr erweitern bzw. implementieren. Das heißt: alle durch den Anwender implementierten Objekte sind POJOs (Plain Old Java Objects) und besitzen somit keine direkten Laufzeitabhängigkeiten vom EJB-Container. Dies führt zu einer Komplexitätsreduktion bei den EJB-Klassen und ermöglicht zudem auch das Testen von Klassen außerhalb des EJB-Containers. Spezielle Data Transfer Objekte zur Weitergabe von Ergebnissen an die Präsentations- oder Webzugangsschicht sind nicht mehr erforderlich.

- **Leichtgewichtiges O/R-Mapping:** EJB 3.0 umfasst ein neues Persistenz-Framework, die *Java Persistence API* (JPA), für das so genannte objektrelationale Mapping (O/R-Mapping). JPA bildet leichtgewichtige Entity-Objekte (POJOs) auf die Einträge einer relationalen Datenbanktabelle ab. Ein Entity-Objekt entspricht dabei einem Tupel in einer relationalen Datenbanktabelle. Das Framework sorgt für Konsistenz zwischen Objektmodell und Datenbank, indem es Änderungen an den Entity-Objekten in die Datenbank übernimmt. JPA ist auch ohne einen EJB-Container nutzbar und ersetzt die schwergewichtigen, Container-abhängigen EntityBeans.

- **Dependency Injection:** Besitzt eine Komponente Abhängigkeiten zu anderen Komponenten, so werden diese Abhängigkeiten nicht durch die Implementierung der Komponente festgelegt, sondern über einen Methodenaufruf zur Laufzeit in die Komponente injiziert. Dieser Ansatz reduziert die Abhängigkeiten zwischen den Komponenten und erhöht gleichzeitig die Wartbarkeit und Wiederverwendbarkeit der Komponenten.

- **Komplexitätsreduktion:** Neben den bereits erwähnten Vereinfachungen verzichtet EJB 3.0 auf einen Großteil der Komponentenschnittstellen und setzt auf das Paradigma der „Configuration by exception“, welches besagt, dass nur diejenigen Einstellungen konfiguriert werden müssen, die nicht den Standardeinstellungen entsprechen.

- **Annotationen:** Durch den Einsatz von Annotationen können die Komponenten mit Querschnittsfunktionen, wie z.B. dem Logging, annotiert werden. Solche Funktionen müssen dann nicht für jede Komponente einzeln implementiert werden. Mithilfe von Annotationen werden auch das POJO-Programmiermodell und die Komplexitätsreduktion, z.B. durch Einsparung des Deployment Descriptors, realisiert.

Web-Frameworks

Ein Web-Framework dient dazu, die Entwicklung von Web-Anwendungen zu unterstützen, indem es generische Lösungen für häufig auftretende Probleme bei der Web-Entwicklung bereit stellt. Üblich sind u.a. Lösungen für Datenbankzugriff, Sicherheit, Sitzungsverwaltung, Seitennavigation, Validierung von Benutzereingaben, Internationalisierung und Templatesysteme zum einfachen Designen von Webseiten. Im Folgenden werden einige bekannte Web-Frameworks für Java EE vorgestellt.

Struts

Das Struts Framework [Apa07a, Hol06] erweitert die Java Servlet API und setzt für die Gestaltung von Webseiten das Model-View-Controller Entwurfmuster (MVC) [GHJV96] um. MVC ist ein gebräuchliches Entwurfsmuster zur Programmierung von Benutzeroberflächen. Die Aufgaben der Benutzeroberfläche werden dabei folgendermaßen auf die drei Komponenten *Model*, *View* und *Controller* aufgeteilt: das Model speichert die Daten der Anwendung, die View ist für die visuelle Repräsentation der Daten verantwortlich, der Controller enthält die Anwendungslogik der Anwendung und steuert das Zusammenspiel zwischen View und Model.

Das Struts Framework stellt einen zentralen Controller, das *ActionServlet*, bereit. Alle Seiten-Request werden an den zentralen Controller in Form von „Actions“, die in einer zentralen Konfigurationsdatei definiert werden, geschickt. Der Controller ruft die mit einer Action verknüpfte Action-Klasse auf, die daraufhin das anwendungsspezifische Model manipuliert. Anhand eines zurückgegebenen „ActionForward“-Codes kann der Controller, die an den Client zu schickende Antwortseite identifizieren. Von zentraler Bedeutung für die Funktionsweise des Struts-Frameworks ist die oben bereits erwähnte Konfigurationsdatei, in der der gesamte Kontrollfluss der Web-Anwendung definiert wird.

Obwohl Struts ein sehr ausgereiftes und populäres Framework ist, werden heute immer häufiger auch neuere leichtgewichtige Frameworks wie JSF, Spring MVC oder Tapestry eingesetzt. Weitere Verbesserungen sollen mit dem Struts 2.0 Framework, das eine Integration von Struts mit dem WebWork-Framewok darstellt, eingeführt werden. WebWork ist aus dem Struts-Framework entstanden und wurde bis zur Zusammenführung einige Jahre parallel zu diesem entwickelt.

Vorteile	Nachteile
Das „Standard“-Framework	Umständliche ActionForms
Gute Dokumentation	Keine Unit-Tests möglich
Gute Tag-Bibliothek	

(Quelle: Raible [Rai07])

Tabelle 5.3: Vor- und Nachteile des Struts-Frameworks.

JavaServer Faces

Ähnlich wie viele andere Web-Frameworks basiert JavaServer Faces (JSF) [Sun07b, HS06] auf dem Model-View-Controller Paradigma. Der Unterschied zu andern Frameworks liegt in der Erzeugung der Webseiten, welche aus einzelnen UI-Komponenten zusammengesetzt werden. Dieser komponentenbasierte Ansatz weist ein höheres Abstraktionsniveau auf und macht die HTML-Programmierung zu weiten Teilen überflüssig.

Vorteile	Nachteile
Java EE-Standard	komplexe Tag-Bibliothek
Leicht und schnell erlernbar	Mehrere konkurrierende Implementierungen
Viele Komponentenbibliotheken	

(Quelle: Raible [Rai07])

Tabelle 5.4: Vor- und Nachteile des JSF-Frameworks.

UI-Komponenten besitzen einen Zustand, der während eines Request-Response-Zykluses erhalten bleibt, und erzeugen ihre eigene HTML-Repräsentation, wodurch die Programmierung einer Web-Oberfläche deutlich leichter und intuitiver gestaltet wird. Darüber hinaus verfügt JSF über ein serverseitiges Ereignismodell. Hierbei werden die Eingaben des Benutzers auf der Clientseite in Ereignisse auf der Serverseite umgewandelt. Dieses Modell ermöglicht eine Oberflächenprogrammierung, die sehr viel Ähnlichkeiten mit der Programmierung von Desktop-Oberflächen aufweist.

Spring MVC

Das Spring MVC-Framework [Int07, JHA+05] zur Entwicklung von Web-Anwendungen ist ein Modul des Spring-Frameworks. Spring selber ist eine auf Java basierende Plattform zur Serverprogrammierung, die ebenso wie die Java EE-Plattform dazu dient, verteilte, mehrschichtige und komponentenbasierte Architekturen zu implementieren. Ziel von Spring ist es, die Entwicklung von Anwendungen für Application-Server durch ein leichtgewichtiges, POJO-basiertes Programmiermodell zu vereinfachen.

Um dieses Ziel zu erreichen, unterstützt Spring, und demzufolge auch Spring MVC, die folgenden Programmierparadigmen:

- **Dependency Injection**: Spring bietet ein auf der JavaBean-Technologie basierendes Konfigurationsmanagement, mit dem die Abhängigkeiten zwischen den einzelnen Komponenten reduziert werden. Hierfür werden die Abhängigkeiten, die eine Komponente besitzt, nicht durch sie selbst bestimmt, sondern von außen über die Argumente eines Methodenaufrufes injiziert.

Vorteile	Nachteile
Gute Integration mit anderen Technologien und Frameworks	Aufwändige Konfiguration durch XML
Leichtes Testen durch Dependency Injection	Erfordert viel Code in JSPs
	Durch hohe Flexibilität u. U. mangelnde Struktur (kein Parent Controller)

(Quelle: Raible [Rai07])

Tabelle 5.5: Vor- und Nachteile des Spring MVC-Frameworks.

- **Integration**: Spring bietet gute Integrationsmöglichkeiten mit anderen Frameworks, so dass sich bereits existierende und bewährte Lösungen leicht integrieren lassen.

- **Aspektorientierte Programmierung**: Durch dieses Paradigma können einzelne Aspekte, das sind in der Regel anwendungsübergreifende Querschnittsfunktionen, wie z.B. Logging, von der reinen Anwendungslogik isoliert werden. Das Ziel ist es, hierdurch die Übersichtlichkeit, Wartbarkeit und Wiederverwendbarkeit des Programmcodes zu erhöhen.

Diese Eigenschaften gehören im Vergleich zu anderen Web-Frameworks zu den Besonderheiten von Spring MVC. Wie der Name bereits andeutet, setzt auch Spring MVC auf das MVC-Paradigma.

Tapestry

Tapestry [Apa07b, Ton06] basiert auf einer komponentenbasierten Architektur und weist daher ähnliche Merkmale und Besonderheiten wie JSF auf. Das Framework unterstützt neben den üblichen Merkmalen wie Seitennavigation, Eingabevalidierung, Internationalisierung und Sicherheit vor allem persistente (d. h. Request-Response-Zyklus überdauernde) Zustände der Komponenten.

Durch die Verwendung von Komponenten erreicht Tapestry ähnlich wie JSF einen relativ hohen Abstraktionsgrad bei der Oberflächenprogrammierung, verwendet jedoch für die Erzeugung des HTML-Codes eine eigene Template-Engine. Mithilfe der Template-Engine wird eine klare Trennung von HTML- und Anwendungscode erreicht, die es ermöglicht, beide parallel und unabhängig von einander zu entwickeln.

Vorteile	Nachteile
Hohe Produktivität	steile Lernkurve
Echte HTML-Templates	Schlechte Dokumentation
Viele Innovationen zwischen Versionszyklen	Lange Versionszyklen

(Quelle: Raible [Rai07])

Tabelle 5.6: Vor- und Nachteile des Tapestry-Frameworks.

5.2.4 .NET

Grundlagen Das .NET Framework [Mic07b] ist eine von Microsoft entwickelte Implementierung der *Common Language Infrastructure* (CLI). Die CLI ist ein ISO/IEC/ECMA Standard, der, ähnlich wie CORBA, eine programmiersprachen- und plattformneutrale Softwarearchitektur spezifiziert. Im Unterschied zu CORBA definiert CLI zusätzlich eine *Common Intermediate Language* (CIL) und ein Dateiformat für das Deployment von Komponenten, den sogenannten Assemblies. In diesen Punkten ähnelt die CLI der Java-Spazifikation, die Java-Bytecode als Zwischensprache und Jar-Dateien für das Deployment definiert.

Die CLI umfasst:

- die **Common Language Specification** (CLS), eine Menge von Regeln, die eine Programmiersprache erfüllen muss, um in die Zwischensprache, die Common Intermediate Language übersetzt werden zu können,
- das **Common Type System** (CTS), ein für alle Sprachen der CLI einheitliches und verbindliches Typsystem, und
- eine Laufzeitumgebung, das **Virtual Execution System** (VES), welches u.a. einen Class-Loader, einen JIT(Just In Time)-Compiler und einen Memory Manager für die Garbage Collection definiert.

Der Aufbau und die Funktionsweise der CLI sind in Abbildung 5.10 dargestellt. Eine CLS-konforme Quellsprache wird in die Zwischensprache CIL übersetzt. Mithilfe einer plattformspezifischen Implementierung der VES wird die CIL interpretiert bzw. in ausführbaren Maschinencode übersetzt.

Das .NET Framework, welches die CLI für Windows-Systeme implementiert, besteht, wie in Abbildung 5.11 dargestellt, im Wesentlichen aus einer Menge von Klassenbibliotheken und der Common Language Runtime.

Die Common Language Runtime (CLR), ist die von Microsoft umgesetzte Implementierung der VES, die nicht nur die Spezifikation der CLI erfüllt, sondern darüber hinaus Zugriffe auf das COM sowie das zugrunde liegende Betriebssystem und DLL-basierte APIs ermöglicht. Durch die direkte Integration von COM und

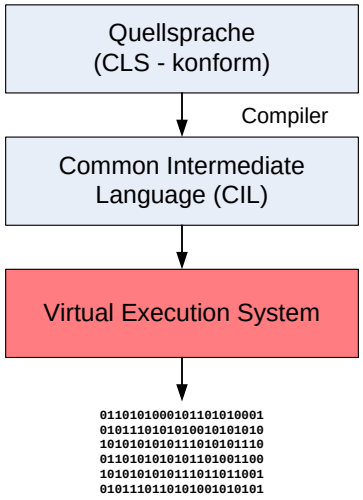


Abbildung 5.10: Aufbau und Funktionsweise der Common Language Infrastructure (CLI).

der Windowsplattform erreichen die durch einen JIT-Compiler übersetzten Zugriffe nahezu die gleiche Geschwindigkeit wie herkömmlicher, präcompilierter Code.

Die Integration mit COM wird durch die Verwendung des Adaptermusters erreicht: Runtime-callable Wrapper ermöglichen den Zugriff auf COM-Komponenten. Microsoft unterstützt eine Vielzahl von Programmiersprachen für die CLR, u.a. C#, J#, JScript, Managed C++ und Visual Basic.NET. Da alle Sprachen vor der Ausführung in dieselbe Zwischensprache übersetzt werden, ermöglicht das .NET Framework Programmiersprachenunabhängigkeit.

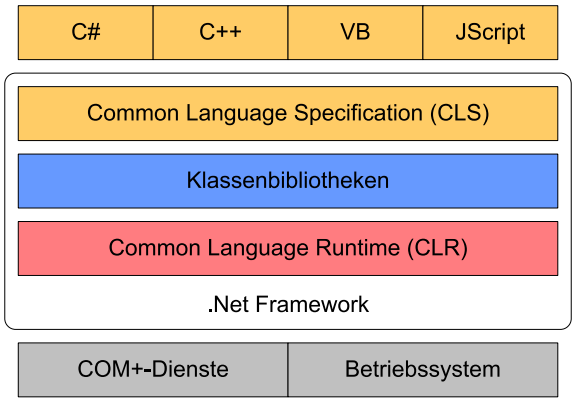
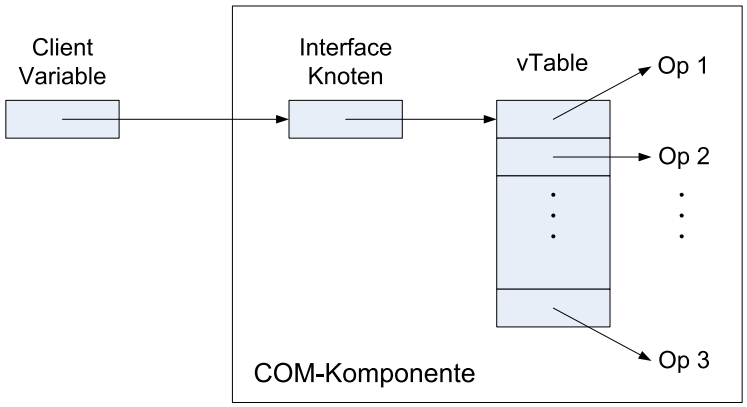


Abbildung 5.11: Architektur des .NET Frameworks.

COM

Das Component Object Model [SGM02, S. 329 ff.] stellt die Grundlage komponentenbasierter Software für die Microsoft Windows Plattform dar. Beim COM handelt es sich um einen binären, sprachunabhängigen Standard zur Definition von Schnittstellen. Das Verhalten einer Schnittstelle wird durch den COM-Standard nicht weiter spezifiziert, d. h. es werden keine Vorgaben gemacht, wie das Verhalten der Schnittstelle umzusetzen ist. Eine COM-Schnittstelle muss daher nicht zwangsläufig durch ein Objekt implementiert werden, sondern kann auch lediglich aus einer Menge von Funktionen bestehen, die die Vorgaben der Schnittstelle umsetzen.

Abbildung 5.12 zeigt den Aufbau einer COM-Schnittstelle. Sie wird auf binärer Ebene durch einen Interface-Knoten repräsentiert. Dieser Knoten enthält einen Zeiger auf eine Tabelle von Funktionszeigern (vTable). Aus der Sicht eines Clients besteht eine COM-Komponente somit aus einem Zeiger auf einen Zeiger, der auf eine vTable zeigt.



(Quelle Szyperski [SGM02, S. 331ff.])
Abbildung 5.12: Aufbau einer COM-Komponente

Mithilfe dieser doppelten Indirektion ist es möglich, für die Komponente ein objektorientiertes Verhalten zu simulieren. Hierfür wird den Methoden der Komponente bzw. der vTable beim Aufruf die Referenz auf den Schnittstellenknoten als zusätzlicher Parameter übergeben. Mithilfe dieser Referenz können verschiedene Instanzen von Interface-Knoten, die auf dieselbe vTable zeigen, unterschieden werden. Die Methoden der vTable können durch diese Referenz die Instanzen der Interface-Knoten unterscheiden und objektorientiertes Verhalten simulieren, indem sie intern den Interface-Knoten Instanzvariablen zuordnen.

Das wichtigste Interface des COM ist das *IUnknown* Interface, dessen Funktionen QueryInterface, AddRef und Release von allen COM Schnittstellen implementiert werden müssen. Die Funktionen AddRef und Release in- bzw. dekrementieren einen Referenzzähler für den Interface-Knoten. Mithilfe des Referenzzählers können nicht

mehr referenzierte COM-Objekte ermittelt und gelöscht werden. Da eine COM-Komponente auch mehrere Schnittstellen besitzen kann, wird die Funktion QueryInterface benötigt, die zu einer übergebenen Schnittstellen-ID, falls vorhanden, eine Referenz auf den zugehörigen Interface-Knoten zurück gibt. Die QueryInterface Funktion wird benötigt, um die von einer Komponente implementierten Schnittstellen zu ermitteln und um die Funktionen der verschiedenen Schnittstellen aufrufen zu können.

Wie bereits erwähnt, dienen die Schnittstellenknoten dazu, die Instanzen einer COM-Komponente zu unterscheiden. Weil eine COM-Komponente auch mehrere Interface-Knoten besitzen kann, werden COM-Objekte über Schnittstellenknoten der IUnknown Schnittstelle identifiziert. Alle Schnittstellen eines COM-Objekts geben beim Aufruf der QueryInterface Funktion mit der ID der IUnknown Schnittstelle als Parameter die Referenz auf den selben Interface-Knoten zurück. Dies gewährleistet die eindeutige Identifizierbarkeit von COM-Objekten.

ASP.NET

ASP.NET [Mic07a] ist das Web-Framework der .NET-Plattform, welches die Erstellung von Web-Anwendungen und Web-Services unterstützt. Im Gegensatz zur Java EE-Plattform, für die es viele Web-Frameworks mit unterschiedlichen Schwerpunkten und Zielsetzungen gibt, ist ASP.NET das einzige Web-Framework für die .NET-Plattform. ASP.NET ist der Nachfolger der Active Server Pages (ASP) Technologie und bietet generische Lösungen für die üblichen Probleme, die bei der Programmierung von Webanwendungen auftreten. Es gibt zum Beispiel Lösungen für: Sitzungsverwaltung, Authentifizierung und Autorisierung, Caching sowie häufig verwendete Steuerelemente für Web-Oberflächen, wie Schaltflächen, Textfelder und Tabellen. Darüber hinaus basiert das Framework auf der Common Language Runtime, daher können ASP.NET-Seiten in jeder .NET-Sprache erstellt werden.

Das Framework unterstützt für die Erzeugung von dynamischen Webseiten das ASPX-Dateiformat. Typischer Weise enthält eine ASPX-Datei statischen HTML-Code, in den über bestimmte Tags dynamische Inhalte eingebunden werden können. Der Programmcode für die dynamischen Inhalte wird nach dem so genannten Code-Behind-Modell in einer separaten Datei gespeichert. Durch diese Form der Trennung von Layout und Programmlogik ist es möglich, ein ereignisbasiertes Programmiermodell umzusetzen. Der dynamische Inhalt einer Webseite muss damit nicht mehr, wie z.B. bei der veralteten ASP Technologie, sequenziell mit der Webseite selbst erzeugt werden, sondern wird durch die Reaktion auf Benutzereingaben im Code-Behind manipuliert. Dieser Ansatz ist deutlich intuitiver und ermöglicht es, Web-Oberflächen ähnlich wie Desktop-Anwendungen zu programmieren. Das ASP.NET-Framework unterstützt zudem ein Komponentenmodell, welches es ermöglicht, Webseiten aus mehreren UI-Komponenten, den sogenannten Web-Controls oder Steuerelementen, zusammenzusetzen. Ein Web-Control behält seinen Zustand über mehre-

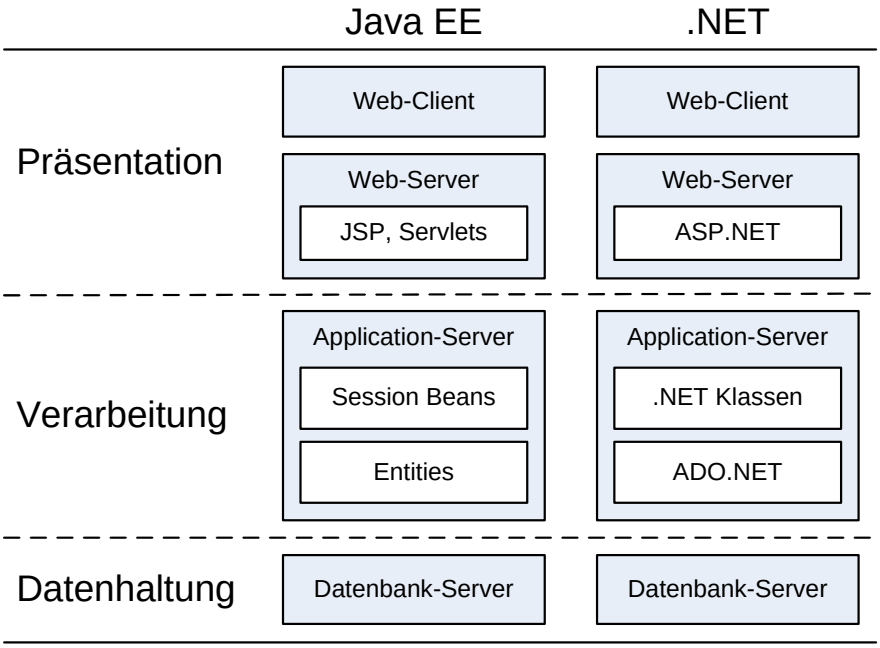


Abbildung 5.13: Architekturvergleich von Java EE- und .NET-Anwendungen.

re Request-Response-Zyklen zwischen Browser und Webserver hinweg, erzeugt seinen eigenen HTML- und JavaScript-Code zur Webclient-seitigen Visualisierung im Browser, kann mit anderen Controls zu einem komplexeren Control kombiniert werden und wandelt Benutzereingaben in entsprechende Ereignisse um. Diese Eigenschaften ermöglichen es, in einer zustandslosen Web-Umgebung ein ereignisbasiertes, zustandsbehaftetes Programmiermodell zu realisieren, ähnlich wie bei den Java EE-Frameworks JSF und Tapestry.

Das ASP.NET-Framework bietet eine hohe Performanz und Skalierbarkeit. Gründe hierfür sind die Präcompilierung der ASPX-Seiten, die vielfältigen Caching-Möglichkeiten und die automatisierte Sitzungsverwaltung, die eine Sitzungsverfolgung auch auf einer Webfarm ermöglicht. Im Vergleich zu anderen Web-Frameworks werden bei ASP.NET große Teile des HTML- und JavaScript-Codes automatisch generiert. Aussehen und Verhalten der Steuerelemente werden im Wesentlichen über deren Eigenschaften gesteuert, das Erzeugen des HTML-Codes geschieht dann automatisch. Weiterführende Informationen über die ASP.NET Technologie finden sich u.a. in [LH05, Loh03].

5.2.5 Vergleich Java EE und .NET

In diesem Abschnitt werden die Komponenten-Frameworks Java EE und .NET gegenübergestellt und wichtige Gemeinsamkeiten und Unterschiede zwischen beiden

Frameworks beschrieben. CORBA wird aufgrund der relativ geringen Verbreitung und abnehmenden praktischen Relevanz nicht in diesen Vergleich mit einbezogen. Abbildung 5.13 stellt die Architekturen einer Java EE- und einer .NET-Anwendung vergleichend gegenüber. Gegenstand des Vergleichs sind die Präsentationsschicht, die Geschäftslogikschicht und die Plattformen, die den beiden Frameworks zugrunde liegen. Die Unterschiede innerhalb der einzelnen Schichten werden im Folgenden genauer beschrieben.

Präsentationsschicht

- **HTML-Erzeugung:** Die dynamische Generierung von HTML-Seiten geschieht bei Java EE mithilfe von Servlets oder JSP-Seiten. Das .NET-Framework verwendet für diese Aufgabe ASPX-Seiten, die von ihrem Aufbau in etwa mit JSP-Seiten vergleichbar sind. Hinsichtlich der automatischen Integration von JavaScript-Code zur Client-seitigen Prüfung von Benutzereingaben zeigt sich, dass die ASPX-Technologie etwas komfortabler ist.

- **Web-Frameworks:** Für Java EE gibt es eine Vielzahl von Frameworks, die JSPs bzw. Servlets um weitere generische Lösungen wie Sitzungsverwaltung, Sicherheit, Validierung, Internationalisierung und Seitennavigation erweitern. Darüber hinaus besitzen die Frameworks unterschiedliche Ansätze zur Trennung von Layout und Geschäftslogik.

  Das Web-Framework der .NET-Plattform heißt ASP.NET und bietet ebenfalls Lösungsansätze für die oben genannten zentralen Aufgaben der Web-Programmierung. Zudem besitzt ASP.NET ein Code-Behind-Modell, das die klare Trennung von Layout und Geschäftslogik ermöglicht.

- **Komponentenbasierung:** Die Einführung von Komponenten ermöglicht auch für Webseiten ein intuitives, ereignisbasiertes Programmiermodell mit Steuerelementen, die einen persistenten Zustand besitzen. Komponenten werden bei Java EE nur durch einige wenige Web-Frameworks wie JavaServer Faces oder Tapestry unterstützt. ASP.NET ermöglicht ein komponentenbasiertes Programmiermodell durch den Einsatz von serverseitigen Web-Controls.

Geschäftslogik

- **Transaktionen:** Beide Frameworks unterstützen sowohl automatisches als auch manuelles Transaktionsmanagement.

- **Entfernte Methodenaufrufe:** Java EE bietet bei entfernten Methodenaufrufen vollständige Ortstransparenz, weil alle Komponenten über JNDI ermittelt werden. Durch die strikte Verwendung von Schnittstellen wird so auch eine automatische Lastverteilung ermöglicht. Bei .NET liegt Ortstransparenz

nur bei expliziter Verwendung von .NET Remoting vor. Ohne Remoting sind nur lokale Aufrufe möglich. Dieser Ansatz bietet nur bedingt Ortstransparenz, dafür sind die expliziten lokalen Aufrufe jedoch schneller als bei Methodenaufrufen mit vollständiger Ortstransparenz.

- **Web-Services:** Beide Frameworks bieten eine gute Integration von Web-Services. Bei Java EE geschieht dies über zusätzliche Spezifikationen, .NET verfügt über native APIs zur Web-Service-Programmierung.

- **Datenhaltung:** Java EE beinhaltet die Java Persistence API (JPA), die über leichtgewichtige Entities ein O/R-Mapping ermöglicht. JPA bietet zudem Diente für Sicherheit, Fehlertoleranz, Skalierbarkeit und den Zugriff auf verschiedenste Typen von Datenquellen. Auch ADO.NET ermöglicht es einer .NET-Anwendung, auf unterschiedliche Arten von Datenquellen zuzugreifen, allerdings unterstützt es bisher kein automatisches O/R-Mapping. Für zukünftige Versionen von ADO.NET ist auch ein automatisches Mapping geplant. Es gibt allerdings auch für .NET O/R-Mapper wie z.B. NHibernate [Mid07]. Hiermit lässt sich dann eine ähnlich komfortable Anbindung der Datenhaltung wie bei Java EE erreichen.

Plattform

- **Laufzeitumgebung:** Java EE basiert auf der Java Virtual Maschine (JVM), die den durch einen Compiler erzeugten Bytecode zur Laufzeit interpretiert bzw. (Just-in-Time) zur Ausführungszeit in Maschinencode übersetzt. Auch .NET verfügt analog über eine virtuelle Maschine (CLR), die präcompilierten Zwischencode (MSIL-Code) interpretiert bzw. Just-in-Time compiliert. Die Ausführung über einen Interpreter bzw. Just-in-Time-Compilierung ermöglichen es, gewisse Aufgaben, wie die Code-Verifikation, das Sicherheitsmanagement und die Speicherbereinigung, zu automatisieren, haben jedoch einen Geschwindigkeitsverlust zur Folge.

- **Plattformunabhängigkeit:** Da für nahezu jedes Betriebssystem eine JVM existiert, ist auch Java EE quasi vollständig Plattformunabhängig. Bei .NET wäre eine Plattformunabhängigkeit aufgrund der vorhandenen virtuellen Maschine theoretisch auch realisierbar, bisher gibt es jedoch, abgesehen von einigen wenige Ausnahmen (z.B. Mono für Linux), ausschließlich eine Implementierung der CLR für Windows-Betriebssysteme.

- **Sprachunterstützung:** Java EE unterstützt ausschließlich die Sprache Java. In anderen Sprachen geschriebene Programme können lediglich über das Java Native Interface (JNI) eingebunden werden. Diese Form der Einbindung ist jedoch mit zusätzlichem Aufwand verbunden. Die CLR kann theoretisch jede

CLS-konforme Sprachen ausführen. Compiler für sehr viele Sprachen existieren bereits.

- **Entwicklungstools:** Während es für Java EE eine Menge verschiedener Werkzeuge und Entwicklungsplattformen gibt, wird für die Entwicklung von .NET-Anwendungen im Wesentlichen die Visual Studio .NET Entwicklungsumgebung eingesetzt.

Fazit

Die wichtigsten Unterschiede zwischen beiden Frameworks liegen in der Plattformstrategie. Während Java EE eine plattformunabhängige, an die Sprache Java gebundene Lösung darstellt, ist .NET sprachunabhängig, jedoch an die Windows-Plattform gebunden. Im Bezug auf die Präsentationsschicht scheint .NET mit dem Code-Behind-Modell, den Web-Controls und der guten Visual Studio Integration den Lösungen für Java EE überlegen zu sein. Im Bereich der Geschäftslogik, insbesondere bei der Anbindung an relationale Datenbanken, bietet Java EE die überzeugenderen Ansätze. Ein Nachteil der .NET Plattform ist hier vor allem darin zu sehen, dass ADO.NET noch kein automatisches O/R-Mapping unterstützt. Für Java EE gibt es neben der JPA viele weiter Frameworks die diese Funktion automatisieren. Sowohl Java EE- als auch .NET-Komponenten können relativ leicht als Web-Services bereitgestellt und eingebunden werden. Hierdurch werden auch heterogene Softwarekomponenten, in denen Java EE, .NET und andere Ansätze verknüpft werden, ermöglicht. Web-Services sind die „lingua franca“ der Software-Integration und entsprechen damit der Sprache „Englisch“ im internationalen Geschäftsleben.

Kapitel 6

Entwurfsmuster

Entwurfsmuster [GHJV96] sind bewährte Lösungen häufig auftauchender Entwurfsprobleme. Durch ihre Verwendung ist eine Wiederverwendung auf Entwurfsebene möglich; das Rad muss nicht mehr ständig neu erfunden werden. Im objektorientierten Kontext besteht ein Muster aus einem System von wenigen Klassen. Muster erlauben, ein Softwaresystem auf einem höheren Abstraktionsniveau zu verstehen. Statt als System von sehr vielen Klassen wird es von den Entwicklern als Kombination von deutlich weniger Mustern wahrgenommen und dadurch leichter überblickt. Jeder professionelle Softwareentwickler, der diese Bezeichnung verdient, sollte zumindest die wichtigsten Muster kennen und bei den übrigen Muster wissen, wozu sie dienen und wo ihre Details bei Bedarf nachgeschlagen werden können. Muster sind in nahezu jeder gut strukturierten objektorientierten Architektur zu finden und besitzen vier grundlegende Elemente.

Jedes Muster wird durch einen *Musternamе* benannt. Entwurfsmuster stellen somit eine Terminologie für den Softwareentwurf zur Verfügung, welche die Kommunikation und die Dokumentation erleichtert und erweitert. Der *Problemabschnitt* beschreibt, wann ein Muster anzuwenden ist, welches Problem adressiert wird und was sein Kontext ist. Oft wird die Problembeschreibung eine Liste von Bedingungen aufführen, die erfüllt sein müssen, wenn die Anwendung des Musters sinnvoll sein soll. Es beschreibt mögliche spezifische Entwurfsprobleme oder Klassen- oder Objektstrukturen, die symptomatisch für einen unflexiblen Entwurf sind. Der *Lösungsabschnitt* beschreibt eine allgemeine Anordnung von Klassen und Objekten, aus denen der Entwurf besteht, sowie ihre Beziehungen, Zuständigkeiten und Interaktionen. Der *Konsequenzenabschnitt* beschreibt die Vor- und Nachteile des aus der Anwendung eines Musters resultierenden Entwurfs und ist oft von zentraler Bedeutung für die Bewertung von Entwurfsalternativen. Die Konsequenzen der Musteranwendung für den Entwurf betreffen oft den Speicherplatzverbrauch, die Laufzeit und Sprach- und Implementierungsaspekte. Da in objektorientierten Systemen oft der Wiederverwendbarkeit eine zentrale Rolle zugeschrieben wird, umfasst der Konsequenzenabschnitt eines Musters auch seinen Einfluss auf die Flexibilität,

Erweiterbarkeit und Portabilität des Systems. In vielen Situationen können mehrere Muster verwendet werden, um ein Problem zu lösen. Welches dann ausgewählt wird, hängt davon ab, welche mit jedem Muster verbundenen Vor- und Nachteile in der jeweiligen Situation am wichtigsten sind. Außerdem werden Muster oft kombiniert.

Entwurfmuster können in drei Kategorien unterteilt werden: Erzeugungsmuster, Strukturmuster und Verhaltensmuster. *Erzeugungsmuster* betreffen die Erzeugung von Objekten. Die Unterscheidung der anderen beiden Kategorien ist nicht sehr trennscharf und ihre Beschreibungen sind recht generisch: *Strukturmuster* befassen sich mit der Zusammensetzung von Klassen und Objekten, während ein *Verhaltensmuster* ein interessantes Verhalten realisiert. Entwurfsmuster basieren auf den beiden Mechanismen Vererbung und Delegation, d.h. Weitergabe einer Anfrage an ein per Assoziation angebundenes Nachbarobjekt. Abhängig davon, ob *Vererbung* oder *Delegation* bei dem jeweils betrachteten Muster von größerer Bedeutung ist, spricht man auch von einem *Klassen-* bzw. *Objekt-basierten* Muster. Bei Mustern, die auf Flexibilität abzielen, wird typischerweise Delegation eingesetzt, da so der verwendete Dienstleister zur Laufzeit ausgetauscht werden kann, während Vererbungshierarchien statisch sind. In den nachfolgenden Abschnitten werden einige Entwurfsmuster aus den drei erwähnten Kategorien vorgestellt. Es wird beschrieben, wann sie einsetzbar sind und welche Konsequenzen deren Einsatz hat.

6.1 Erzeugungsmuster

Entwurfsmuster, die der Erzeugung von Objekten dienen, verstecken den Erzeugungsprozess und helfen so, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden.

6.1.1 Abstrakte Fabrik

Das wichtigste Erzeugungsmuster ist wohl die abstrakte Fabrik (Abstract Factory). Dieses objektbasierte Erzeugungsmuster bietet eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu spezifizieren.

Betrachten wir als Beispiel eine Klassenbibliothek für Benutzerschnittstellen, die mehrere Look-and-Feel-Standards wie Swing oder Java AWT unterstützt. Unterschiedliche Look-and-Feel-Standards legen jeweils ein unterschiedliches Aussehen und Verhalten der Interaktionselemente einer Benutzungsschnittstelle (Widgets) fest. Typische Widgets sind Scrollbars, Fenster oder Schaltflächen. Damit eine Anwendung zwischen verschiedenen Look-and-Feel-Standards portierbar bleibt, sollte sie sich nicht auf die Widgets eines spezifischen Standards festlegen und die Erzeugung von Look-and-Feel-spezifischen Widgetklassen nicht über die ganze Anwendung verteilen.

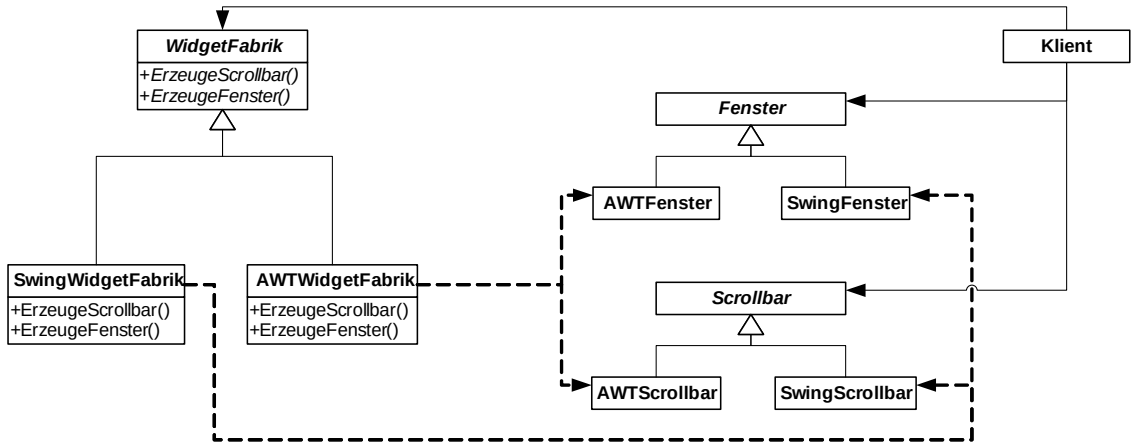


Abbildung 6.1: Entwurfsmuster Abstrakte Fabrik am Beispiel.

Dieses Designproblem kann mit einer abstrakten WidgetFabrik gelöst werden, die eine Schnittstelle zum Erzeugen jeder grundlegenden Art von Widget deklariert (s. Abbildung 6.1). Jede Widgetart wird durch eine abstrakte Klasse repräsentiert, deren konkrete Unterklassen den jeweiligen Look-and-Feel-Standard implementieren. Für jeden Look-and-Feel-Standard gibt es eine konkrete Unterklasse der abstrakten Widgetfabrik, welche für jede abstrakte Widgetklasse eine Operation zum Erzeugen eines neuen Widgets implementiert. Eine abstrakte Fabrik verlagert folglich die Erzeugung von Objekten auf ihre konkreten Unterklassen. Klienten erzeugen neue Widgets ausschließlich über die durch die abstrakte WidgetFabrik definierte Schnittstelle, ohne die konkreten Klassen zu kennen, die sie benutzen. Auf diese Weise bleiben sie unabhängig vom aktuellen Look-and-Feel. Zudem sichert eine Widgetfabrik die Abhängigkeiten zwischen den konkreten Widgetklassen ab. Ein Swing-Fenster sollte nur mit einem Swing-Scrollbar ausgestattet werden. Diese Konsistenzbedingung wird automatisch als Konsequenz des Einsatzes einer SwingWidgetFabrik sichergestellt.

Das Abstrakte-Fabrik-Muster kann verwendet werden, wenn (1) ein System unabhängig von der Erzeugung, Zusammensetzung und Repräsentation seiner erzeugten Objekte sein soll, (2) ein System mit einer von mehreren Produktfamilien konfiguriert werden soll, (3) Konsistenzbedingungen bei der Zusammenarbeit von verwandten Objekten sichergestellt werden müssen, und (4) nur eine Schnittstelle, aber nicht die konkrete Implementierung einer Klassenbibliothek offen gelegt werden soll.

Das Abstrakte-Fabrik-Muster hat die folgenden Vorteile. Zunächst ermöglicht es, Klassen von Objekten zu steuern, welche durch die Anwendung erzeugt werden. Da eine Fabrik für den Prozess des Erzeugens von Objekten zuständig ist und ihn kapselt, isoliert es Klienten von den Implementierungs- und Produktklassen. Des Weiteren lässt sich ein einfacher Austausch von Produktfamilien realisieren. Die Klasse einer konkreten Fabrik erscheint nur einmal in der Anwendung, nämlich genau dort,

wo von ihr ein Exemplar erzeugt wird. Dies ermöglicht einen einfachen Austausch der von der Anwendung benutzten Fabriken und der damit zusammenhängenden Produktkonfigurationen (aus dem gleichen Grund implementiert man abstrakte Fabriken am besten als Singleton (s.u.) [GHJV96]). Um verschiedene Look-and-Feel-Objekte zu erzeugen, sollten Klienten unterschiedliche konkrete Fabriken haben. Ein weiterer Vorteil ist die einfache Konsistenzsicherung unter den Produkten, um zu gewährleisten, dass nur Objekte einer Familie zu einer Zeit verwendet werden.

Als Nachteil von abstrakten Fabriken ist deren Erweiterbarkeit bzw. die Unterstützung neuer Produkte zu nennen. Dies liegt daran, dass die Schnittstelle der Fabrik die Menge von generierbaren Produkten festlegt. Die Unterstützung neuer Arten von Produkten erfordert es, diese Schnittstelle zu erweitern, was dazu führt, die abstrakte Fabrik-Klasse und all ihre Unterklassen zu verändern.

6.1.2 Weitere Erzeugungsmuster

Details zu den im Folgenden kurz skizzierten weiteren Erzeugungsmustern findet man in [GHJV96]:

Fabrikmethode: wird häufig in so genannten Frameworks, d.h. in objektorientierten Klassenbibliotheken, eingesetzt. Ein Framework stellt oft sehr allgemeine Klassen zur Verfügung, die bei seiner Nutzung durch Anwendungs-bezogene Klassen ergänzt werden müssen. Problem ist, dass bei der Framework-Erstellung diese Klassen nicht bekannt sind und das Framework dennoch manchmal Objekte dieser Klassen erzeugen muss. Letzteres wird von dem Fabrikmethode-Muster dadurch ermöglicht, dass die betreffende Klasse des Frameworks eine abstrakte Methode enthält, die in einer vom Framework-Nutzer erstellten Unterklasse so überschrieben wird, dass bei Aufruf der Fabrikmethode das gewünschte Objekt erzeugt wird.

Singleton: stellt ein Einzelobjekt zur Verfügung und verhindert, dass weitere Objekte der betreffenden Klasse erzeugt werden. Es wird häufig verwendet, um in einem Softwaresystem globale Einstellungen zu speichern.

Prototyp: erzeugt Objekte und insbesondere komplexe Objektstrukturen durch Klonen von Vorlagen statt durch direkte Verwendung von Konstruktoren.

Erbauer: trennt den Konstruktionsprozess von komplexen Objektstrukturen von deren Repräsentation.

6.2 Strukturmuster

Strukturmuster befassen sich mit der Komposition von Klassen und Objekten, um größere Strukturen zu bilden. Als Beispiel wird im Folgenden der Adapter (bzw.

Wrapper) vorgestellt, der, je nach Realisierung, sowohl ein klassen- oder objektbasiertes Strukturmuster sein kann.

6.2.1 Adapter

Mitunter kann eine als wiederverwendbar entwickelte Klasse einer Klassenbibliothek nicht wiederverwendet werden, weil ihre Schnittstelle nicht der von der Anwendung verlangten spezifischen Schnittstelle entspricht. Das Adaptermuster löst dieses Problem und lässt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten nicht dazu in der Lage wären. Im Allgemeinen passt ein Adapter die Schnittstelle des zu adaptierenden Objekts an eine andere Schnittstelle an und bietet so eine einheitliche Abstraktion von mehreren unterschiedlichen Schnittstellen. Die Schnittstelle des Adapters wird auf Basis der Schnittstelle des adaptierten Objekts realisiert.

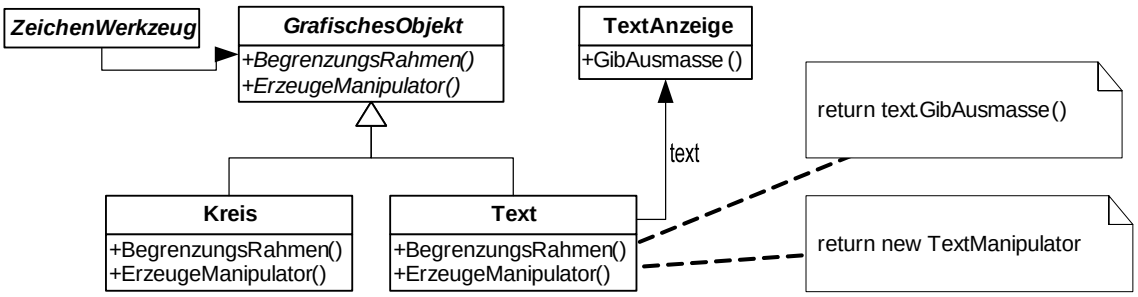


Abbildung 6.2: Entwurfsmuster Adapter am Beispiel.

Betrachten wir als Beispiel ein Zeichenwerkzeug, mit dem Linien, Polygone, Kreise, Texte usw. gezeichnet und zu Bildern und Diagrammen arrangiert werden können (vgl. Abbildung 6.2). Jedes dieser grafischen Objekte kann editiert werden und einen passenden, rechteckigen Begrezungsrahmen ermitteln. Der Editor definiert hierzu für jedes grafische Objekt eine konkrete Unterklasse, z.B. eine Linien-Klasse für Linien, eine Polygon-Klasse für Polygone usw., die von einer abstrakten Klasse GrafischesObjekt abgeleitet werden. Diese Oberklasse stellt eine einheitliche Schnittstelle für die konkreten grafischen Objekte bereit. Elementare geometrische Objekte können sehr einfach implementiert werden, da ihre Zeichen- und Editiermöglichkeiten beschränkt sind. Nehmen wir an, dass eine vorhandene Klasse TextAnzeige bereits die Begrenzung eines Texts ermitteln kann und dass wir diese nutzen wollen. Das Problem ist nun, dass die Klasse TextAnzeige nicht für GrafischesObjekt-Klassen entworfen wurde und die Schnittstellen daher nicht übereinstimmen. Eine mögliche Lösung dieses Problems ist, die TextAnzeige-Klasse an die durch GrafischesObjekt definierte Schnittstelle anzupassen. Dies ist jedoch nicht sinnvoll, da dann auch alle Klienten der Klasse angepasst werden müssten. Besser ist es, die neue Adapterklasse Text so dazwischen zu schalten, dass sie die Schnittstelle von TextAnzeige an

die Schnittstelle von GrafischesObjekt anpasst. Hierbei gibt es zwei Möglichkeiten: (1) Text erbt sowohl von GrafischesObjekt als auch von TextAnzeige oder (2) Text erbt nur von GrafischesObjekt und ein TextAnzeige-Objekt wird per Delegation an ein Text-Objekt angebunden, wobei die Klasse Text die Methode GibAusmasse der Klasse TextAnzeige nutzt, um die geforderte Methode BegrenzungsRahmen zu implementieren. Diese beiden Ansätze entsprechen der klassen- bzw. objektbasierten Version des Adaptermusters. In Abbildung 6.2 wird der Objektadapter-Ansatz verwendet. Man beachte, dass die Adapterklasse Funktionalität, die von der zu adaptierenden Klasse nicht geboten, von der zu implementierenden Schnittstelle aber erwartet wird, selbst implementiert; im Beispiel ist dies die Methode ErzeugeManipulator.

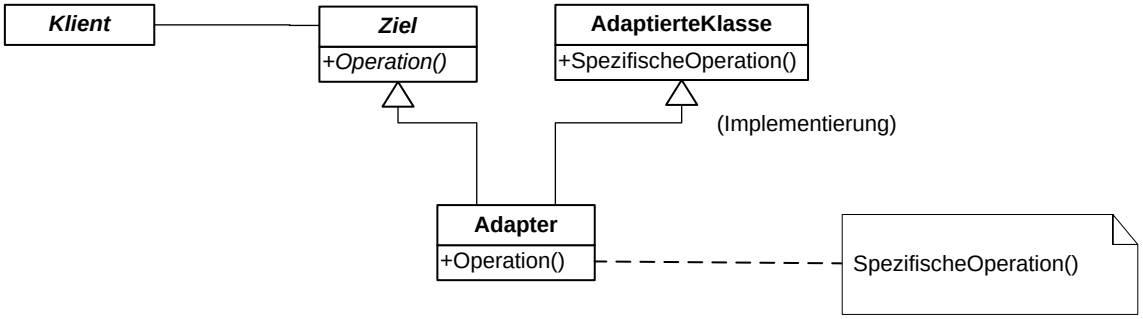


Abbildung 6.3: Klassen-Adapter.

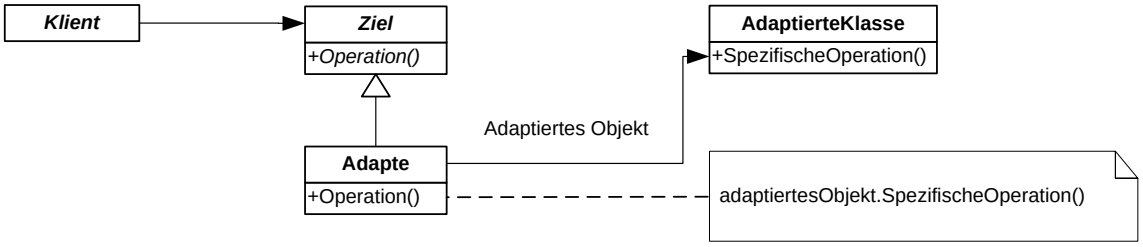


Abbildung 6.4: Objekt-Adapter.

Das Adaptermuster kann verwendet werden, wenn (1) eine existierende Klasse benutzt werden soll, deren Schnittstelle nicht der benötigten Schnittstelle entspricht, (2) eine wiederverwendbare Klasse erstellt werden muss, die mit unabhängigen oder nicht vorhersehbaren Klassen zusammenarbeiten soll, welche nicht notwendigerweise über kompatible Schnittstellen verfügen, und (3) (nur für Objektadapter) verschiedene existierende Unterklassen benutzt werden müssen und es unpraktisch ist, die Schnittstellen jeder einzelnen Unterklassen durch Ableiten anzupassen.

Ein Klassenadapter verwendet Mehrfachvererbung, um eine Schnittstelle an eine andere anzupassen (vgl. Abb. 6.3), und ist daher in Sprachen wie Java, die nur

Einfachverbung bieten, nicht verwendbar. Ein Objektadapter verwendet Delegation (vgl. Abb. 6.4). In beiden Fällen ruft der Klient die Operationen von Adapterobjekten auf, die sich auf Methoden der adaptierten Klasse abstützen.

Ein Klassenadapter passt die zu adaptierende Klasse an genau eine konkrete Zielklasse an. Daraus ergibt sich der Nachteil, dass ein Klassenadapter nicht funktioniert, wenn wir eine Klasse und all ihre Unterklassen anpassen wollen. Der Vorteil eines Klassenadapters ist, dass er Teile des von der adaptierten Klasse geerbten Verhaltens überschreiben kann, weil er Unterklasse von ihr ist. Zudem wird lediglich ein einzelnes Objekt verwendet. Es wird keine Zeigerindirektion benötigt, um zum adaptierten Objekt zu gelangen. Im Gegensatz dazu erlaubt ein Objektadapter sowohl mit der anzupassenden Klasse selbst als auch mit allen ihren Unterklassen zusammenzuarbeiten. Des Weiteren kann der Adapter neue Funktionalität allen adaptierten Klassen auf einmal hinzuzufügen. Jedoch können die Methoden der anzupassende Klasse nicht überschrieben werden.

6.2.2 Weitere Strukturmuster

Details zu den im Folgenden kurz skizzierten weiteren Strukturmustern findet man in [GHJV96]:

Brücke: trennt eine Schnittstelle von ihrer Implementierung und ermöglicht, beide unabhängig zu modifizieren.

Dekorierer: erlaubt, eine Klasse dynamisch um Funktionalität zu erweitern bzw. ihr Funktionalität zu entziehen. In solchen Anwendungen vermeidet dieses Muster die Erstellung einer sehr komplexen Klassenhierarchie.

Fassade: vereinfacht die Nutzung eines Subsystems durch Bereitstellung einer kompakten, einfach zu verwendenden Schnittstelle. Intern leitet die Fassadenklasse Aufrufe an die passenden Klassen des Subsystems weiter.

Fliegengewicht: dient dem Sparen von Speicherplatz bei vielen, ähnlichen Kleinstobjekten.

Kompositum: repräsentiert rekursiv aufgebaute Objektstrukturen wie z.B. Bäume. Die Klassen `Ausdruck`, `Konstante` und `Geschachtelt` in Abb. 4.6 sind eine Ausprägung des Kompositum-Musters zur Darstellung arithmetischer Ausdrücke, z.B. im Rahmen eines Compilers.

Stellvertreter: verschiebt die Kontrolle über ein Objekt auf ein vorgelagertes Stellvertreter-Objekt mit gleicher Schnittstelle. Der Stellvertreter kann beispielsweise die Funktionalität eines entfernten Objekts lokal zur Verfügung stellen oder ein Speicherplatz-intensives Objekt solange im Hauptspeicher vertreten, bis es wirklich gebraucht wird, und es dann nachladen.

6.3 Verhaltensmuster

Verhaltensmuster befassen sich mit Algorithmen und der Zuweisung von Zuständigkeiten zu Objekten und beschreiben nicht nur Muster von Objekten oder Klassen, sondern auch Muster der Interaktion zwischen ihnen, d.h. komplexe Kontrollflüsse. Einige dieser Muster beschreiben, wie eine Gruppe von Objekten zusammenarbeitet, um eine Aufgabe zu erfüllen, die keines der Objekte alleine ausführen kann.

6.3.1 Befehl

Das Befehlsmuster ist ein objektbasiertes Verhaltensmuster. Hierbei wird ein Befehl (eine Operation) als ein Objekt gekapselt. Durch Anbindung eines Befehlsobjekts per Delegation wird einem Klienten-Objekt die Operation als Dienstleistung zur Verfügung gestellt. Durch Anbindung eins anderen Befehlsobjekts kann zur Laufzeit die Funktionalität des Klientenobjekts indirekt geändert werden. Die Kapselung einer Operation als Objekt erlaubt eine Vielzahl von Verwendungsmöglichkeiten (ggf. in Kombination mit anderen Mustern) wie z.B. einen Klienten mit einem austauschbaren Befehl zu parametrisieren, Operationen in eine Queue zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen. In manchen Situationen ist es notwendig, Anfragen an Objekte zu stellen, ohne irgendetwas über die auszuführende Operation zu wissen oder das Objekt zu kennen, an das die Anfrage gerichtet wird. Ein Beispiel hierfür sind Klassenbibliotheken für Benutzungsschnittstellen, welche Menüs oder Schaltflächen bereitstellen, die als Reaktion auf eine Benutzereingabe eine bestimmte Operation auslösen. Der Entwickler eines solchen Frameworks kennt das Zielobjekt einer Anfrage und die auszuführenden Operationen nicht.

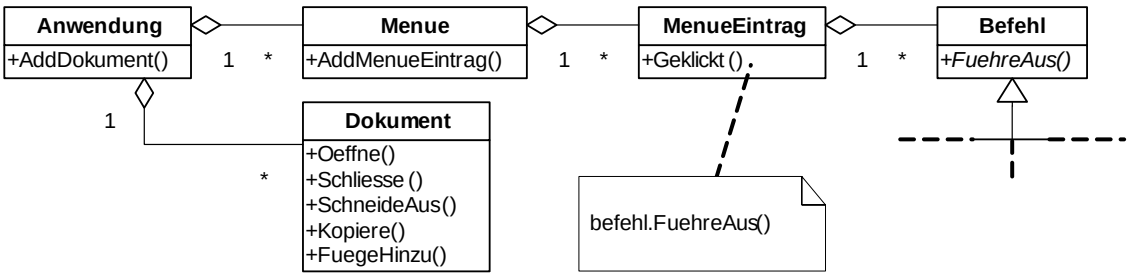


Abbildung 6.5: Entwurfsmuster Befehl am Beispiel.

Mit dem Befehlsmuster können Anfragen an unbekannte Anwendungsobjekte gerichtet werden, indem die Anfrage selbst zu einem Objekt gemacht wird. Der Kern dieses Musters ist eine abstrakte Klasse Befehl, die eine Schnittstelle zum Ausführen von Operationen deklariert. Im einfachsten Fall ist dies eine abstrakte FuehreAus-Operation, welche von konkreten Unterklassen implementiert wird. Exemplare dieser Unterklassen werden einem Empfänger übergeben, der die entspre-

chende FuehreAus-Operation umsetzt. Beispielsweise kann jede Auswahlmöglichkeit eines Menüs durch ein Exemplar der Klasse MenueEintrag realisiert werden. Ein Objekt der Klasse Anwendung erzeugt diese Menüs mit allen ihren MenüEinträgen und verwaltet Objekte der Klasse Dokument, die ein Benutzer geöffnet hat (Abb. 6.5). Jeder MenueEintrag wird mit einer konkreten Unterklasse von Befehl konfiguriert. Bei Auswahl eines MenueEintrags ruft das MenueEintrag-Objekt die Operation FuehreAus auf seinem Befehlsobjekt auf, das die entsprechende Operation umsetzt. Welche konkrete Unterklasse sich hinter ihrem Befehlsobjekt verbirgt, ist einem Exemplar der Klasse MenueEintrag nicht bekannt.

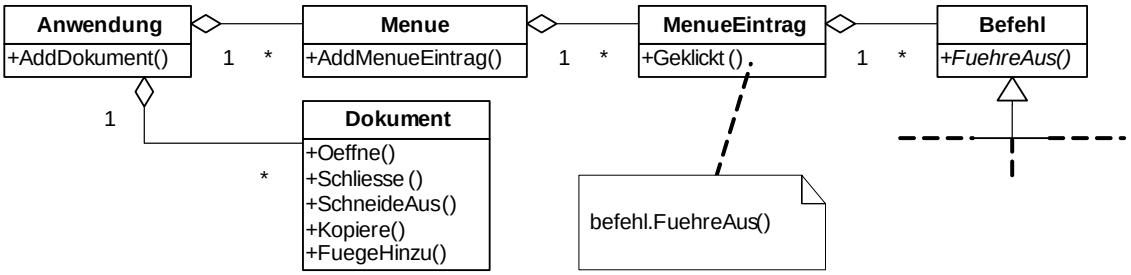


Abbildung 6.6: Entwurfsmuster Befehl.

Abbildung 6.6 stellt die Struktur des Befehlsmusters dar. Der Klient (Anwendung) erzeugt ein KonkreterBefehl-Objekt und übergibt es dem Empfänger, welcher weiß, wie die an die Ausführung einer Anfrage gebundenen Operationen auszuführen sind. Ein Aufrufer (MenueEintrag) speichert das Befehlsobjekt und ruft bei einem bestimmten registrierten Ereignis (Geklickt) die FuehreAus-Operation des Befehlsobjekts auf.

Das Befehlsmuster hat mehrere Vorteile. Zum einem entkoppelt das Befehlsmuster das Objekt, das die Anfrage auslöst, von dem, das weiß, wie sie umzusetzen ist. Zum anderen ist es einfach, neue Befehlsobjekte hinzuzufügen, weil keine bereits existierenden Klassen geändert werden müssen. Weiterhin kann ein Befehlsobjekt wie jedes andere Objekt auch manipuliert und erweitert werden. So lassen sich mehrere Befehlsobjekte zu einem komplexeren Makro-Befehl zusammensetzen.

Die Klassen `BinOp` und `Addition` in Abb. 4.6 sind ein weiteres Beispiel für eine Verwendung des Befehlsmusters.

6.3.2 Beobachter

Oftmals muss man die Konsistenz zwischen den miteinander in Beziehung stehenden Objekten eines Anwendungssystems aufrechterhalten. Eine enge Kopplung der Klassen sollte hierbei aus Gründen der Wiederverwendbarkeit und Übersichtlichkeit vermieden werden. Beobachter ist ein objektbasiertes Verhaltensmuster, bei dem eine 1:n-Abhängigkeit zwischen Objekten definiert wird, so dass die Änderung des

Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

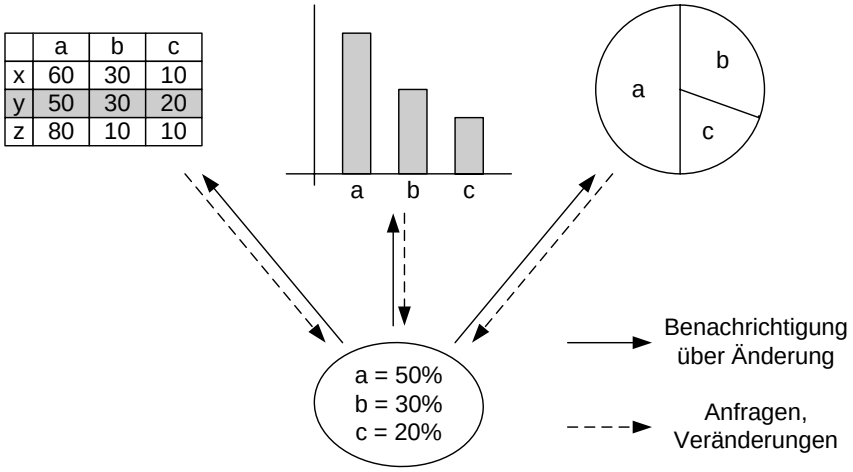


Abbildung 6.7: Objekt mit unterschiedlichen Visualisierungen.

Abbildung 6.7 zeigt verschiedene gleichzeitig genutzte Objekte zur Darstellung (Beobachter) desselben Anwendungsdatenobjekts (Subjekt), hier einer Tabelle. Wenn sich die Tabelle ändert, so wird zunächst das Datenobjekt aktualisiert, welches dann alle Darstellungsobjekte über seine Zustandsänderung informiert. Als Reaktion darauf synchronisiert sich jedes Darstellungsobjekt mit dem Zustand des Datenobjekts mit Hilfe von Anfragen.

Abbildung 6.8 zeigt die Struktur des Beobachtermusters. Die Basisklasse Subjekt definiert das Protokoll zur Benachrichtigung über Änderungen und die Schnittstelle zum An- und Abmelden einer beliebigen Anzahl von Beobachtern. Ein Beobachter definiert eine Aktualisierungsschnittstelle, über welche die Synchronisation mit den geänderten Daten des Subjekts erfolgt. Der Zustand eines KonkretesSubjekt-Objekts kann über die Operation SetzeZustand geändert werden. Eine Zustandsänderung des konkreten Subjekts bewirkt die Benachrichtigung aller registrierten Beobachter, welche sich daraufhin mit dem Subjekt über die Operation GibZustand synchronisieren.

Das Beobachtermuster kann verwendet werden, wenn (1) die Änderung eines Objekts die Änderung anderer Objekte verlangt, wobei die Anzahl der zu ändernden Objekte nicht bekannt ist, (2) ein Objekt andere Objekte benachrichtigen sollte, ohne Annahmen darüber treffen zu dürfen, wer diese Objekte genau sind, und (3) wenn eine Abstraktion zwei voneinander abhängige Aspekte besitzt, die aus Gründen der Wiederverwendung in unterschiedlichen Objekten gekapselt werden sollen. Somit ermöglicht das Beobachtermuster, Subjekte und Beobachter voneinander unabhängig zu variieren. Subjekte können wiederverwendet werden, ohne ihre Beobachter wiederverwenden zu müssen, und umgekehrt. Zudem können neue Beob-

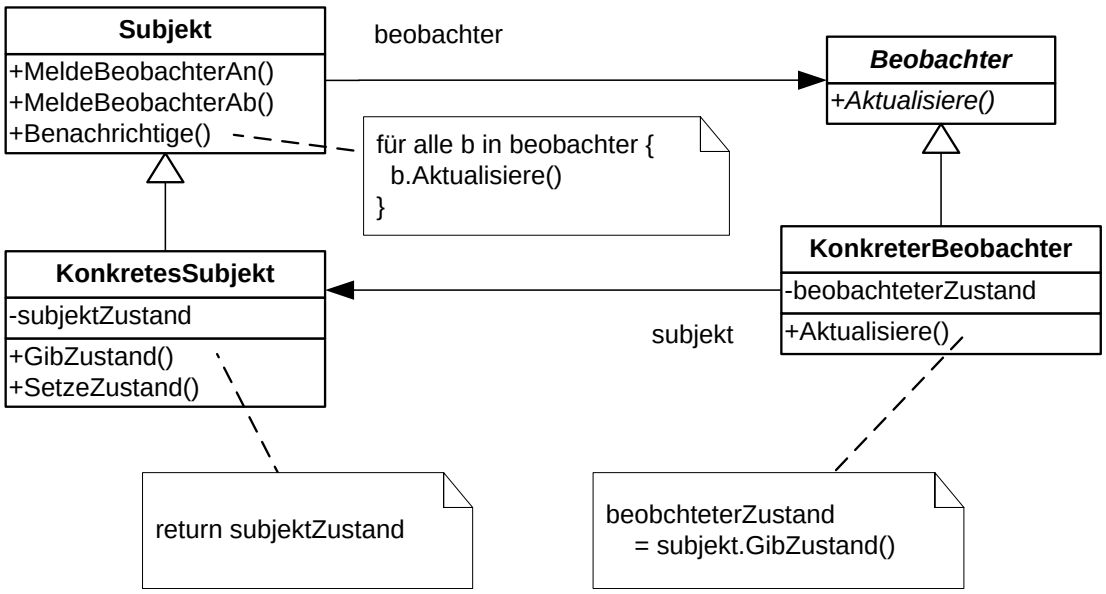


Abbildung 6.8: Entwurfsmuster Beobachter.

achter ohne Modifikation des Subjekts oder anderer Beobachter hinzugefügt werden. Nachteil des Musters ist, dass eine scheinbar harmlose Operation auf dem Subjekt zu aufwändigen Aktualisierungen bei den Beobachtern führen kann. Schlecht definierte und gewartete Abhängigkeiten führen des Weiteren schnell zu unsinnigen Aktualisierungen, die schwer aufzufinden sind.

6.3.3 Weitere Verhaltensmuster

Details zu den im Folgenden kurz skizzierten weiteren Verhaltensmustern findet man in [GHJV96]:

Besucher: erlaubt bei Klassen mit sehr umfangreichen Schnittstellen und Operationen, die sich in Themenbereiche untergliedern lassen, die Operationen jeweils eines Themenbereichs in eine eigene Besucherklasse herauszuziehen. Hierdurch kann der Themenbereich getrennt betrachtet und verstanden werden sowie die Klasse übersichtlich gehalten werden.

Iterator: erlaubt das schrittweise Durchlaufen einer Kollektion, wie z.B. einer Liste.

Memento: ermöglicht, den Zustand eines Objektes extern zu sichern und bei Bedarf wiederherzustellen.

Schablonenmethode: ähnlich wie bei der Fabrikmethode wird in einer Oberklasse eine abstrakte (Schablonen-)Methode vorgesehen, die in einer Unterklasse überschrieben wird.

Strategie: erlaubt es, Algorithmen in eigenen Klassen zu kapseln und so austauschbar zu halten.

Vermittler: steuert die Kooperation von Objekten.

Zustand: erlaubt, das Verhalten eines Objekts von seinem Zustand abhängig zu machen. Für jeden Zustand wird eine eigene Unterklasse der Zustandsklasse mit hierfür spezifischem Verhalten bereitgestellt. Durch Anbindung eines Objekts an ein Objekt der passenden Unterklasse wird dafür gesorgt, dass es zum Zustand passende Dienstleistungen bereitgestellt bekommt und sich dadurch selbst zustandsabhängig verhält.

Zuständigkeitskette: schaltet mehrer Objekte hintereinander. Eine eingehende Anfrage wird die entstandene Kette entlang geleitet, bis ein Objekt die Anfrage beantworten kann.

6.4 Weitere Entwurfsmuster

Neben den klassischen Entwurfsmuster der „Gang-of-Four“ haben inzwischen noch einige andere Muster eine gewisse Bedeutung erlangt. Einige hiervon werden im Folgenden kurz skizziert:

Datentransferobjekt: (data transfer object, DTO) stellt eine Zeile des Ergebnisses einer Datenbankabfrage als Objekt zur Verfügung und erlaubt auf die Attribute über get-Methoden zuzugreifen.

Datenzugrifsobjekt: (data access object, DAO) kapselt den Zugriff auf eine Datenquelle, wie z.B. eine Datenbank, und hält diese daher austauschbar.

Business-Delegate: entkoppelt Präsentations- und Geschäftslogik.

Kapitel 7

Testen

Zum Thema Testen sollen im folgenden drei Aspekte genauer betrachtet werden. Zunächst wird erläutert, wie zu testende Softwarebausteine mit möglichst geringem Änderungsaufwand isoliert getestet werden können. Dann wird die Automatisierung des Testens behandelt. Im letzten Abschnitt dieses Kapitels wird dann gezeigt, nach welchen Prinzipien Testfälle erzeugt werden können.

7.1 Isolation zu testender Einheiten

Beim Testen ist es nicht sinnvoll, sofort das zu komplette, betrachtete Gesamtsystem im Rahmen eines Big-Bang-Ansatzes zu testen. Das Ergebnis eines solche Versuchs wäre absehbar: es würde ein Fehler auftreten, der aber im Rahmen des Gesamtsystems nur sehr schwer lokalisieren ließe. Stattdessen beginnt man typischerweise mit einzenen Basiseinheiten (Modulen, Klassen) und setzt diese dann sukzessiv zu immer größeren Teilsystemen zusammen, wobei auftretende Fehler jeweils vor einer weiteren Integration beseitigt werden. Ein Problem beim Testen von Basiseinheiten und Teilsystemen ist, dass diese oft andere Bausteine erfordern, die beim Testen zunächst noch nicht berücksichtigt werden sollen. Durch eine geschickt gewählte Testreihenfolge lässt es sich oft erreichen, dass von dem zu testenden Baustein nur solche Bausteine benötigt werden, die bereits sorgfältig getestet wurden. In diesem Fall kann man direkt auf die getesteten Bausteine zurückgreifen und benötigt keine weiteren Vorkehrungen. Manchmal benötigen sich Bausteine aber wechselseitig. Dann lässt es sich oft nicht umgehen, einen der Bausteine zunächst durch einen vereinfachten Baustein (auch Stumpf (Stub) oder Mock genannt) zu ersetzten, der die Funktionalität des eigentlichen Bausteins auf möglichst einfache Weise simuliert (vgl. Abb. 7.1).

Ein Problem bei der Verwendung von Teststümpfen ist, dass der zu testende Baustein oft geändert werden muss, um mit den Teststümpfen verknüpft zu werden. Dies ist nicht nur umständlich, sondern birgt auch das Risiko, dass diese Änderungen

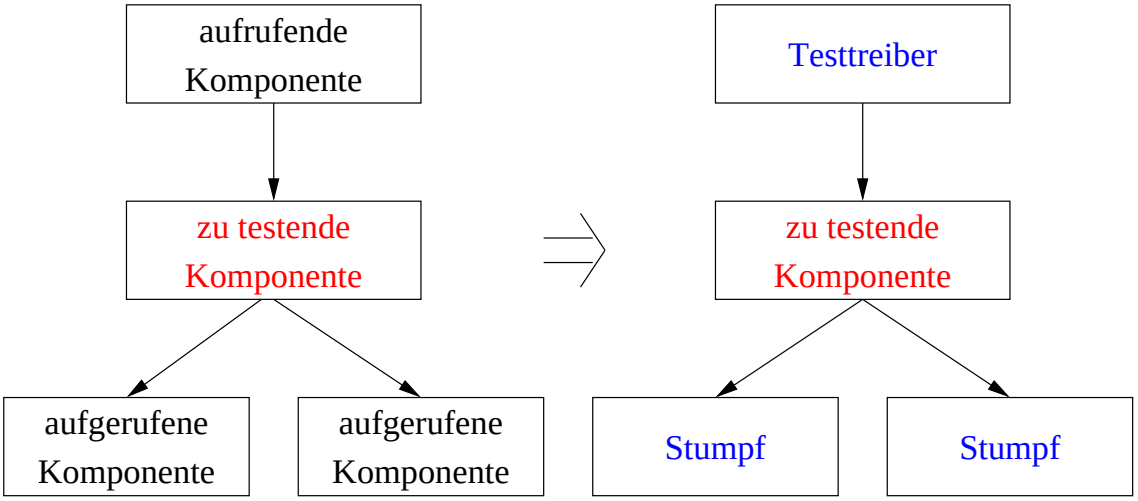


Abbildung 7.1: Ersetzen von benachbarten Bausteinen durch vereinfachte Teststümpfe.

nach dem Testen nicht alle wieder revidiert werden, so dass ein vereinfachter Teststumpf irrtümlich statt des eigentlichen Softwarebausteins im System verbleibt.

Bei der objektorientierten Softwareentwicklung lässt sich dieses Problem zumindest bei einem geeigneten Design umgehen. Wie das erreicht werden kann, wird in [Jun04, Jun02] im Detail erläutert. Die Grundidee besteht darin, dass eine Mock-Klasse Unterklasse der Klasse wird, die sie im Test ersetzen soll (vg. Abb. 7.2).

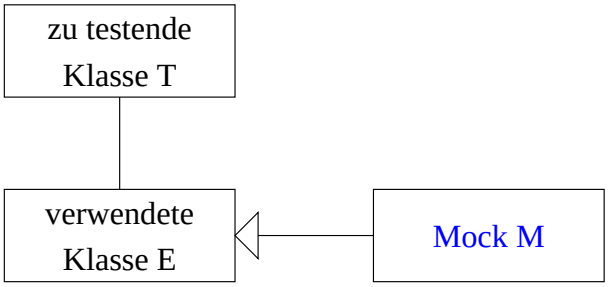


Abbildung 7.2: Verwendung von Mock-Klassen.

Eine Änderung der zu testenden Klasse T kann vermieden werden, wenn in T die zu ersetzende Klasse E nicht „fest verdrahtet “ wird, sondern wenn ein T-Objekt stattdessen auch mit Objekten der Mock-Klasse M verknüpft werden kann. Insbesondere darf in T nicht der Konstruktor von E verwendet werden. Stattdessen sollte beispielsweise eine abstrakte Fabrik (vgl. Abschnitt 6.1.1) genutzt werden, um die benötigten Nachbarobjekte eines T-Objekts zu erzeugen. Im Normalbetrieb wird dann eine abstrakte Fabrik verwendet, die E-Objekte erzeugt. Beim Testen von T würde im Testtreiber stattdessen eine abstrakte Fabrik bereitgestellt, die statt

E-Objekten M-Objekte erzeugt.

Voraussetzung für diesen Ansatz ist natürlich, dass die Klasse E überhaupt Unterklassen zulässt. Im Falle von Java bedeutet dies, dass E nicht `final` sein darf.

7.2 Automatisierung des Testens

Um Zeit und Kosten zu sparen, ist es sinnvoll, das Testen so weit wie möglich zu automatisieren. Einige Prozessmodelle für die Softwareentwicklung, wie z.B. Extreme Programming, verlangen eine solche Automatisierung sogar (vgl. Abschnitt 2.3).

Für die Automatisierung von Tests haben in den letzten Jahren einfache, frei verfügbare Frameworks wie JUnit [JUn08, Lin05] (für Java) und Varianten wie NUnit [NUn08] (für alle .NET-Sprachen, inklusive C#, C++, Visual Basic, ...) eine große Beliebtheit erreicht; nicht zuletzt, weil Hersteller von Entwicklungsumgebungen diese in ihre Werkzeuge integriert haben.

Bei diesem Ansatz wird zu einer zu testenden Klasse eine Testklasse als Testtreiber erstelllt. Hierin wird für jeden Testfall eine Methode angelegt. In diesen Testmethoden werden im Framework vordefinierte Methoden wie z.B. `assertEquals` verwendet, mit denen das erhalte Ergebnis mit dem erwarteten verglichen werden kann. Außerdem bietet die Testklasse die Möglichkeit, durch geeignetes Überschreiben der Methode `setUp` das zu testende Objekt vor den Tests in einen definierten Anfangszustand zu bringen. Hierbei können z.B. auch Datenbank- oder Netzwerkverbindungen aufgebaut. Entsprechend erlaubt die Methode `tearDown`, nach den Tests solche Ressourcen wieder frei zu geben.

Im folgenden Beipiel wird eine JUnit-Testklasse `ListTest` gezeigt, mit der eine (hier nicht gezeigte) Klasse `MyList` getestet wird. `MyList` implementiert eine lineare Liste und bietet neben einer Iteratorschnittstelle eine Methode `insertFirst`. In der Testklasse `ListTest` wird beispielhaft eine Methode `testList` gezeigt, in der eine Liste mit den Elementen 42, 17 und 39 erzeugt wird, für die anschließend mit `assertEquals` überprüft wird, ob diese Elemente auch ordnungsgemäß eingetragen wurden.

```
import junit.framework.*;

public class ListTest extends TestCase{
  public static Test suite(){
    return new TestSuite(ListTest.class);}

  public static void main(String[] args){
    junit.textui.TestRunner.run(suite());}

  protected void setUp(){}
  protected void tearDown(){}
```

```
  public void testList(){
    MyList<Integer> list = new MyList<Integer>();
    int testvalues[] = {42,17,39};
    for(int i: testvalues) list.insertFirst(i);
    int i = testvalues.length - 1;
    for(Integer val: list)
      assertEquals(val+0,testvalues[i--]);
  }
}
```

Varianten von JUnit gibt es auch für das Testen von Webapplikationen. HttpUnit [Fej08] erlaubt beispielsweise das Testen von Webapplikationen über die aus HTML-Seiten bestehende Benutzeroberfläche. Ähnliche Tools hierfür sind Cactus [Pro08], Selenium [ope08] und Watij [Wat08].

Neben den erwähnten Frameworks gibt es zum automatisierten Abspielen von Testfällen auch kommerzielle Tools wie den IBM Rational TestManager (und andere Rational Tools) [Rat08b, Rat08a] und HP Functional Testing (sowie weitere HP Tools) [HP08a, HP08b]. Solche Tools erlauben auch das Testen von (konventionellen, d.h. nicht HTML-basierten) graphischen Benutzeroberflächen. Man kann Ereignisfolgen bestehend aus Klicks, Mausbewegungen, Textfeldeinträgen usw., wie sie bei der Bedienung einer graphischen Benutzeroberfläche anfallen, aufzeichnen und zur Durchführung eines Tests automatisiert wieder abspielen. Hierbei lässt sich dann prüfen, ob das System erwartungsgemäß reagiert. Problematisch bei diesem Testen der graphischen Benutzeroberfläche kann die mangelnde Robustheit der Testfälle gegenüber Änderungen an der Benutzeroberfläche sein. Nur die ausgereifteren Tools erlauben eine Weiterverwendung von Testfällen nach nicht-trivialen Änderungen an den entsprechenden Fenstern. Die oben genannten Tools zum Testen von Webapplikationen sind hier deutlich robuster, da HTML einen gezielteren Zugriff auf Bedienelemente zulässt, als das die Pixeldarstellung eines Fensters einer konventionellen graphischen Benutzeroberfläche bietet. Dies ist nicht nur ein (weiteres) Argument für Webapplikationen, sondern eröffnet auch die Möglichkeit, in der Testphase mit einer Weboberfläche zu arbeiten, die dann später ggf. durch eine konventionelle graphische Benutzeroberfläche ersetzt bzw. hierum ergänzt wird. Insbesondere bei .NET und den dort vorliegenden Ähnlichkeiten zwischen Webforms und konventionellen Fenstern sollte dies mit überschaubarem Aufwand realisierbar sein.

7.3 Black-Box-Testen

Für das Erstellen von Testfällen gibt es im Wesentlichen zwei Ansätze: Black-Box-Testen und Glass-Box-Testen. Beim Black-Box-Testen geht man von der Spezifikation aus, wie sie z.B. in Form des Pflichtenhefts oder der Schnittstellenbeschreibung von Klassen vorliegt, und versucht, alle hieraus ableitbaren Nutzungsszenarien des betrachteten Softwarebausteins abzuleiten und durch je einen Testfall überprüfen.

Diesen Ansatz bezeichnet man als Black-Box-Testen, da man die Implementierung des Bausteins hierbei nicht berücksichtigt. Er funktioniert häufig gut, wenn man sich auf das normale Verhalten eines Systems konzentiert. Fehler in Sonderfällen, insbsondere solchen, die sich nicht aus der Spezifikation ablesen lassen, sondern durch die bei der Implementierung verwendeten Datenstrukturen und Algorithmen entstehen, lassen sich mit diesem Ansatz nicht besonders gut aufspüren. Ein Vorteil von Black-Box-Testen ist, dass es sich nicht nur für einzelne Klassen und Module sondern auch für Teilsysteme und das Gesamtsystem einsetzen lässt. Für eine Testfall-getriebene Softwareentwicklung, wie sie bei Extreme Programming (vgl. Abschnitt 2.3) verwendet wird, lässt sich ausschließlich mit Black-Box-Testen realisieren.

Ein interessantes Tool zur automatischen Erzeugung von Black-Box-Testfällen für (u.a.) Java ist QuickCheck [Sou08]. Dieses Werkzeug erzeugt mit Hilfe eines Zufallsgenerators (z.B. 100) Testfälle mit steigender Komplexität, die anhand einer vom Tester vorgegebenen Methode, einer so genannten *property*, auf Korrektheit überprüft werden. QuickCheck arbeitet meist sehr schnell und findet Fehler mit erstaunlich großer Zuverlässigkeit. Lediglich Fehler in Codeteilen, die nur mit einer sehr geringen Wahrscheinlichkeit durchlaufen werden, werden, sofern es solche gibt, teilweise nicht gefunden. Problematisch ist allerdings, dass in dem nicht seltenen Fall, dass gewisse, aufgrund der Paramatertypen mögliche Eingabeparameter einer zu testenden Methode unzulässig sind, vom Tester zunächst passende Eingabegeneratoren erstellt werden müssen, da QuickCheck sonst nicht genügend zulässige Testeingaben findet.

7.4 Glass-Box-Testen

Beim Glass-Box-Testen geht man nicht von der Spezifikation sondern von dem Code des betrachteten Bausteins aus und versucht, alle Kontroll- und/oder Datenflüsse, die in diesem Baustein möglich sind, durch möglichst wenige Testfälle abzudecken.

Abbildung 7.3 veranschaulicht den Code für die binäre Suche in einem Array graphisch und zeigt, wie sich der Kontrollfluss mit zwei Testfällen überdecken lässt. In dem ersten, rot dargestellten Testfall wird der Wert 5 in einem 2-elementigen Array mit den Werten 17 und 42 gesucht; im anderen, blau dargestellten Testfall wird im gleichen Array der Wert 42 gesucht. Die rote bzw. blaue Linie zeigen den sich jeweils ergebenden Kontrollfluss.

Ein entsprechendes Tools, was für Java-Klassen ein solches System von Testfällen für die Überdeckung der Kontroll- und Datenflüsse erzeugt, wurde als Prototyp an der Universität Münster entwickelt [MLK04, LMK04]. Glass-Box-Testen findet auch Fehler, die nur in Sonderfällen auftreten. Allerdings ist der Rechenaufwand hier so groß, dass es sich zwar für das Testen von einzelnen, algorithmisch komplexen Klassen eignet, i. allg. aber nicht für die Überprüfung ganzer Subsysteme oder Systeme. Durch Verwenden beider Ansätze lassen sich die Stärken von Black-Box- und Glass-

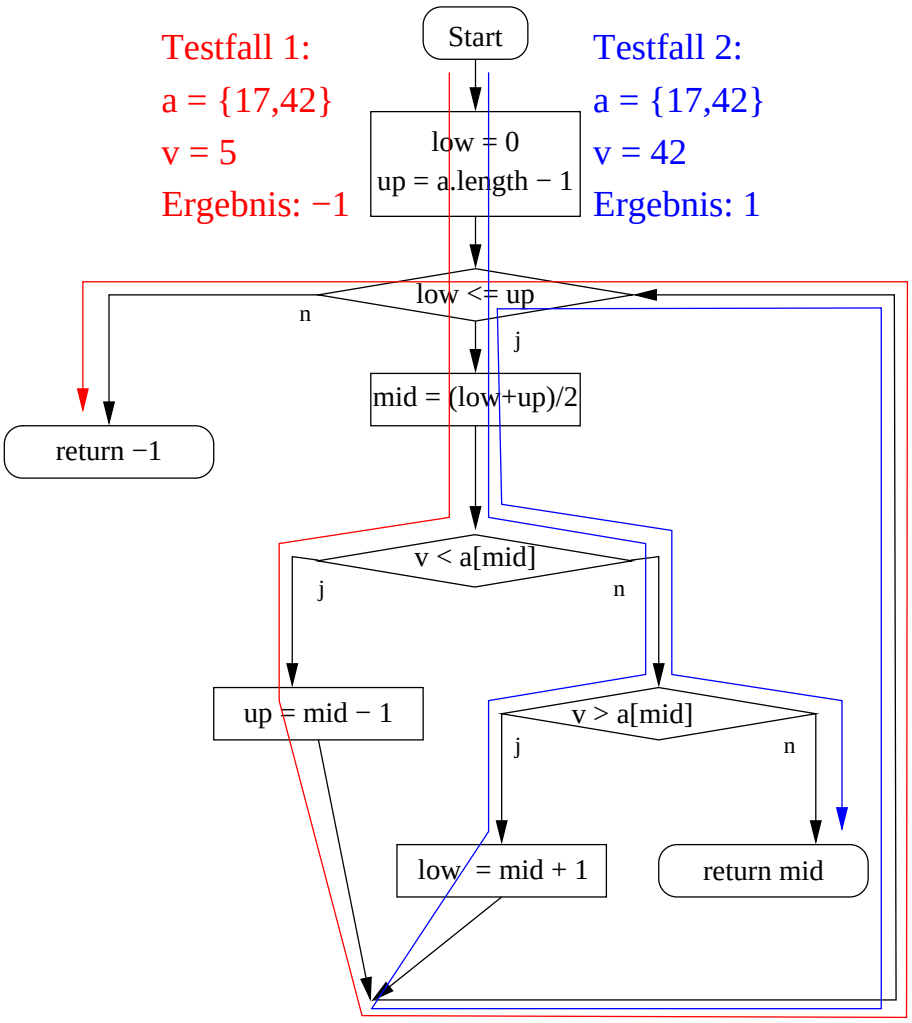


Abbildung 7.3: Kontrollflussüberdeckung für binäres Suchen.

Box-Testen kombinieren.

Kapitel 8

Tools

In diesem Kapitel sollen einige Werkzeuge vorgestellt werde, deren Einsatz bei der Softwareentwicklung zu empfehlen ist oder zumindest überlegt werden sollte.

8.1 Versionsverwaltung

8.1.1 Aufgaben

Die Versionsverwaltung dient in der Softwareentwicklung der Verwaltung unterschiedlicher Versionen von digitalen Dokumenten, wie Quellcode- oder Dokumentationsdateien. Die wesentliche Aufgabe einer Versionsverwaltung besteht darin, die unterschiedlichen Versionen der Dateien eines Projektes zu speichern. Hierfür werden zu jeder Version eine Versionsnummer sowie die Person, welche die neue Version erstellt hat, gespeichert. Dies ermöglicht es, die Entwicklung einzelner Teile des Projektes durch Versionsvergleiche vollständig nachzuvollziehen und Änderungen unter Umständen gezielt rückgängig zu machen.

Neben der historischen Versionsfortschreibung, dient die Versionsverwaltung aber auch als zentrale und gemeinsame Datenbasis für ein Softwareprojekt. Der Einsatz einer Versionsverwaltung ist daher vor allem bei größeren Projekten, an denen mehrere Entwickler gleichzeitig arbeiten, sinnvoll.

Zu den Hauptaufgaben eines Versionsverwaltungssystems zählen:

- **Protokollierung von Änderungen:** Es kann jederzeit nachvollzogen werden, wer wann was geändert hat.
- **Wiederherstellung:** Versehentliche oder fälschliche Änderungen können jederzeit rückgängig gemacht werden.
- **Archivierung:** Die verschiedenen Releases eines Systems können archiviert werden. So kann auf diese Releases jederzeit zurückgegriffen werden.

- **Koordinierung:** Es wird der gleichzeitige Zugriff von mehreren Anwendern auf die Dateien des Projektes ermöglicht.

- **Verwaltung mehrerer Entwicklungszweige:** Es ist möglich, gleichzeitig mehrere Entwicklungszweige (Branches) eines Projektes zu verwalten. Dabei kann bestimmt werden, auf welche Zweige eine Änderung Auswirkungen hat.

8.1.2 Funktionsweise

Ein klassisches Versionsverwaltungssystem besteht in der Regel aus einem Server, der die gemeinsame Datenbasis, das *Repository*, verwaltet und ein oder mehreren Clients, die mithilfe des Servers auf das Repository zugreifen. Die Clients stellen die Anwenderschnittstelle dar, über die die Benutzer auf die Dienste des Versionsverwaltungssystems zugreifen können.

Zu den wichtigsten Diensten einer Versionsverwaltung zählen das *Checkout* und das *Checkin*. Beim Checkout erstellt ein Client eine *lokale Kopie* von einzelnen oder mehreren Dateien aus dem Repository. Die Änderungen an einer lokalen Kopie haben zunächst keinen Einfluss auf das im Repository gespeicherte Original und sind somit auch für die übrigen Clients nicht sichtbar. Sollen die Änderungen einer lokalen Kopie in das Repository übernommen werden, so ist dafür eine besondere Bestätigung notwendig. Diese Bestätigung wird als Checkin oder Commit bezeichnet. Dabei wird für die betroffenen Dateien eine neue Version im Repository erzeugt.

Bei der Koordinierung des Mehrbenutzerbetriebes tritt das Problem auf, gleichzeitige Änderungen an derselben Datei zu behandeln. Hierfür gibt es im Wesentlichen zwei Strategien, die *pessimistische* und die *optimistische Versionskontrolle*.

Der Mechanismus der pessimistischen Versionskontrolle wird als *„Lock Modify Write“* bezeichnet. Beim Checkout werden dabei alle betroffenen Dateien gesperrt, was zur Folge hat, dass dieselbe Datei nur von einem Benutzer ausgecheckt und bearbeitet werden kann. Erst nach dem Checkin wird die Sperre aufgehoben und die Datei für Änderungen durch andere Benutzer wieder freigegeben. Auf diese Weise werden Änderungskonflikte vermieden.

Die optimistische Versionskontrolle benutzt ein Verfahren, welches als *„Copy Modify Merge“* bezeichnet wird. Dieses Verfahren lässt gleichzeitige Änderungen von unterschiedlichen Benutzern an derselben Datei zu. Wird beim Checkin einer Datei festgestellt, dass sich sowohl das Original im Repository als auch die lokale Kopie geändert haben, so müssen die beiden Versionen automatisch oder manuell zusammengeführt werden. Eine manuelle Auflösung der Konflikte ist in der Regel nur notwendig, wenn die Konflikte sich nicht automatisch beheben lassen.

Die optimistische Versionskontrolle besitzt im Vergleich zur pessimistischen den Nachteil, dass sie durch die manuelle Konfliktbeseitigung zusätzlichen Aufwand für den Anwender bedeutet. Dafür wird durch das optimistische Verfahren der

Mehrbenutzerbetrieb deutlich weniger eingeschränkt. Dies ist insbesondere bei einer größeren Anzahl räumlich getrennter Anwender von Vorteil.

Wie in Abbildung 8.1 dargestellt, gibt es grundsätzlich zwei Möglichkeiten, die verschiedenen Versionen einer Datei in einem Repository zu speichern. Die einfachste Möglichkeit besteht darin, von jeder Version einen vollständigen *Schnappschuss* zu speichern. Bei dieser Form der Speicherung treten jedoch Redundanzen auf, die umso größer werden, je kleiner die Änderungen (Deltas) zwischen den Versionen sind. Bei geringfügigen Änderung werden daher zwei nahezu identische Versionen gespeichert. Um ein Repository mit vielen unterschiedlichen Versionen effektiv und platzsparend verwalten zu können, ist es wichtig, Redundanzen möglichst zu vermeiden. Viele Versionsverwaltungssysteme speichern daher üblicherweise lediglich die Unterschiede zwischen den einzelnen Versionen, die *Änderungsmenge*. Um dies zu erreichen, wird in vielen Systemen nur die neueste Version vollständig gespeichert, während alle älteren Versionen als „Rückwärts-Patch“ vorliegen. Ältere Versionen lassen sich dann durch Anwendung der Rückwärts-Patches aus der aktuellsten Version rekonstruieren.

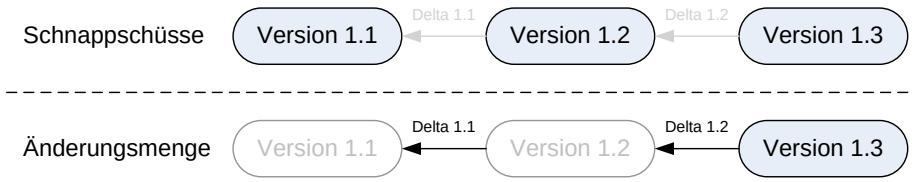


Abbildung 8.1: Speichermodelle für die Versionskontrolle. Verschiedene Versionen einer Datei können entweder vollständig in Form von *Schnappschüssen* oder als *Änderungsmenge*, die ausgehend von einer vollständigen Version nur die Versionsunterschiede protokolliert, gespeichert werden.

Es existiert eine Vielzahl von kommerziellen und nicht kommerziellen Versionsverwaltungssystemen, die aufgrund ihrer individuellen Eigenschaften für unterschiedliche Anforderungen geeignet sind. Eine Gegenüberstellung der Eigenschaften bekannter Systeme findet sich z.B. bei [Wik07] und [Bet07]. In einer von Forrester durchgeführten Studie [Sch07], die 11 führende Versionsverwaltungssysteme vergleicht, werden die kostenlos erhältliche Open-Source-Lösung *Subversion* [Col07] und das von IBM kommerziell vertriebene *ClearCase* [IBM07a] am besten bewertet werden.

8.2 CASE-Tools

8.2.1 Funktionsweise

Computer Aided Software Engineering (CASE) bezeichnet den Einsatz von Softwaretools zur Unterstützung der Entwicklung und Wartung von Software. Die Werkzeuge, die hierfür zum Einsatz kommen, werden als *CASE-Tools* bezeichnet. Die Unterstützung durch ein CASE-Tool kann sich auf verschiedenste Aspekte des Softwareentwicklungsprozesses beziehen. Ziel des Einsatzes von CASE-Werkzeugen ist die qualitative und quantitative Verbesserung der Softwareentwicklung. Zu den CASE-Werkzeugen zählen daher nicht nur die unabdingbaren Hilfsmittel wie Compiler, Editor, Linker und Debugger.

Der Einsatzbereich für CASE-Tools umfasst sämtliche Phasen des Entwicklungsprozesses, wie Planung, Definition, Entwurf, Implementierung sowie Einführung und Wartung. Wie in Abbildung 8.2 dargestellt, lassen sich CASE-Werkzeuge in Abhängigkeit von ihren Schwerpunkten zu Kategorien zusammenfassen. Diejenigen Werkzeuge, die die ersten Phasen der Softwareentwicklung (Planung, Definition und Entwurf) unterstützen, werden auch als *Front-End-* oder *Upper-CASE-Werkzeuge* bezeichnet. Zu den *Back-End-* oder *Lower-CASE-Werkzeugen* gehören alle Werkzeuge, die sich den späteren Phasen der Softwareentwicklung (Implementierung, Entwurf, Einführung und Wartung) zuordnen lassen. Werkzeuge, die sowohl die früheren als auch die späteren Phasen der Entwicklung abdecken, werden als I(integrated)-CASE-Werkzeuge bezeichnet. Werden mehrere CASE-Werkzeuge in einer Plattform integriert, so spricht man von einer CASE-Umgebung.

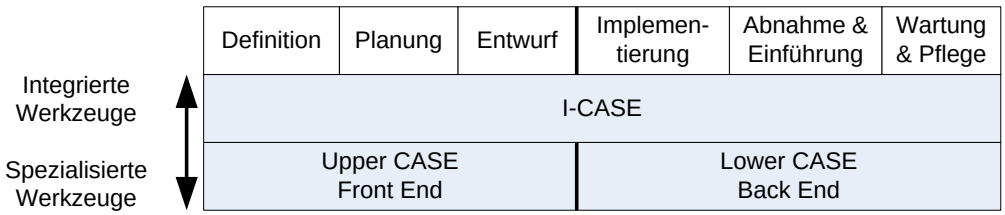


Abbildung 8.2: Kategorisierung von CASE-Werkzeugen [Bal93, S. 126].

8.2.2 Auswahl von CASE-Plattformen

Nach [Bal98, S. 591ff.] spielen bei der Auswahl von CASE-Umgebungen drei Faktoren eine entscheidende Rolle:

- allgemeine Anforderungen,
- firmenspezifische Anforderungen und

- derzeitiges Marktangebot.

Die *allgemeinen Anforderungen* leiten sich aus dem Stand der Technik ab und sind unabhängig von den firmenspezifischen Anforderungen. Zu ihnen zählen im wesentlichen Anforderungen an die CASE-Plattform, die einzelnen CASE-Werkzeuge und die Entwicklungsumgebung. Wichtige Anforderungen sind (vgl. [Bal98]):

- **Integrationsfähigkeit:** Für eine CASE-Plattform ist die Integrationsfähigkeit von CASE-Werkzeugen von entscheidender Bedeutung. Hierfür ist häufig auch eine sinnvolle Integration der Datenbestände der einzelnenen Werkzeuge in ein gemeinsames Repository notwendig, um die Interaktionsfähigkeit der Werkzeuge zu garantieren.

- **Offenheit:** Die Plattform sollte möglichst über Export- und Importschnittstellen verfügen, und auch Drittanbietern muss es möglich sein, Werkzeuge in die Plattform zu integrieren.

- **Multi- und Interprojektfähigkeit:** Die CASE-Plattform sollte in der Lage sein, mehrere Projekte mit verschiedenen Mitarbeitern parallel verwalten zu können. Zudem sind projektübergreifende Informationen separat zu verwalten, um Redundanzen und Inkonsistenzen zu vermeiden.

Die *firmenspezifischen Anforderungen* leiten sich aus der individuellen Situation und den Bedürfnissen der betroffenen Firma ab. Hier nennt Balzert [Bal93, S. 144 ff.] folgende Kategorien, die sich gegenseitig beeinflussen: die *Eigenschaften der Zielprodukte*, die zu unterstützende *Zielumgebung*, die zu unterstützenden *Methoden* und *Vorgehensmodelle* sowie das *Entwicklungsumfeld*.

Nach Abgleich der allgemeinen und firmenspezifischen Anforderungen ergibt sich ein Kriterienkatalog, der die ideale, auf die Anforderungen der Firma zugeschnittene CASE-Umgebung beschreibt. Die ideale Firmenumgebung sollte mit den tatsächlich am Markt vorhandenen Lösungen verglichen werden, um schließlich die bestmögliche Lösung auszuwählen. Bei der Begutachtung der Marktangebote gilt es vor allem zwei Punkte zu berücksichtigen, die technischen und die kaufmännischen Aspekte der angebotenen Umgebungen.

Um einen Überblick über die Unterstützungsmöglichkeiten durch CASE-Tools zu erlangen, ist es daher sinnvoll, sich die Aufgaben innerhalb der einzelnen Phasen genauer anzuschauen. Ein Verzeichnis von CASE-Werkzeugen findet sich bei [Lam07]. Für einen modellgetriebenen Softwareentwicklungsprozess gibt es u. a. kommerziell erhältliche Plattformen, wie die *Rational Suite* [IBM07b] von IBM oder *Borland Together* [Bor07] von Borland. Es existieren aber auch kostenfreie Lösungen, wie StarUML [Sta07] oder EclipseUML [Omo07].

8.2.3 Test-Werkzeuge

Insbesondere was den Werkzeugeinsatz beim Testen angeht, gibt es bei vielen Firmen noch Nachholbedarf. Hier lässt sich sowohl die Durchführung von Tests als auch die Erstellung von Testfällen zumindest teilweise automatisieren. Das Spektrum erstreckt sich bezüglich der automatisierten Testdurchführung von frei verfügbaren Tools für das Unit-Testing wie JUnit [Lin05, JUn08] für Java (und Varianten hiervon für andere Sprachen, z.B. NUnit für C#) bis hin zu kommerziellen Werkzeugen wie dem IBM Rational TestManager [Rat08b] und HP Functional Testing [HP08a], die neben funktionalen Tests auch das Testen von Benutzerschnittstellen durch Aufzeichnen und Abspielen von Ereignisfolgen unterstützen. Varianten von JUnit wie HttpUnit [Fej08] und Cactus [Pro08] unterstützen ebenso wie z.B. Selenium [ope08] und Watij [Wat08] das Testen von Webapplikationen.

Schließlich gibt es Werkzeuge für das Performance-Testing, die festzustellen gestatten, ob Leistungsanforderungen vom betrachteten System eingehalten werden. Als Beispiel seien hier der HP LoadRunner [HP08b] und IBM Rational Performance Tester [Rat08a] erwähnt.

Das automatische Generieren von Testfällen kann z.B. von QuickCheck [Sou08] übernommen werden. Hier werden per Zufallsgenerator zufällige Testeingaben erzeugt, für die dann automatisch geprüft wird, ob sie das gewünschte Ergebnis berechnen. Dies funktioniert vielfach erstaunlich gut, und ein Großteil der Fehler wird durch die so erzeugten Testfälle aufgedeckt. Vorausgesetzt wird hierbei, dass eine spezielle Methode, eine so genannte Property vom Tester bereitgestellt wird, die prüft, ob ein berechnetes Ergebnis stimmt. Das Erstellen solcher Properties kann manchmal aufwändig werden. Auch werden manchmal spezielle Generatoren für sinnvolle Eingaben benötigt. Dies ist insbesondere dann der Fall, wenn ein Großteil der aufgrund der Datentypen der Eingabeparameter möglichen Eingaben unzulässig ist und das Verhalten des Systems für diese illegalen Eingaben nicht interessiert.

Schließlich gibt es Werkzeuge, die feststellen, welcher Anteil des betrachteten Codes durch die bisher untersuchten Testfälle überdeckt wird. So lässt sich Feststellen, ob Teile des Codes noch nicht getestet wurden und daher weitere Testfälle erforderlich sind.

8.2.4 Werkzeuge zur Architekturüberwachung

Weiterhin gibt es (kommerzielle) Werkzeuge wie z.B. Sotograph [Tom08], die insbesondere bei lange eingesetzten und häufig gewarteten Softwaresystemen sicherstellen können, dass die ursprüngliche Architektur im Laufe der Zeit nicht immer mehr untergraben und die Verständlichkeit und Wartbarkeit somit verschlechtert wird, sondern dass die urspünglichen *Architekturrichtlinien* eingehalten werden, z.B. bezüglich der Schichten. Solche Tools können auch eine Vielzahl von *Kennzahlen* zu einem Softwareprodukt ermitteln, die u.a. Aufschluss über dessen Zustand geben.

Kapitel 9

Enterprise Application Integration

Die Anwendungsintegration oder Enterprise Application Integration (EAI) [CHK06, Lin00] gehört zu den Themengebieten der Informatik, die in Wissenschaft und Praxis vielfach diskutiert werden. Trotzdem wird das Konzept der EAI häufig verkürzt oder sogar grundsätzlich falsch verstanden. Neben einer uneinheitlichen Definition der relevanten Begriffe trägt vor allem die Fülle der im Kontext der EAI verfügbaren Produkte und Technologien zu der Verwirrung bei. Nicht selten wird EAI mit dem Einsatz bestimmter Technologiekonzepte (z.B. Middleware-Plattformen) oder Integrationsarchitekturen (z.B. Service Orientierte Architekturen) gleichgesetzt. Das Spektrum der verfügbaren Integrationstechnologien ist aufgrund der sich wandelnden Anforderungen an die Anwendungsintegration fortwährenden Veränderungen unterworfen. Die Dominanz aktueller Technologietrends am Markt für Integrationsprodukte und in den wissenschaftlichen Publikationen begünstigt eine einseitige und technologiespezifische Diskussion von Integrationsproblemen. Die Frage, inwieweit die favorisierte Integrationslösung dabei wirklich den gestellten Anforderungen genügt und ob ihr Einsatz möglicherweise mit Nachteilen oder Risiken verbunden ist, gerät dabei allzu leicht ins Hintertreffen. Als Beispiel hierfür kann die derzeitige Euphorie um Web-Service basierte Integrationslösungen angesehen werden.

Statt um eine konkrete Technologie handelt es sich bei EAI vielmehr um ein Konzept, welches mit einer Vielzahl möglicher Technologien realisiert werden kann. Das Ziel der Anwendungsintegration besteht in der umfassenden Verknüpfung von Anwendungen auf innerbetrieblicher oder zwischenbetrieblicher Ebene. Dabei sollen die bestehenden Anwendungen in ihrer ursprünglichen Form erhalten bleiben, um einerseits die in ihre Erstellung investierten Ressourcen und andererseits das darin gebundene Wissen für die Zukunft zu erhalten. Zur Umsetzung dieses Ziels stellt das EAI-Konzept eine Reihe von Methoden, Architekturkonzepten sowie Standards und Technologien zur Verfügung.

9.1 Heterogene Systemlandschaften

Der Ausgangspunkt von Integrationsvorhaben ist stets eine Menge von bereits existierenden Systemen, die aufgrund von Heterogenitäten nicht oder nur mit Einschränkungen in der Lage sind, miteinander zu interagieren. Diese Heterogenitäten können sich auf verschiedenen Ebenen der Interaktion zwischen zwei Informationssystemen vollziehen. Bestehen zwischen diesen wesentliche Unterschiede in der technischen Repräsentation, Speicherung und Bereitstellung der verwalteten Informationen, so spricht man von einer *technischen Heterogenität*. Diese kann zum Beispiel durch zueinander inkompatible Hardwareplattformen, Datenbankmanagementsysteme, Kommunikationsmechanismen oder Programmiersprachen entstehen. Für eine einfache Überwindung derartiger Integrationsprobleme ist am Markt für Integrationsprodukte eine Vielzahl von Werkzeugen verfügbar. Wesentlich schwieriger zu handhaben sind *syntaktische Heterogenitäten*. Diese betreffen die Datenmodelle, welche der Speicherung von Informationen in einer Anwendung zugrunde liegen. Unterscheiden sich die Datenmodelle zweier Informationssysteme deutlich voneinander, so ist ein Informationsaustausch nur dann möglich, wenn die zu übertragenden Informationen vom Datenmodell des Quellsystems auf das Datenmodell des Zielsystems oder auf ein zentralisiertes Datenmodell abgebildet werden kann. Außerdem geht die Bedeutung eines Informationsobjektes manchmal nicht eindeutig aus dem Datenmodell hervor. Als Beispiel lässt sich ein Attribut „Preis“ nennen. Ohne weitere Anreicherung dieses Attributes mit zusätzlicher Information ist zum Beispiel nicht ersichtlich, ob es sich um einen Bruttopreis oder einen Nettopreis handelt. Dieser Bedeutungsinhalt ist jedoch für die korrekte Verwendung eines Informationsobjektes relevant. Wenngleich im Kontext eines bestimmten Informationssystems implizit gelten kann, dass es sich bei einem Attribut mit dem Namen Preis stets um einen Bruttopreis handelt, so kann das Vorhandensein dieser impliziten Bedeutung (Semantik) in einem anderen Informationssystem nicht vorausgesetzt werden. Eine Interaktion zwischen zwei Informationssystemen wird demnach nur dann reibungslos funktionieren, wenn beide für die auszutauschenden Informationsobjekte die gleiche implizite Semantik verwenden. Ist dies nicht gegeben, so spricht man von einer *semantischen Heterogenität*. Semantische Heterogenitäten lassen sich umgehen, indem die auszutauschenden Informationsobjekte mit zusätzlicher Semantik (beispielsweise einer Unterscheidung zwischen Bruttopreis und Nettopreis) angereichert oder auf ein vereinheitlichtes semantisches Referenzmodell abgebildet werden.

9.2 Motivationen für Integrationsmaßnahmen

Vorhandene Heterogenitäten sind nicht in jedem Fall kritisch, wachsen sich aber spätestens dann zu einem Problem aus, wenn die Systeme aufgrund betrieblicher Erfordernisse miteinander interagieren bzw. Informationen austauschen müssen. Dies

ist zum Beispiel dann der Fall, wenn zwei Systeme Aktivitäten in einem innerbetrieblichen oder zwischenbetrieblichen Geschäftsprozess unterstützen, welche in einer Kunde-Lieferant-Beziehung stehen. Um Informationen entsprechend der Geschäftsprozesslogik über die Systemgrenze hinweg von einer in die andere Aktivität zu übertragen, wird eine technische Schnittstelle benötigt, die eine Interaktion zwischen den jeweiligen Systemen ermöglicht. Ist dies nicht oder nur unzureichend gewährleistet, so kommt es innerhalb des Prozesses unweigerlich zu einem Medienbruch. Die fehlende oder mangelhafte Integration der Systeme zwingt zu einer Konvertierung der Informationen auf ein alternatives Medium, welches in beiden Aktivitäten verarbeitet werden kann. Der Austausch von Informationen per Email oder auf gedrucktem Papier sind nur zwei Beispiele für eine derartige Situation. Die Konvertierung und anschließende Rückkonvertierung der Informationen (z.B. durch Erzeugung einer Excel-Tabelle im Quellsystem, welche auf elektronischem Wege an das Zielsystem übertragen und dort in die Datenbank eingelesen wird) kostet Zeit und Geld. Im Rahmen der Konvertierung können darüber hinaus Informationen verloren gehen oder Fehler auftreten. Medienbrüche schränken demnach nicht nur die Effizienz, sondern auch die Effektivität von Geschäftsprozessen ein. Den Anstoß für konkrete Integrationsprojekte können vielerlei Ursachen geben. Ein Beispiel ist die Optimierung oder Reorganisationen von Geschäftsprozessen. Bei einer an aufbauorganisatorischen Gesichtspunkten wie Abteilungen, Stellen, Kompetenzen und Verantwortlichkeiten orientierten Betrachtung der Unternehmensorganisation geht der Blick für Schnittstellenprobleme zwischen den Informationssystemen der einzelnen Organisationseinheiten schnell verloren. Bei einer prozessorientierten Betrachtung werden die Anforderungen an die Integration der betrieblichen Informationssysteminfrastruktur unmittelbar sichtbar. Abweichungen zwischen der Ist-Situation und dem gewünschten Soll-Zustand lassen sich somit leicht feststellen. Auch die Verschmelzung zweier Unternehmen mit jeweils eigener Informationssysteminfrastruktur im Rahmen von Unternehmenszusammenschlüssen oder Übernahmen, kann Integrationsmaßnahmen erforderlich machen.

In den folgenden Abschnitten sollen die wesentlichen Konzepte und Technologien der EAI vorgestellt werden. Ausgehend von einer Betrachtung verschiedener Topologien für den Aufbau einer Integrationsinfrastruktur werden mit der Datenebene, der Funktionsebene und der Ebene der Benutzerschnittstelle drei Ansatzpunkte für die Durchführung einer Anwendungsintegration vorgestellt. Nachdem kurz auf die Eigenschaften sowie die Vor– und Nachteile der einzelnen Integrationsansätze eingegangen wurde, werden konkrete Technologien zur Umsetzung einer entsprechenden Integration vorgestellt und gegeneinander abgewägt.

9.3 EAI-Architekturkonzepte

Die Auswahl einer geeigneten Architektur für die Gestaltung einer Integrationsinfrastruktur ist von tragender Bedeutung für deren Eigenschaften und zukünftige Entwicklungspotentiale. Aufgrund ihres grundlegenden Charakters und der mit ihrer Einführung verbundenen hohen Kapitalbindung ist sie von längerfristiger Wirkung und kann nachträglich nur mit hohem Aufwand korrigiert oder zurückgenommen werden. Der auf kurzfristige Sicht attraktivste Ansatz kann sich längerfristig als Sackgasse für die Fortentwicklung der betrieblichen Informationsverarbeitung erweisen. Eine Entscheidung für einen bestimmten Architekturansatz sollte demnach wohl überlegt sein und genügend Flexibilität bieten, um nicht nur gegenwärtigen sondern auch zukünftigen Anforderungen begegnen zu können. Im den folgenden Abschnitten werden drei grundlegende Ansätze für die Gestaltung einer Integrationsinfrastruktur vorgestellt. Um Handlungsempfehlungen für die Praxis ableiten zu können, sollen dabei vor allem deren spezifische Vor– und Nachteile sowie mögliche Restriktionen herausgestellt werden.

9.3.1 Punkt-zu-Punkt-Integration

Die Punkt-zu-Punkt-Integration (vgl. Abbildung 9.1) stellt weniger eine Integrationstopologie dar, als vielmehr einen Hinweis auf das Fehlen einer solchen.

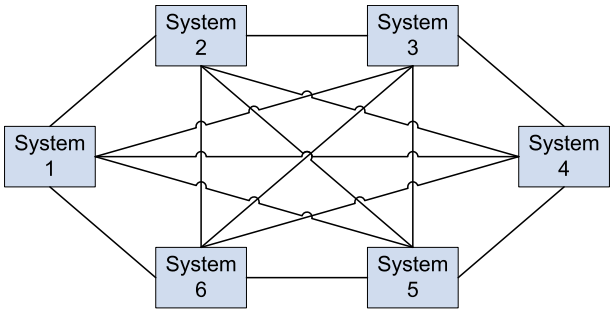


Abbildung 9.1: Vollständige Punkt-zu-Punkt-Integration von sechs Systemen.

Bei der Punkt-zu-Punkt-Integration werden jeweils zwei Systeme bedarfsgetrieben über eine individuelle Schnittstelle miteinander gekoppelt. Der Vorteil dieses Ansatzes liegt bei kleinen Systemen in seiner schnellen und kostengünstigen Realisierbarkeit. Sobald jedoch mehrere Systeme miteinander verbunden werden müssen, zeigen sich die Grenzen dieses Integrationsansatzes. Um in einer Infrastruktur mit n Systemen ein weiteres System mit allen bestehenden Anwendung zu integrieren, müssen n Schnittstellen geschaffen werden. In einer Infrastruktur mit n Systemen müssen demnach maximal n*(n–1)/2 Schnittstellen gewartet werden. Diese „Schnittstellenexplosion“ führt trotz der Vorteile dieses Ansatzes in überschaubaren Inte-

grationsszenarien bei einer wachsenden Zahl der zu integrierenden Systeme zu einer unflexiblen und schwierig zu wartenden Schnittstellenlandschaft.

9.3.2 Hub-and-Spokes-Integration

Bei der Hub-and-Spokes-Integration (vgl. Abbildung 9.2) stellt eine zentrale Integrationskomponente alle grundlegenden Infrastrukturdienste für die Anwendungsintegration zur Verfügung. Die einzelnen Systeme werden nicht direkt, sondern über den Hub miteinander integriert.

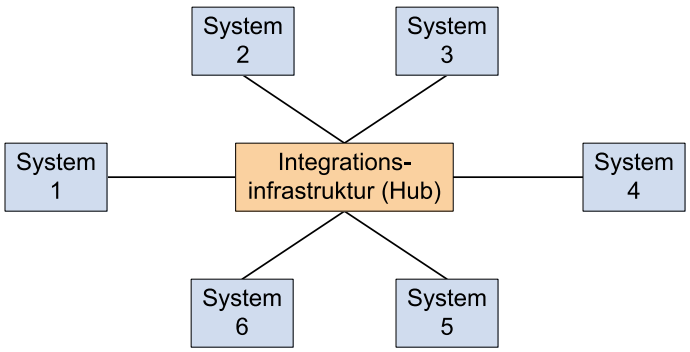


Abbildung 9.2: Hub-and-Spokes-Integration.

Um ein neues System in eine Infrastruktur mit n Systemen einzubinden, muss nur eine standardisierte Schnittstelle zum Hub geschaffen werden. Verglichen mit der Punkt-zu-Punkt-Integration entsteht bei der Verwendung einer Hub-and-Spokes-Topologie ein hoher initialer Aufwand für die Einrichtung des Hubs, während die Anbindung von Systemen an den Hub mit vergleichsweise geringen Kosten verbunden ist. Ein Nachteil dieser Topologie entsteht dadurch, dass jegliche Kommunikation zwischen den Systemen über den Hub läuft. Dessen Leistungsfähigkeit könnte sich zum Performanz-Engpass für die gesamte Integrationsinfrastruktur entwickeln. Bei einem Ausfall des Hubs wäre gar jegliche Kommunikation zwischen den beteiligten Systemen unterbrochen. Die geschilderten Probleme lassen sich durch den Aufbau von Rechnerverbünden (Load Balancing Cluster, Hochverfügbarkeitscluster) und redundante Bereitstellung der Hub-Funktionalitäten umgehen.

9.3.3 Bus-Integration

Ausgehend von der Kritik an der Hub-and-Spokes-Topologie macht sich die Bus-Integration (vgl. Abbildung 9.3) einen verteilten Ansatz zu nutzen, der die geschilderten Performanz– und Verfügbarkeitsprobleme vermeidet. Bei der Bus-Integration werden die Integrationsfunktionalitäten verteilt durch die angebundenen Einheiten bereitgestellt. Aufgrund des Fehlens einer zentralen Integrationskomponente ist eine

flexible Anpassung an Performanz– und Verfügbarkeitsanforderungen möglich. Wie auch bei der Hub-and-Spokes-Topologie ist zur Integration eines Systems in eine Infrastruktur mit n bestehenden Systemen lediglich eine Schnittstelle erforderlich.

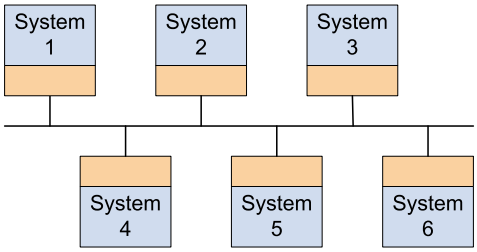


Abbildung 9.3: Bus-Integration.

Problematisch an diesem Ansatz ist, dass die Integrationsfunktionalitäten nicht einmalig an einer zentralen Stelle, sondern mehrfach implementiert sind, wodurch sich ein erhöhter Wartungs– und Anpassungsaufwand ergibt. Darüber hinaus bewirkt die Koordination und Verwaltung der dezentral agierenden Einheiten einen Overhead, der zu Performanzeinbußen und erhöhtem Netzwerkverkehr führen kann.

9.4 Integrationsebenen

Maßnahmen zur Anwendungsintegration sind auf drei Ebenen eines Informationssystems möglich, der *Datenebene*, der *Funktionsebene* und der Ebene der *Benutzerschnittstelle*. Eine Integration auf diesen Ebenen ist mit spezifischen Vor– und Nachteilen verbunden. Darüber hinaus stellt jede Integrationsebene gewisse Anforderungen an die zu integrierenden Systeme, weshalb eine uneingeschränkte Wahlfreiheit in vielen Fällen nicht gegeben ist. Verfügt eine Anwendung beispielsweise über keine Schnittstelle, die einen Zugriff auf die Programmlogik erlaubt und ist darüber hinaus der Quellcode nicht verfügbar bzw. eine Anpassung des Systems nicht möglich, so scheidet eine Integration auf Funktionsebene von vornherein als praktikable Lösung aus. Bei der Entscheidung für eine Integrationsebene muss demnach ein Kompromiss zwischen den Anforderungen an die Integration und den technischen Möglichkeiten für deren Realisierung gefunden werden.

9.4.1 Integration auf Datenebene

Bei der Datenintegration erfolgt die Integration durch unmittelbaren Zugriff auf die Daten, die von den jeweiligen Anwendungen erzeugt, verwaltet und gespeichert werden (vgl. Abbildung 9.4).

Ein wesentlicher Vorteil dieser Methode liegt darin, dass keine Modifikationen der Datenstrukturen oder der Anwendungslogik der zu integrierenden Anwendun-

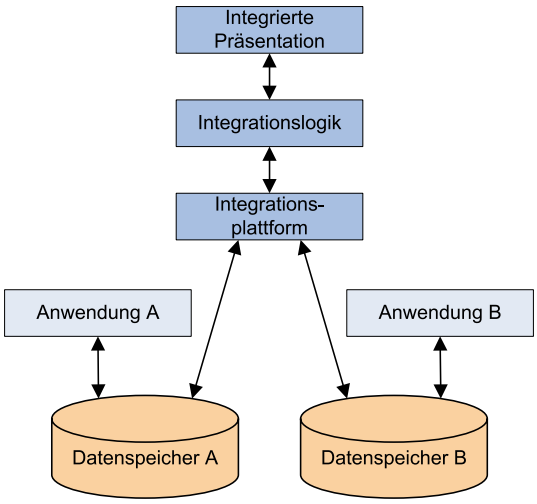


Abbildung 9.4: Integration auf Datenebene.

gen erforderlich ist. Dies ermöglicht vielmals eine schnelle Lösung von Integrationsproblemen und erlaubt eine Integration auch dann, wenn eine Anpassung der Systeme nicht möglich ist, da der Quellcode nicht verfügbar gemacht werden kann. Für die Überwindung der technischen Heterogenität zwischen verschiedenen Datenbankmanagementsystemen und syntaktischer Heterogenitäten als Folge von unterschiedlichen Konzepten der Datenmodellierung (z.B. Entity Relationship Modellierung, Objektorientierte Modellierung), bietet sich eine Vielzahl von Werkzeugen an. Dennoch ist eine Datenintegration keine triviale Angelegenheit. Ursache sind die syntaktischen und semantischen Heterogenitäten, welche sich aus den spezifischen Anforderungen der einzelnen Anwendungen ergeben. Unterschiede auf Daten– und Schemaebene, ergeben sich durch die Nutzung verschiedener Attribute eines Datenobjektes, unterschiedliche Datentypen, Bezeichnungen, Maßeinheiten oder Wertebereiche sowie die verschiedenartige Anwendung von Abstraktionsmechanismen wie der Vererbung. Um die Datenbestände verschiedener Anwendungen im Zuge einer Integration zusammenzuführen, müssen die Heterogenitäten durch Transformation der Daten überwunden werden. Eine Transformation kann dabei ausgehend vom Datenmodell einer Datenquelle auf das Datenmodell eines Zielsystems oder auf ein verallgemeinertes föderiertes Datenmodell stattfinden.

Für die Umsetzung einer Datenintegration bieten sich verschiedene Technologien, wie JDBC (Java Database Connectivity) [ME00], ODBC (Open Database Connectivity) [BJ97], Datentransformationsdienste, OLAP (Online Analytical Processing) [Oeh00] und Data-Warehousing [BG01] an. Eine direkte Vergleichbarkeit der Technologien ist nicht gegeben, da der Funktionsumfang sehr stark variiert. Während sich die Funktionalität von JDBC und ODBC darauf beschränkt, eine definierte und universell verwendbare Schnittstelle für den Zugriff auf verschiedene Datenbanksysteme

bereitzustellen, bieten Datentransformationsdienste und Data-Warehouse-Systeme darüber hinaus einfach nutzbare Funktionalitäten für die Umwandlung und Zusammenführung von Daten.

Der Hauptvorteil einer Integration auf Datenebene besteht darin, dass sie zur Integration einer Anwendung auch dann herangezogen werden kann, wenn diese über keinerlei offen gelegte Schnittstellen verfügt und eine Modifikation des Quellcodes nicht möglich ist. Darüber hinaus bietet ein breit gefächertes Angebot an ausgereiften Werkzeugen dem Entwickler ein hohes Maß an Unterstützung. Nachteilig ist jedoch, dass eine Datenintegration keinen Zugriff auf die Funktionalität eines Systems ermöglicht. Daher ist dieser Integrationsansatz nur dann zu empfehlen, wenn ausschließlich ein Zugriff auf die Daten einer Anwendung erforderlich ist. Soll hingegen die Anwendungslogik eines Systems genutzt werden, so ist eine Integration auf Funktionsebene der Datenintegration vorzuziehen.

9.4.2 Integration auf Funktionsebene

Bei der Funktionsintegration wird die Programmfunktionalität systemübergreifend genutzt, d.h. die integrierten Systeme können gegenseitig auf ihre Anwendungslogik zugreifen (vgl. Abbbilung 9.5).

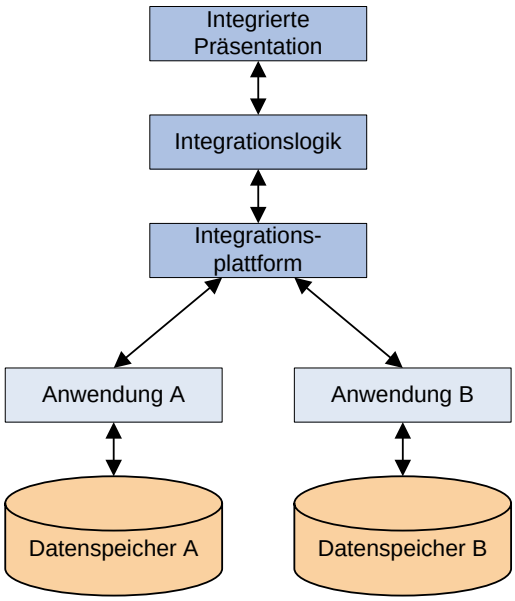


Abbildung 9.5: Integration auf Funktionsebene.

Sie stellt das weitestgehende Integrationskonzept dar und umfasst die Möglichkeiten der Benutzerschnittstellenintegration, wie auch der Datenintegration. Die integrierte Informationssysteminfrastruktur kann als System von mehr oder weniger lose miteinander gekoppelten Komponenten betrachtet werden. Für die Rea-

lisierung dieses Integrationskonzeptes bietet sich ein ganzes Bündel von Technologien an. Diese unterscheiden sich zum Teil erheblich in Aspekten wie Einrichtungsaufwand, Performanz, Skalierbarkeit, Austauschbarkeit und der Flexibilität der durch sie bewirkten Kopplung zwischen den Systemen. Darüber hinaus stellen die verschiedenen Integrationstechnologien sehr unterschiedliche Anforderungen an die zu integrierenden Systeme oder sind mit Restriktionen verbunden, die ihre Einsatzmöglichkeiten auf bestimmte Integrationsszenarien einschränken. Historisch ist eine Entwicklung ausgehend von RPC (Remote Procedure Call) basierten Integrationstechnologien über komponentenorientierte Middleware-Plattformen wie CORBA oder DCOM bis hin zu Web-Service-basierten Integrationslösungen zu beobachten. Eine der wichtigsten Triebfedern für die zu beobachtende technologische Entwicklung im Bereich der Anwendungsintegration ist der Trend hin zu flexiblen, unternehmensübergreifenden und kurzfristig an neue Anforderungen und Rahmenbedingungen anpassbaren Geschäftsprozessen. Dies erfordert eine flexible und skalierbare Integration der zugrunde liegenden Informationssysteme.

Aufgrund der großen Bedeutung der Integration auf Funktionsebene im Kontext der EAI, soll eine Auswahl relevanter Technologien vorgestellt werden. Da zahlreiche technologische Entwicklungen auf vorangegangene Technologien aufbauen, indem sie diese weiterentwickeln oder ausgehend von deren Problemen einen vollständig neuen Lösungsansatz versuchen, werden nicht nur aktuelle Technologietrends sondern auch bestehende Technologien berücksichtigt. Als Ordnungsrahmen dient dabei eine Unterscheidung nach komponentenorientierten, objektorientierten und dienstorientierten Integrationstechnologien (vgl. Abbildung 9.6).

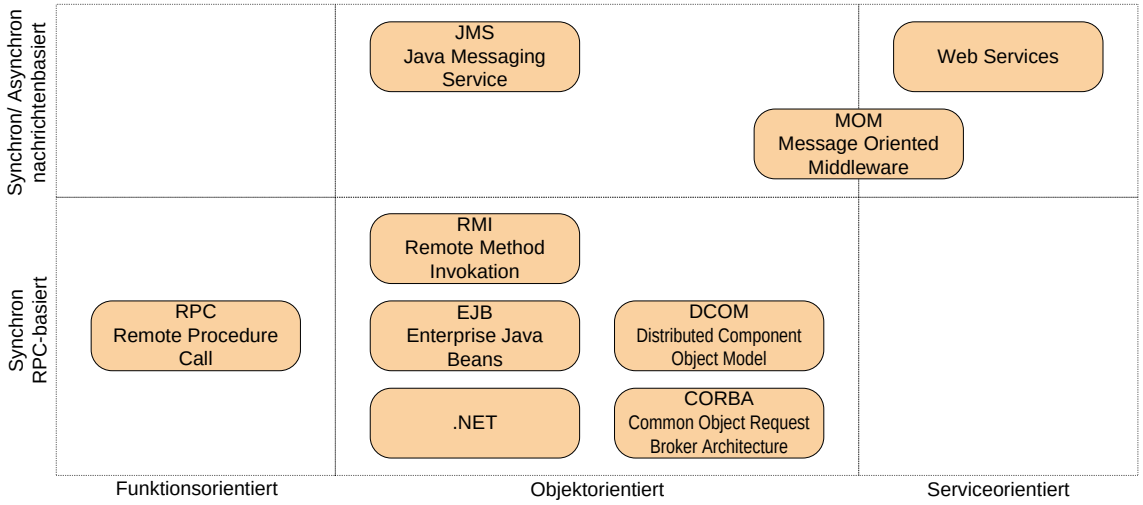


Abbildung 9.6: Übersicht Integrationstechnologien.

Remote Procedure Call

Verfügt eine Anwendung über eine API (Application Programming Interface) so können ihre Funktionalitäten durch andere Systeme auf dem gleichen Rechner über diese Schnittstelle direkt angesprochen und genutzt werden. Der Quellcode der Anwendung selbst wird dabei in der Regel nicht angepasst. Der Hauptvorteil einer derartigen Integration liegt in der Möglichkeit einer Nutzung der Typüberprüfung des Compilers zur Übersetzungszeit. Fehler können so erheblich leichter gefunden und korrigiert werden, als in einem Umfeld, wo die Interaktion der Systeme durch die Verwendung von Nachrichten voneinander entkoppelt ist. Die Einsatzpotentiale dieses Ansatzes sind jedoch auf Integrationsszenarien beschränkt, in welchen die zu integrierenden Anwendungen nicht auf unterschiedliche Rechner verteilt sind.

In einem verteilten Umfeld sind Mechanismen erforderlich, die es ermöglichen, Methoden einer entfernten Anwendung aufzurufen. Für diese Aufgabe bieten sich Dienste wie Remote Procedure Call (RPC) oder der aus dem Java Umfeld bekannte Mechanismus der Remote Method Invocation (RMI) an. Der RPC-Mechanismus führt eine zusätzliche Ebene der Abstraktion zwischen dem Klienten und dem Server ein, der die Implementierungen der beiden Systeme voneinander entkoppelt und die erforderliche Netzwerkkommunikation vermittelt, ohne dass diese durch den Entwickler explizit implementiert werden muss. Die Schnittstelle einer Funktion oder Methode wird mithilfe einer Schnittstellenbeschreibungssprache (Interface Definition Language, IDL) spezifiziert. Auf der Grundlage dieser abstrakten Beschreibung generiert ein IDL-Compiler einen Client Stub und einen Server Stub. Der Client Stub fungiert als lokaler Stellvertreter der durch den Server bereitgestellten Methode. Für den Klienten gestaltet sich der Aufruf der entfernten Methode als lokaler Prozeduraufruf. Die technischen Einzelheiten der Kommunikation wie der Aufbau einer Kommunikationsverbindung über den TCP-IP-Protokollstack oder die Erzeugung von Nachrichten, werden durch den Client Stub bzw. in umgekehrter Richtung durch den Server Stub realisiert. Letzterer reicht den Aufruf serverseitig als lokalen Aufruf an die adressierte Methode weiter und veranlasst die Übermittlung der Rückgabewerte an den entfernten Client. Durch diesen Mechanismus ist die Benutzung eines Kommunikationskanals zum Aufruf einer entfernten Servermethode für den Entwickler der Client-Anwendung transparent. Problematisch an RPC-basierten Integrationsansätzen ist die enge Kopplung an implementierungsspezifische Details der zu integrierenden Anwendung. Wird Letztere erweitert oder abgeändert, so zieht dies im Regelfall Anpassungsentwicklungen am Quellcode der integrierenden Anwendung nach sich. Da die Kommunikation über RPC und RMI Anfragen synchron abläuft, ist die aufrufende Anwendung so lange blockiert (blockierendes Warten), bis vom Empfänger eine Antwort eintrifft. Um eine reibungslose Kommunikation sicherzustellen, muss dafür Sorge getragen werden, dass zwischen den kommunizierenden Anwendungen stets eine stabile Netzwerkverbindung besteht und der Server permanent verfügbar ist. Alternativ besteht die Möglichkeit, einen asynchronen

Kommunikationsmechanismus mittels synchroner Kommunikation und einem Puffer nachzubilden. Wenngleich die Problematik des blockierenden Wartens auf diese Weise vermieden werden kann, so ist dies doch mit zusätzlichem Aufwand verbunden.

Komponentenorientierte Middleware

Einen stärker objektorientierten Integrationsansatz vertreten komponentenorientierte Middleware-Plattformen wie die Common Object Request Broker Architekture (CORBA) [LP98, Hen06], das von Microsoft entwickelte Distributed Component Object Model (DCOM) [RB98] oder die im Rahmen der Java EE (ehemals J2EE) spezifizierten Enterprise JavaBeans (EJB) [BMH06, Sun07c]. Hier werden die in einer zu integrierenden Anwendung enthaltenen Funktionen als Methoden in verteilten Objekten zugänglich gemacht. Objekte verbergen gemäß dem Paradigma der Objektorientierten Programmierung ihre internen Strukturen und Eigenschaften wie auch die meisten Details bezüglich Plattform, Programmiersprachen und Implementierung hinter einer öffentlich zugänglichen Schnittstelle.

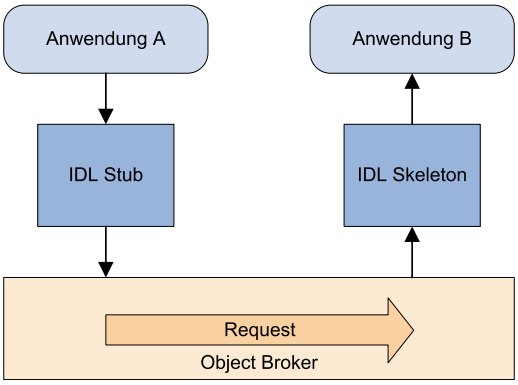


Abbildung 9.7: Entfernter Methodenaufruf über einen Objekt Request Broker.

Durch einen so genannten Object Broker (CORBA, DCOM) bzw. RMI (EJB) wird ein Kommunikationskanal etabliert, der es ermöglicht, einen entfernten Methodenaufruf gegenüber einem Klienten wie einen lokalen Aufruf erscheinen zu lassen. Er stellt somit eine protokollunabhängige Implementierung eines Kommunikationskanals zwischen verteilten Objekten zur Verfügung. Ein Entwickler kann somit auf entfernte Objekte zugreifen, ohne auf Einzelheiten der verwendeten Hardwareplattformen oder Netzwerkprotokolle Rücksicht nehmen zu müssen. Die Beschreibung der Schnittstellen der entfernten Objekte erfolgt mithilfe von Java (EJB) oder programmiersprachenunabhängig mittels einer Interface Definition Language (CORBA, DCOM). Abbildung 9.7 verdeutlicht den Kommunikationsmechanismus, welcher komponentenorientierter Middleware zugrunde liegt. Lokale Proxies (Stub

und Skeleton) fungieren als schnittstellenäquivalente lokale Stellvertreterobjekte des jeweiligen entfernten Server-Objektes. Der Client ruft eine Methode des Stubs, welcher diese in ein plattformunabhängiges Zwischenformat umwandelt (Marshalling) und über den Kommunikationskanal an das Skeleton auf der Serverseite übermittelt. Dieser konvertiert die Parameter in ein lokales Format und ruft die gewünschte Methode des Server Objektes auf. Zur Übertragung der Rückgabewerte des Aufrufs verläuft die Kommunikation in umgekehrter Richtung über das Skeleton und den Kommunikationskanal bis hin zum Stub, welcher die Ergebnisse lokal an die aufrufende Anwendung übergibt (Demarshalling).

Auf eine vertiefende Darstellung der Architektur und Funktionsweise der skizzierten Ansätze soll an dieser Stelle zugunsten einer differenzierten Bewertung der Alternativen im Hinblick auf ihre Einsatzpotentiale im Kontext der Anwendungsintegration verzichtet werden. Für weiterführende Informationen zu CORBA, Java EE und COM/DCOM sei daher auf die angegebene Literatur sowie auf Abschnitt 5.2 in dieser Broschüre verwiesen.

Da Objekte ihre innere Struktur hinter einer Schnittstelle verbergen, kann sich die Implementierung einer Methode ändern, ohne dass sich zwangsläufig die Implementierung der zugreifenden Klienten ändern muss. Darüber hinaus können Objekte aufgrund der objektorientierten Mechanismen der Vererbung und der Polymorphie entsprechend ihrer Klassenzugehörigkeit beim Aufruf der gleichen Methode ein völlig unterschiedliches Verhalten zeigen. Die zentrale Aufgabe einer komponentenorientierten Middleware-Plattform besteht demnach darin, eine Verbindung zwischen einem Klienten und einem entfernten Objekt zu etablieren und die Interaktion zwischen diesen zu vermitteln. Ergänzend werden von den einzelnen Middleware-Systemen zusätzliche zumeist plattform– und herstellerspezifische Funktionen bereitgestellt, welche Aspekte wie die Suche und Verwaltung von Objekten, die Durchführung von Transaktionen oder die Authentifizierung und Autorisierung von Benutzern betreffen. Diese sollen an dieser Stelle jedoch nicht weiter vertieft werden.

Das Hauptproblem nicht nur in RPC-basierten Integrationsansätzen, sondern auch bei der Verwendung komponentenorientierter Middleware, ist die enge Kopplung zwischen den integrierten Systemen. Bei einer Funktionsintegration auf Code– oder Komponentenebene kann neue Funktionalität nur durch Anpassung und anschließende Neuübersetzung eines Systems integriert werden. Sind die Anforderungen an die Integrationsinfrastruktur und die Anbindung neuer Systeme im Zeitverlauf relativ unverändert, so kann dieser Nachteil hingenommen werden. In einem dynamischen Umfeld mit rasch wechselnden Integrationsanforderungen erweist sich eine derartige Lösung schnell als kostspielig und unflexibel. In diesem Fall ist eine Integrationsinfrastruktur wünschenswert, die es ermöglicht, Funktionalitäten zur Laufzeit dynamisch einzubinden. Das heißt jegliche Referenzen oder Adressierungsinformationen, die zum Auffinden einer Funktionalität erforderlich sind, dürfen nicht

zur Übersetzungszeit sondern erst zur Laufzeit angelegt werden.

Ein weiterer gravierender Nachteil komponentenorientierter Middleware ist die unzureichende Kompatibilität der Systeme untereinander. Die Entscheidung für eine bestimmte Middleware-Plattform bedingt, dass alle in die Integrationslösung eingebundenen Systeme den gleichen RPC-Mechanismus, das gleiche Objektmodell oder ein einheitliches Nachrichtenformat unterstützen müssen. Da einige der am Markt verfügbaren Middleware-Plattform darüber hinaus keine vollständige Plattform– und Programmiersprachenunabhängigkeit bieten, ist die Flexibilität bei der Auswahl von Betriebssystemen und Programmiersprachen stark eingeschränkt. So bietet DCOM die größten Einsatzpotentiale in einer Windows-Umgebung, wenngleich Implementierungen für weitere Betriebssysteme verfügbar sind. Enterprise JavaBeans in Verbindung mit RMI sind an die Verwendung der Programmiersprache Java gebunden. Die Integration von Fremdsystemen, die nicht in Java geschrieben sind, ist möglich, lässt sich jedoch nur mit zusätzlichem Aufwand realisieren. CORBA bietet lediglich eine Spezifikation ohne konkrete Implementierung. Die verfügbaren Implementierungen weisen dabei herstellerspezifische Erweiterungen und Besonderheiten auf, weshalb die Interoperabilität zweier Implementierungen oftmals nur sichergestellt werden kann, indem auf die Nutzung derartiger Erweiterungen verzichtet wird. Darüber hinaus setzt der objektorientierte Ansatz von CORBA implizit voraus, dass die zu integrierenden Systeme in einer Sprache implementiert wurden, die dieses Paradigma unterstützt. Sollen Funktionen und Daten integriert werden, die mit einer Skriptsprache wie Perl oder PHP erstellt wurden, so sind diese zunächst durch einen so genannten Wrapper (vgl. Adapter Muster in [GHJV96]) in ein Objekt zu kapseln. Ähnlich wie der RPC-Mechanismus basiert komponentenorientierte Middleware auf einem synchronen Kommunikationsmodell.

Nachrichtenbasierte Middleware-Plattformen

Im Unterschied zu RPC und RMI unterstützen nachrichtenbasierte Middleware-Plattformen (engl. Message Oriented Middleware, MOM) sowohl synchrone, als auch asynchrone Kommunikation. Dadurch eignen sie sich gleichermaßen für den Aufruf von Funktionen wie auch für die unidirektionale Übertragung von Informationen. Bei der Anwendung asynchroner Kommunikation müssen Sender und Empfänger für eine funktionierende Kommunikation nicht zwingend aktiv sein. Ist der Empfänger nicht verfügbar, so werden die an ihn adressierten Nachrichten in einer Warteschlange persistent zwischengespeichert. Die sendende Anwendung wird somit auch dann nicht blockiert, wenn der Empfänger zum Zeitpunkt des Versendens einer Nachricht nicht erreichbar ist. Aus diesem Grund zeichnet sich asynchrone nachrichtenbasierte Kommunikation vor allem in einem verteilten heterogenen Umfeld aus, in welchem die zu integrierenden Anwendungen unterschiedlichen Verantwortlichkeiten unterstehen oder über das Internet interagieren müssen. Nachrichtenbasierte Kommunikation ermöglicht eine Entkopplung der zentralen Integrationsinfrastruktur von

der Implementierung der einzelnen Anwendungen. Dies erlaubt gegenüber dem Einsatz von RPC- oder RMI-basierten Integrationslösungen einen zusätzlichen Grad an Flexibilität. Ein Nachteil nachrichtenbasierter Kommunikation besteht darin, dass sie die Suche nach Fehlern erschwert. Während bei einer engen Kopplung zweier Systeme aufgrund der Implementierungsabhängigkeiten viele Fehler bereits zur Übersetzungszeit gefunden werden können, zeigen sich bei einer nachrichtenbasierten, losen Kopplung viele Fehler erst zur Laufzeit. Dabei ist in vielen Fällen zunächst nicht direkt erkennbar, ob ein Fehler mit der Erzeugung einer Nachricht im Quellsystem, dem Nachrichtenformat selbst, der Übertragung der Nachricht oder ihrer Auswertung im Zielsystem zusammenhängt.

Java Messaging Service

In einem Java-Umfeld bietet sich Java Message Service (JMS) (vgl. [Sun07d]) als mögliche Implementierungsplattform für eine nachrichtenbasierte Kommunikation an. Die JMS-API erlaubt es Java EE-Anwendungen, Nachrichten zu erzeugen, zu versenden und zu empfangen. Sie unterstützt die Umsetzung einer lose gekoppelten, verteilten und asynchronen Kommunikationsplattform. Vorteilhaft ist die Verfügbarkeit einer ausgereiften API, die dem Anwender eine einfache Benutzung ermöglicht und darüber hinaus Dienste wie Transaktionssicherung oder die Unterstützung verschiedener Nachrichtenmodelle bereitstellt. Neben der Punkt-zu-Punkt-Kommunikation (Unicast), welche mit einem E-Mail-Versand von einem Sender an einen Empfänger verglichen werden kann, lassen sich Nachrichten im Publish-and-Subscribe-Modus versenden. Beim Publish-and-Subscribe werden Nachrichten nicht an einen bestimmten Empfänger, sondern an ein so genanntes Topic gesendet. Empfänger einer Nachricht sind dabei nur solche Systeme, die das jeweilige Topic abonniert haben. Auf diese Weise lässt sich auch eine Multicast- oder Broadcast-Kommunikation nachbilden, welche mit einem E-Mail-Versand eines Senders an mehrere Mitglieder bzw. eine ganze Gruppe von Empfängern vergleichbar ist. Probleme sind zu erwarten, wenn Anwendungen zu integrieren sind, die in einer Programmiersprache vorliegen, für die keine JMS Implementierung verfügbar ist. Eine Integration kann in diesem Fall nur mit erhöhtem Implementierungsaufwand oder unter Einschränkungen vorgenommen werden. Sollen Anwendungen über Organisationsgrenzen hinweg angebunden oder die Systeme geografisch verteilter Einheiten einer einzelnen Organisation integriert werden, so findet die Kommunikation der Anwendungen zumeist über eine Firewall oder einen Proxy statt. Hierbei ist zu beachten, dass der JMS-Dienst standardmäßig den Port 7676 nutzt, weshalb JMS-Anfragen häufig durch die Firewall blockiert werden. Lösen lässt sich dieses Problem, indem der betreffende Port explizit für die JMS-Kommunikation geöffnet wird. Wenn dies nicht erwünscht ist, so können JMS-Nachrichten alternativ in spezielle HTTP-Pakete verpackt (HTTP-Tunneling) und auf diese Weise über Firewallgrenzen hinweg versandt werden.

Service Orientierte Architekturen und Web-Services

Eine nicht nur plattformunabhängige, sondern auch programmiersprachen- und protokollunabhängige Alternative stellen Web-Services dar. Ein Web-Service lässt sich als Komponente auffassen, die ihre Funktionalität über eine öffentlich zugängliche Schnittstelle anbietet und über offene, im Internet verwendete Protokolle zugreifbar ist. Mithilfe von Web-Services kann eine Service Orientierte Architektur (SOA) [KBS05] realisiert werden. Analog zur Definition eines Web-Service, stellen Dienste im Sinne einer SOA ein durch eine wohldefinierte und über ein öffentliches Verzeichnis zugängliche Schnittstellenbeschreibung repräsentiertes sowie wiederverwendbares Stück Software dar, das eine abgegrenzte Funktionalität anbietet und durch einen Klienten integriert werden kann. Eine Service Orientierte Architektur stellt ein Netzwerk von lose gekoppelten und miteinander interagierenden Diensten dar. Diese Kommunizieren durch den Austausch von Nachrichten und können untereinander auf Daten oder auf ihre Anwendungslogik zugreifen. Wenngleich sich das SOA-Konzept mithilfe von Web-Services realisieren lässt, so muss betont werden, dass die Verwendung von Web-Services nicht zwangsläufig zu einer SOA führt und eine SOA andererseits nicht notwendigerweise mithilfe von Web-Services implementiert werden muss.

Aus Sicht eines Geschäftsprozessverantwortlichen macht weder die Integration eines zu komplexen, noch die eines zu simplen Dienstes Sinn. Während ein komplexer Dienst sich aufgrund seiner ausgeprägten Semantik nur in wenigen Fällen ohne aufwändige Anpassungen einsetzen lässt, kann ein elementarer Dienst zwar leicht wiederverwendet werden, liefert dabei jedoch nur einen beschränkten Nutzen. Dienste sind demnach von mittlerer Komplexität und bilden einen wiederverwendbaren Ausschnitt aus einem Geschäftsprozess oder eine inhaltlich klar abgegrenzte und für den praktischen Einsatz geeignete Funktionalität ab. Sie können durch den Dienstanbieter über ein öffentlich zugängliches Verzeichnis (Dienst Registry) zur Verfügung gestellt und von einem Dienstkonsumenten anhand ihrer Beschreibung zur Laufzeit eingebunden werden. Elementare Dienste lassen sich durch eine so genannte Orchestrierung zu komplexen Diensten kombinieren, welche wiederum veröffentlicht werden können.

Technologisch beruhen Web-Services auf den XML-basierten Standards SOAP, WSDL (Web-Services Description Language) und UDDI (Universal Description, Discovery and Integration) [Wor07a]. Ein SOAP-Aufruf lässt sich als XML-Dokument verstehen, welches einen Methodenaufruf beschreibt. Die grundlegende Semantik einer SOAP-Nachricht entspricht somit im Wesentlichen der eines RPC-Aufrufes. SOAP-Nachrichten können mithilfe verschiedener Protokolle, wie HTTP, UDP oder SMTP (Simple Mail Transfer Protocol) übertragen werden. Firewallprobleme wie sie im Zusammenhang mit JMS beschrieben wurden, können häufig umgangen werden, wenn HTTP als Übertragungsprotokoll gewählt wird. WSDL erlaubt es, die

durch einen Klienten zugreifbaren Funktionen eines Web-Service zu definieren sowie deren Schnittstellen zu beschreiben. Im Einzelnen beinhaltet ein WSDL-Dokument Angaben über Parameter und Rückgabewerte der Funktionen eines Web-Service sowie allgemeine Zugriffs– und Deploymentinformationen. Um für eine bestimmte Anforderung einen adäquaten Web-Service finden und adressieren zu können, kann der Standard UDDI verwendet werden. UDDI stellt mithilfe einer SOAP-Schnittstelle einen Verzeichnisdienst bereit, der Informationen über Web-Services und deren Anbieter enthält. Die Interaktionen zwischen einem Dienstanbieter und einem Konsumenten sind in Abbildung 9.8 veranschaulicht. Der Anbieter veröffentlicht in einer UDDI-Registry Informationen zu seinem Unternehmen bzw. seiner Organisation sowie die WSDL-Beschreibung der zu veröffentlichen Web-Services.

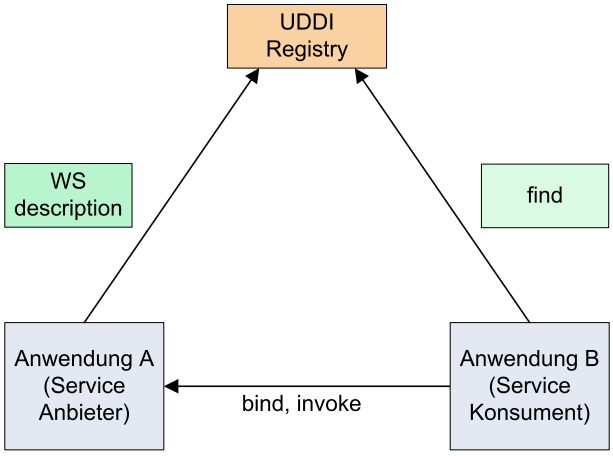


Abbildung 9.8: Interaktionen zwischen Web-Services.

Der Konsument durchsucht dieses Verzeichnis, findet anhand der Anbieter– und Dienstbeschreibungen einen passenden Dienst und erhält die WSDL-Beschreibung des ausgewählten Web-Services. Mit dieser kann er den entsprechenden Dienst integrieren und über SOAP-Nachrichten auf dessen Funktionen zugreifen. Sollen Web-Services lediglich innerhalb der eigenen Organisation oder eines beschränkten Kreises von Anbietern und Nutzern veröffentlicht werden, so kann als Alternative zu UDDI auch ein eigener Web-Service-Verzeichnisdienst eingerichtet werden. In vielen Fällen kann auf die Verwendung eines Registry-Dienstes sogar gänzlich verzichtet werden, da Schnittstellen und Funktionalitäten der eingesetzten Web-Services hinreichend bekannt sind. Verglichen mit den anderen betrachteten Standards kommt UDDI im praktischen Einsatz daher eine eher geringe Bedeutung zu.

Der Hauptvorteil von Web-Services liegt in der Verwendung offener Standards, die sowohl Plattformunabhängigkeit, als auch Protokoll- und Programmiersprachenunabhängigkeit ermöglichen und somit eine hohe Flexibilität und Interoperabilität Web-Service-basierter Integrationslösungen sicherstellen.

Durch das üblicherweise zur Nachrichtenübertragung verwendete HTTP-Protokoll ist eine Kommunikation auch über Organisationsgrenzen hinweg problemlos möglich. Nachteile betreffen vor allem die Performanz, die durch die XML-Verarbeitung und die Größe der zu übertragenen Nachrichten negativ beeinflusst wird. Kritisch zu hinterfragen ist weiterhin, ob die verfügbaren Standards wie XML-Encryption [Wor07b], XML-Signature [Wor07c] oder Web-Services-Security [OAS07c] ausreichen, um auch in sensiblen Bereichen ein angemessenes Sicherheitsniveau zu gewährleisten.

Orchestrierung von Web-Services

Mithilfe der Sprache WS-BPEL (Web Service Business Process Execution Language) [OAS07a, JMS06] lassen sich Web-Services zu komplexeren Diensten „orchestrieren“. Dies ermöglicht es, Geschäftsprozesse aus elementaren Web-Service-Bausteinen mit definierten Schnittstellen flexibel zusammenzusetzen. Zur Erstellung von Geschäftsprozessen bietet WS-BPEL die Möglichkeit, bereits existierende Web-Services zu adressieren und aufzurufen. Darüber hinaus können Kontrollstrukturen wie Verzweigungen und Schleifen implementiert werden. Jeder mithilfe von WS-BPEL erstellte Prozess stellt wiederum einen Web-Service dar, der durch ein WSDL-Dokument beschrieben wird und veröffentlicht werden kann. Um einen WS-BPEL-Prozess ausführen zu können, wird eine BPEL-Engine als Laufzeitumgebung benötigt, welche auch einen transparenten Zugriff auf die eingebundenen Web-Services ermöglicht. Zudem stellt sie Funktionen für das Deployment und die Verwaltung der WS-BPEL-Prozesse sowie Persistenzdienste zur Verfügung. Die Beschreibung der WS-BPEL-Prozesse erfolgt mittels XML, wobei die einzelnen Sprachelemente wie Variablendeklarationen, Zuweisungen oder Kontrollstrukturen durch entsprechende Tags dargestellt werden. Verglichen mit anderen Programmiersprachen wie zum Beispiel Java wird der Quellcode eines WS-BPEL-Prozesses daher schnell unübersichtlich. Durch die zunehmende Verfügbarkeit grafischer Entwicklungswerkzeuge relativiert sich dieser Nachteil jedoch zusehends. Schwerer wiegt, dass dem Entwickler nur wenige Konstrukte zur Verfügung stehen, die eine angemessene Strukturierung von Prozessen ermöglichen. Dieser Umstand schränkt die Einsatzmöglichkeiten von WS-BPEL auf die Beschreibung relativ überschaubarer Abläufe ohne komplexe Logiken oder Berechnungen ein.

9.4.3 Integration über die Benutzerschnittstelle

Bei einer Integration auf Benutzerschnittstellenebene werden Daten, Funktionen und Prozesse einer Anwendung durch Einbindung ihrer Benutzerschnittstelle zugreifbar gemacht (vgl. Abbildung 9.9). Die Programmlogik oder die Daten der Anwendung bleiben bei dieser Form der Integration unberührt.

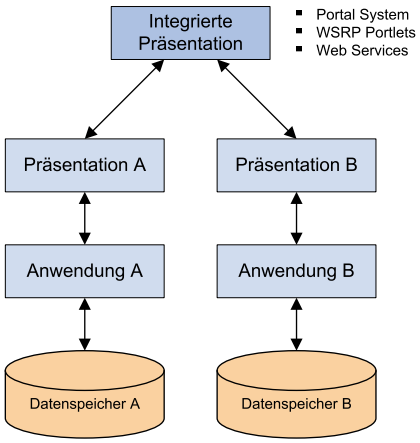


Abbildung 9.9: Integration auf Benutzerschnittstellenebene.

Eine derartige Integration kann mithilfe so genannter *Screen-Scraping-Systeme* oder durch ein webbasiertes Portal gewährleistet werden. Welche Art der Integration in speziellen Situation sinnvoller ist, hängt vor allem davon ab, ob die Daten eines Systems für eine automatisierte Weiterverarbeitung in einem Fremdsystem verfügbar gemacht oder Benutzern in integrierter Form präsentiert werden sollen. Ein Screen-Scraping-System ermöglicht es, die Bildschirmmasken einer Anwendung zeilenweise auszuwerten und einen Benutzerdialog zu simulieren. Die Ausgaben der Anwendung an den Benutzer können auf diese Weise automatisch erfasst und einem Fremdsystem zugeführt werden. Screen-Scraping gilt gemeinhin als Notlösung, welche mit enormen Performanzeinbußen erkauft werden muss. Darüber hinaus müssen Änderungen an der Benutzerschnittstelle einer Anwendung im Screen-Scraping-System nachgeführt werden, um Probleme wie Abstürze oder fehlerhafte Ergebnisse zu vermeiden. Andererseits stellt Screen-Scraping vielmals die einzige Alternative für eine Integration dar, wenn alle anderen Ansätze mangels Verfügbarkeit geeigneter Schnittstellen oder einer geeigneten Dokumentation bzw. des Quellcodes ausscheiden.

Ein *Portal* kann als eine webbasierte Anwendung verstanden werden, die Daten und Funktionen verschiedener Anwendungen als Dienste auf einer Plattform integriert und dem Benutzer über eine einheitlich gestaltete Bedienoberfläche zugänglich macht. Für den Benutzer ergeben sich Vorteile wie eine einheitliche Bedienung, Single-Sign-On, benutzerspezifische Anpassung von Funktionsumfang und Layout sowie eine einfach einzurichtende Verknüpfung von Diensten und Inhalten verschiedener Anwendungen. Die Präsentation einer Anwendung auf der Portalseite erfolgt in einem Fensterbereich, der als Portlet (vgl. [Zï¿06] für die Entwicklung von Portlets mit Java) bezeichnet wird. Ein Portlet bildet die Benutzerschnittstelle einer einzelnen Anwendung oder einer Gruppe von logisch und/oder inhaltlich zusammengehörigen Anwendungen ab. Durch Kommunikationsmechanismen können eingeschränkte Interaktionen zwischen Portlets realisiert werden. Der OASIS-Standard

WSRP (Web-Services for Remote Portlets) [OAS07b] beschreibt wie WSRP-kompatible Portale, Anwendungen und Inhalte mithilfe standardisierter Web-Services dynamisch integrieren können. Im Unterschied zu Web-Services erlaubt WSRP, nicht nur Inhalte und Anwendungslogik sondern auch Darstellungselemente (Markup-Fragmente) zu veröffentlichen und zu konsumieren. Auf diese Weise können Portalbetreiber flexibel Angebote nebst ihrer Präsentation aus verschiedenen organisationsinternen und –externen Quellen integrieren und dem Benutzer bereitstellen.

9.5 Zusammenfassende Bewertung

Verglichen mit einer Integration auf Funktionsebene lässt sich eine Datenintegration relativ einfach bewerkstelligen. Dies liegt nicht zuletzt daran, dass hierfür ein breit gefächertes Angebot an ausgereiften Integrationstechnologien zur Verfügung steht. Eine Datenintegration ist daher vor allem dann zu empfehlen, wenn ein lesender Zugriff auf die Daten einer Anwendung benötigt wird. Soll über eine Integration der zugrunde liegenden Datenbasis auch schreibend auf die Daten einer Anwendung zugegriffen werden, so muss mit Problemen durch semantische Heterogenitäten gerechnet werden. So ist es zum Beispiel denkbar, dass eine Anwendung Datensätze in die Datenbank einer Zielanwendung einträgt, die durch diese nicht korrekt interpretiert werden können. Eine Integration auf Datenebene verleitet dazu, in Ermangelung eines Zugriffs auf die Verarbeitungslogik einer Anwendung, diese in unterschiedlichen Systemen redundant zu implementieren. Schwierig nachzuvollziehende Fehler sind dann möglich, wenn Änderungen an der Programmlogik nicht konsistent nachgeführt werden oder transaktionsverarbeitende Logik dupliziert wurde. Aus diesen Gründen stellt die Datenintegration nur dann eine unproblematische Integrationslösung dar, wenn lediglich ein lesender Zugriff auf die Daten einer Anwendung benötigt wird.

Die Integration auf Funktionsebene ist die semantisch reichhaltigste Art der Integration, da hier nicht nur die Semantik der Daten sondern zudem die Semantik der Programmlogik einer Anwendung zugänglich gemacht wird. Darüber hinaus lassen sich Redundanzen, welche in den zu integrierenden Systemen sowohl auf Datenebene als auch auf Funktionsebene bestehen können, weitestgehend beseitigen. Andererseits lässt sich eine Funktionsintegration nicht in jedem Fall ohne Schwierigkeiten durchführen. Probleme sind vor allem dann zu erwarten, wenn Anwendungen zu integrieren sind, die über keine Schnittstellen zur Anbindung an Fremdsysteme verfügen und keinen Einblick in den Quellcode erlauben. Darüber hinaus ist die Funktionsintegration mit einem nicht zu unterschätzenden technischen Aufwand verbunden. Hierzu trägt nicht zuletzt die Fülle der verfügbaren Integrationstechnologien bei. Aufgrund der unterschiedlichen Eigenschaften der Technologiealternativen sollte der Auswahl einer geeigneten Integrationstechnologie mindestens genauso viel Beachtung geschenkt werden, wie der grundsätzlicheren Entscheidung für eine Integrationsebene. Da die Integrationsinfrastruktur grundlegende Funktionalitäten

bereitstellt, welche die Interaktionsfähigkeit zwischen den gegenwärtigen wie auch den zukünftigen Informationssystemen eines Unternehmens ermöglichen soll, ist sie sowohl mit den Anforderungen, Rahmenbedingungen und Restriktionen der existierenden Informationssysteme, als auch mit den aus der Unternehmensstrategie abgeleiteten Zielvorgaben für die zukünftige Entwicklung der betrieblichen Informationsverarbeitung abzustimmen. Aspekte wie Nachhaltigkeit, Skalierbarkeit, Flexibilität und Sicherheit spielen hier eine tragende Rolle. Eine zu kurzsichtige und alleine an der gegenwärtigen Situation orientierte Planung einer Anwendungsintegration sollte vermieden werden, da dies leicht zu einer Integrationsinfrastruktur führt, die zukünftigen Anforderungen nicht mehr gewachsen ist oder mit dem Wachstum des Unternehmens nicht schritt hält und sich somit zum kostspieligen Engpass für die Fortentwicklung der betrieblichen Informationsverarbeitung entwickelt.

Eine Integration auf Benutzerschnittstellenebene wird in der Literatur zur EAI [CHK06, Lin00] gemeinhin als Notlösung angesehen, die erst dann zur Anwendung kommt, wenn eine Integration auf Daten– oder Funktionsebene nicht möglich ist. Die Hauptgründe für die Kritik sind deutliche Defizite dieses Integrationsansatzes hinsichtlich Performanz und Skalierbarkeit. Hinzu kommt die Problematik, dass sich die Benutzerschnittstelle einer Anwendung verglichen mit der Anwendungslogik oder ihren Datenstrukturen relativ häufig ändert. Bei der Verwendung von Screen-Scraping-Werkzeugen bewirkt jedoch oftmals schon die Verschiebung eines Anzeigeelementes um wenige Millimeter, dass es nicht mehr zuverlässig erkannt wird. Darüber hinaus werden in den einzelnen Anwendungen bestehende Redundanzen auf Daten- oder Funktionsebene nicht beseitigt.

Kapitel 10

Modell-getriebene Software-Entwicklung

Mit dem Siegeszug der Objektorientierung hat sich auch die hierfür entwickelte, vereinheitlichte Modellierungsnotation, die Unified Modeling Language (UML), durchgesetzt (vgl. Abschnitt 4.2). Mit dem Vorliegen der Modelle entstand dann die Idee, Teile eines objektorientierten Softwaresystems nicht von Hand zu erstellen, sondern automatisch aus den Modellen zu generieren.

Die wohl beliebtesten Tools für die Modell-getriebene Software-Entwicklung sind androMDA [And08] und openArchitectureWare (oAW) [EFH+07]. Beide bieten bereits vordefinierte Transformationsregeln (so genannte Cartridges) für typische Plattformen wie Enterprise JavaBeans, Spring, Hibernate usw.. oAW ist etwas flexibler und leichter an eigene Plattformen anpassbar, auch solche die nicht auf Java basieren.

Die benötigten Transformationen werden häufig in speziell für diese Aufgabe entwickelten Templatesprachen entwickelt. Für jede Zielplattform gibt es jeweils eigene Transformationsregeln. Damit in den Modellen plattform-spezifische Einstellungen vorgenommen werden können, müssen sie semantisch angereichert werden. Dies geschieht durch so genannte *Stereotype*, mit denen sich Modellelemente wie z.B. Klassen und Methoden genauer charakterisieren lassen. Wenn beispielsweise eine Klasse `Exemplar` vom Modellierer mit dem Stereotyp `Entity` versehen wird (vgl. Abb. 10.1), so bedeutet dies für eine Transformation des Modells im Hinblick auf die Zielplattform Enterprise JavaBeans (vgl. Abschitt 5.2.3), dass diese Klasse als Entitätsklasse realisiert und deren Objekte vom EJB-Container verwaltet und mit den zugehörigen Tupeln der entsprechenden Datenbanktabelle synchronisiert werden sollen. Der in diesem Beispiel generierte Java-Code kann wie in Abbildung 10.2 aussehen. Neben EJB-spezifischen Annotationen (@Entity, @ManyToOne, usw.) wurden get- und set-Methoden für das Attribut `inventarnr` und die Assoziation zur Klasse `Buch` angelegt. Die Transformation wird bei oAW durch ein Template in der Sprache xPand

beschrieben. Das hier verwendete Template ist in Abbildung 10.3 zu sehen. In diesem Template werden einige Hilfsfunktionen wie z.B. `getPackageName`, `getVisibility`, `getStatic`, `getTypeName`, `getAttributeName` und `isMultiple` verwendet. Diese werden bei oAW in einer weiteren Domänen-spezifischen Sprache, nämlich Xtend, definiert. Für die Funktion `getMultiplicityAnnotation` sieht eine solche Xtend-Definition beispielsweise wie folgt aus:

```
String getMultiplicityAnnotation(uml::Property p) :
  "@"+ (isMultiple(getOpposite(p)) ? "Many" : "One") + "To" +
       (isMultiple(p) ? "Many" : "One");
```

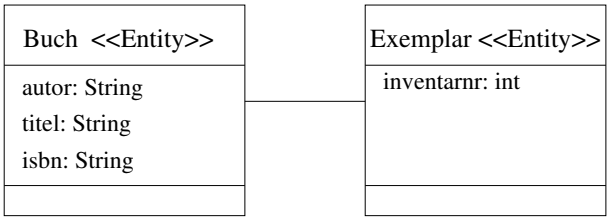


Abbildung 10.1: Klassendiagramm mit Stereotypen.

```
package Data;
@Entity
public  class Exemplar implements java.io.Serializable{
   protected int inventarnr;
   protected Data.Buch buch;

   public int getInventarnr(){return inventarnr;}
   public void setInventarnr(int p_inventarnr){inventarnr = p_inventarnr;}

   @ManyToOne
   @JoinColumn(name = "buch")
   public Data.Buch getBuch(){return buch;}
   public void setbuch(Data.Buch p_buch){buch= p_buch;}
}
```

Abbildung 10.2: Generierter EJB-Code für die Klasse Exemplar im Buch-Beispiel.

Durch die Modell-getriebene Softwareentwicklung [SV06] lässt sich die Produktivität bei der Softwareentwicklung besonders dann steigern, wenn viele, sehr ähnliche Versionen oder Varianten eines Softwaresystems für jeweils die gleichen Plattformen erstellt werden müssen. Dann lassen sich die Transformationen wiederholt nutzen. Bei Einzelentwicklungen bringt dieser Ansatz wenig Vorteile.

In einem Pilotprojekt an der Universität Münster ließen sich Webanwendungen, bei denen ausschließlich so genannte CRUD-Operationen (create, read, update, delete) benötigt wurden, inklusive der hierfür nötigen Navigationsstrukturen zu

```
≪EXTENSION templates::Library≫
≪DEFINE Root FOR Metamodel::Entity≫
  ≪FILE getClassPath(this)≫
    package ≪getPackageName(this.package)≫;
    @Entity
    public class ≪this.name≫ implements java.io.Serializable{
    ≪EXPAND Property FOREACH this.ownedAttribute≫
    ≪EXPAND Accessor FOREACH this.ownedAttribute≫ }
  ≪ENDFILE≫
≪ENDDEFINE≫

≪DEFINE Property FOR uml::Property≫
  ≪getVisibility(this)≫ ≪getStatic(this)≫
  ≪getTypeName(this)≫ ≪getAttributeName(this)≫
  ≪IF !isMultiple(this)≫  ;
  ≪ELSE≫
    = new java.util.ArrayList<≪getQualifiedTypeName(this.type)≫>();
  ≪ENDIF≫
≪ENDDEFINE≫

≪DEFINE Accessor FOR uml::Property≫
  ≪IF this.association != null≫
    ≪getMultiplicityAnnotation(this)≫
    @JoinColumn(name = "≪this.name≫")
  ≪ENDIF≫
  public ≪getStatic(this)≫ ≪getTypeName(this)≫ ≪getGetterName(this)≫(){
    return ≪getAttributeName()≫; }
  public ≪getStatic(this)≫ void ≪getSetterName(this)≫
      (≪getTypeName(this)≫ p_≪getParameterName(this)≫){
    ≪getAttributeName()≫ = p_≪getParameterName()≫;  }
≪ENDDEFINE≫

≪DEFINE Root FOR uml::Model≫
  ≪EXPAND Root FOREACH this.allOwnedElements()≫
≪ENDDEFINE≫

≪DEFINE Root FOR uml::Element≫≪ENDDEFINE≫
```

Abbildung 10.3: xPand-Transformationsregeln im Buch-Beispiel.

100 % aus den UML-Modellen erzeugen [Wol08]. Im Wesentlichen wurden hierbei Klassendiagramme zur Datenmodellierung und Aktivitätsdiagramme zur Modellierung der Navigationsstruktur in der webbasierten Benutzeroberfläche benutzt. Exemplarisch wurde so ein Bibliotheksverwaltungssystem vollkommen automatisch aus UML-Modellen erzeugt. Bei algorithmisch anspruchsvolleren Anwendungen wird eine Codegenerierung zu 100 % nicht möglich sein, jedoch kann auch hier vielfach ein großer Teil erzeugt werden. Nur der verbleibende, algorithmisch komplexere Rest muss dann von Entwicklern in so genannten geschützten Regionen per Hand erstellt werden.

Die Modell-getriebene Softwareentwicklung ist ein noch junger und in Entwicklung befindlicher Zweig der Softwareentwicklung. Gleichwohl wird dieser Ansatz auch von einigen Firmen im Münsterland bereits produktiv eingesetzt und bietet auch für andere erhebliches Potenzial zur Steigerung der Produktivität. Zu beachten

ist hierbei allerdings, dass der Ansatz einen nicht unerheblichen Einarbeitungsaufwand erfordert und nur geeignet ist, wenn die oben erwähnten Rahmenbedingungen (viele Versionen bzw. Varianten eines Systems für die jeweils gleichen Plattformen) gegeben sind. Ein weiteres Problem ist in der komplexeren Qualitätssicherung zu sehen. Während es bei konventioneller Softwareentwicklung ausreicht, den entwickelten Code zu testen, können sich Fehler nun nicht nur im handgeschriebenen Programmcode sondern auch in den Modellen, Transformationsregeln und Metamodellen verbergen. Debugger, die die Fehlersuche über diese verschiedenen Ebenen hinweg unterstützen, fehlen noch.

Literaturverzeichnis

[And08] AndroMDATeam. *AndroMDA*. http://www.andromda.org. 2008

[Apa07a] Apache Software Foundation. *Struts*. `http://struts.apache.org/`. Abrufdatum: 17.07.2007

[Apa07b] Apache Software Foundation. *Tapestry*. `http://tapestry.apache.org/`. Abrufdatum: 17.07.2007

[BA04] Beck, Kent ; Andres, Dirk: *Extreme Programming Explained - Embrace Change*. 1. Boston : Addison Wesley, 2004

[Bal93] Balzert, Helmut: *CASE - Systeme und Werkzeuge*. 1. Mannheim : B.I. Wissenschaftsverlag, 1993

[Bal98] Balzert, Helmut: *Lehrbuch der Software-Technik: Software-Management, Software- Qualitï¿½ssicherung, Unternehmensmodellierung*. 1. Heidelberg u.a. : Spektrum Akademischer Verlag, 1998

[Bal01] Balzert, Helmut: *Lehrbuch der Software-Technik: Software-Entwicklung*. 1. Heidelberg u.a. : Spektrum Akademischer Verlag, 2001

[Bet07] Better SCM Initiative. *Version Control System Comparison*. `http://better-scm.berlios.de/comparison/comparison.html`. Abrufdatum: 17.07.2007

[BG01] Bauer, Andreas ; Gnzel, Holger: *Data-Warehouse-Systeme - Architektur, Entwicklung, Anwendung*. 2. Heidelberg : dpunkt-Verlag, 2001

[BJ97] Bï¿½nke, Jana ; Johannes, Hermann: *ODBC. Optimaler Einsatz im Client/Server-Umfeld*. 1. Mnchen : Addison Wesley, 1997

[BMH06] Burke, Bill ; Monson-Haefel, Richard: *Enterprise JavaBeans 3.0*. 5. O'Reilly Media, 2006

[Boe89] Boehm, Barry W.: *Software Risk Management*. 1. IEEE Computer Society Press, 1989

[Boe91] Boehm, Barry W.: Software Risk Management: Principles and Practices. In: *IEEE Software* 8 (1991), Nr. 1, S. 32–41

[Bor07] Borland. *Borland Together*. `http://www.borland.com/us/products/together/index.html`. Abrufdatum: 17.07.2007

[CDKM02] Coulouris, George ; Dollimore, Jean ; Kindberg, Tim ; Muhr, Judith: *Verteilte Systeme: Konzepte und Design*. 1. Mnchen : Pearson Studium, 2002

[CHK06] Conrad, Stefan ; Hasselbring, Wilhelm ; Koschel, Arne: *Enterprise Application Integration - Grundlagen, Konzepte, Entwurfsmuster, Praxisbeispiele*. 1. Mnchen : Spektrum Akademischer Verlag, 2006

[Col07] Collabnet. *Subversion*. `http://subversion.tigris.org/`. Abrufdatum: 17.07.2007

[EFH+07] Efftinge, Sven ; Friese, Peter ; Haase, Arno ; Kadura, Clemens ; Kolb, Bernd ; Moroff, Dieter ; Thoms, Karsten ; Völter, Markus. *openArchitectureWare User Guide, Version 4.2*. http://www.eclipse.org/gmt/oaw/doc/4.2/openArchitectureWare-42-reference.pdf. 2007

[EL07] Eberling, Werner ; Lessner, Jan: *Enterprise JavaBeans 3*. 1. Mnchen u.a. : Carl Hanser Verlag, 2007

[Fai94] Fairley, Richard: Risk Management for Software Projects. In: *IEEE Software* 11 (1994), Nr. 3, S. 57–67

[Fej08] Fejes, Balasz. *Test Web Applications with HttpUnit*. `http://www.javaworld.com/javaworld/jw-04-2004/jw-0419-httpunit.html?page=1`. Abrufdatum: 28.03.2008

[Fow05] Fowler, Martin: *Refactoring - Oder wie Sie das Design vorhandener Software verbessern*. 2. Mnchen : Addison-Wesley, 2005

[GHJV96] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 3. Bonn : Addison Wesley, 1996

[Hen06] Henning, Michi: The Rise and Fall of CORBA. In: *ACM Queue: Component Technologies* 4 (2006), Nr. 5

[Hol06] Holmes, James: *Struts: The Complete Reference*. 2. McGraw-Hill Osborne Media, 2006

[HP08a] HP. *HP Functional Testing*. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_home.jsp?zn=bto&cp=1_4011_100__. Abrufdatum: 28.03.2008

[HP08b] HP. *HP LoadRunner*. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_home.jsp?zn=bto&cp=1_4011_100__. Abrufdatum: 28.03.2008

[HS06] Holmes, James ; Schalk, Chris: *JavaServer Faces: The Complete Reference (Complete Reference Series)*. 1. McGraw-Hill Osborne Media, 2006

[IBM07a] IBM. *Rational ClearCase Change Management Solution*. http://www-306.ibm.com/software/awdtools/changemgmt/. Abrufdatum: 17.07.2007

[IBM07b] IBM. *Rational Suite*. http://www-306.ibm.com/software/awdtools/suite/. Abrufdatum: 17.07.2007

[IFP08] IFPUG. *International Function Point User Group*. http://www.ifpug.org/. Abrufdatum: 28.03.2008

[Int07] Interface21. *Spring Framework*. http://www.springframework.org/. Abrufdatum: 17.07.2007

[JHA+05] Johnson, Rod ; Hoeller, Juergen ; Arendsen, Alef ; Risberg, Thomas ; Kopylenko, Dmitriy: *Professional Java Development with the Spring Framework*. 1. Birmingham : Wrox Press Ltd., 2005

[JMS06] Juric, Matjaz ; Mathew, Benny ; Sarang, Poornachandra: *Business Process Execution Language for Web Services*. 2. Packt Publishing, 2006

[Jun02] Jungmayr, Stefan: Design for Testability. In: *Proceedings of CONQUEST 2002*, 2002, S. 57–64

[Jun04] Jungmayr, Stefan: *Improving testability of object-oriented systems*. dissertation.de, 2004. – ISBN 3–89825–781–9

[JUn08] JUnit.org. *JUnit, Testing Ressources for Extreme Programming*. http://www.junit.org/. Abrufdatum: 28.03.2008

[KBS05] Krafzig, Dirk ; Banke, Karl ; Slama, Dirk: *Enterprise SOA - Service Oriented Architecture Best Practices*. 1. New Jersey : Prentice Hall, 2005

[Lam07] Lamb, David A. *Software Engineering Archives*. http://www.cs.queensu.ca/Software-Engineering. Abrufdatum: 17.07.2007

[LH05] Liberty, Jesse ; Hurwitz, Dan: *Programming ASP.NET*. 3. O'Reilly Media, Inc., 2005

[Lin00] Linthicum, David S.: *Enterprise Application Integration*. 1. Boston et al. : Addison Wesley, 2000

[Lin05] Link, Johannes: *Softwaretests mit JUnit*. dpunkt.verlag, 2005

[LMK04] Lembeck, Christoph ; Müller, Roger A. ; Kuchen, Herbert: Testfallerzeugung mit einer symbolischen virtuellen Maschine und Constraint Solvern. In: *GI Jahrestagung (2)* Bd. 51, GI, 2004, S. 418–427

[Loh03] Lohrer, Matthias: *Einstieg in ASP.NET*. 1. Galileo Computing, 2003

[LP98] Linnhoff-Popien, Claudia: *CORBA - Kommunikation und Management*. 1. Berlin et al. : Springer, 1998

[LR06] Lahres, Bernhard ; Rayman, Gregor: *Praxisbuch Objektorientierung – Von den Grundlagen zur Umsetzung*. 1. Bonn : Galileo Press, 2006

[ME00] Melton, Jim ; Eisenberg, Andrew: *Understanding SQL and Java Together - A Guide to SQLJ, JDBC, and Related Technologies*. 1. San Francisco : Morgan Kaufmann, 2000

[Mic07a] Microsoft. *ASP.NET*. `http://www.asp.net/`. Abrufdatum: 17.07.2007

[Mic07b] Microsoft. *Microsoft .NET Homepage*. `http://www.microsoft.com/net/default.aspx`. Abrufdatum: 17.07.2007

[Mid07] Red Hat Middleware. *NHibernate for .NET*. `http://www.hibernate.org/343.html`. Abrufdatum: 17.07.2007

[MLK04] Müller, Roger A. ; Lembeck, Christoph ; Kuchen, Herbert: A symbolic Java virtual machine for test case generation. In: *IASTED Conf. on Software Engineering*, 2004, S. 365–371

[NUn08] NUnit.org. *NUnit*. `http://www.nunit.org/`. Abrufdatum: 11.04.2008

[OAS07a] OASIS - Organization for the Advancement of Structured Information Standards. *Web Services Business Process Execution Language Version 2.0*. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`. Abrufdatum: 19.09.2007

[OAS07b] OASIS - Organization for the Advancement of Structured Information Standards. *Web Services for Remote Portlets Version 1.0 - Standard Specification.* `http://www.oasis-open.org/committees/download.php/3343/oasis-200304-wsrp-specification-1.0.pdf`. Abrufdatum: 19.09.2007

[OAS07c] OASIS - Organization for the Advancement of Structured Information Standards. *Web Services Security: SOAP Message Security 1.1 - Standard Specification.* `http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf`. Abrufdatum: 19.09.2007

[Obj07] Object Management Group. *CORBA.* `http://www.corba.org/`. Abrufdatum: 17.07.2007

[Oeh00] Oehler, Karsten: *OLAP - Grundlagen, Modellierung und betriebswirtschaftliche Lï¿½ungen.* 1. Mnchen/ Wien : Verlag Hanser, 2000

[Oes05] Oestereich, Bernd: *Analyse und Design mit UML 2 - Objektorientierte Softwareentwicklung.* 7. Mnchen : Oldenbourg, 2005

[Omo07] Omondo. *EclipseUML.* `http://www.eclipsedownload.com/`. Abrufdatum: 17.07.2007

[OOS08] OOSE Innovative Informatik. *UML Tools.* `http://www.oose.de/umltools.htm`. Abrufdatum: 05.03.2008

[ope08] openQA. *Selenium.* `http://selenium.openqa.org/`. Abrufdatum: 28.03.2008

[PB05] Poensgen, Benjamin ; Bock, Bertram: *Function-Point-Analyse.* 1. Heidelberg : dpunkt.verlag, 2005

[Por07] Portland Pattern Repository. *Anti-Patterns - definition and catalog.* `http://c2.com/cgi/wiki?AntiPattern`. Abrufdatum: 19.09.2007

[Pro08] Project, The J. *Jakarta Cactus.* `http://jakarta.apache.org/cactus/`. Abrufdatum: 28.03.2008

[Rai07] Raible, Matt. *Comparing Web Frameworks.* `https://equinox.dev.java.net/framework-comparison/WebFrameworks.pdf`. Abrufdatum: 17.07.2007

[Rat07] Rational Software Corporation. *Rational Unified Process: Best Practices for Software Development Teams.* `http:`

//www.augustana.ab.ca/~mohrj/courses/2000.winter/csc220/papers/rup_best_practices/rup_bestpractices.pdf. Abrufdatum: 19.09.2007

[Rat08a] Rational, IBM. *Rational Performance Tester*. http://www-306.ibm.com/software/de/rational/. Abrufdatum: 28.03.2008

[Rat08b] Rational, IBM. *Rational TestManager*. http://www-306.ibm.com/software/de/rational/. Abrufdatum: 28.03.2008

[RB98] Rubin, William ; Brain, Marshall: *Understanding DCOM*. 1. New Jersey : Prentice Hall, 1998

[SBS06] Sriganesh, Rima P. ; Brose, Gerald ; Silverman, Micah: *Mastering Enterprise JavaBeans 3.0*. 1. Indianapolis : Wiley Publishing Inc., 2006

[Sch07] Schwaber, Carey. *The Forrester Wave: Software Change And Configuration Management*. http://www.collab.net/forrester_wave_report/index.html. Abrufdatum: 17.07.2007

[SGM02] Szyperski, Clemens ; Gruntz, Dominik ; Murer, Stephan: *Component software: beyond object-oriented programming*. 2. London u.a. : Addison-Wesley, 2002

[Sou08] SourceForge. *QuickCheck for Java*. http://sourceforge.net/projects/qc4j/. Abrufdatum: 28.03.2008

[Sta07] StarUML. *StarUML*. http://staruml.sourceforge.net/en/. Abrufdatum: 17.07.2007

[Stï05] Stï¿½rle, Harald: *UML 2 fr Studenten*. 1. Mnchen : Pearson Studium, 2005

[Sun07a] Sun Microsystems. *Java EE at a Glance*. http://java.sun.com/javaee/. Abrufdatum: 17.07.2007

[Sun07b] Sun Microsystems. *JavaServer Faces Technology*. http://java.sun.com/javaee/javaserverfaces/. Abrufdatum: 17.07.2007

[Sun07c] Sun Microsystems. *The Java EE 5 Tutorial*. http://java.sun.com/javaee/5/docs/tutorial/doc/. Abrufdatum: 19.09.2007

[Sun07d] Sun Microsystems. *Java Message Service - Version 1.1, Documentation*. http://java.sun.com/products/jms/docs.html. Abrufdatum: 19.09.2007

[SV06] Stahl, Thomas ; Völter, Markus: *Model-driven Software Development: Technology, Engineering, Management*. Wiley, 2006

[Tom08] Tomography, Software. *Sotograph*. `http://www.software-tomography.de/html/sotograph.html`. Abrufdatum: 28.03.2008

[Ton06] Tong, Kent Ka L.: *Enjoying Web Development with Tapestry*. 1. `Lulu.com`, 2006

[Wat08] Watij. *Watij – Web Application Testing in Java*. `http://watij.com/`. Abrufdatum: 28.03.2008

[Wel07] Wells, Don. *Extreme Programming: A gentle introduction*. `http://www.extremeprogramming.org/index.html`. Abrufdatum: 19.09.2007

[Wik07] Wikipedia. *Comparison of revision control software*. `http://en.wikipedia.org/wiki/Comparison_of_revision_control_software`. Abrufdatum: 17.07.2007

[Wol08] Wolffgang, Ulrich: *Modellgetriebene Entwicklung von Webapplikationen*, University of Münster, Department of Information Systems, Diplomarbeit, 2008

[Wor07a] World Wide Web Consortium (W3C). *Web Service Glossary, February 2004*. `http://www.w3.org/TR/ws-gloss/`. Abrufdatum: 19.09.2007

[Wor07b] World Wide Web Consortium (W3C). *XML Encryption Syntax and Processing, December 2002*. `http://www.w3.org/TR/xmlenc-core/`. Abrufdatum: 19.09.2007

[Wor07c] World Wide Web Consortium (W3C). *XML-Signature Syntax and Processing, February 2002*. `http://www.w3.org/TR/xmldsig-core/`. Abrufdatum: 19.09.2007

[Zep04] Zeppenfeld, Klaus: *Objektorientierte Programmiersprachen*. 1. Spektrum Akademischer Verlag, 2004

[Zï¿½06] Zï¿½ner, Stefan: *Portlets - Portalkomponenten in Java*. 1. Frankfurt a. M. : Entwickler.Press, 2006

Förderkreis der Angewandten Informatik
an der Westfälischen Wilhelms-Universität Münster e.V.

Einsteinstraße 62
D-48149 Münster
Tel.: +49 251 83 33797
Fax : +49 251 83 33755

ISSN 1868-0801