

Helmut Balzert

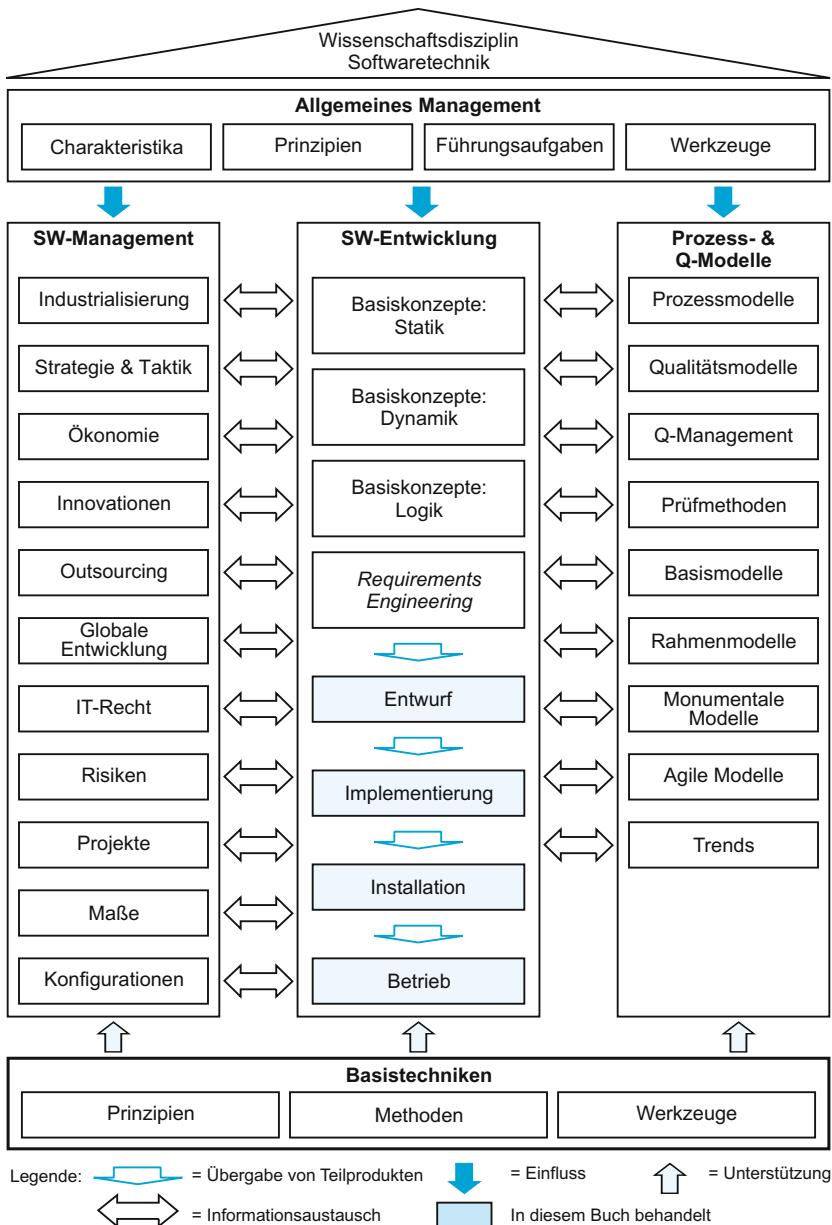
Lehrbuch der Softwaretechnik

Entwurf, Implementierung, Installation und Betrieb

3. Auflage

Lehrbücher der Informatik

Spektrum
AKADEMISCHER VERLAG



Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb

Lehrbücher der Informatik

Herausgegeben von
Prof. Dr.-Ing. habil. Helmut Balzert

Helmut Balzert
Lehrbuch der Softwaretechnik:
Softwaremanagement
2. Auflage

Helmut Balzert
Lehrbuch der Softwaretechnik:
Basiskonzepte und Requirements
Engineering
3. Auflage

Heide Balzert
Lehrbuch der Objektmodellierung
Analyse und Entwurf mit der UML 2, 2. Auflage

Helmut Balzert
Lehrbuch Grundlagen der Informatik
Konzepte und Notationen in UML 2, Java 5, C# und C++,
Algorithmik und Softwaretechnik, Anwendungen, 2. Auflage

Klaus Zeppenfeld
Lehrbuch der Grafikprogrammierung
Grundlagen, Programmierung, Anwendung

Helmut Balzert
Objektorientierte Programmierung
mit Java 5

Helmut Balzert

Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb

3. Auflage

Unter Mitwirkung von
Peter Liggesmeyer
Holger Schwichtenberg

Spektrum
AKADEMISCHER VERLAG

Autor

Prof. Dr. Helmut Balzert
E-Mail: hb@W3L.de

Weitere Informationen zum Buch finden Sie unter
www.spektrum-verlag.de/978-3-8274-1706-0

Wichtiger Hinweis für den Benutzer

Der Verlag und der Autor haben alle Sorgfalt walten lassen, um vollständige und akkurate Informationen in diesem Buch zu publizieren. Der Verlag übernimmt weder Garantie noch die juristische Verantwortung oder irgendeine Haftung für die Nutzung dieser Informationen, für deren Wirtschaftlichkeit oder fehlerfreie Funktion für einen bestimmten Zweck. Ferner kann der Verlag für Schäden, die auf einer Fehlfunktion von Programmen oder ähnliches zurückzuführen sind, nicht haftbar gemacht werden. Auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultieren. Eine telefonische oder schriftliche Beratung durch den Verlag über den Einsatz der Programme ist nicht möglich. Der Verlag übernimmt keine Gewähr dafür, dass die beschriebenen Verfahren, Programme usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften. Der Verlag hat sich bemüht, sämtliche Rechteinhaber von Abbildungen zu ermitteln. Sollte dem Verlag gegenüber dennoch der Nachweis der Rechtsinhaberschaft geführt werden, wird das branchenübliche Honorar gezahlt.

Bibliografische Information der Deutschen Nationalbibliothek

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

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

3. Auflage 2011

© Spektrum Akademischer Verlag Heidelberg 2011

Spektrum Akademischer Verlag ist ein Imprint von Springer

11 12 13 14 15 5 4 3 2 1

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Planung und Lektorat: Dr. Andreas Rüdinger

Redaktion und Gestaltung: Miriam Platte, Witten

Herstellung: Crest Premedia Solutions (P) Ltd, Pune, Maharashtra, India

Umschlaggestaltung: SpieszDesign, Neu-Ulm

Titelbild: Anna Solecka-Zach: „Ohne Titel“ (1995)

Satz: W3L GmbH, Witten – automatischer Satz aus der W3L-Plattform.

ISBN 978-3-8274-1706-0

Vorwort zur 3. Auflage

Die Softwaretechnik bildet einen Grundpfeiler der Informatik. Sie hat sich zu einem umfassenden Wissenschaftsgebiet entwickelt. Um Ihnen, liebe Leserin, lieber Leser, ein optimales Erlernen dieses Gebietes zu ermöglichen, habe ich – zusammen mit Koautoren – ein dreibändiges Lehr- und Lernbuch über die Softwaretechnik geschrieben. Es behandelt die Kerngebiete »Basiskonzepte und Requirements Engineering« (Band 1), »Entwurf, Implementierung, Installation und Betrieb« (Band 2) und »Softwaremanagement« (Band 3) sowie die Abhängigkeiten zwischen diesen Gebieten.

Ich habe das Gebiet der Softwaretechnik in mehrere große Bereiche gegliedert – der Tempel der Softwaretechnik (Abb. 0.0-1) veranschaulicht diese Gliederung. Ziel jeder Softwareentwicklung ist es, ein lauffähiges Softwareprodukt zu erstellen, zu warten und zu pflegen. Im Mittelpunkt der Gliederung steht daher die Softwareentwicklung (mittlere Säule). Jede der Aktivitäten der Softwareentwicklung trägt dazu bei, Teilprodukte zu erstellen, die dann in ein Gesamtprodukt münden.

Eine Softwareentwicklung läuft aber *nicht* von alleine ab. Sie basiert auf und nutzt Basistechniken, das sind Prinzipien, Methoden, Werkzeuge, *Best Practices*.

Die eigentliche Softwareentwicklung wird durch die zwei Säulen »Softwaremanagement« und »Prozess- und Qualitätsmodelle« und das Dach »Allgemeines Management« eingerahmt.

Bei den Aktivitäten des Managements und der Prozess- & Qualitätssicherung handelt es sich um *begleitende* Aktivitäten, deren Ergebnisse aber selbst *nicht* Bestandteil des Endprodukts sind. Dennoch sind sie eminent wichtig – was die große Anzahl fehlgeschlagener Softwareprojekte deutlich zeigt.

In diesem Buch werden der Entwurf, die Implementierung, die Installation und der Betrieb einer Anwendungssoftware behandelt. Dieses Buch besteht aus folgenden Teilen:

I Der Entwurf

II Die Implementierung

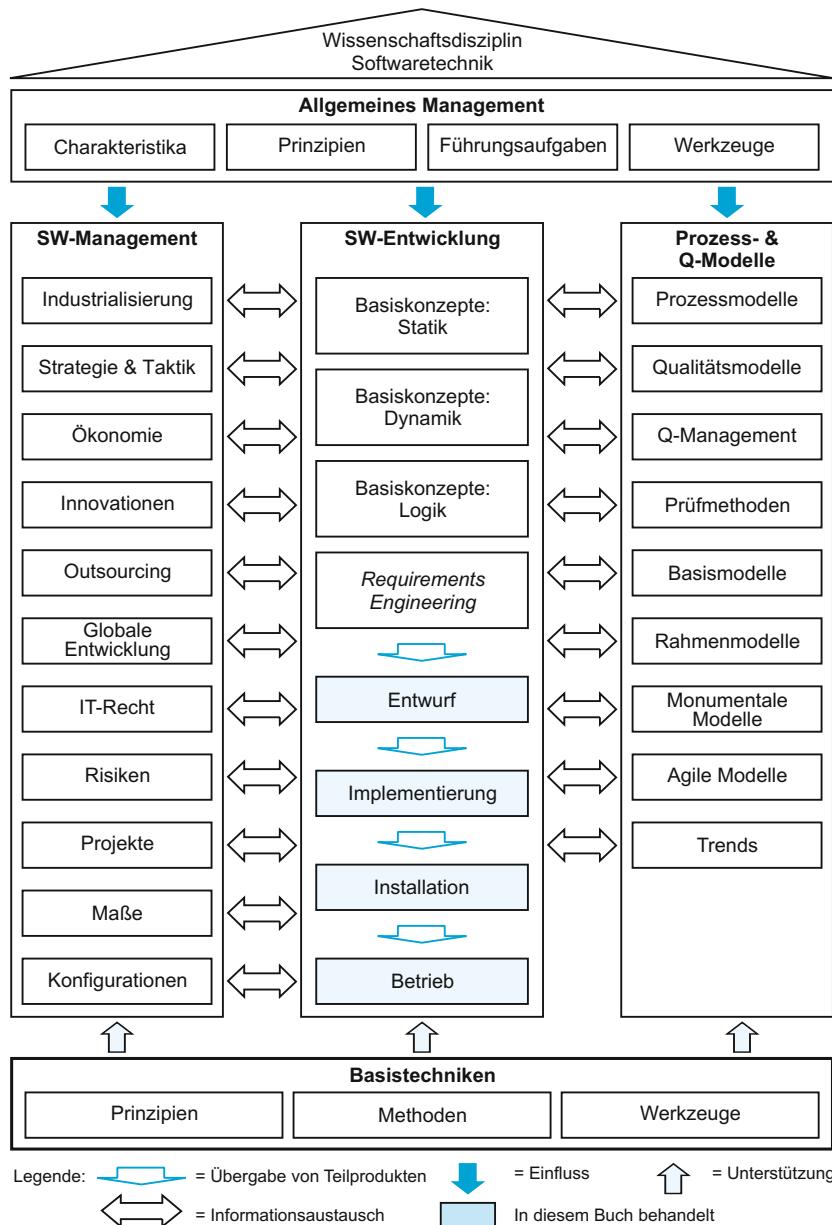
III Verteilung, Installation, Abnahme und Einführung

IV Der Betrieb

Aufbau &
Gliederung

Vorwort zur 3. Auflage

Abb. 0.0-1:
Gliederung der
Softwaretechnik.



Eine Softwareentwicklung darf nicht isoliert gesehen werden, sondern es muss immer der gesamte Software-Lebenszyklus betrachtet werden. Daher beginnt dieses Buch mit einer Betrachtung des Lebenszyklus:

■ »Der Software-Lebenszyklus«, S. 1

Vorwort zur 3. Auflage

Im Mittelpunkt dieses Buches steht der Entwurf von Softwarearchitekturen. Dabei wird eine ganzheitliche Sicht eingenommen. Die Kunst besteht darin, eine Softwarearchitektur zu entwerfen, die die funktionalen und nichtfunktionalen Anforderungen unter Berücksichtigung von Architekturprinzipien, Architektur- und Entwurfsmustern sowie weiteren Einflussfaktoren erfüllt. Dabei sind vielfältige Abhängigkeiten zu berücksichtigen. Nicht umsonst wird daher der Entwurf auch als »Königsdisziplin« der Softwaretechnik bezeichnet.

Entwurf im
Mittelpunkt

Erfahrungen haben gezeigt, dass viele Studierende der Informatik zwar in der Lage sind, »Entwürfe im Kleinen« vorzunehmen, aber oft überfordert sind, eine globale Softwarearchitektur zu entwerfen. Betrachtet man heutige Lehrbücher zur Softwarearchitektur, dann stellt man zwei Extreme fest:

- Es gibt Lehrbücher, die die Softwarearchitektur sehr abstrakt und auf einer Modellebene betrachten und beschreiben.
- Auf der anderen Seite gibt es Lehrbücher, die sehr konkret spezifische plattformorientierte Architekturen vorstellen und auch die zugehörige Programmierung dazu zeigen, zum Beispiel Lehrbücher zu Java EE (*Enterprise Edition*) oder .Net.

2 Extreme

Beide Wege sind *nicht* zufriedenstellend. Bei der abstrakten Behandlung von Softwarearchitekturen fehlt die eigene praktische Erfahrung für die Umsetzung der Konzepte. Bei der Behandlung einer plattformspezifischen Architektur nimmt die Plattformarchitektur dem Entwerfer viel Arbeit ab, ohne dass er sich über die einzelnen Aspekte jeweils im Klaren ist. Außerdem werden Architekturalternativen meist nicht behandelt.

In diesem Buch wird daher versucht, einen Mittelweg zwischen beiden Extremen einzuschlagen. Ausgehend von globalen Architekturmustern werden zunächst Einzelaspekte mit ihren Alternativen behandelt. Damit immer der Bezug zur Realität vorhanden ist, wird eine durchgängige Fallstudie in verschiedenen Varianten zunächst für Einzelaspekte entworfen und implementiert. Dadurch wird es auch möglich, gute Softwarearchitekturen zu entwerfen, auch wenn keine Standardplattform, wie z. B. Java EE, zur Verfügung steht, nicht geeignet ist oder nicht benötigt wird:

Mittelweg

- »Der Entwurf«, S. 5

Natürlich ist es in einem Lehrbuch nicht möglich, alle Aspekte einer Softwarearchitektur im Detail zu behandeln. Zur Implementierung der Fallstudie wird daher immer die Programmiersprache Java einschließlich verfügbaren Java-Frameworks verwendet. Neben der Java EE-Plattform wird aber auch die .Net-Plattform behandelt. Auf architektonische Unterschiede wird eingegangen. Zusätzlich werden die Besonderheiten bei softwareintensiven Systemen behandelt.

Vorwort zur 3. Auflage

Programme zum
Herunterladen

Im kostenlosen E-Learning-Kurs zu diesem Buch finden Sie alle Programme zum Herunterladen sowie Beschreibungen zu den eingesetzten Entwicklungswerkzeugen und Plattformen.

Auf die Programmierung bzw. Implementierung von entworfenen Softwarekomponenten wird nur ein kurzer Blick geworfen, da die meisten Studierenden der Informatik sich mit diesem Teil der Softwareentwicklung meist intensiv auseinandergesetzt haben:

- »Die Implementierung«, S. 491

Die Verteilung und Installation – im Englischen *Deployment* genannt – von Komponenten eines Softwaresystems auf Client- und Serversysteme spielt heute eine wichtige Rolle. Die Anforderungen an die Effizienz bei der Verteilung und Installation haben Rückwirkungen auf die Softwarearchitektur:

- »Verteilung, Installation, Abnahme und Einführung«, S. 519

Früher ging man davon aus, dass Softwaresysteme nur eine begrenzte Lebensdauer haben. Man war der Meinung, wenn ein Softwaresystem den Anforderungen nicht mehr genügt, dass man es dann durch ein neues Softwaresystem abgelöst.

Die Erfahrung hat gezeigt, dass der Aufwand, um Altsysteme durch vollständig neue Systeme zu ersetzen, oft so groß ist, dass ein Austausch nicht rentabel ist. Daher muss man – wie auch bei Städten mit alten und neuen Stadtteilen – in der Lage sein, Altsysteme in neue Umgebungen einzubetten. Eine Konsequenz aus dieser Erkenntnis ist außerdem, dass die Weiterentwickelbarkeit von neuen Systemen heute eine wesentlich größere Priorität besitzt.

Daher spielt der Betrieb von Softwaresystemen einschließlich der Wartung, Pflege und Weiterentwicklung heute eine große Rolle, mit Rückwirkungen auf die Softwarearchitektur:

- »Der Betrieb«, S. 529

Zur 3. Auflage

Die Inhalte der 2. Auflage wurden in zwei Bände aufgeteilt, um die Bücher handlicher und zielgruppenorientierter zu gestalten. Im Gegensatz zur 2. Auflage ist die 3. Auflage unabhängig von einem Vorgehensmodell.

Viele Themen, die in der 2. Auflage von 2001 behandelt wurden, können heute als bekannt vorausgesetzt werden, z. B. Einführung in relationale Datenbanksysteme und SQL. Objektorientierte Datenbanksysteme haben sich nicht durchgesetzt und werden daher in diesem Buch *nicht* mehr behandelt. Ebenso werden die heute veralteten Methoden SD (*Structured Design*) und MD (*Modular Design*) *nicht* mehr betrachtet.

An diesen Beispielen sieht man auch die dynamische Entwicklung der Softwaretechnik und insbesondere der Softwarearchitekturen. Sehr viel mehr Wert wird daher in diesem Buch auf Architektur- und Entwurfsmuster gelegt.

Vorwort zur 3. Auflage

Kapitel aus der 2. Auflage, die nicht mehr in dieser neuen Auflage berücksichtigt werden konnten, finden Sie als *Open Content* auf der Website www.W3L.de/OpenContent.

Das Buch besteht aus 33 Kapiteln. Die meisten Kapitel enthalten Unterkapitel. Am Ende des Buches finden Sie ein umfangreiches Glossar, das Literaturverzeichnis und das Sachregister. Dadurch kann das Buch auch ideal als Nachschlagewerk benutzt werden.

Ziel der Didaktik ist es, einen Lehrstoff so zu strukturieren und aufzubereiten, dass der Lernende sich leicht ein mentales Modell von dem Lehrstoff aufbauen kann und genügend Übungsmöglichkeiten erhält, um zu überprüfen, ob er den Lehrstoff – nach der Beschäftigung mit ihm – entsprechend den vorgegebenen Lernzielen beherrscht. Dieses didaktische Ziel habe ich versucht in diesem Lehrbuch umzusetzen. Die Übungsmöglichkeiten befinden sich jedoch nicht im Lehrbuch, sondern in dem ebenfalls verfügbaren E-Learning-Kurs (siehe unten).

In den meisten Lehrbüchern wird die Welt so erklärt, wie sie ist – ohne dem Lernenden vorher die Möglichkeit gegeben zu haben, über die Welt nachzudenken. Ich stelle daher in vielen Kapiteln Fragen an Sie. Diese Fragen sollen Sie dazu anregen, über ein Thema nachzudenken. Erst nach dem Nachdenken sollten Sie weiter lesen. (Vielleicht sollten Sie die Antwort nach der Frage zunächst durch ein Papier abdecken).

Die Softwaretechnik ist komplex. Es gibt viele gegenseitigen Abhängigkeiten. Um diese Abhängigkeiten zu verdeutlichen, enthält dieses Buch viele (absolute) Querverweise auf andere Kapitel, damit Sie sich mit verschiedenen Perspektiven auf ein Themengebiet befassen können.

Dieses Buch kann zur Vorlesungsbegleitung, zum Selbststudium und zum Nachschlagen verwendet werden. Um den Umfang und die Kosten des Buches zu begrenzen, enthält dieses Buch *keine* Tests und Aufgaben. Für diejenigen Leser unter Ihnen, die Ihr Wissen durch Tests und Aufgaben aktiv überprüfen möchten, gibt es einen (kostenpflichtigen) E-Learning-Online-Kurs, der zusätzlich zahlreiche Animationen und Selbsttestaufgaben mit Musterlösungen enthält. Mentoren und Tutoren betreuen Sie bei der Bearbeitung von Tests und Aufgaben. Das Bestehen eines Abschlussstests und einer Abschlussklausur wird durch Zertifikate dokumentiert. Dieser Online-Kurs ist Bestandteil des Online-Bachelor-Studiengangs »Web- und Medieninformatik« der FH Dortmund und dem Berufsprofil »Software-Architekt/in (FH Dortmund)« im Rahmen der »Wissenschaftlichen Informatik-Weiterbildung Online«. Sie finden Informationen dazu und den E-Learning-Kurs auf der W3L-Plattform (<http://www.W3L.de>).

Buchaufbau

Didaktik & Methodik

Querverweise

Einsatz des Buches

Vorwort zur 3. Auflage

Kostenloser
E-Learning-Kurs

Ergänzend zu diesem Buch gibt es den kostenlosen E-Learning-Kurs »Nichtfunktionale Anforderungen«, der Installationsanleitungen und zusätzlich zahlreiche Tests enthält, mit denen Sie Ihr Wissen überprüfen können. Sie finden den Kurs auf der Website <http://Akademie.W3L.de>. Unter Startseite & Aktuelles finden Sie in der Box E-Learning-Kurs zum Buch den Link zum Registrieren. Nach der Registrierung und dem Einloggen geben Sie bitte die folgende Transaktionsnummer (TAN) ein: 5403882130.

Zielgruppen

Dieses Buch ist für folgende Zielgruppen geschrieben:

- Studierende der Informatik und Softwaretechnik an Universitäten, Fachhochschulen und Berufsakademien.
- Software-Ingenieure, Softwaremanager und Software-Qualitätsicherer in der Praxis.

Vorkenntnisse

Vorausgesetzt werden gute Kenntnisse der Programmiersprache Java, der Modellierungssprache UML, der objektorientierten Analyse sowie Erfahrungen bei der Entwicklung von Softwaresystemen.

Beispiele

Zur Vermittlung der Lerninhalte werden viele Beispiele verwendet. Um Ihnen diese unmittelbar kenntlich zu machen, sind sie in blauer Schrift gesetzt.

Visualisierung

Da ein Bild oft mehr aussagt als 1000 Worte, habe ich versucht, möglichst viele Sachverhalte zu veranschaulichen.

Hinweis

Viele der komplexen UML-Diagramme wurden mit einem UML-Werkzeug erstellt. Da dieses Werkzeug keine Vektorgrafiken erzeugt, ist die Darstellungsqualität dieser Grafiken im Buch nicht optimal. Um Übertragungsfehler zu vermeiden, wurde jedoch auf ein Neuzeichnen verzichtet.

Begriffe, Glossar
halbfett, blau

In diesem Lehrbuch wurde sorgfältig überlegt, welche Begriffe eingeführt und definiert werden. Ziel ist es, die Anzahl der Begriffe möglichst gering zu halten. Alle wichtigen Begriffe sind im Text halbfett und blau gesetzt. Die so markierten Begriffe sind am Ende des Buches in einem Glossar alphabetisch angeordnet und definiert. Dabei wurde oft versucht, die Definition etwas anders abzufassen, als es im Text der Fall war, um Ihnen noch eine andere Sichtweise zu vermitteln.

Als
Begleitunterlage &
zum
Selbststudium

Bücher können als Begleitunterlage oder zum Selbststudium ausgelegt sein. In diesem Buch versuche ich einen Mittelweg einzuschlagen. Ich selbst verwende das Buch als begleitende und ergänzende Unterlage zu meinen Vorlesungen. Viele Lernziele dieses Buches können aber auch im Selbststudium erreicht werden – insbesondere wenn ergänzend dazu der E-Learning-Kurs eingesetzt wird.

Englische vs.
deutsche Begriffe

Ein Problem für ein Informatikbuch stellt die Verwendung englischer Begriffe dar. Da die Wissenschaftssprache der Softwaretechnik Englisch ist, gibt es für viele Begriffe – insbesondere in Spezialgebieten – keine oder noch keine geeigneten oder üblichen deutschen

Vorwort zur 3. Auflage

Fachbegriffe. Auf der anderen Seite gibt es jedoch für viele Bereiche der Softwaretechnik sowohl übliche als auch sinnvolle deutsche Bezeichnungen, z.B. Entwurf für *Design*.

Da mit einem Lehrbuch auch die Begriffswelt beeinflusst wird, bemühe ich mich in diesem Buch, sinnvolle und übliche deutsche Begriffe zu verwenden.

Ist anhand des deutschen Begriffs *nicht* unmittelbar einsehbar oder allgemein bekannt, wie der englische Begriff lautet, dann wird in Klammern und kursiv der englische Begriff hinter dem deutschen Begriff aufgeführt. Dadurch wird auch das Lesen der englischsprachigen Literatur erleichtert.

Gibt es noch keinen eingebürgerten deutschen Begriff, dann wird der englische Originalbegriff verwendet. Englische Bezeichnungen sind immer *kursiv* gesetzt, sodass sie sofort ins Auge fallen.

Ziel der Buchgestaltung war es, Ihnen als Leser viele Möglichkeiten zu eröffnen, dieses Buch nutzbringend für Ihre eigene Arbeit einzusetzen. Sie können dieses Buch sequenziell von vorne nach hinten lesen. Die Reihenfolge der Kapitel ist so gewählt, dass die Voraussetzungen für ein Kapitel jeweils erfüllt sind, wenn man das Buch sequenziell liest.

Eine andere Möglichkeit besteht darin, jeweils eine der Teildisziplinen »Entwurf«, »Implementierung«, »Verteilung, Installation, Abnahme und Einführung« und »Betrieb« durchzuarbeiten. Auf Querbezüge und notwendige Voraussetzungen wird jeweils hingewiesen.

Außerdem kann das Buch themenbezogen gelesen werden. Möchte man sich in die Architektur- und Entwurfsmuster einarbeiten, dann kann man die dafür relevanten Kapitel durcharbeiten.

Durch das Buchkonzept ist es natürlich auch möglich, punktuell einzelne Kapitel durchzulesen, um eigenes Wissen zu erwerben, aufzufrischen und abzurunden, z.B. Durchlesen des Kapitels über Globalisierung.

Durch ein ausführliches Sachregister und Glossar kann dieses Buch auch gut zum Nachschlagen verwendet werden.

Ich habe versucht, ein innovatives wissenschaftliches Lehrbuch der Softwaretechnik zu schreiben. Ob mir dies gelungen ist, müssen Sie als Leser selbst entscheiden.

Ein Buch soll aber nicht nur vom Inhalt her gut sein, sondern Form und Inhalt sollten übereinstimmen. Daher wurde auch versucht, die Form anspruchsvoll zu gestalten.

Da ich ein Buch als »Gesamtkunstwerk« betrachte, ist auf der Buchtitelseite ein Bild der Malerin Anna Solecka-Zach abgedruckt. Sie setzt den Computer als Hilfsmittel ein, um ihre künstlerischen Vorstellungen umzusetzen.

Englische Begriffe
kursiv gesetzt

Lesen des Buches:
sequenziell

Nach
Teildisziplinen

Themenbezogen

Punktuell

Zum
Nachschlagen

Vorwort zur 3. Auflage

| | |
|-----------|---|
| Koautoren | <p>Die dynamische Entwicklung der Softwaretechnik und die Themenbreite machen es für einen Autor fast unmöglich, alleine ein Lehrbuch der Softwaretechnik zu schreiben. Ich habe daher zwei Kollegen gebeten, Teilgebiete durch eigene Beiträge abzudecken.</p> <p>Prof. Dr.-Ing. Peter Liggesmeyer, Technische Universität Kaiserslautern und Institutsleiter des Fraunhofer Instituts für Experimentelles Software Engineering, und seine Mitarbeiter Dr. Mario Trapp, Dr. Martin Becker und Thorsten Keuler haben das Buch durch Beiträge zu eingebetteten Systemen ergänzt:</p> <ul style="list-style-type: none">■ »Betriebssicherheit und Funktionssicherheit«, S. 121 (Ergänzung des Abschnitts zur Funktionssicherheit)■ »Zuverlässigkeit«, S. 124 (Ergänzung)■ »Architekturen »Eingebetteter Systeme««, S. 397 <p>Dr. Holger Schwichtenberg, Technologieberater und Softwarearchitekt, hat das Buch um Microsoft-spezifische Gesichtspunkte bereichert:</p> <ul style="list-style-type: none">■ »Transaktionen in .NET«, S. 186■ »Netzkommunikation in .NET«, S. 301■ »Die .NET-Plattform«, S. 360 (Kapitel und Unterkapitel)■ »Persistenz in .NET«, S. 442■ »GUIs in der .NET-Plattform«, S. 467 |
| Dank | <p>Ich danke den Koautoren für ihre Beiträge zu diesem Lehrbuch. Mein besonderer Dank gilt meinem wissenschaftlichen Mitarbeiter M.Sc. Michael Goll, der viele Beispiele in diesem Buch implementiert hat. Mein wissenschaftlicher Mitarbeiter M.Sc. Jakob Pytlik hat die Fallstudie zum Subsystem Benutzungsoberfläche (siehe »Fallstudie: Kundenverwaltung – GUI«, S. 458) konzipiert und programmiert. Danke dafür.</p> <p>Um nichts zu übersehen, habe ich zwei Fachleute gebeten, den Inhalt dieses Buches kritisch zu reflektieren und mir eine entsprechende Rückmeldung zu geben. Ich möchte daher folgenden Personen herzlich danken:</p> <ul style="list-style-type: none">■ Dr.-Ing. Olaf Zwintscher, Geschäftsführer der W3L GmbH in Witten und Lehrbeauftragter für Softwaretechnik an der Ruhr-Universität Bochum.■ Dr.-Ing. Doga Arinir, Leiter der Softwareentwicklung der W3L GmbH in Witten und Lehrbeauftragter für Softwaretechnik an der Ruhr-Universität Bochum. <p>Die Grafiken erstellte meine Mitarbeiterin Frau Anja Schartl. Danke!</p> <p>Trotz der Unterstützung vieler Personen bei der Erstellung dieses Buches enthält ein so umfangreiches Werk sicher immer noch Fehler und Verbesserungsmöglichkeiten: »<i>Nobody is perfect</i>«. Kritik und Anregungen sind daher jederzeit willkommen.</p> |

Vorwort zur 3. Auflage

Eine aktuelle Liste mit Korrekturen und Informationen zu diesem Buch finden Sie im kostenlosen E-Learning-Kurs zu diesem Buch (siehe oben).

Nach soviel Vorrede wünsche ich Ihnen nun viel Spaß beim Lesen. Möge Ihnen dieses Buch – trotz der manchmal »trockenen« Materie – ein wenig von der Faszination und Vielfalt der Softwaretechnik vermitteln. Werden Sie ein guter Softwareingenieur – Sie werden gebraucht!

Ihr

A handwritten signature in black ink, appearing to read "Helmut Balmer".

Inhalt

| | | |
|----------|---|-----|
| 1 | Der Software-Lebenszyklus | 1 |
| I | Der Entwurf | 5 |
| 2 | Artefakte | 9 |
| 3 | Verteilungsdiagramme | 11 |
| 4 | Fallstudie: KV – Überblick | 15 |
| 5 | Fallstudie: KV – Einzelplatz | 21 |
| 6 | Was ist eine Softwarearchitektur? | 23 |
| 7 | Architekturprinzipien | 29 |
| 7.1 | Architekturprinzip: Konzeptionelle Integrität | 30 |
| 7.2 | Architekturprinzip: Trennung von Zuständigkeiten | 31 |
| 7.3 | Architekturprinzip: Ökonomie | 32 |
| 7.4 | Architekturprinzip: Symmetrie | 33 |
| 7.5 | Architekturprinzip: Sichtbarkeit | 34 |
| 7.6 | Architekturprinzip: Selbstorganisation | 34 |
| 8 | Architektur- und Entwurfsmuster | 37 |
| 8.1 | Exkurs: <i>Callback</i> | 42 |
| 8.2 | Das Schichten-Muster (<i>layers pattern</i>) | 46 |
| 8.3 | Das Beobachter-Muster (<i>observer pattern</i>) | 54 |
| 8.4 | Das MVC-Muster (<i>model view controller pattern</i>) | 62 |
| 8.5 | Das Fassaden-Muster (<i>facade pattern</i>) | 69 |
| 8.6 | Das Kommando-Muster (<i>command pattern</i>) | 75 |
| 8.7 | Das Proxy-Muster (<i>proxy pattern</i>) | 83 |
| 8.8 | Fabrikmethoden-Muster (<i>factory method pattern</i>) | 89 |
| 8.9 | Das Strategie-Muster (<i>strategy pattern</i>) | 96 |
| 8.10 | Das Brücken-Muster (<i>bridge pattern</i>) | 103 |
| 9 | Nichtfunktionale Anforderungen | 109 |
| 9.1 | Wartbarkeit | 116 |
| 9.2 | Weiterentwickelbarkeit | 119 |
| 9.3 | Betriebssicherheit und Funktionssicherheit | 121 |
| 9.4 | Zuverlässigkeit | 124 |
| 9.5 | Leistung und Effizienz | 128 |
| 9.6 | Benutzbarkeit | 130 |

Inhalt

| | |
|-----------|---|
| 9.7 | Portabilität 132 |
| 10 | Einflussfaktoren auf die Architektur 135 |
| 11 | Globalisierung von Software 143 |
| 11.1 | Globalisierung in Java 145 |
| 11.2 | Fallstudie: KV – Globalisiert 151 |
| 12 | Authentifizierung und Autorisierung 153 |
| 12.1 | A & A-Entwurfsmuster 156 |
| 12.2 | JAAS 161 |
| 12.3 | Fallstudie: KV – JAAS 167 |
| 13 | Transaktionen 177 |
| 13.1 | Transaktionsverarbeitung 178 |
| 13.2 | Fallstudie: KV – JTA 180 |
| 13.3 | Transaktionen in .NET 186 |
| 14 | Verteilte Architekturen 191 |
| 14.1 | Client-Server-Architektur 193 |
| 14.2 | Web-Architektur 196 |
| 14.3 | SOA – Serviceorientierte Architekturen 201 |
| 15 | Arten der Netzkomunikation 205 |
| 15.1 | <i>Sockets</i> 207 |
| 15.1.1 | TCP-Sockets 208 |
| 15.1.2 | UDP-Sockets 213 |
| 15.1.3 | Fallstudie: KV – Sockets 216 |
| 15.2 | RMI 220 |
| 15.2.1 | »Hello World« mit Java-RMI 222 |
| 15.2.2 | RMI-Begriffe 229 |
| 15.2.3 | Stummel-Objekte (<i>stubs</i>) 231 |
| 15.2.4 | Fallstudie: KV – RMI 236 |
| 15.3 | CORBA 241 |
| 15.3.1 | Die Architektur von CORBA 242 |
| 15.3.2 | Die Schnittstellendefinitionssprache IDL 244 |
| 15.3.3 | Standardisierte CORBA-Dienste 245 |
| 15.3.4 | Fallstudie: KV – CORBA in Java 246 |
| 15.4 | XML-RPC 253 |
| 15.4.1 | Fallstudie: KV – XML-RPC 256 |
| 15.5 | SOAP 262 |
| 15.5.1 | SOAP-Nachrichten, Webservices und WSDL 263 |
| 15.5.2 | Webservices bereitstellen und nutzen mit JAX-WS 267 |
| 15.5.3 | Webservices nutzen 274 |
| 15.5.4 | UDDI 280 |
| 15.5.5 | Fallstudie: KV – SOAP 283 |

| | | |
|--------|--|-----|
| 15.6 | REST | 286 |
| 15.6.1 | Die Konzepte von REST | 287 |
| 15.6.2 | Webservices mit JAX-RS | 289 |
| 15.6.3 | Fallstudie: KV – REST | 297 |
| 15.7 | Netzkommunikation in .NET | 301 |
| 15.8 | Entwurfskonzepte für verteilte Anwendungen | 306 |
| 15.8.1 | Fabrik-Dienst | 307 |
| 15.8.2 | Wert-Objekte | 309 |
| 15.8.3 | Fassaden | 312 |
| 15.9 | Vergleich der Konzepte | 315 |

16 Softwaretechnische Infrastrukturen 319

| | | |
|--------|--|-----|
| 16.1 | Anforderungen an Unternehmensanwendungen | 319 |
| 16.2 | Die Java EE-Plattform | 321 |
| 16.2.1 | Die Java EE-Architektur | 321 |
| 16.2.2 | EJBs | 326 |
| 16.2.3 | JNDI | 328 |
| 16.2.4 | Erstellung und Nutzung einer <i>Session Bean</i> | 330 |
| 16.2.5 | Fallstudie: KV mit Java EE als Client-Server-Anwendung | 333 |
| 16.2.6 | Fallstudie: KV mit Java EE als Web-Anwendung | 351 |
| 16.3 | Die .NET-Plattform | 360 |
| 16.3.1 | Eigenschaften des .NET Framework | 361 |
| 16.3.2 | Zur .NET-Architektur | 368 |
| 16.3.3 | Werkzeuge | 372 |
| 16.4 | Infrastrukturen für serviceorientierte Architekturen | 375 |
| 16.5 | Fallstudie: KV mit Java EE als Webservice | 377 |

17 Architekturen »Eingebetteter Systeme« 397

| | | |
|--------|--|-----|
| 17.1 | Anforderungsspezifikation | 398 |
| 17.2 | Typische Sichten Eingebetteter Systeme | 400 |
| 17.3 | Architekturmuster für Eingebettete Systeme | 406 |
| 17.3.1 | Das PSC-Muster (<i>protected single channel pattern</i>) | 407 |
| 17.3.2 | Das <i>Homogeneous Redundancy</i> -Muster | 409 |

18 Das Subsystem Applikation 413

| | | |
|------|---|-----|
| 18.1 | Web-Architektur für das Subsystem Applikation | 414 |
|------|---|-----|

19 Das Subsystem Persistenz 421

| | | |
|------|---|-----|
| 19.1 | Vom Direktzugriff bis zum JPA | 422 |
| 19.2 | Exkurs: ORM – Objektrelationale Abbildung | 426 |
| 19.3 | JPA – <i>Java Persistence API</i> | 431 |
| 19.4 | Fallstudie: KV – JPA | 436 |
| 19.5 | Persistenz in .NET | 442 |

Inhalt

| | | |
|------------|---|-----|
| 20 | Das Subsystem Benutzungsoberfläche | 449 |
| 20.1 | GUI-Entwurfsmuster MVP | 457 |
| 20.2 | Fallstudie: Kundenverwaltung – GUI | 458 |
| 20.3 | GUIs in der .NET-Plattform | 467 |
| 21 | Der Entwurfsprozess | 481 |
| 22 | Qualitätssicherung der Architektur | 487 |
| II | Die Implementierung | 491 |
| 23 | Implementierungsprinzipien | 495 |
| 24 | Schnittstellen, Fabriken und Komposition | 503 |
| 25 | Restrukturieren (<i>refactoring</i>) | 511 |
| III | Verteilung, Installation, Abnahme und Einführung | 519 |
| 26 | Verteilung und Installation | 521 |
| 27 | Abnahme und Einführung | 525 |
| IV | Der Betrieb | 529 |
| 28 | Wartung | 533 |
| 29 | Pflege | 537 |
| 30 | Reverse Engineering | 543 |
| 31 | Reengineering (Teil 1) | 551 |
| 32 | Reengineering (Teil 2) | 557 |
| 33 | Reengineering (Teil 3) | 561 |
| | Glossar | 575 |
| | Literatur | 583 |
| | Sachindex | 591 |

1 Der Software-Lebenszyklus

Jedes Softwaresystem durchläuft einen Software-**Lebenszyklus**. Dieser Lebenszyklus muss bei jeder Softwareentwicklung beachtet werden – insbesondere bei der Architektur des Softwaresystems. In der Regel leben Softwaresysteme nämlich wesentlich länger als ursprünglich gedacht. Die Abb. 1.0-1 zeigt die einzelnen Phasen des Software-Lebenszyklus. Den einzelnen Phasen lassen sich nichtfunktionale Anforderungen zuordnen (siehe »Nichtfunktionale Anforderungen«, S. 109).

Lebenszyklus

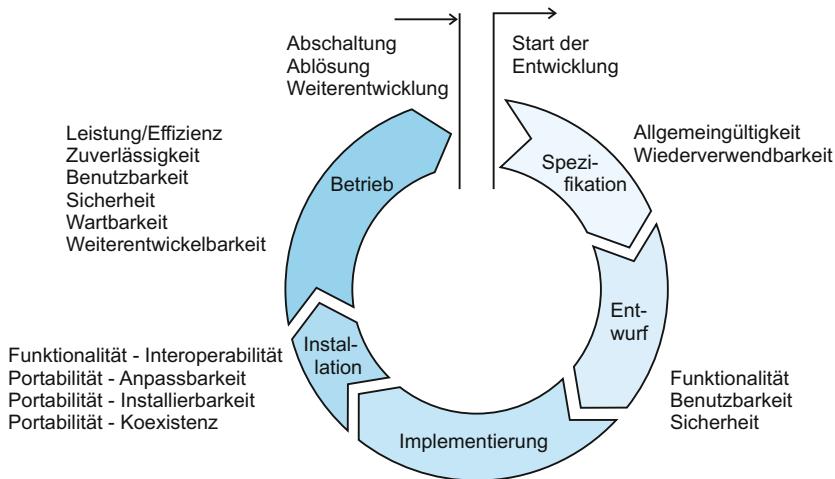


Abb. 1.0-1: Der Lebenszyklus eines Softwaresystems und zugeordnete nichtfunktionale Anforderungen.

Der Lebenszyklus eines Softwareproduktes beginnt in der Regel mit einer Spezifikationsphase und endet mit der Abschaltung, Ablösung oder Weiterentwicklung des Produkts:

Phasen der Softwareerstellung

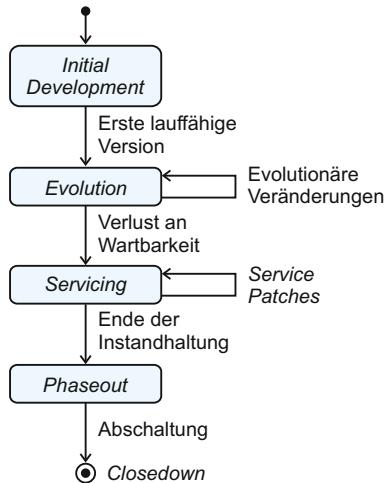
- Das Ergebnis des **Requirements Engineering** ist die **Spezifikation** des Softwaresystems.
- Ausgehend von der Spezifikation ist es die Aufgabe im **Entwurf**, die **Software-Architektur** zu erstellen. Beim Entwurf wird oft noch zwischen »Entwurf im Großen« und »Entwurf im Kleinen« unterschieden.
- Ist der Entwurf fertiggestellt, dann erfolgt in der **Implementierung** die Umsetzung der Architektur in den Programmcode.
- Das fertig gestellte Softwaresystem muss nun auf die Zielrechner **verteilt** und **installiert** sowie abgenommen und eingeführt werden – Verteilung und Installation werden oft als *Deployment* oder *Rollout* bezeichnet.

1 Der Software-Lebenszyklus

- Betrieb
- Nach der Installation beginnt der **Betrieb** des Softwaresystems.
 - Die Betriebsphase lässt sich nach [RaBe00] noch weiter unterteilen (Abb. 1.0-2):
 - *Initial Development*: Ist die Softwareentwicklung abgeschlossen, dann liegt die erste funktionierende Version vor.
 - *Evolution*: Die Fähigkeiten und die Funktionalität des Systems werden erweitert, um die Benutzeranforderungen, unter Umständen auf verschiedene Weise, zu erfüllen. Diese evolutionären Veränderungen führen zu einem Verlust an Wartbarkeit.
 - *Servicing*: Während der Instandhaltung werden kleinere Defekte repariert und einfache funktionale Änderungen vorgenommen (*Service Patches*).
 - *Phaseout*: Der Softwarehersteller entscheidet sich dazu, keine Instandhaltung mehr vorzunehmen und versucht – so lange wie möglich – Einnahmen durch das System zu erhalten.
 - *Closedown*: Der Softwarehersteller nimmt das System vom Markt und versucht, den Benutzer davon zu überzeugen, ein Ersatzsystem anzuschaffen, wenn eins existiert.

In der Regel läuft die Betriebsphase jedoch versioniert ab. Während der evolutionären Weiterentwicklung entscheidet sich der Hersteller in bestimmten Intervallen, eine neue Version der Software herzustellen und ein Release an den Kunden auszuliefern (Abb. 1.0-3).

Abb. 1.0-2:
Unterteilung der
Betriebsphasen (in
Anlehnung an
[RaBe00, S.67]).



Aufwand

Im Software-Lebenszyklus werden 80% des Aufwands für die Wartung benötigt, davon 40%, um die Software zu verstehen [TVS10, S. 46].

1 Der Software-Lebenszyklus

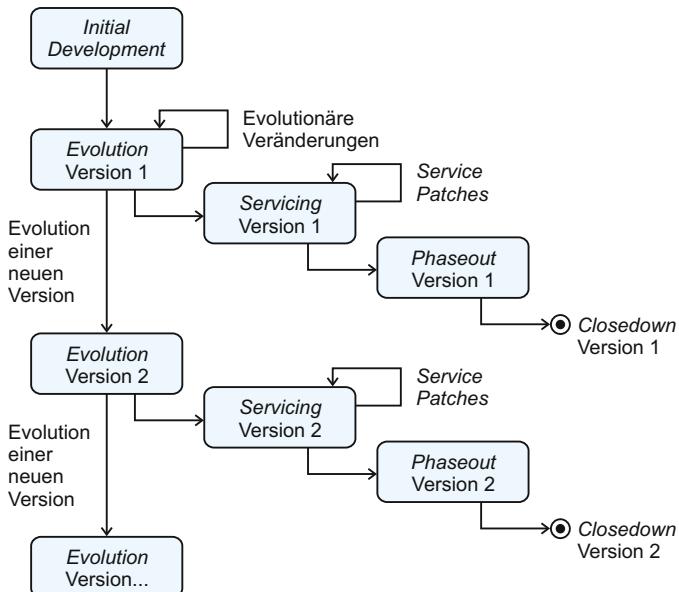


Abb. 1.0-3:
Unterteilung der
Betriebsphase mit
mehreren
Versionen (in
Anlehnung an
[RaBe00, S.67]).

Wie der Software-Lebenszyklus zeigt, beeinflussen die Anforderungen, niedergelegt in der Spezifikation, ganz wesentlich den Entwurf. Die Softwarearchitektur wiederum legt die Implementierung und die Installationsart fest. Beim Betrieb der Software zeigt sich, ob in der Spezifikation und der Architektur die Randbedingungen des Betriebs und die Evolution der Software ausreichend berücksichtigt worden sind.

Aufbau &
Gliederung

In diesem Buch wird davon ausgegangen, dass die Spezifikation bereits erstellt wurde und vorliegt. Das Thema *Requirements Engineering* wird in dem »Lehrbuch der Softwaretechnik – Basiskonzepte und Requirements Engineering« [Balz09a] und in dem zugehörigen E-Learning-Kurs behandelt.

Der Entwurf der Softwarearchitektur ist neben dem *Requirements Engineering* eine zentrale Aufgabe der Softwareentwicklung – oft als Königsdisziplin der Softwaretechnik bezeichnet. Er wird daher im ersten Teil dieses Buches ausführlich behandelt, da die Architektur wesentliche Auswirkungen auf alle folgenden Phasen des Softwarelebenszyklus hat:

■ Teil I: »Der Entwurf«, S. 5

Es wird in diesem Buch davon ausgegangen, dass der Leser bereits ausreichend Erfahrung mit der Implementierung von Software gesammelt hat. Es wird daher nur auf einige spezielle Aspekte der Implementierung eingegangen:

■ Teil II: »Die Implementierung«, S. 491

1 Der Software-Lebenszyklus

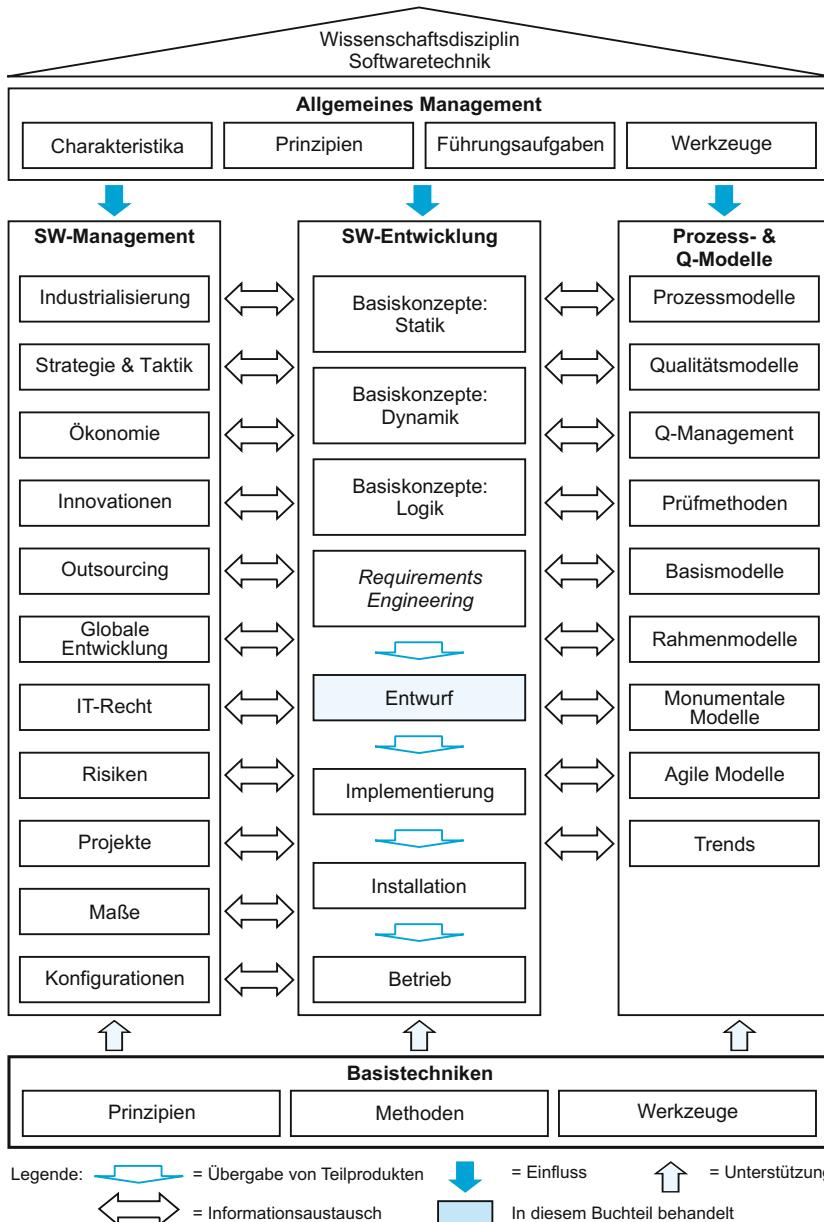
Bei den Überlegungen zur Architektur eines Softwaresystems wird bereits der Aspekt der späteren Verteilung und Installation der Software berücksichtigt. Die Erkenntnisse werden noch einmal zusammengefasst dargestellt:

- Teil III: »Verteilung, Installation, Abnahme und Einführung«, S. 519

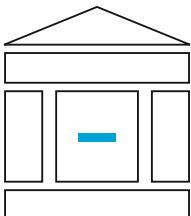
Da die Aspekte des Betriebs in der Regel nicht oder nicht ausreichend genug behandelt werden, wird dieses Thema ausführlicher behandelt:

- Teil IV: »Der Betrieb«, S. 529

I Der Entwurf



I I Der Entwurf



Aufgabe des **Entwerfens** (*design*) ist es, aus den gegebenen Anforderungen an ein Softwaresystem eine softwaretechnische Lösung im Sinne einer Softwarearchitektur zu entwickeln. Oft erfolgt der Entwurf in zwei Schritten:

- Beim »Entwerfen im Großen« (*high-level-design, global-design*) wird zunächst die globale Softwarearchitektur festgelegt – auch **Grobentwurf** genannt.
- Anschließend werden beim »Entwerfen im Kleinen« die einzelnen Subsysteme und Komponenten im Detail entworfen – auch **Feinentwurf** genannt

Das Problem beim Entwerfen besteht darin, dass eine **Vielzahl von Einflussfaktoren**, die sich oft gegenseitig beeinflussen, berücksichtigt werden müssen. Oft entstehen Zielkonflikte, die durch geeignete Kompromisse gelöst werden müssen. Damit diese Probleme nicht nur abstrakt dargestellt werden, wird eine durchgehende Fallstudie vorgestellt, die die Probleme konkret zeigt. Die Implementierung dieser Fallstudie steht im E-Learning-Kurs zum Herunterladen zur Verfügung.

i Bereits beim Entwurf muss der gesamte Lebenszyklus des zu entwickelten Softwaresystems betrachtet werden (siehe »Der Software-Lebenszyklus«, S. 1). Es muss also nicht nur die Softwarearchitektur modelliert und dokumentiert werden, sondern auch die physische Architektur und die Verteilung der Artefakte auf die technische Infrastruktur. Während die Softwarearchitektur in der Regel mit den **UML**-Elementen Komponenten, Pakete, Klassen und Sequenzdiagrammen beschrieben wird, werden physisch vorhandene Elemente durch UML-Artefakte beschrieben:

- »Artefakte«, S. 9

Bereits beim Entwurf muss man sich Gedanken über die Verteilung und die Installation des entwickelten Softwaresystems machen. In der UML können dafür Verteilungsdiagramme verwendet werden:

- »Verteilungsdiagramme«, S. 11

Bevor auf den Entwurf eingegangen wird, wird die Fallstudie »Kundenverwaltung« vorgestellt und als Einzelplatzanwendung entworfen und implementiert:

- »Fallstudie: KV – Überblick«, S. 15
- »Fallstudie: KV – Einzelplatz«, S. 21

Nach dieser intuitiven Einführung in die Probleme anhand der Fallstudie wird der Begriff der Softwarearchitektur nun genauer betrachtet:

- »Was ist eine Softwarearchitektur?«, S. 23

Architekturprinzipien sollten globale Leitlinien für jeden Entwurf sein:

- »Architekturprinzipien«, S. 29

Basis für die »Architektur im Großen« sollen Architekturmuster, Basis für die »Architektur im Kleinen« Entwurfsmuster sein:

- »Architektur- und Entwurfsmuster«, S. 37

Nichtfunktionale Anforderungen betreffen in der Regel die gesamte Softwarearchitektur und müssen daher sorgfältig überlegt und gegeneinander abgewogen werden:

- »Nichtfunktionale Anforderungen«, S. 109

Anschließend wird auf wichtige Einflussfaktoren eingegangen, die bei der Softwarearchitektur zu berücksichtigen sind:

- »Einflussfaktoren auf die Architektur«, S. 135

Unabhängig von der fachspezifischen Funktionalität müssen heutige Softwaresysteme eine Reihe von allgemeinen Funktionen zur Verfügung stellen bzw. eine Reihe von Techniken unterstützen – oft spricht man in diesem Zusammenhang auch von den »quer schneidenden Belangen«. Im Folgenden werden die entsprechenden Themen zunächst getrennt voneinander behandelt:

- »Globalisierung von Software«, S. 143
- »Authentifizierung und Autorisierung«, S. 153
- »Transaktionen«, S. 177
- »Verteilte Architekturen«, S. 191
- »Arten der Netzkomunikation«, S. 205

Zur Realisierung einer Softwarearchitektur wird in der Regel auf softwaretechnische Infrastrukturen aufgesetzt. Diese Infrastrukturen können entweder von vornherein vorgegeben sein, was die Architekturüberlegungen (wesentlich) einschränken kann, oder sie müssen neu beschafft werden. Sie unterstützen – mehr oder weniger stark – die Realisierung der einzelnen Subsysteme:

- »Softwaretechnische Infrastrukturen«, S. 319

Für eingebettete Systeme gibt es eine Reihe von Besonderheiten zu beachten:

- »Architekturen »Eingebetteter Systeme««, S. 397

Jedes Softwaresystem lässt sich grob in drei Subsysteme gliedern, die je nach Anwendungsgebiet ein unterschiedliches Gewicht und eine unterschiedliche Komplexität besitzen:

- »Das Subsystem Applikation«, S. 413
- »Das Subsystem Persistenz«, S. 421
- »Das Subsystem Benutzungsoberfläche«, S. 449

Nachdem alle wichtigen Aspekte für eine Architekturüberlegung behandelt worden sind, wird der eigentliche Entwurfsprozess näher betrachtet:

- »Der Entwurfsprozess«, S. 481

Parallel zum Entwurfsprozess muss die Qualität der Architektur sichergestellt werden:

- »Qualitätssicherung der Architektur«, S. 487

I I Der Entwurf

Hinweis

Für die Fallstudie »Kundenverwaltung« und für die Codebeispiele wird die Programmiersprache Java verwendet. Zusätzlich werden an relevanten Stellen Hinweise auf die Möglichkeiten der .NET-Plattform gegeben. Als Beispiel wird für die .NET-Plattform teilweise eine »Fluggesellschaft« verwendet. Wenn Sie bisher noch keine Kenntnisse über die .NET-Plattform besitzen, dann sollten Sie zuerst das Kapitel »Die .NET-Plattform«, S. 360, lesen.

2 Artefakte

Bei der objektorientierten Modellierung einer Anwendung denkt und arbeitet man vorwiegend mit Elementen wie Klassen, Objekten, Attributen, Assoziationen usw.¹ Dabei handelt es sich um nicht materielle, d. h. rein logische Dinge. Ein Klassen-Diagramm sagt nichts darüber aus, in welcher Form eine Klasse später existieren wird. Durch die Wahl der Implementierungssprache wird dies zwar meist implizit vorgegeben, die Vorstellung bleibt jedoch vage.

Logische & physische Elemente

In Java wird eine Klasse in einer java-Datei implementiert. Daraus entsteht durch Übersetzung eine class-Datei. Aber bei inneren Klassen gibt es schon wieder mehrere Möglichkeiten der Aufteilung auf Dateien und ob mehrere class-Dateien zu einem JAR-Archiv zusammengefasst werden oder nicht, ist ebenfalls zunächst unklar.

Klassen in Java

Ein **Artefakt** (*artifact*) ist in der UML etwas physisch Vorhandenes, z. B. eine Datei auf der Festplatte.² In Diagrammen wird für Artefakte das Klassen-Symbol mit einem Dokument-Piktogramm in der rechten oberen Ecke verwendet (Abb. 2.0-1). Alternativ kann der Artefakt-Charakter des Elements als **Stereotyp** angegeben werden.

Grafische Darstellung

Die Zuordnung von logischen Elementen zu Artefakten erfolgt durch eine Abhängigkeitsbeziehung, also eine gestrichelte Linie mit offenem Pfeil, die mit dem Stereotyp *manifest* bezeichnet wird.

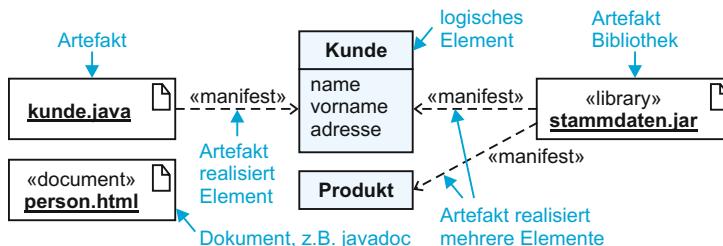


Abb. 2.0-1: Die UML-Notation für Artefakte.

Die UML sieht einige Standard-Stereotypen vor, mit denen Artefakte näher spezifiziert werden können:

- «file»: Eine physische Datei. Es handelt sich um einen übergeordneten Stereotyp. Die Bedeutung wird von folgenden Stereotypen weiter spezialisiert.

¹ Teile dieses Textes und der Grafiken wurden mit freundlicher Genehmigung des W3L-Verlags aus [ZiAr10, S. 301 ff.] übernommen.

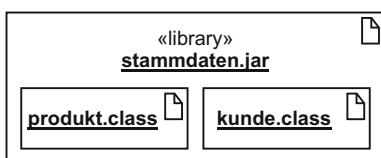
² Physisch vorhanden wird aus der Sichtweise eines Informatikers benutzt. Ein Philosoph mag Probleme damit haben, eine Datei als physisch vorhanden zu bezeichnen.

I 2 Artefakte

- «document»: Eine Datei, die keinen Programmcode (Quelltext oder Objektcode) enthält.
- «executable»: Eine Datei mit einem ausführbaren Programm.
- «library»: Eine Datei einer statischen oder dynamischen Bibliothek.
- «script»: Eine Datei mit einem Programm, das nicht übersetzt, sondern zur Laufzeit interpretiert wird (Java-Bytecode fällt definitiv *nicht* in diese Kategorie).
- «source»: Eine Datei mit einem Quellprogramm.

Schachtelung Artefakte können beliebig ineinander geschachtelt werden. Eine class-Datei kann z.B. in einem JAR-Archiv ausgeliefert werden. Dies kann in der UML einfach durch Schachtelung der Symbole dargestellt werden (Abb. 2.0-2).

Abb. 2.0-2:
Artefakte können
ineinander
geschachtelt
werden.



Praktischer
Einsatz

Es macht in der Praxis wenig Sinn, die Zuordnung aller logischen Modell-Elemente zu Artefakten zu modellieren. Eine komplexe Anwendung besteht leicht aus mehreren hundert Klassen. Die Abbildung auf java- und class-Dateien ist meist klar und auch immer gleich. Sie braucht nicht vollständig in einem Diagramm modelliert zu werden. Besser ist die Beschränkung auf wirklich wesentliche Elemente, im Java-Umfeld vor allem auf JAR-Archive. Diese sind deshalb so wichtig, weil sie ausgeliefert und auf dem Zielsystem installiert werden müssen. Quellcode-Dateien existieren zwar auch physisch, bleiben jedoch normalerweise auf Entwicklungsrechnern. Genaugenommen realisieren JAR-Archive natürlich keine Klassen, sondern enthalten die class-Dateien, die dies tun. Dieser Zusammenhang darf aber implizit vorausgesetzt werden und sollte in einem Diagramm nicht explizit dargestellt werden.

Ausführbare
Programme

Außerhalb der Java-Welt können vor allem Dateien mit ausführbarem Programmcode als Artefakte beschrieben werden. Unter Windows sind dies exe- und dll-Dateien, unter Linux z.B. rpm-Dateien. Auf dieser Ebene macht die Modellierung von Artefakten Sinn.

3 Verteilungsdiagramme

Bei der Planung und Dokumentation der Verteilung einer Anwendung geht es darum, auf welchen Computersystemen die Anwendung laufen soll und welche Bestandteile der Anwendung auf welchen Systemen vorliegen müssen, damit ein reibungsloser Betrieb möglich ist.¹

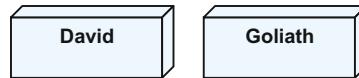
Die UML unterstützt die Modellierung durch **Verteilungsdiagramme** (*deployment diagram*). Die wichtigsten Elemente in diesem Diagrammtyp sind Artefakte (siehe »Artefakte«, S. 9) und Knoten (*nodes*).

Knoten

Die Verteilung einer Anwendung erfolgt in der UML auf einzelne Knoten. Knoten repräsentieren eine Ausführungsumgebung mit Speicher- und Verarbeitungskapazität. Meist ist damit Hardware gemeint, also ein Computersystem mit Speicher, Prozessoren und Systemsoftware. Statt Knoten wird dann häufig der Begriff Maschine oder *host* verwendet. Je nach Bedarf können aber auch eine **JVM**, ein Anwendungsserver oder ein Betriebssystem als Knoten modelliert werden. In einem Verteilungsdiagramm werden sie als dreidimensionales Rechteck dargestellt (Abb. 3.0-1).

Knoten können ineinander geschachtelt werden. Dadurch kann angedeutet werden, dass bestimmte Knoten vom Vorhandensein anderer Knoten abhängig sind.

Die Abb. 3.0-2 zeigt einen komplexen Systemaufbau für eine Web-Anwendung mit Anwendungsserver.



Zielsetzung

Diagramm-
Elemente

Abb. 3.0-1:
Computer (Knoten)
werden in der UML
durch ein 3D-
Rechteck
dargestellt.

Realisierung von Artefakten

Die Artefakte einer Anwendung werden auf die Knoten verteilt. Das geschieht am einfachsten dadurch, dass die Artefakt-Symbole innerhalb der Knoten-Symbole dargestellt werden (Abb. 3.0-3). Alternativ kann eine Abhängigkeitsbeziehung mit dem Stereotypen *manifest* zwischen einem Knoten- und einem Artefakt-Symbol gezeichnet werden.

Die Verteilung von Artefakten auf Knoten kann nach zwei unterschiedlichen Gesichtspunkten erfolgen:

¹ Teile dieses Textes wurden mit freundlicher Genehmigung des W3L-Verlags aus [ZiAr10, S. 303 ff.] übernommen.

I 3 Verteilungsdiagramme

Abb. 3.0-2:
Geschachtelte Knoten mit einem Computer mit Linux. Auf ihm läuft ein Webserver innerhalb einer JVM. Ein JSP-Server und ein Anwendungsserver laufen gemeinsam in einer zweiten JVM.

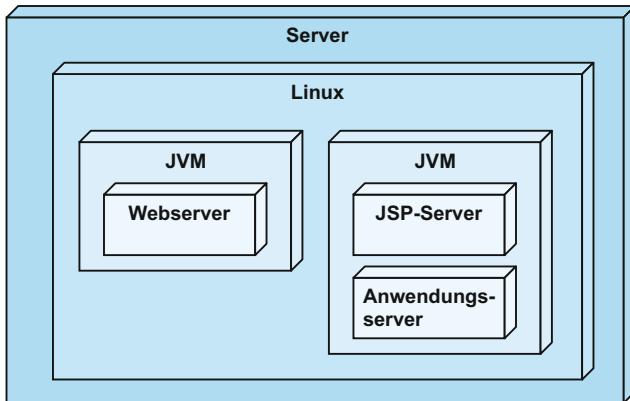
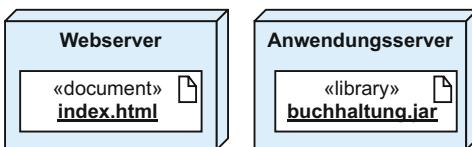


Abb. 3.0-3:
Artefakte werden auf Knoten verteilt.



- typische Verteilungs-Szenarien
- konkrete Installationen

Es wird ein typisches Verteilungs-Szenario dargestellt, das alle Arten von Artefakten und Knoten enthält (Abb. 3.0-4). Ein Knoten repräsentiert dann z.B. einen Server mit bestimmten Leistungsmerkmalen, aber nicht eine konkrete Maschine mit einer IP-Adresse, die sie von anderen, gleichartigen Maschinen unterscheidet.

In einem solchen Diagramm können Knoten-Typen definiert werden, z.B. Server, Client, Mobiles Gerät usw. Es ist dann ersichtlich, dass bestimmte Teile der Anwendung für den serverseitigen Betrieb entwickelt wurden, während andere Teile auf mobilen Geräten laufen sollen.

Abb. 3.0-4:
Mehrere konkrete Computer oder Geräte mit gleichen Merkmalen können zu einem Knoten-Typ zusammengefasst werden.



Manchmal müssen Verteilungsdiagramme so detailliert sein, dass sie auf der Ebene konkreter Maschinen arbeiten. Ein Knoten ist dann z.B. ein bestimmter Server mit einer bestimmten IP-Adresse. Wenn ein Artefakt auf zwei Servern laufen soll, wird es beiden Knoten zugeordnet und zweimal eingezeichnet.

Ein solches Diagramm beschreibt eine konkrete Installation der Anwendung (Abb. 3.0-5).

Man kann beide Arten von Verteilungsdiagrammen unabhängig voneinander verwenden, je nachdem welche Sichtweise benötigt wird. Ihre volle Stärke spielen sie aber erst in Kombination aus. Das Verhältnis zwischen Verteilungs-Szenarien und konkreten Installa-

Klassen & Objekte

3 Verteilungsdiagramme I

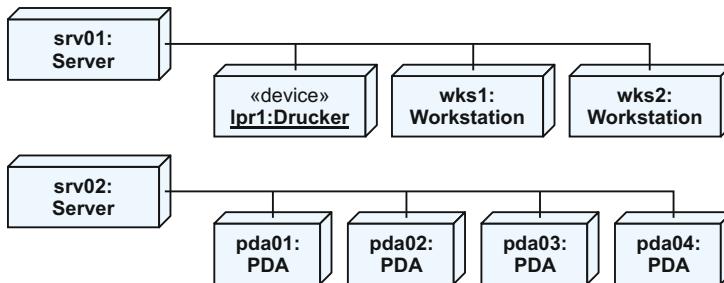


Abb. 3.0-5:
Physisch vorhandene Computer oder Geräte werden als Exemplare von Knoten in Verteilungsdiagrammen dargestellt.

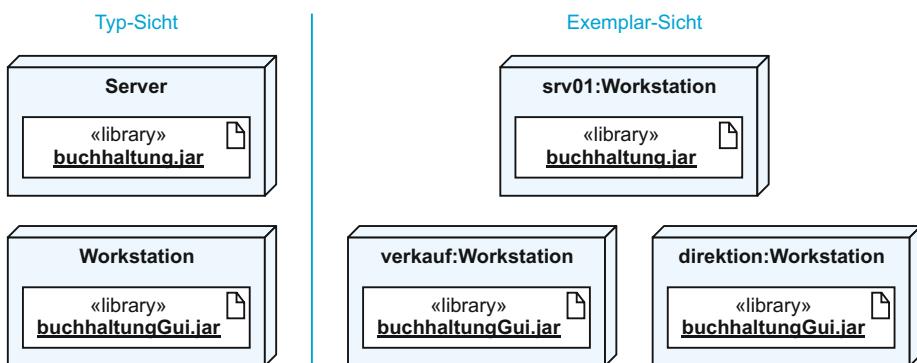
tionen ist prinzipiell das gleiche wie zwischen Klassen und Objekten. Jeder konkrete Knoten ist ein Exemplar eines Knoten-Typs im Verteilungs-Szenario. Ein konkreter Server mit einer bestimmten IP-Adresse ist ein Exemplar eines Server-Knotens. Der Knoten-Typ gibt an, welche Artefakte auf einem konkreten Knoten installiert sein müssen.

Auf einem Knoten-Exemplar liegen Exemplare von Artefakten. Was unter einem Exemplar eines Artefakts zu verstehen ist, verdeutlicht am besten ein Beispiel.

Exemplare von Artefakten

Wenn im Rahmen eines Projekts ein JAR-Archiv mit Klassen zur Buchhaltung entwickelt wird, ist dies ein Artefakt. Bei der Installation wird das JAR-Archiv auf drei Server kopiert und auf diesen ausgeführt. Die Kopien sind dann Exemplare des Artefakts (Abb. 3.0-6).

Beispiel



Manchmal stellt sich die Frage, ob ein bestimmtes Stück Software als Knoten oder als Artefakt modelliert werden sollte. Als Faustregel kann das Projekt betrachtet werden, für das die Modellierung erfolgen soll. Wenn die Software als Teil des Projekts entwickelt wurde, oder während der Entwicklung benötigt wird, ist sie ein Artefakt. Im Projekt entwickelte oder zugekauft Bibliotheken fallen in diese Kategorie. Knoten bieten den Artefakten eine Ausführungsumgebung und sind nicht Teil des Projekts. Die Grenzen sind dabei fließend.

Abb. 3.0-6: Typ-Sicht vs. Exemplar-Sicht in einem Verteilungsdiagramm.

I 3 Verteilungsdiagramme

Bei der Entwicklung eines Anwendungsservers sind seine Bestandteile Artefakte. Bei seiner Verwendung, also als Ablaufumgebung für eine andere Anwendung, wird er zu einem Knoten. Mit etwas Erfahrung wird man aber immer in der Lage sein, die jeweils günstigste Modellierungsform ohne große Probleme zu erkennen.

4 Fallstudie: KV – Überblick

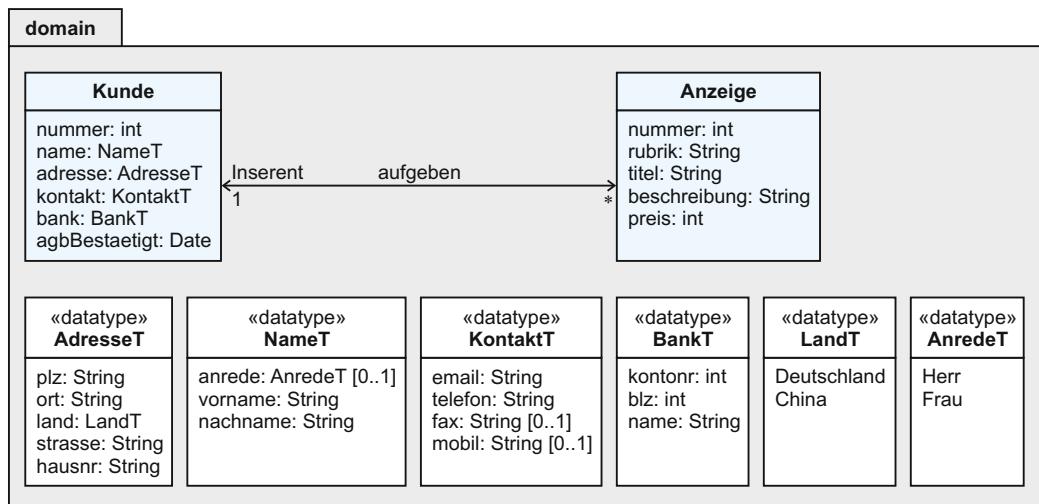
Im Folgenden wird davon ausgegangen, dass für ein Softwaresystem bereits eine Spezifikation vorliegt. Es wird gezeigt, welche weiteren Entscheidungen notwendig sind, um von der Spezifikation zu einem Entwurf und zu einer Implementierung zu gelangen.

Die im Folgenden beschriebene Fallstudie wird schrittweise ausgebaut. Es wird gezeigt, wie sie mit verschiedenen Techniken und Architekturen realisiert werden kann.

Hinweis

In dieser Fallstudie »Kundenverwaltung« – kurz KV genannt – sollen für einen Zeitungsverlag die Inserenten verwaltet werden, die eine Anzeige aufgeben. Die Abb. 4.0-1 zeigt die Spezifikation in Form eines UML-OOA-Diagramms.

Aufgabenstellung



Um aus dem modellierten und spezifizierten Fachkonzept ein einsatzfähiges Softwaresystem zu erstellen, ist es erforderlich, drei **Subsysteme** im Entwurf zu modellieren:

- das Subsystem **Benutzungsoberfläche/Präsentation** (oft *User Interface* – kurz UI genannt – oder *Graphical User Interface* – kurz GUI genannt – wenn es sich um eine grafische Benutzungsoberfläche handelt),
- das Subsystem **Applikation** (auch »Anwendungslogik« oder »Applikationslogik« oder *Business Architecture* genannt),
- das Subsystem **Persistenz**.

Abb. 4.0-1: OOA-Klassendiagramm für die Fallstudie »Kundenverwaltung«.

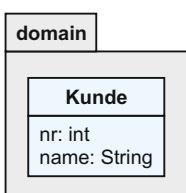
I 4 Fallstudie: KV – Überblick

Das Subsystem Benutzungsoberfläche wickelt die Interaktion mit dem Benutzer ab. Es hat Zugriff auf das Subsystem Applikation. In diesem Subsystem werden die Kunden und ihre Anzeigen verwaltet. Das Subsystem Applikation hat Zugriff auf das Subsystem Persistenz, in dem die Daten dauerhaft gespeichert werden.

Vereinfachung

Das in der Abb. 4.0-1 dargestellte OOA-Modell entspricht einer realen Spezifikation. Damit die folgenden Architekturdarstellungen übersichtlich bleiben und die Implementierung leicht nachzuvollziehen ist, wird die obige Spezifikation zunächst drastisch vereinfacht. Es wird nur von einer Kundenklasse mit zwei Attributten (Kundennummer und Kundenname) ausgegangen (Abb. 4.0-2) – im Folgenden »Kundenverwaltung-Mini« (KV-Mini) genannt.

Abb. 4.0-2: OOA-Paketdiagramm für die vereinfachte Kundenverwaltung.



Ausgangspunkt für das Architekturmodell ist das vorhandene OOA-Modell. Für das Subsystem Applikation wird ein Kundencontainer benötigt, der alle Kundenobjekte verwaltet.

Das Subsystem Benutzungsoberfläche – im Folgenden GUI genannt – muss es ermöglichen, einen Kunden zu erfassen, anzuzeigen und zu ändern. Das Subsystem Persistenz kann die Kundendaten beispielsweise in einer XML-Datei speichern.

Das so entstehende OOD-Modell wird als **logische Architektur** bezeichnet. Die Abb. 4.0-3 zeigt die logische Architektur in Form von UML-Paketen. Da das Paket gui nur Zugriff auf das Paket application und das Paket application nur Zugriff auf das Paket persistence hat (dargestellt durch die Pfeile) spricht man auch von einem strikten Schichtenmodell.

Statische Sicht

Das hier dargestellte OOD-Modell zeigt die statische Sicht der Architektur (Abb. 4.0-3).

Die erste Spalte zeigt das aus dem OOA-Modell entstehende OOD-Modell mit der logischen Architektur.

Bevor mit dem Entwurf fortgefahren werden kann, sind unter anderem folgende Entscheidungen zu treffen:

- Soll das Softwaresystem für einen Benutzer oder für mehrere Benutzer (nacheinander oder gleichzeitig) einsetzbar sein?
- Soll das Softwaresystem auf einem Computer-Einzelplatz eingesetzt werden oder soll über einen Client auf die installierte Anwendung auf einem Server zugegriffen werden?
- Welche Infrastruktur steht zur Verfügung bzw. wird zur Verfügung gestellt?

Annahmen Im Folgenden wird zunächst davon ausgegangen, dass das Softwaresystem nur von einem Benutzer zu einer Zeit oder von mehreren Benutzern nacheinander auf einem Einzelplatz-Computer benutzt werden soll.

4 Fallstudie: KV – Überblick I

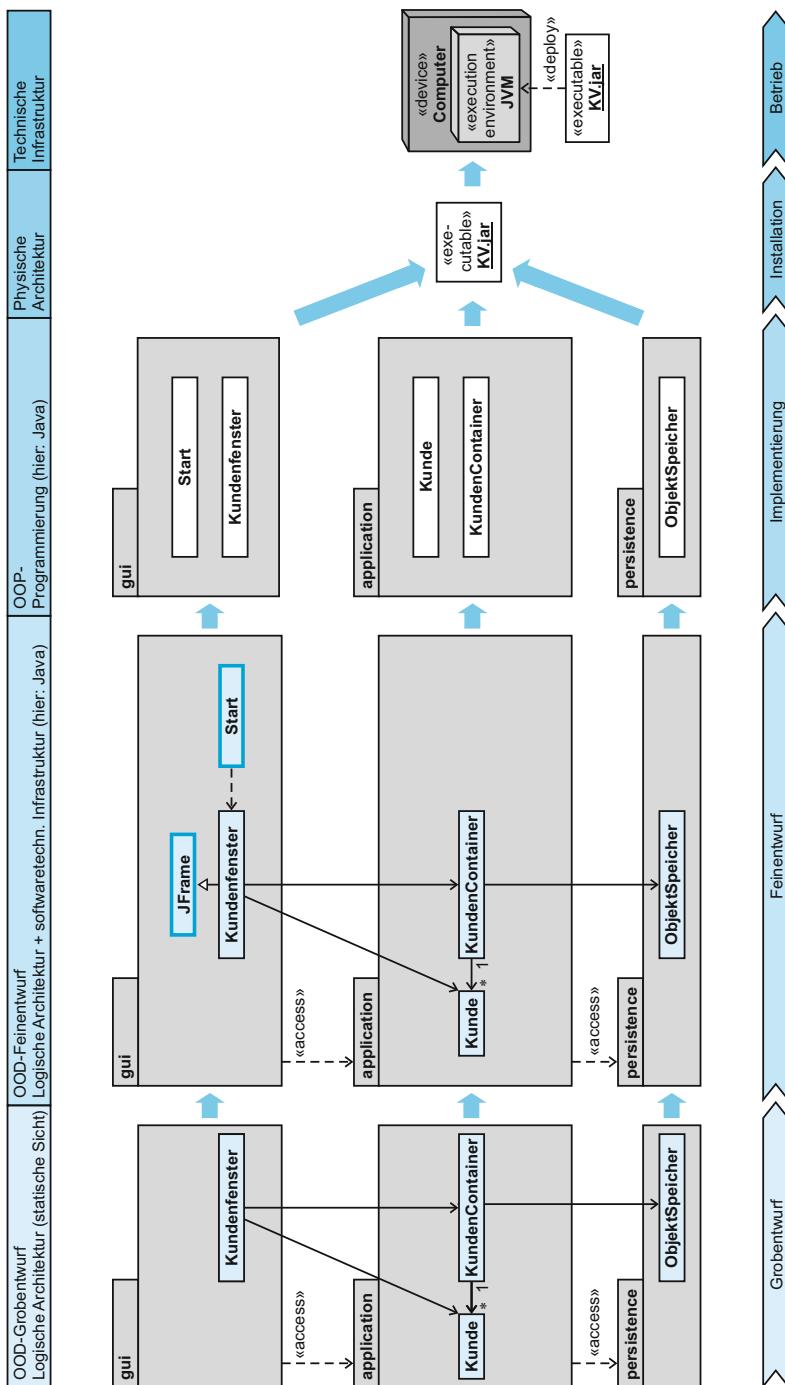


Abb. 4.0-3: Entwicklungsschritte einer Einzelplatzanwendung bei der Fallstudie »Kundenverwaltung«.

I 4 Fallstudie: KV – Überblick

| | |
|-----------------|--|
| | Als softwaretechnische Infrastruktur steht für die Entwicklung eine Java-Entwicklungsumgebung zur Verfügung. Auf dem Ziel-Computersystem steht eine Java-Laufzeitumgebung zur Verfügung. |
| Feinentwurf | Unter diesen Annahmen kann der Entwurf verfeinert werden. Im Subsystem gui können nun beispielsweise die Java-Swing-Klassen für den Feinentwurf eingesetzt werden. Im Subsystem Persistenz kann auf die Java-XML-Klassen zurückgegriffen werden (2. Spalte in der Abb. 4.0-3). |
| Implementierung | Nach dem Feinentwurf kann die Implementierung erfolgen. Das Ergebnis sind Java-Quellcode-Dateien und – nach der Übersetzung – Java-Objektcode-Dateien (3. Spalte in der Abb. 4.0-3). |
| Installation | Die sogenannte physische Architektur ist recht einfach. Alle Objektcode-Dateien müssen zu einer jar-Datei zusammengefasst werden. Diese Datei wird für die Installation ausgeliefert (4. Spalte in der Abb. 4.0-3.) |
| Betrieb | Die technische Infrastruktur besteht hier aus einem oder mehreren einzelnen Computersystemen, auf denen die jar-Datei installiert wird (5. Spalte in der Abb. 4.0-3). |
| Frage | Überlegen Sie, welche Änderungen erforderlich sind, wenn als Infrastruktur anstelle von Java eine .NET-Umgebung zur Verfügung steht. |
| Antwort | Beim Feinentwurf müssen anstelle von Java-Klassen entsprechende .NET-Klassen verwendet werden. Die Auslieferung des Softwaresystems erfolgt nicht in Form einer jar-Datei, sondern in Form einer exe-Datei. Außerdem muss auf dem Ziel-Computersystem eine .NET-Laufzeitumgebung zur Verfügung stehen. Die Fähigkeit eines Softwaresystems, von einer Umgebung in eine andere Umgebung übertragen zu werden, bezeichnet man auch als Portabilität. Portabilität zählt zu den nichtfunktionalen Anforderungen (siehe »Portabilität«, S. 132). |
| Frage | Überlegen Sie, welche Änderungen erforderlich sind, wenn die Daten nicht in einer XML-Datei, sondern in einer relationalen Datenbank gespeichert werden sollen. |
| Antwort | Das Subsystem Persistenz muss geändert werden. Eventuell hat dies auch Rückwirkungen auf das Subsystem Applikation. Die softwaretechnische Infrastruktur muss eine relationale Datenbank zur Verfügung stellen. Das Subsystem Persistenz muss direkt oder über eine Schnittstelle auf die relationale Datenbank zugreifen. Auf dem Ziel-Computersystem muss ein Laufzeitsystem der relationalen Datenbank zur Verfügung stehen bzw. das Laufzeitsystem muss mit ausgeliefert werden. |
| | Befindet sich das bisherige Softwaresystem bereits bei einem oder bei mehreren Kunden im Einsatz, dann muss eine Migrationsstrategie überlegt werden, die festgelegt, wie bereits vorhandene Daten |

4 Fallstudie: KV – Überblick I

aus der XML-Datei in die relationale Datenbank übertragen werden können. Eventuell muss ein besonderes Migrationsprogramm geschrieben und mit ausgeliefert werden, um die Daten zu migrieren.

Die Änderungsbeispiele zeigen deutlich, dass jede Änderung an einem fertigen und bereits ausgelieferten Softwaresystem umfangreiche und aufwändige Änderungen nach sich ziehen kann.

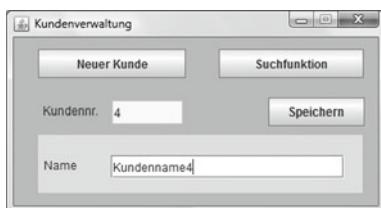
Damit Sie die beschriebenen Programme auf Ihrem Computersystem installieren und ablaufen lassen können, empfiehlt es sich, die kostenlose Entwicklungsumgebung »Eclipse IDE for Java EE Developers« herunterzuladen und auf Ihrem Computersystem zu installieren. Die in dem kostenlosen E-Learning-Kurs zu diesem Buch beschriebenen, herunterladbaren Programme können mit der Import-Funktion von Eclipse direkt in die Entwicklungsumgebung importiert werden. Im kostenlosen E-Learning-Kurs zu diesem Buch finden Sie einen Schnelleinstieg in die Entwicklungsumgebung Eclipse.

Hinweis

5 Fallstudie: KV – Einzelplatz

Das OOA-Modell der Fallstudie »Kundenverwaltung-Mini« zeigt die Abb. 5.0-1 (siehe »Fallstudie: KV – Überblick«, S. 15).

Um den Programmieraufwand für die grafische Benutzungsoberfläche gering zu halten, wird ein einfaches Fenster mit Druckknöpfen gewählt und auf Menüs verzichtet. Die Abb. 5.0-2 zeigt, wie ein neuer Kunde angelegt werden kann.



| Kunde |
|-------------------------|
| - nr: int |
| - name: String |
| + getNr(): int |
| + getName(): String |
| + setNr(int): void |
| + setName(String): void |

Abb. 5.0-1: OOA-Klassendiagramm der Fallstudie Kundenverwaltung-Mini.

Abb. 5.0-2: Erfassung eines Kunden bei der Fallstudie Kundenverwaltung-Mini.

Wird der Druckknopf Neuer Kunde betätigt, wird die nächste Kundennummer angezeigt und es kann ein Kundenname eingegeben werden. Wird der Druckknopf Speichern gedrückt, dann wird der eingegebene Kundenname gespeichert. Für die Suche eines Kunden gibt es eine Suchfunktion (Abb. 5.0-3).



Abb. 5.0-3: Suchen eines Kunden in der Fallstudie Kundenverwaltung-Mini.

Wird der Druckknopf Suchfunktion gedrückt, dann wird ein neuer Druckknopf Suchen eingeblendet. Wird eine Kundennummer eingegeben und der Druckknopf Suche betätigt, dann wird der zur Kundennummer gehörige Name angezeigt. Für eine Einzelplatz-Anwendung als *Stand-alone*-Anwendung (unabhängig von anderen Anwendungen) mit der softwaretechnischen Infrastruktur »Java« ergibt sich das OOD-Modell der Abb. 5.0-4.

I 5 Fallstudie: KV - Einzelplatz

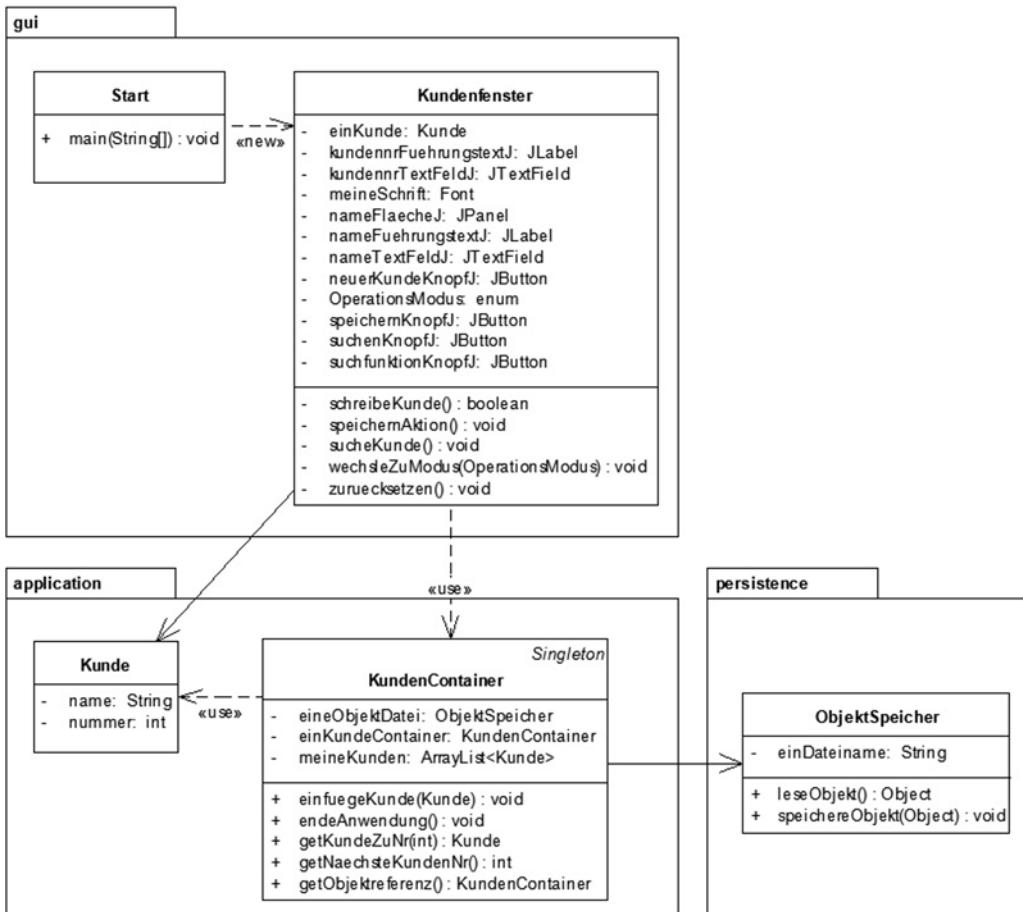


Abb. 5.0-4: OOD-Klassendiagramm der Kundenverwaltung-Mini (Einzelplatz). Wie die Abb. 5.0-4 zeigt, führt die Realisierung dieses einfachen Beispiele bereits zu einem OOD-Modell mit fünf Java-Klassen. Das OOA-Modell bleibt unverändert – es wird nur ein Konstruktor hinzugefügt. Bei der Klasse KundenContainer wird das Singleton-Entwurfsmuster eingesetzt.

OOP Die Programme zu dieser Fallstudie können Sie im E-Learning-Kurs herunter laden.

Sehen Sie sich den Programmcode der einzelnen Klassen genauer an. Führen Sie das Programm auf Ihrem Computersystem aus und machen Sie sich mit der Bedienung vertraut.

6 Was ist eine Softwarearchitektur?

Eine **Softwarearchitektur** beschreibt die Strukturen eines Softwaresystems durch Architekturbausteine und ihre Beziehungen und Interaktionen untereinander sowie ihre physikalische Verteilung. Die extern sichtbaren Eigenschaften eines Architekturbausteins werden durch Schnittstellen spezifiziert.

Definition

Sichten

Beim Bau eines Hauses gibt es verschiedene Sichten. Der Elektriker hat eine andere Sicht als der Sanitär-Installateur. Analogie

Dementsprechend gibt es verschiedene Architekturplan-Varianten, d. h. es gibt Architekturpläne für den Elektriker, für den Sanitär-Installateur usw.

Analog gibt es unterschiedliche Sichten auf eine Softwarearchitektur. Folgende Sichten werden in der Regel unterschieden:

- **Statische Sicht** (auch Bausteinssicht genannt): Zeigt die statische Struktur der Architektur.
- **Laufzeitsicht** (auch dynamische Sicht genannt): Zeigt, welche Bausteine des Systems zur Laufzeit existieren und wie sie zusammenwirken, d. h. wie die Prozessstruktur aussieht.
- **Verteilungssicht** (auch Infrastruktursicht genannt): Gibt an, welche Architekturbausteine auf welchen Hardwarekomponenten ablaufen. Dokumentiert die Computer, die Netztopologie und Netzprotokolle sowie sonstige Bestandteile der physischen Systemumgebung. Es wird die Sicht des Systembetreibers dargestellt.
- **Kontextsicht**: Beschreibt die Einbettung des Systems in seine Umgebung. Aus abstrakter Sicht werden die wichtigsten Schnittstellen und die wesentlichen Teile der umgebenden Infrastruktur dargestellt. Stellt man die Analogie zu einem Hausarchitekten her, dann beschreibt die Kontextsicht die Schnittstelle zwischen einem Hausarchitekten und einem Stadtplaner.

Die Abb. 6.0-1 zeigt den Zusammenhang zwischen den verschiedenen Sichten (in Anlehnung an [Star05, S. 86]).

I 6 Was ist eine Softwarearchitektur?

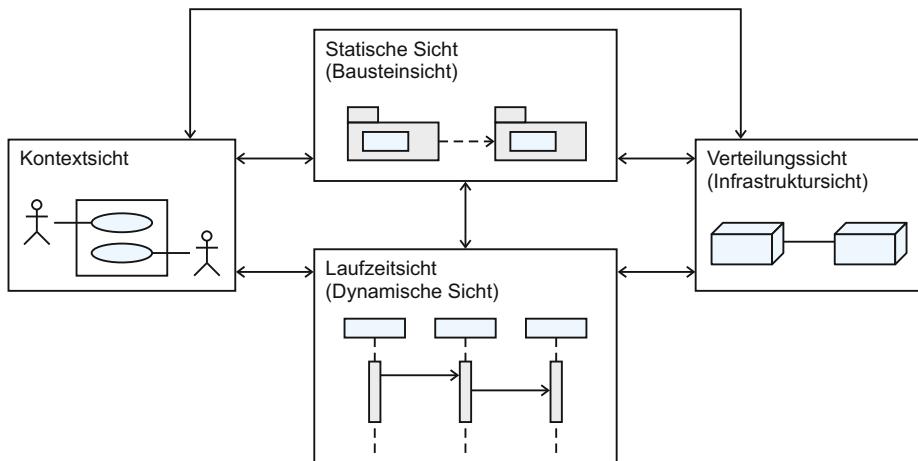


Abb. 6.0-1: Sichten
auf eine Software-
architektur.

Verschiedene Bausteinarten

Je nach Abstraktionsgrad kann man verschiedene Architektur-Bausteinarten unterscheiden:

- Subsysteme
- Komponenten
- Frameworks
- Pakete
- Klassen

- Subsystem Auf der obersten Abstraktionsebene einer Softwarearchitektur werden oft zunächst **Subsysteme** gebildet. Der Begriff Subsystem ist in der Softwaretechnik *nicht* einheitlich definiert. Subsysteme helfen bei großen Systemen, geeignete Teilsysteme zu identifizieren. Ein Subsystem soll
- eine logische Einheit bilden, d. h. es soll so strukturiert sein, dass es den Leser durch das Architekturmödell führt,
 - einen Themenbereich enthalten, der für sich allein betrachtet und verstanden werden kann,
 - eine zusammenhängende Funktionalität besitzen und in sich selbst abgeschlossen sein,
 - einen Teil der an das System gestellten Anforderungen befriedigen,
 - Komponenten, Pakete und Klassen enthalten, die logisch zusammengehören,
 - für sich entworfen und eventuell implementiert werden können, wobei eine wohl definierte Schnittstelle zur Umgebung vorhanden ist.
- Dabei soll die Schnittstelle
- Vererbungsstrukturen nur in vertikaler Richtung schneiden,

6 Was ist eine Softwarearchitektur? I

- keine Aggregation durchtrennen und
- möglichst wenig Assoziationen enthalten.

Der Begriff **Komponente** wird in zwei unterschiedlichen Bedeutungen verwendet. In der UML 2 spezifiziert eine Komponente sein Verhalten durch bereitgestellte und benötigte Schnittstellen [OMG09a, S. 146]. Eine Komponente ist in der UML eine Spezialisierung der Klasse und besitzt somit alle Eigenschaften einer Klasse. Sie wird durch das Klassensymbol (Rechteck) mit dem Stereotyp <<component>> dargestellt. Komponenten sind eine Art von »großen Klassen«. Sie können Beziehungen eingehen und haben nach außen sichtbare oder versteckte Teile. Durch sie ist es möglich, einen Zugriffsschutz über sauber definierte Schnittstellen zu gewährleisten. Komponenten können beliebig ineinander geschachtelt werden.

Im Rahmen von konkreten Komponentenmodellen besitzen Komponenten eine spezifische Bedeutung.

Eine Komponente ist bei diesen Techniken ein *binärer* Softwarebaustein, der Dritten Funktionalität über Schnittstellen zur Verfügung stellt und unabhängig ausgeliefert werden kann.

Eine Komponente benötigt eine Komponentenplattform und besitzt die Fähigkeit zur Selbstbeschreibung.

Sie weist nur explizite Kontextabhängigkeiten auf und verfügt über die Fähigkeit zur Anpassung, um eine möglichst hohe Kompositionsfähigkeit zu erreichen [Balz09a, S. 135].

Eine komponentenbasierte Entwicklung basiert darauf, dass Schnittstellen von ausgelieferten Komponenten *nicht* mehr modifiziert werden, während die interne Realisierung beliebig geändert werden darf.

Es gibt verschiedene Komponentenmodelle, z. B. JavaBeans, EJBs (*Enterprise JavaBeans*), CORBA Components, COM (*Components Object Model*) und .NET components. Komponenten im Sinne von Komponentenmodellen können eine Art von Architekturbaustein im Sinne der obigen Definition von Softwarearchitektur sein.

Ein **Framework** ist ein durch den Softwareentwickler anpassbares oder erweiterbares System kooperierender Klassen, die einen wiederverwendbaren Entwurf für einen bestimmten Anwendungsbereich implementieren. Es besteht aus konkreten und abstrakten Klassen, die Schnittstellen definieren.

Pakete (*packages*) sind ein Strukturierungsmechanismus, um Klassen und Pakete zu einer Einheit zusammenzufassen. In der UML gruppieren ein Paket Modellemente und Diagramme. Ein Paket kann selbst Pakete enthalten.

Auf der Architekturebene sind Subsysteme, Komponenten, Frameworks und Pakete typische Architekturbausteine.

I 6 Was ist eine Softwarearchitektur?

Abstraktionsebenen

Um von den Anforderungen an ein Softwaresystem zur technischen Lösung zu gelangen, sind mehrere Abstraktionsebenen zu »überwinden«. Die Softwarearchitektur beschreibt die Softwarestruktur auf den oberen Abstraktionsebenen. Sie abstrahiert und verbirgt Details.

Beispiel Implementiert ein Baustein einen Sortieralgorithmus, dann ist der Sortieralgorithmus nicht Bestandteil der Architektur. Die Schnittstelle, über die der Sortieralgorithmus in Anspruch genommen werden kann, gehört jedoch zur Architektur. Ebenfalls müssen das Laufzeitverhalten (z. B. $O(n \log_2 n)$) und das Speicherverhalten (z. B. $O(n)$) bekannt sein sowie weitere Informationen zum Sortierverhalten.

Fachkonzept Ausgangspunkt für alle Architekturüberlegungen ist ein Modell des Fachkonzepts, das im *Requirements Engineering* entstanden ist. Dieses Fachkonzept kann bereits in Subsysteme, Komponenten und Pakete untergliedert sein.

Logische Architektur Basierend auf dem Fachkonzept wird eine logische Architektur entworfen, die das Fachkonzept um Subsysteme ergänzt, um zu einem lauffähigen System zu gelangen. In der Regel wird zuerst die statische Sicht modelliert. In vielen Anwendungssystemen wird insbesondere ein Subsystem Benutzungsoberfläche und ein Subsystem Persistenz benötigt.

Hierarchische Verfeinerung Jedes Subsystem kann dann hierarchisch weiter in Komponenten und Pakete unterteilt werden.

Eine logische Architektur kann also aus mehreren Abstraktionsebenen bestehen. Die Granularität auf der untersten Abstraktionsebene muss so fein sein, dass die Verknüpfung zur softwaretechnischen Infrastruktur hergestellt werden kann.

Softwaretechnische Infrastruktur Jedes Softwaresystem benötigt eine softwaretechnische Infrastruktur, deren Dienstleistungen es in Anspruch nimmt. Im einfachsten Falle wird beispielsweise für die Entwicklung ein Compiler in der jeweiligen Programmiersprache und ein entsprechendes Laufzeitssystem auf dem Zielcomputersystem benötigt. Bei einer objektorientierten Entwicklung gehören zur jeweiligen Programmiersprache Klassenbibliotheken, die für die Realisierung der logischen Architektur benötigt werden.

Beispiel Für die Realisierung des Subsystems Benutzungsoberfläche werden die Swing-Klassen (softwaretechnische Infrastruktur) von Java verwendet.

Die **logische Architektur** wird also mit der softwaretechnischen Infrastruktur verknüpft. Dies bezeichnet man oft auch als **Feinentwurf**. Bei unternehmensweiten, verteilten Anwendungen besteht die

6 Was ist eine Softwarearchitektur? I

softwaretechnische Infrastruktur oft aus einer sogenannten *Middleware*, die eine Vielzahl von Dienstleistungen zur Verfügung stellt (siehe »Softwaretechnische Infrastrukturen«, S. 319).

Ist die logische Architektur mit der softwaretechnischen Infrastruktur modellmäßig verknüpft, dann können die einzelnen Bausteine implementiert, das heißt programmiert werden. Es entsteht eine Vielzahl von miteinander verknüpften Dateien.

Neben einer logischen Architektur muss auch eine physische Architektur entworfen werden, insbesondere wenn es sich um eine verteilte Anwendung handelt. Unter einer physischen Architektur werden die Computer, die Netzwerke, die sie verbinden, und weitere Hardwarekomponenten verstanden. Die implementierten Bausteine müssen nun auf die physische Architektur abgebildet werden. In der Regel bedeutet dies, dass gepackte Dateien der softwaretechnischen Infrastruktur der physischen Architektur zuordnet werden müssen.

Bei der Installation müssen die der physischen Architektur zugeordneten Dateien auf der realen technischen Infrastruktur verteilt, installiert und konfiguriert werden, oft *Deployment* genannt (siehe »Verteilung und Installation«, S. 521).

Implementierung

Physische Architektur

Technische Infrastruktur

7 Architekturprinzipien

Prinzipien sind Grundsätze, die man seinem Handeln zugrunde legt. Prinzipien sind allgemeingültig, abstrakt, allgemeinsten Art. Sie bilden eine theoretische Grundlage. Prinzipien werden aus der Erfahrung und Erkenntnis hergeleitet und durch sie bestätigt.

Definition

In dem »Lehrbuch der Softwaretechnik – Basiskonzepte und Requirements Engineering« [Balz09a, S. 25 ff.] werden folgende allgemeinen Prinzipien vorgestellt, die für die Softwaretechnik relevant sind:

- Prinzip der Abstraktion
- Prinzip der Strukturierung
- Prinzip der Bindung und Kopplung
- Prinzip der Hierarchisierung
- Prinzip der Modularisierung
- Geheimnisprinzip
- Prinzip der Lokalität
- Prinzip der Verbalisierung

Alle diese Prinzipien sind auch für den Softwareentwurf relevant, in Abhängigkeit von den funktionalen und nichtfunktionalen Anforderungen in unterschiedlicher Gewichtung und Ausprägung.

In der Literatur werden noch weitere spezifische Architekturprinzipien diskutiert, oft auch unter anderen Bezeichnungen wie Leitprinzipien, Qualitätsindikatoren, Hinweise (*considerations*). Bisweilen wird auch von »architektonischer Schönheit« gesprochen. Eine Reihe dieser Architekturprinzipien wird näher vorgestellt:

- »Architekturprinzip: Konzeptionelle Integrität«, S. 30
- »Architekturprinzip: Trennung von Zuständigkeiten«, S. 31
- »Architekturprinzip: Ökonomie«, S. 32
- »Architekturprinzip: Symmetrie«, S. 33
- »Architekturprinzip: Sichtbarkeit«, S. 34
- »Architekturprinzip: Selbstorganisation«, S. 34

Die aufgeführten Prinzipien können durch verschiedene Maßnahmen erreicht werden, beispielsweise durch den Einsatz geeigneter Architektur- und Entwurfsmuster (siehe »Architektur- und Entwurfsmuster«, S. 37).

I 7 Architekturprinzipien

7.1 Architekturprinzip: Konzeptionelle Integrität

Frage Was stellen Sie sich unter dem Architekturprinzip »Konzeptionelle Integrität« vor?

Antwort Eine Softwarearchitektur besitzt *keine* »Konzeptionelle Integrität«, wenn für gleichartige Aufgabenstellungen unterschiedliche Architekturansätze, z. B. unterschiedliche Entwurfsmuster oder Variationen davon, verwendet werden. Das führt zu einer schlechten Wartbarkeit und Verständlichkeit.

Das **Architekturprinzip** der **Konzeptionellen Integrität** (*conceptual integrity*) ist eingehalten, wenn Entwurfsentscheidungen im gesamten System durchgängig angewendet und Speziallösungen vermieden werden.

- Beispiel**
- Entwickler A verwendet für die Fehlerbehandlung das Konzept der Ausnahmebehandlung.
 - Entwickler B verwendet für Fehlermeldungen Rückgabewerte.
 - Entwickler C protokolliert Fehler lediglich in einem Logbuch.

Beispiel Das Architekturprinzip wird verletzt, wenn in einer verteilten Architektur für die Komponenten mehrere verschiedene Schnittstellen und Implementierungen für die Kommunikation angeboten werden.

Die Beachtung dieses Architekturprinzips bringt folgende Vorteile mit sich:

- + Das Verständnis der Architektur wird durch die konzeptionelle Integrität erleichtert.
- + Durch die Vermeidung von Speziallösungen und punktuellen Ausnahmen wird die Komplexität der Architektur reduziert.

Die Einhaltung dieses Prinzips wird bei einer inkrementellen Entwicklung einer Architektur schwierig, da das Entstehen einer Architektur über einen längeren Zeitraum oft zu Stilbrüchen führt. Bei Änderungen einer Architektur wird oft nicht die gesamte Architektur bearbeitet, sondern nur Teile, was Integritätsbrüche zur Folge haben kann. Ebenso kommt es bei der Implementierung einer Architektur zur Verletzung der Integrität, wenn Entwickler wegen besonderer Anforderungen, Optimierungsanforderungen oder fehlendem Problembewusstsein Speziallösungen erfinden.

Terminologie Synonym zu dem Begriff »Konzeptionelle Integrität« findet man in der Literatur oft noch folgende Begriffe: Prinzip der kleinstmöglichen Überraschung, Entwurfssymmetrie, Einheitlichkeit.

Literatur [Stal09, S. 141], [ReHa09, S. 89 f.], [PBG07, S. 108]

7.2 Architekturprinzip: Trennung von Zuständigkeiten I

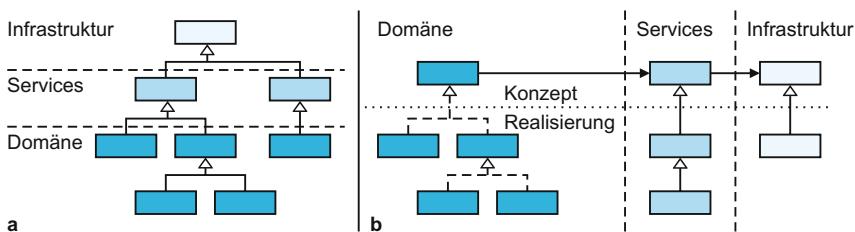
7.2 Architekturprinzip: Trennung von Zuständigkeiten

Was können Sie sich unter dem Architekturprinzip »Trennung von Zuständigkeiten« vorstellen? [Frage](#) [Antwort](#)

Jede Komponente einer Architektur ist bei dem Prinzip **Trennung von Zuständigkeiten** (*separation of concerns*) für eine Aufgabe oder einen Aufgabenkomplex verantwortlich. Die Aufgabe ist *nicht* über mehrere Komponenten verteilt. Eine Komponente ist also verantwortlich für eine einzige Aufgabe und erledigt diese Aufgabe richtig. Ist eine Komponente für einen Aufgabenkomplex zuständig, dann soll die Komponente für jede von ihr zu erledigende Aufgabe eine eigene Schnittstelle zur Verfügung stellen.

Die Abb. 7.2-1 zeigt zwei verschiedene Möglichkeiten, Zuständigkeiten zu trennen [BuHe10a, S. 65]. Auf der linken Seite werden Zuständigkeiten durch eine Vererbungshierarchie abgebildet, die zu überfrachteten Blatt-Klassen führen können. Die Wurzel-Klasse stellt die Infrastruktur zur Verfügung, Unterklassen sorgen für die Anpassung an die aktuellen Domänen-Unterklassen. Diese Art der Trennung von Zuständigkeiten erhöht die Abhängigkeiten der Blatt-Klassen.

Auf der rechten Seite der Abb. 7.2-1 ist eine Trennung der Zuständigkeiten durch die Verwendung der Komposition vorgenommen worden (siehe »Schnittstellen, Fabriken und Komposition«, S. 503). Der Fokus liegt auf den Domänen-Klassen. Die Infrastruktur-Klassen befinden sich erst an dritter Stelle. Die Delegation durch konzeptionelle Wurzeln, ausgedrückt durch reine Schnittstellen, führt zu einer strikten und besser separierten Schichtenbildung.



Bei der Implementierung wird dieses Prinzip manchmal verletzt, weil aus Effizienzgründen oder wegen der Beschränkung der Programmiersprache mehrere Aspekte im Code einer Komponente realisiert sind, *Tangled Code* genannt.

Die Einhaltung dieses Prinzips bringt folgende Vorteile mit sich:

- + Die Sichtbarkeit wird erhöht, da die Zuständigkeiten klar voneinander abgegrenzt sind.

Abb. 7.2-1: Zwei Optionen, um die Trennung von Zuständigkeiten zu realisieren (in Anlehnung an [BuHe10, S. 65]).

I 7 Architekturprinzipien

- + Die Trennung von Zuständigkeiten führt zu einer modularen Architektur.

Die Aufteilung von Zuständigkeiten muss ausgewogen sein. Eine zu starke Trennung von Zuständigkeiten führt zu einer Fragmentierung und einem Verlust an Sichtbarkeit. Durch die Analyse von Gemeinsamkeiten und Unterschieden kann identifiziert werden, was in einer Architektur getrennt werden sollte und was als ganze, kohärente Einheit zu betrachten ist.

Terminologie In der Literatur werden für dieses Prinzip auch die Namen *Spacing* [BuHe10a, S. 65] und »Trennung von Belangen« verwendet [Stal09, S. 143].

Literatur [Stal09, S. 143], [ReHa09, S. 79 f.], [BuHe10a, S. 65]

7.3 Architekturprinzip: Ökonomie

Frage Was könnte das Architekturprinzip »Ökonomie« für eine Softwarearchitektur bedeuten?

Antwort Ein ökonomischer Architekturentwurf sorgt dafür, dass Übersichtlichkeit, Verdichtung und Abstraktion in einem ausgewogenen Verhältnis berücksichtigt werden (in Anlehnung an [BuHe10a, S. 63]). Verdichtung meint Geradlinigkeit und Klarheit im Entwurf.

Übersichtlichkeit dient dazu, ein Durcheinander zu vermeiden. Verdichtung erleichtert das Verständnis der Architektur. Abstraktion sorgt dafür, dass unnötiges Detail verschwindet. Durch dieses Prinzip soll verhindert werden, dass Softwarearchitekturen mit unnötiger Komplexität belastet werden.

Beispiele Beispiele für unnötige Komplexität sind: Willkürliche Flexibilität; unnötige Funktionen; Entwurfsentscheidungen, deren Komplexität nicht problemgerecht ist; Fokus auf Wiederverwendbarkeit statt auf die Benutzbarkeit.

Terminologie Im Zusammenhang mit dem Prinzip Ökonomie werden oft in ähnlicher oder leicht abgewandelter Bedeutung die Prinzipien **Einfachheit** und **Minimalismus** genannt.

Zitat »Je weniger Subsysteme, Komponenten und Interaktionen in einem Softwaresystem auftreten, desto verständlicher die zugehörige Architektur und desto geringer die Gefahr von Fehlern. Unbeabsichtigte Komplexität ergibt sich beispielsweise, sobald Architekten nach Entwurfsperlen streben oder das System auf alle Eventualitäten vorbereiten wollen. Unnötige Indirektionsstufen (accidental complexity) reduzieren in diesem Fall direkt die architektonische Einfachheit« [Stal09, S. 142].

7.4 Architekturprinzip: Symmetrie I

Zu beachten ist allerdings, dass jede Aufgabe eine inhärente Komplexität besitzt. Daher können keine Lösungen mit geringerer Komplexität vorhanden sein.

[BuHe10a, S. 63 f.], [Stal09, S. 142]

Literatur

7.4 Architekturprinzip: Symmetrie

Was können Sie sich unter dem Architekturprinzip »Symmetrie« vorstellen? Frage

Symmetrie macht sich durch einen konsistenten, geordneten und ausgewogenen Entwurf bemerkbar. Elemente wiederholen und stützen sich gegenseitig, machen es leichter das Verhalten des Systems vorherzusagen, es zu warten und auszubauen [BuHe10b, S. 12]. Antwort

Es lassen sich verhaltensorientierte und strukturelle Symmetrien unterscheiden.

- Zu jeder Initialisierungsfunktion soll es auch eine komplementäre Abschlussfunktion geben.
- Zu einem Verbindungsaufbau in einer verteilten Anwendung soll es auch einen Verbindungsabbau im selben Kontext geben, z.B. Herstellung einer Datenbank-Verbindung durch `connect()` und Trennen durch `disconnet()`.
- Jedes erzeugende Entwurfsmuster (siehe »Architektur- und Entwurfsmuster«, S. 37) wird vollständiger und nützlicher, wenn es auch ein Konzept zur »Entsorgung« bzw. »Zerstörung« enthält.
- Jede gestartete Transaktion muss auch beendet werden.

Beispiele für verhaltensorientierte Symmetrien

Verhaltensorientierte Symmetrien führen oft zu strukturellen Symmetrien.

Die Erkenntnis, dass in einem erzeugenden Entwurfsmuster auch ein Konzept zur »Entsorgung« bzw. »Zerstörung« fehlt, führt zu einem modifizierten Muster, das strukturell ein Konzept zur »Entsorgung« bzw. »Zerstörung« enthält. Beispiel

Wenn Asymmetrie jedoch zu einem einfacheren Entwurf führt, dann sollte die Symmetrie *nicht* erzwungen werden.

In Java werden Objekte explizit erzeugt. Ihre Zerstörung wird implizit durch den *Garbage Collector* vorgenommen. Beispiel

Die Beachtung des Prinzips der Symmetrie bringt folgende Vorteile mit sich:

- ⊕ Ein symmetrischer Entwurf ist leichter zu verstehen, nachzuvollziehen und zu kommunizieren.
- ⊕ Die Paarung von »Erzeugung« und »Zerstörung« führt in manchen Fällen zu einer Verbesserung der Architekturqualität.

I 7 Architekturprinzipien

Symmetriebrüche sind erlaubt, müssen aber gut begründet sein.

Tipp Die meisten Softwarearchitekturen benötigen mehr Symmetrie und nicht weniger. Wenn Sie im Zweifel sind, dann machen Sie Ihre Architektur symmetrisch.

Literatur [BuHe10b, S. 12 f.], [Stal09, S. 141]

7.5 Architekturprinzip: Sichtbarkeit

Frage Was stellen Sie sich unter dem Architekturprinzip »Sichtbarkeit« (*visibility*) vor?

Antwort Eine Architektur ist dann gut »sichtbar«, wenn ihre Artefakte deutlich und ausdrucksstark beschrieben sind. Das **Prinzip der Verbalisierung** ist eine Untermenge des Prinzips der Sichtbarkeit. Eine schlechte Sichtbarkeit liegt vor, wenn wichtige Entwurfs- und Fachkonzepte nicht beschrieben oder implizit in der Architektur verborgen sind.

Beispiel Schnittstellen sind oft nicht ausdrucksstark beschrieben und ihre Aufteilung ist unklar. Eine Dokumentation fehlt oder ist ausschweifend. Die Folgen davon sind eine schlechte Modifizierbarkeit und Wartbarkeit. Die Einarbeitung ist aufwändig.

Beispiel Variablen vom Typ *String* repräsentieren oft Konzepte der Domäne, z. B. ISBN-Nummern oder SQL-Anweisungen. Dies ist jedoch nicht die geeignete Repräsentation in einem Programm. Besser ist es, einen expliziten Typ für ein solches Konzept einzuführen. Dadurch werden duplizierter Code und unausgesprochene Entwurfsannahmen vermieden.

Die Verwendung von Mustern macht Konzepte sichtbar.

Das Prinzip der Sichtbarkeit bringt folgenden Vorteil mit sich:

+ Änderbarkeit, Wartbarkeit und Einarbeitung werden erleichtert.

Literatur [BuHe10a, S. 64 f.]

7.6 Architekturprinzip: Selbstorganisation

Frage Was fällt Ihnen zum Architekturprinzip »Selbstorganisation« ein?

Antwort In vielen Softwaresystemen gibt es einen Trend zu zentralisierten und expliziten Kontroll-, Steuerungs- und Entscheidungsstrukturen. Eine Reihe von Aufgaben lassen sich aber besser über dezentrale Ansätze lösen. Die Kontrolle ist dann nicht einer einzelnen Komponente zugeordnet, sondern die beteiligten Komponenten organisieren sich selbst.

7.6 Architekturprinzip: Selbstorganisation I

Gibt es für wichtige Funktionen nur eine Server-Instanz, dann wird Beispiel diese zum Flaschenhals bezogen auf Zuverlässigkeit, Verfügbarkeit und Skalierbarkeit.

Das Schreiben von nebenläufigen Programmen, die in mehreren Beispiel *threads* ablaufen, ist schwierig. Wenn sie nicht sorgfältig entworfen werden, leiden sie unter *race conditions*. Um solche Probleme zu vermeiden, werden häufig übermäßig Synchronisationsmechanismen und eine strikte zentralisierte Laufzeitkontrolle eingesetzt. Als Effekte ergeben sich oft *deadlocks* sowie langsame, instabile und kaum skalierbare Programme – genau das Gegenteil von dem, was man erreichen wollte.

Eine dezentralisierte Steuerung ist oft besser, um die Vorteile der Nebenläufigkeit zu nutzen.

Der Entwurf einer Architektur mit sich selbst organisierenden Komponenten erfordert ein tiefes Verständnis der System-Domäne.

Anstelle des Begriffs Selbstorganisation wird oft auch die Bezeichnung »Emergenz« (*emergence*) verwendet. Terminologie

Die Beachtung des Prinzips der Selbstorganisation bringt folgenden Vorteil mit sich:

- ✚ Eine sich selbst organisierende Steuerung ist oft der Schlüssel zu skalierbaren, effizienten und ökonomischen Softwarearchitekturen.

[BuHe10b, S. 13 f.], [Stal09, S. 144]

Literatur

8 Architektur- und Entwurfsmuster

In klassischen Ingenieurdisziplinen werden große Teile früherer Lösungen wiederverwendet. Diese Entwurfsform wird in [ShGa96] als Entwurf-durch-Routine bezeichnet.

Entwurf-durch-Routine

Werden für unbekannte Aufgabenstellungen neue Lösungen benötigt, dann spricht man von Entwurf-durch-Innovation [Star05, S. 160].

Entwurf-durch-Innovation

Bereits 1995 hat man erkannt, dass viele Softwaresysteme auf bekannten Lösungen aufsetzen können. In dem berühmten Buch [GHJ+95] wurden zum ersten Mal systematisch bewährte Entwurfslösungen klassifiziert beschrieben – in der Literatur werden die vier Autoren auch oft als Viererbande (*the Gang of Four* oder GoF) bezeichnet. In dem Buch werden 23 Entwurfsmuster beschrieben. Die Autoren haben diese Entwurfsmuster aus ihren Erfahrungen mit sequenziellen, objektorientierten *User Interface Frameworks* in den Programmiersprachen Smalltalk und C++ abgeleitet.

Zur Historie

Heute unterscheidet man oft zwischen Architekturmustern und Entwurfsmustern.

Architekturmuster (*architecture pattern*) beschreiben Systemstrukturen, die die Gesamtarchitektur eines Systems festlegen. Sie spezifizieren, wie Subsysteme zusammenarbeiten.

Definition

Architekturmuster sind weitgehend unabhängig von der fachlichen Domäne des Softwaresystems. Sie bieten Lösungen für technische, meist querschnittliche Aspekte von Softwaresystemen, wie Verteilung, Transaktionen, Nebenläufigkeit, Persistenz oder Echtzeitverhalten. Oft werden Architekturmuster auch als **Basisarchitekturen** bezeichnet – bisweilen auch als **Architekturstil**.

Entwurfsmuster (*design patterns*) geben bewährte generische Lösungen für häufig wiederkehrende Entwurfsprobleme an, die in bestimmten Situationen auftreten. Sie legen die Struktur von Subsystemen fest.

Definition

Entwurfsmuster werden auch als **Mikro-Architekturen** bezeichnet. Sie werden oft als Teile einer Lösung eines Architekturmusters eingesetzt. Sie sind spezifisch und wirken lokal.

I 8 Architektur- und Entwurfsmuster

Architekturmuster und Entwurfsmuster lassen sich *nicht* scharf voneinander absetzen. Es gibt einen fließenden Übergang zwischen ihnen.

Tipp

Bevor Sie etwas Neues entwickeln, prüfen Sie, ob es nicht bereits bewährte Muster für Ihr Problem gibt. Praktizieren Sie einen Entwurf-durch-Routine. Dadurch erhöhen Sie die Wiederverwendbarkeit und senken gleichzeitig das Entwicklungsrisiko.

Beschreibung von Mustern

Um Muster miteinander vergleichen zu können, sollten sie in einem einheitlichen Format beschrieben werden. In der Literatur werden verschiedene Beschreibungsschablonen mit unterschiedlichem DetAILierungsgrad angegeben, z.B. in [BMR+96, S. 20 f.] und [GHJ+95, S. 6 ff.]. Hier wird folgende Beschreibungsschablone verwendet:

- **Name(n)**: Der Name bzw. die Namen – falls mehrere bekannt – des Musters und eine kurze Zusammenfassung.
- **Grundidee**: Kurze Beschreibung der Idee, die dem Muster zugrunde liegt.
- **Anwendungsbereich(e)**: Angabe, wo das Muster vorwiegend eingesetzt wird.
- **Beispiel(e)**: Ein oder mehrere echte Beispiele, die zeigen, dass ein Problem vorhanden ist, und dass ein Muster benötigt wird.
- **Problem/Kontext**: Beschreibung des Problems und des Kontextes, für das das Muster geeignet ist.
- **Lösung**: Das grundsätzliche Lösungsprinzip des Musters einschl. einer Beschreibung der strukturellen und dynamischen Aspekte sowie Hinweise zur Implementierung.
- **Varianten**: Eine kurze Beschreibung von Varianten oder Spezialisierungen des Musters.
- **Vor- und Nachteile**: Die Vorteile und die Nachteile, die das Muster mit sich bringt.
- **Zusammenhänge**: Verweise auf Muster, die ähnliche Probleme lösen, und zu Mustern, die helfen, das beschriebene Muster zu verfeinern. Bezüge zu nichtfunktionalen Anforderungen.

Klassifizierung von Architektur- und Entwurfsmustern

Muster lassen sich nach verschiedenen Kriterien klassifizieren.

Klassifikation nach Zweck und Geltungsbereich

Entwurfsmuster lassen sich zu Familien verwandter Muster zusammenfassen. Eine Klassifikation kann nach den Kriterien

- Zweck (*purpose*) und
- Geltungsbereich (*scope*) erfolgen (Tab. 8.0-1) [GHJ+95].

8 Architektur- und Entwurfsmuster I

| Zweck→ ↓Geltungsbereich | Erzeugendes Muster | Strukturelles Muster | Verhaltensmuster |
|----------------------------|---|---|--|
| Klasse | <ul style="list-style-type: none"> ■ Fabrikmethode <i>(factory method)</i> | <ul style="list-style-type: none"> ■ Adapterklasse | <ul style="list-style-type: none"> ■ Interpreter <i>(interpreter)</i> ■ Schablonenmethode <i>(template)</i> |
| Objekt | <ul style="list-style-type: none"> ■ Abstrakte Fabrik <i>(abstract factory)</i> ■ Erbauer <i>(builder)</i> ■ Prototyp <i>(prototype)</i> ■ Singleton <i>(singleton)</i> | <ul style="list-style-type: none"> ■ Adapter <i>(adapter)</i> ■ Brücke <i>(bridge)</i> ■ Kompositum <i>(composite)</i> ■ Dekorierer <i>(decorator)</i> ■ Fassade <i>(facade)</i> ■ Fliegengewicht <i>(flyweight)</i> ■ Proxy <i>(proxy)</i> | <ul style="list-style-type: none"> ■ Zuständigkeitskette <i>(chain of responsibility)</i> ■ Kommando <i>(command)</i> ■ Iterator <i>(iterator)</i> ■ Vermittler <i>(mediator)</i> ■ Memento <i>(memento)</i> ■ Beobachter <i>(observer)</i> ■ Zustand <i>(state)</i> ■ Strategie <i>(strategy)</i> ■ Besucher <i>(visitor)</i> |

Hinweis: Hier behandelte Muster sind in der Tab. 8.0-1 fett dargestellt.

Der Zweck gibt an, was ein Muster bewirkt:

- Ein erzeugendes Muster (*creational pattern*) befasst sich mit der Erzeugung von Objekten.
- Ein strukturelles Muster (*structural pattern*) beschreibt die Komposition von Klassen und Objekten.
- Ein Verhaltensmuster (*behavioral pattern*) beschreibt, wie Klassen oder Objekte miteinander kommunizieren und wie die Verantwortlichkeiten verteilt sind.

Der Gültigungsbereich gibt an, ob sich das Muster primär auf

- Klassen oder
- Objekte bezieht.

Klassenmuster behandeln Beziehungen zwischen Klassen und ihren Unterklassen. Diese Beziehungen werden durch Vererbungen ausgedrückt und sind statisch, d. h. sie werden zur Übersetzungszeit festgelegt.

Objektmuster beschreiben demgegenüber Beziehungen zwischen Objekten, die zur Laufzeit geändert werden können und dynamisch sind. Fast alle Muster benutzen bis zu einem gewissen Grad die Vererbung.

Erzeugende Klassenmuster verschieben einen Teil der Objekterzeugung hin zu Unterklassen, während erzeugende Objektmuster einen Teil der Objekterzeugung zu anderen Objekten hin verschieben.

*Tab. 8.0-1:
Klassifizierung
von
Entwurfsmustern
[GHJ+95, S. 10].*

Klassenmuster

Objektmuster

I 8 Architektur- und Entwurfsmuster

Strukturelle Klassenmuster verwenden die Vererbung, um Klassen miteinander zu kombinieren, während strukturelle Objektmuster Wege angeben, um Objekte zusammenzufügen.

Verhaltensmuster für Klassen benutzen die Vererbung, um Algorithmen und Kontrollflüsse zu beschreiben, während Verhaltensmuster für Objekte beschreiben, wie eine Gruppe von Objekten zusammenarbeitet, um eine Aufgabe auszuführen, die ein Objekt alleine nicht erledigen kann.

Klassifikation nach Anwendungs- und Verteilungsarten

- **Muster für verteilte Systeme:** Allein für verteilte Systeme werden in [BHS07] über 250 Muster aufgeführt. Weitere Muster sind in [VKZ02] beschrieben.
- **Muster für sicherheitsrelevante Systeme:** Für sicherheitsrelevante Bereiche wie Authentifizierung, Autorisierung und Vertraulichkeit gibt es eine große Bandbreite von Mustern, z.B. veröffentlicht in [SFH+06] und [Stee06].
- **Muster für fehlertolerante Systeme:** Die zunehmende Integration von Software in sicherheitskritische Systeme erfordert robuste Techniken, um die geforderten Anforderungen zu erfüllen. Eine ganze Reihe von Mustern beschäftigen sich daher mit der Fehlertoleranz und dem Fehlermanagement von Systemen: [Uta05] und [Pont01].
- **Muster für eingebettete Systeme:** Software ist immer mehr untrennbarer Bestandteil von Systemen wie Herzschrittmachern, Kraftwerken, Bordelektronik von Flugzeugen usw. In diesen Systemen müssen physikalische Anforderungen und Randbedingungen unter Beachtung von Zeitbedingungen beachtet werden. Eine Reihe von Mustern berücksichtigt diese Randbedingungen, ohne die Vorteile von Abstraktionen aufzugeben: [Pont01], [NoWe00], [DiGi09]. Einige konkrete Architekturmuster werden im Kapitel »Architekturmuster für Eingebettete Systeme«, S. 408 ff., näher betrachtet. Neben dem Erkennungs-, Wiederherstellung- und Vermeidungsmuster werden noch das PSC- und das Homogeneous Redundancy-Muster behandelt.

Weitere Kriterien

Andere Klassifikationskriterien sind:

- Muster, die oft zusammen verwendet werden.
- Muster, die Alternativen darstellen.
- Muster, die zu ähnlichen Entwürfen führen.
- Muster, die miteinander in Beziehung stehen.
- Muster, die Architektur- und Entwurfsprinzipien unterstützen.
- Muster, die helfen, nichtfunktionale Anforderungen zu realisieren.

Beziehungen zwischen Mustern

Es gibt Muster, die für sich allein stehen (*stand-alone-patterns*), und Muster, die Beziehungen zu anderen Mustern haben.

Allein stehende Muster sind

Beispiele

- das Strategie-Muster, das ein plug-in-Verhalten ermöglicht, und
- das Muster *Wrapper Facade*, das es ermöglicht, vorhandene Subsysteme einzukapseln.

Es lassen sich drei Beziehungsarten zwischen Mustern unterscheiden:

- **Sich ergänzende Muster:** Ein Muster ergänzt ein anderes Muster oder stellt eine alternative Lösung zur Verfügung. Kooperative Ergänzungen führen zu einem vollständigeren und balancierten Entwurf. Muster können auch miteinander konkurrieren.
- **Zusammengesetzte Muster:** Muster, die oft zusammengesetzt verwendet werden, werden als eigenständiges Muster betrachtet. Beispielsweise wird das Muster *Command* oft als Composite-Muster implementiert, sodass sich der Musternname *Composite Command* eingeprägt hat.
- **Sequenzen von Mustern:** Muster, die in einer Reihenfolge angeordnet werden, wobei die Vorgänger-Muster die Voraussetzungen für die Nachfolge-Muster bereitstellen.

Der Begriff **Muster-Sprachen** (*pattern languages*) wird verwendet, wenn eine Menge von Mustern ganzheitlich dazu dienen, Software für eine spezifische technische Domäne oder eine spezifische Anwendungs-Domäne zu entwerfen.

Inzwischen gibt es eine Vielzahl von beschriebenen Mustern. In diesem Buch können und sollen nur einige wichtige und häufig verwendete Muster beschrieben werden, die das »Entwerfen im Großen« unterstützen und die dazu beitragen, Prinzipien und nichtfunktionale Anforderungen zu realisieren.

Zunächst wird der Callback-Mechanismus erläutert, der in vielen Entwurfsmustern eine Rolle spielt:

- »Exkurs: *Callback*«, S. 42

Das Schichten-Muster ist eines der wichtigsten Muster für die »Softwarearchitektur im Großen«:

- »Das Schichten-Muster (*layers pattern*)«, S. 46

Das Beobachter-Muster ermöglicht es, Komponenten zu entkoppeln:

- »Das Beobachter-Muster (*observer pattern*)«, S. 54

Das MVC-Muster ist eine Verfeinerung des Beobachter-Musters und strukturiert – insbesondere bei Web-Anwendungen – die gesamte Architektur:

- »Das MVC-Muster (*model view controller pattern*)«, S. 62

I 8 Architektur- und Entwurfsmuster

| | |
|-----------|---|
| | Das Fassaden-Muster ermöglicht einen vereinfachten Zugriff auf ein komplexes Subsystem: ■ »Das Fassaden-Muster (<i>facade pattern</i>)«, S. 69 |
| | Das Kommando-Muster entkoppelt Sender vom Empfänger: ■ »Das Kommando-Muster (<i>command pattern</i>)«, S. 75 |
| | Das Proxy-Muster kapselt den Zugriff auf ein Objekt durch ein vor-gelagertes Stellvertreter-Objekt: ■ »Das Proxy-Muster (<i>proxy pattern</i>)«, S. 83 |
| | Das Fabrikmethoden-Muster delegiert die Objekterzeugung an Unterklassen: ■ »Fabrikmethoden-Muster (<i>factory method pattern</i>)«, S. 89 |
| | Das Strategie-Muster erlaubt es, einen Algorithmus – ohne Änderung an der nutzenden Klasse – auszutauschen: ■ »Das Strategie-Muster (<i>strategy pattern</i>)«, S. 96 |
| | Das Brücken-Muster ermöglicht es, eine Abstraktion von ihrer Imple-mentierung zu entkoppeln: ■ »Das Brücken-Muster (<i>bridge pattern</i>)«, S. 103 |
| Website | Eine der umfangreichsten Sammlungen von Architektur- und Ent-wurfsmustern enthält die Website Handbook of Software Architec-ture (http://www.handbookofsoftwarearchitecture.com). Über 2000 Mus-ter sind auf dieser Website katalogisiert. Nach folgenden Kriterien kann gesucht werden: nach Disziplin, nach Domäne, nach Technik, nach Typ, nach Entwicklungsphase. Zusätzlich kann nach Name oder nach Quelle recherchiert werden. |
| Literatur | [BMR+96], [GHJ+95], [BHS07b] |

8.1 Exkurs: *Callback*

Lesehinweis

In vielen Entwurfsmustern wird der *Callback*-Mechanismus di-rekt oder indirekt verwendet. Wenn Sie diesen Mechanismus noch nicht kennen, dann sollten Sie sich zunächst damit befassen, be-vor Sie die einzelnen Entwurfsmuster durcharbeiten.

Name(n)

Der *Callback*-Mechanismus wird auch Rückruf-Mechanismus, *Inver-sion of Control* (IoC) oder Steuerungsumkehr genannt.

Grundidee

Ein Rückruf (*callback*) ist eine Referenz auf einen ausführbaren Codeteil. Die Referenz auf diesen Codeteil wird als Parameter über eine Parameterliste an einen anderen Codeteil übergeben. Von diesem Codeteil aus kann dann über die Referenz zu einem beliebi-gen Zeitpunkt der referenzierte Codeteil aufgerufen werden. In Pro-grammiersprachen wie C/C++ wird eine Methode bei einem anderen

8.1 Exkurs: Callback I

Objekt registriert. Dies anderes Objekt kann dann die Methode zurück aufrufen. In Java gibt es *keine* Möglichkeit, um eine Methode zu registrieren, d. h. einen Funktionszeiger über die Parameterliste zu übergeben. In Java wird daher über die Parameterliste ein Objekt an ein anderes Objekt übergeben. Dieses andere Objekt ruft dann die Methode bei dem übergebenen Objekt auf.

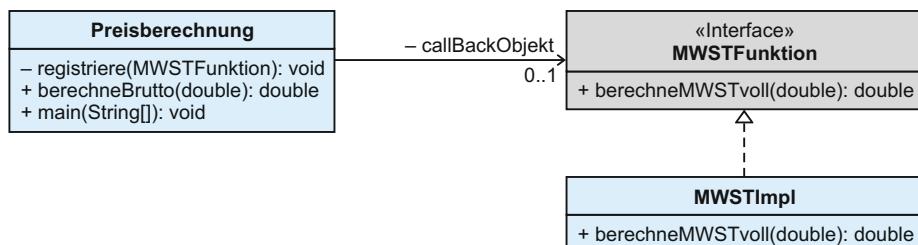
Anwendungsbereich(e)

- Es ist dadurch möglich, aus einer niedrigeren Softwareschicht eine Funktion oder Methode aufzurufen, die in einer höheren Softwareschicht angeordnet ist. Dadurch werden Schichten bzw. Komponenten einer Software entkoppelt.
- Nützlich für die Implementierung verschiedener Entwurfsmuster wie Besucher (*visitor*), Beobachter (siehe »Das Beobachter-Muster (*observer pattern*)«, S. 54) und Strategie (siehe »Das Strategie-Muster (*strategy pattern*)«, S. 96).
- Eine Methode einer Anwendung wird bei einer Standardbibliothek registriert und von dieser zu einem späteren Zeitpunkt aufgerufen.
- Ein Zeichenprogramm erhält über die Parameterliste die Funktion übergeben, die gezeichnet werden soll, zum Beispiel $\sin(x)$ oder $\cos(x)$.

Beispiel(e)

In einer Klassen-Bibliothek befindet sich eine Funktion zur Berechnung der Mehrwertsteuer. In einem Anwendungsprogramm wird eine Referenz auf diese Funktion dadurch gesetzt, dass eine entsprechende Objektreferenz gespeichert wird. Die Abb. 8.1-1 zeigt das zugehörige UML-Diagramm.

Beispiel



Die Java-Klassen sehen wie folgt aus:

```
public interface MWSTFunktion
{
    public double berechneMWSTvoll(double wert);
}

//Klasse z. B. in einer Bibliothek
public class MWSTImpl implements MWSTFunktion
```

Abb. 8.1-1: Veranschaulichung des Callback-Mechanismus am Beispiel einer MWST-Funktion.

I 8 Architektur- und Entwurfsmuster

```
{  
    public double berechneMWSTvoll(double wert)  
    {  
        return wert*1.19;  
    }  
}  
  
//Klasse z. B. in einer Anwendung  
public class Preisberechnung  
{  
    private MWSTFunktion callBackObjekt = null;  
  
    private void registriere(MWSTFunktion callBackObj)  
    {  
        callBackObjekt = callBackObj;  
    }  
  
    public double berechneBrutto(double netto)  
    {  
        return callBackObjekt.berechneMWSTvoll(netto);  
    }  
  
    public static void main(String[] args)  
    {  
        Preisberechnung brutto = new Preisberechnung();  
        //Objekt erzeugen  
        MWSTFunktion mwstf = new MWSTImpl();  
        //Verweis auf das Objekt  
        //mit der gewünschten Funktion setzen  
        brutto.registriere(mwstf);  
        //Aufruf der gewünschten Funktion  
        //des registrierten Objekts  
        System.out.println(brutto.berechneBrutto(200.0));  
    }  
}
```

Problem/Kontext

Der normale Parametermechanismus in Java versorgt die aufgerufene Methode mit Informationen. Die aufgerufene Methode kann jedoch nur einfache Typen oder ein Objekt zurückgeben. Außerdem kann die aufgerufene Methode keine weiteren Informationen an die aufrufende Methode zurückgeben, wenn der Aufruf einmal beendet ist.

Lösung

Der *Callback*-Mechanismus erlaubt einen zweiseitigen Informationsfluss zwischen Aufrufer und Aufgerufenem. Die größte Flexibilität wird erreicht, wenn ein Objekt, das eine Schnittstelle für die *Callback*-Methoden implementiert, über die Parameterliste übergeben wird.

8.1 Exkurs: **Callback** I

```
public interface CallBack  
{  
    void methodToCallBack();  
}  
  
public class CallBackImpl implements CallBack  
{  
    public void methodToCallBack()  
    {  
        System.out.println("Ich wurde zurückgerufen!");  
    }  
}  
  
public class Caller  
{  
    public void register(CallBack callback)  
    {  
        callback.methodToCallBack();  
    }  
  
    public static void main(String[] args)  
    {  
        Caller caller = new Caller();  
        CallBack callBack = new CallBackImpl();  
        caller.register(callBack);  
    }  
}
```

Beispiel

Java

Varianten

Eine Rückruffunktion bekommt oft gar keinen Namen, sondern wird als anonyme Funktion direkt beim Aufruf definiert.

```
public class DemoCallback  
{  
    public interface Funktion  
    {  
        double berechne(double wert1, double wert2);  
    }  
  
    public static void main(String[] args)  
    {  
        Funktion addiere = new Funktion()  
        {  
            //Anonyme innere Klasse  
            public double berechne(double a, double b)  
            {  
                return a + b;  
            }  
        };  
        Funktion multiplizierte = new Funktion()  
        {  
            //Anonyme innere Klasse  
            public double berechne(double a, double b)  
            {  
                return a * b;  
            }  
        };  
    }  
}
```

Beispiel

Java

I 8 Architektur- und Entwurfsmuster

```
    };
    System.out.println(addiere.berechne(1.5,2.5));
    System.out.println(multipliziere.berechne(1.5,2.5));
}
}
```

Vor- und Nachteile

- + Der *Callback*-Mechanismus ermöglicht eine lose Kopplung zwischen einzelnen Komponenten.
- + Funktionen können allgemein definiert und erst beim Aufrufen der Funktion kann durch Angabe der Rückruffunktionen das Verhalten exakt bestimmt werden.
- Die Programme sind schwerer zu verstehen.

Zusammenhänge

Der *Callback*-Mechanismus unterstützt das Prinzip der Bindung und Kopplung (siehe »Architekturprinzipien«, S. 29).

8.2 Das Schichten-Muster (*layers pattern*)

Name(n)

Das Schichten-Muster (*layers pattern*) ist ein häufig eingesetztes Architekturmuster. In der Literatur wird die Schichtenarchitektur bisweilen auch als ein Architekturstil bezeichnet.

Grundidee

Subsysteme werden verschiedenen horizontalen Schichten zugeordnet. Schichten sind meist dadurch gekennzeichnet, dass Komponenten innerhalb eines Subsystems *beliebig* aufeinander zugreifen können. Zwischen den Schichten gelten dann strengere Zugriffsregeln. Jede Schicht stellt den darüber liegenden Schichten eine bestimmte Menge von Diensten (*services*) zur Verfügung. Die interne Struktur einer Schicht ist nach außen in der Regel *nicht* sichtbar.

Anwendungsbereich(e)

Das Schichten-Muster wird eingesetzt, um große Systeme zu strukturieren. Dabei sollen folgende Anforderungen erfüllt werden:

- Späte Änderungen der Anforderungen und des Quellcodes sollen lokal bleiben.
- Schnittstellen sollen stabil bleiben. Wünschenswert ist die Verwendung von Standardschnittstellen.
- Teile des Systems sollen austauschbar sein. Komponenten sollten durch alternative Implementierungen ersetzt werden können, ohne den Rest des Systems zu beeinträchtigen.

8.2 Das Schichten-Muster (*layers pattern*) I

- Die unteren Abstraktionsebenen sollen in späteren Systemen wieder verwendet werden können.
- Die Systemstrukturierung soll es auch ermöglichen, dass jedes Subsystem getrennt von den anderen Subsystemen entwickelt werden kann.

Viele Middleware-Plattformen basieren auf Schichtenarchitekturen, zum Beispiel Java-EE (siehe »Die Java EE-Plattform«, S. 321) und Microsoft .NET (siehe »Die .NET-Plattform«, S. 360).

Beispiel(e)

Bei dialogorientierten Anwendungen oder Informationssystemen, die eine dauerhafte Datenhaltung erfordern, werden im Regelfall mindestens die drei Schichten »Benutzungssoberfläche« (oft *User Interface* – kurz UI genannt), »Fachdomäne« (auch »Anwendungslogik« oder »Applikationslogik« oder *Business Architecture*) und »Persistenz« unterschieden. Es wird dann von einer **Drei-Schichten-Architektur** gesprochen. Gibt es in der Anwendung unterschiedliche Geschäftsprozesse, dann wird die Fachdomäne oft in zwei Schichten aufgeteilt: eine Steuerungsschicht (auch *Workflow*-Schicht genannt) und eine Anwendungsschicht. Es liegt dann eine Vier-Schichten-Architektur vor.

Das ISO/OSI-7-Schichtenmodell und die TCP/IP-4-Schichtenarchitektur besitzen eine Schichten-Architektur mit linearer Ordnung. Beispiel 2

Betriebssystemarchitekturen lassen sich in die Schichten Hardware-Abstraktionsschicht (Treiber), die eigentliche Betriebssystemschicht (z. B. Kernel und Dateisystem) und die grafische Benutzungsoberfläche gliedern. Beispiel 3

Problem/Kontext

Ein System stellt Funktionen auf unterschiedlichen Abstraktionsebenen zur Verfügung. Funktionen auf höheren Abstraktionsebenen benutzen Funktionen auf niedrigeren Abstraktionsebenen. Der Kommunikationsfluss verläuft typischerweise von einer höheren Abstraktionsebene zu einer niedrigeren Abstraktionsebene. Die Antworten auf Anfragen oder die Meldung von Ereignissen laufen in der umgekehrten Richtung. Solche Systeme benötigen oft eine horizontale Strukturierung die orthogonal zu ihrer vertikalen Unterteilung ist. Dies ist dann der Fall, wenn es mehrere Funktionen auf demselben Abstraktionsniveau gibt, die aber weitgehend unabhängig voneinander sind.

I 8 Architektur- und Entwurfsmuster

Lösung

Das zu entwickelnde System wird in Schichten gegliedert. Die unterste Schicht enthält die Funktionen auf dem niedrigsten Abstraktionsniveau und erhält die Bezeichnung Schicht 1. Die Schicht 2 benutzt die Funktionen der Schicht 1. Die Schicht 3 wiederum benutzt die Schicht 2 usw. Komponenten einer höheren Schicht können zur Erledigung ihrer Aufgabe also auf Komponenten der gleichen Schicht und der unmittelbar darunter liegenden Schicht zurückgreifen. Innerhalb einer Schicht sollen sich alle Komponenten auf demselben Abstraktionsniveau befinden.

Anforderungen

Eine Schichtenarchitektur ist dann sinnvoll, wenn

- die Dienstleistungen einer Schicht sich auf demselben Abstraktionsniveau befinden und
- die Schichten entsprechend ihrem Abstraktionsniveau geordnet sind, sodass eine Schicht nur die Dienstleistungen der tieferen Schichten benötigt.

Beispiel In einer Infrastrukturschicht werden technische oder sogar physikalische Aspekte von Systemen angeordnet.

Es muss ein natürliches Abstraktionskriterium geben, nach dem die Subsysteme geordnet werden können, eine »Benutzt«-Beziehung allein reicht nicht aus. Außerdem muss eine geeignete »Granularität« für die Schichten gefunden werden. Zu wenige Schichten erschweren die Wiederverwendbarkeit, die Anpassbarkeit und Portabilität. Zu viele Schichten erhöhen die Komplexität und den Aufwand für die Trennung der Schichten.

Struktur Für die Schichtenanordnung gibt es u. E. folgende drei verschiedene Varianten:

- Schichten mit linearer Ordnung,
- Schichten mit strikter Ordnung (*strict layering, relaxed layered system*),
- Schichten mit baumartiger Ordnung.

Beispiel **Schichten-Muster mit strikter Ordnung:**

Die Schichten werden entsprechend ihrem Abstraktionsniveau angeordnet. Von Schichten mit höherem Abstraktionsniveau kann auf *alle* Schichten mit niedrigerem Abstraktionsniveau zugegriffen werden, aber nicht umgekehrt (Abb. 8.2-1, Benutztpfeil **c**). Der Vorteil der Flexibilität und Performanz bei diesem Modell wird mit einem Verlust an Wartbarkeit bezahlt.

Schichten-Muster mit linearer Ordnung:

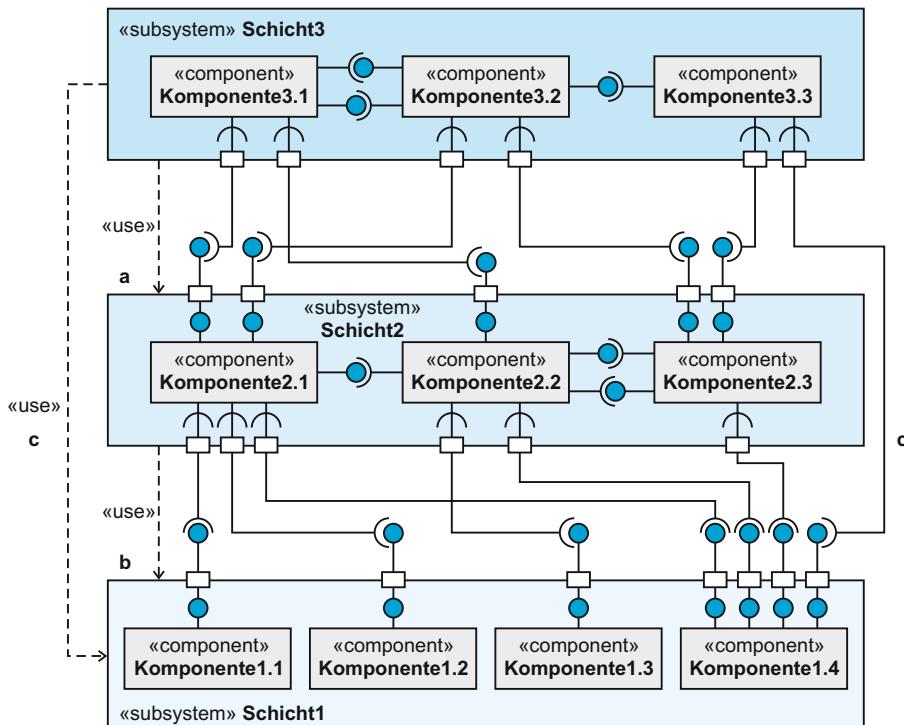
Von einer Schicht kann immer nur auf die nächst niedrigere zugegriffen werden (Abb. 8.2-1, Benutztpfeile **a** und **b**). Der Benutztpfeil **c** wäre dann *nicht* erlaubt. Ein solches Schichtenmodell schirmt alle tieferen Schichten so ab, dass ein direkter Zugriff durch höhere Schichten nicht möglich ist.

8.2 Das Schichten-Muster (*layers pattern*) I

Sollen Schichten bewusst gebildet werden, dann müssen Subsysteme explizit Schichten zugeordnet werden. Das Schichtenmodell selbst muss explizit angegeben werden oder implizit vorgegeben sein.

Eine Schichtenarchitektur kann in der UML durch Komponenten (siehe Abb. 8.2-1) oder Pakete modelliert werden. Zur Notation

(siehe Abb. 8.2-1) oder Pakete modelliert werden.



Paketdiagramme werden als logisches Strukturierungsmittel eingesetzt. Die Paketdarstellung bietet jedoch keinerlei Schnittstellenbeschreibung. Demgegenüber erlaubt es die **Komponentendarstellung**, Schnittstellen zu definieren. Aufgrund der besseren Schnittstellenbeschreibung ist die hierarchische Darstellung mithilfe von Komponenten vorteilhaft. Jede Schicht wird dabei als Komponente interpretiert.

Folgende Schritte helfen beim Entwurf einer Schichtenarchitektur (in Anlehnung an [BMR+96, S. 38 f.]):

1 Festlegung eines Abstraktionskriteriums für die Gruppierung von Aufgaben in Schichten.

2 Festlegung der Anzahl der Schichten unter Beachtung des Abstraktionskriteriums. Zu viele Schichten führen zu einem *Overhead*, zu wenige Schichten zu einer schlechten Struktur.

Abb. 8.2-1: Drei-Schichten-Architektur dargestellt als UML-Komponentendiagramm.

Hinweise zur Implementierung

I 8 Architektur- und Entwurfsmuster

3 Jeder Schicht einen Namen geben und Aufgaben zuordnen.

Die Aufgabe auf der höchsten Schicht ist die Aufgabe, die das System zur Verfügung stellt. Die Aufgaben aller anderen Schichten helfen den höheren Schichten. Wird Bottom-Up vorgegangen, dann stellen die tiefen Schichten die Infrastruktur für die höheren Schichten zur Verfügung.

4 Festlegung der Dienstleistungen. Ziel ist es, die Schichten strikt voneinander zu trennen. Keine Komponente ist über mehr als eine Schicht verteilt. Oft ist es besser, mehr Dienstleistungen in höheren Schichten anzugeben als in tieferen Schichten. Die Basisschichten sollten schlank sein, während die höheren Schichten ein breiteres Anwendungsspektrum anbieten können. Dieses Phänomen wird auch bezeichnet als die umgekehrte Pyramide der Wiederverwendung (*inverted pyramid of reuse*).

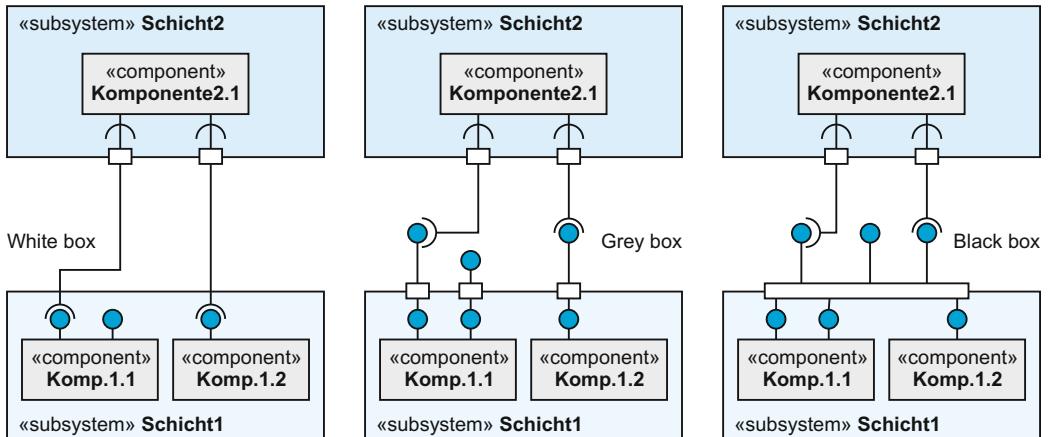
5 Überarbeitung der Schichtenbildung. In der Regel müssen die Schritte 1 bis 4 mehrmals durchlaufen werden, da es nicht möglich ist, das Abstraktionskriterium präzise zu definieren, bevor die betroffenen Schichten und ihre Dienstleistungen näher betrachtet worden sind.

6 Festlegen einer Schnittstelle für jede Schicht. Soll eine Schicht i eine *Black box* für die Schicht $i+1$ sein, dann ist eine schmale Schnittstelle zu entwerfen, die alle Dienstleistungen der Schicht i anbietet. Es ist zu prüfen, ob die Schnittstelle in ein Fassaden-Objekt eingekapselt werden sollte (siehe »Das Fassaden-Muster (*facade pattern*)«, S. 69). Ein *White box*-Ansatz liegt vor, wenn die Schicht $i+1$ alle Interna der Schicht i sieht. Ein *Grey box*-Ansatz liegt vor, wenn die Schicht $i+1$ sieht, aus welchen Komponenten die Schicht i besteht und diese Komponenten direkt anspricht, aber Interna der individuellen Komponenten nicht sieht (Abb. 8.2-2).

Anzustreben ist immer ein *Black box*-Ansatz, da er die Weiterentwickelbarkeit besser unterstützt als die anderen Ansätze. Ausnahmen können aus Gründen der Effizienz gemacht werden oder wenn auf das Innere einer anderen Schicht unbedingt zugegriffen werden muss. Der letzte Fall tritt selten auf und kann mithilfe des Reflexions-Musters abgemildert werden. Das Reflexions-Muster ermöglicht es, die Struktur und das Verhalten eines Software-systems dynamisch zu ändern [BMR+96, S. 193 ff.].

7 Festlegung der Kommunikation zwischen benachbarten Schichten. Am meisten wird das Push-Modell für die Kommunikation zwischen Schichten verwendet (siehe »Das Beobachter-Muster (*observer pattern*)«, S. 54). Wenn die Schicht i eine Dienstleistung der Schicht $i-1$ aufruft, dann wird jede benötigte Information als Teil des Aufrufs übergeben. Bei dem Pull-Modell holt sich eine tiefere Schicht auf eigene Veranlassung verfügbare In-

8.2 Das Schichten-Muster (*layers pattern*) I



formationen von einer höheren Schicht. Solche Modelle führen jedoch zu zusätzlichen Abhängigkeiten zwischen einer Schicht und seiner angrenzenden höheren Schicht. Wenn Abhängigkeiten von tieferen Schichten zu höheren Schichten, die durch das Pull-Modell entstehen, vermieden werden sollen, dann sollten *Callbacks* (siehe »Exkurs: *Callback*«, S. 42) verwendet werden.

8 Entkoppeln benachbarter Schichten. Oft kennt die obere Schicht die nächst niedrigere Schicht, aber die niedrigere Schicht kennt nicht die sie benutzenden Schichten. Daraus ergibt sich eine einseitige Kopplung. Änderungen in der Schicht i können ohne Rücksicht auf die Schicht $i+1$ vorgenommen werden. Voraussetzung ist, dass die Schnittstelle und die Semantik der Schicht unverändert bleiben. Eine solche einseitige Kopplung ist ideal, wenn die Anforderungen von oben nach unten verlaufen und die Rückgabewerte ausreichend für den Transport der Ergebnisse in der umgekehrten Richtung sind.

Läuft die Kommunikation von unten nach oben, dann können *Callbacks* verwendet werden und trotzdem kann die einseitige Kopplung von oben nach unten beibehalten werden. Die oberen Schichten registrieren *Callback*-Funktionen bei den tieferen Schichten. Dies ist effektiv, wenn nur eine feste Anzahl von möglichen Ereignissen von unten nach oben gesendet werden. Beim Start teilen die höheren Schichten den tieferen Schichten mit, welche Funktionen bei festgelegten Ereignissen aufzurufen sind. Die tiefere Schicht verwaltet die Abbildung von Ereignissen zu *Callback*-Funktionen. Das Beobachter-Muster ermöglicht eine Realisierung dieses Konzepts (siehe »Das Beobachter-Muster (*observer pattern*)«, S. 54). Das Kommando-Muster zeigt, wie Funktionen für eine Parameterübergabe verkapselt werden können (siehe »Das Kommando-Muster (*command pattern*)«, S. 75).

Abb. 8.2-2: White-box-, Grey-box- und Black-box-Schichten.

I 8 Architektur- und Entwurfsmuster

9 Entwerfen einer Fehlerbehandlungs-Strategie. Ein Fehler kann entweder in der Schicht behandelt werden, in der er auftritt, oder er kann an die nächst höhere Schicht weitergegeben werden. Wird der Fehler an die nächst höhere Schicht weitergegeben, dann muss die tiefere Schicht den Fehler so beschreiben, dass er für die höhere Schicht bedeutungsvoll ist. In der Regel sollten Fehler auf der tiefsten möglichen Schicht behandelt werden. Dadurch wird vermieden, dass die höheren Schichten mit einer Vielzahl verschiedener Fehler überschwemmt werden. Ähnliche Fehlertypen sollten auf jeden Fall in allgemeinere Fehlertypen transformiert werden und nur diese Fehlertypen sollten weitergegeben werden.

Varianten

Die Abb. 8.2-3 zeigt eine Schichtenarchitektur mit baumartiger Ordnung. Zwischen Schichten gleicher Knotenebene findet keine Kommunikation statt.

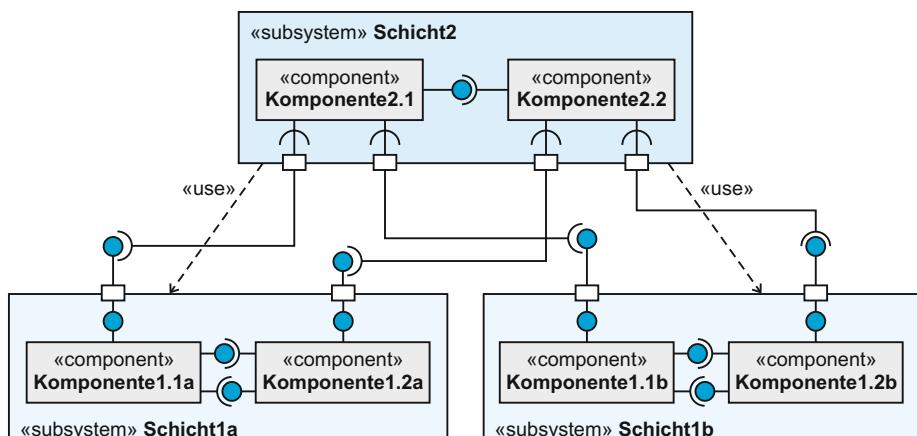


Abb. 8.2-3: Beispiel für eine baumartige Struktur.

Eine Variante der Schichtenarchitektur besteht darin, dass eine Schicht partiell undurchsichtig ist.

Das bedeutet, dass einige ihrer Komponenten nur in der angrenzenden äußeren Schicht sichtbar sind, während die anderen für alle oberen Schichten sichtbar sind.

Oft gibt es parallel zu den horizontalen Schichten eine vertikale Schicht, die Dienstleistungen für alle horizontalen Schichten zur Verfügung stellt (Abb. 8.2-4).

Vor- und Nachteile

Vorteile

Eine Schichtenarchitektur besitzt folgende Vorteile:

- + Übersichtliche Strukturierung in Abstraktionsebenen oder virtuelle Maschinen.

8.2 Das Schichten-Muster (*layers pattern*) I

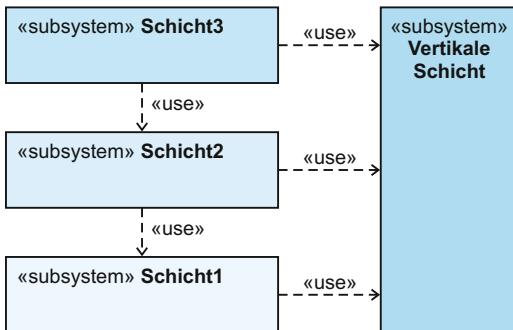


Abb. 8.2-4: Schichtenarchitektur mit vertikaler Dienstleistungsschicht.

- + Keine zu starke Einschränkung des Entwerfers, da er neben einer strengen Hierarchie noch eine liberale Strukturierungsmöglichkeit innerhalb der Schichten besitzt.
- + Schichten sind voneinander unabhängig, sowohl beim Entwurf als auch beim Betrieb des Systems.
- + Eine Schicht kann gegen eine andere Schicht ausgetauscht werden, wenn sie die gleichen Dienste anbietet.
- + Durch die Schichtenbildung werden die Abhängigkeiten zwischen Subsystemen minimiert.
- + Muss ein System portiert werden, dann genügt es in der Regel die unterste Schicht zu portieren.
- + Klar definierte und allgemein akzeptierte Abstraktionsebenen erleichtern die Entwicklung von Schnittstellenstandards.
- + Durch die Einführung von Überwachungsschichten kann die Betriebssicherheit (*safety*) und der Schutz vor Einbrüchen ins System (*security*) verbessert werden (siehe »Authentifizierung und Autorisierung«, S. 153).
- + Es werden die Wiederverwendbarkeit, die Änderbarkeit, die Wartbarkeit, die Portabilität und die Testbarkeit unterstützt.

Dem stehen folgende Nachteile gegenüber:

Nachteile

- Effizienzverlust, da alle Daten über verschiedene Schichten transportiert werden müssen (bei linearer Ordnung). Dies gilt auch für Fehlermeldungen. Durch das Überspringen von Zwischenschichten kann die Effizienz verbessert werden (*Layer-Bridging*). Dadurch werden jedoch zusätzliche Abhängigkeiten erzeugt.
- Bestimmte Änderungen müssen in allen Schichten vorgenommen werden. Wird beispielsweise ein neues Datenfeld hinzugefügt, das auf der Benutzungsoberfläche angezeigt und gespeichert werden muss, dann müssen Änderungen an allen Schichten vorgenommen werden.
- Eindeutig voneinander abgrenzbare Abstraktionsschichten lassen sich *nicht* immer definieren.
- Innerhalb einer Schicht kann Chaos herrschen.

I 8 Architektur- und Entwurfsmuster

Hinweis

In der englischen Literatur wird für den Begriff Schicht entweder der Begriff *Layer* oder der Begriff *Tier* verwendet. Mit dem Begriff *Tier* ist oft die Zuordnung einer Schicht zu einem Computersystem verbunden.

Zusammenhänge

Das Schichten-Muster unterstützt die nicht-funktionalen Anforderungen (siehe »Nichtfunktionale Anforderungen«, S. 109) Wartbarkeit (Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit), Weiterentwickelbarkeit, Portabilität und Wiederverwendbarkeit.

Es beeinträchtigt die Effizienz (Zeitverhalten, Verbrauchsverhalten).

Das Schichten-Muster unterstützt die physikalische Verteilung eines Softwaresystems auf verschiedene Computer (Verteilbarkeit).

Literatur

[BMR+96, S. 31–51]

8.3 Das Beobachter-Muster (*observer pattern*)

Name(n)

Das Beobachter-Muster (*observer pattern*) ist ein objektbasiertes Verhaltensmuster (*behavioral pattern*). Es wird auch Verleger-Abonnenten-Muster, *publisher subscriber pattern* oder *listeners pattern* genannt.

Grundidee

Das Beobachter-Muster sorgt dafür, dass bei der Änderung eines Objekts alle davon abhängigen Objekte benachrichtigt werden, sodass sie sich aktualisieren können. Die abhängigen Objekte müssen sich dazu bei dem gewünschten Objekt registrieren. Dadurch wird eine lose Kopplung zwischen den Objekten erreicht.

Anwendungsbereich(e)

Das Beobachter-Muster wird zur Implementierung von Benutzeroberflächen und in GUI-Frameworks intensiv eingesetzt. Es ist eines der Muster, das in Java am häufigsten verwendet wird, z. B. in Swing und JavaBeans.

Der Einsatz des Beobachter-Musters ist in folgenden Fällen sinnvoll:

- Die Änderung eines Objekts verlangt die Änderung anderer Objekte und es ist nicht bekannt, wie viele Objekte geändert werden müssen.

8.3 Das Beobachter-Muster (*observer pattern*) I

- Objekte sollen automatisch über Zustandsänderungen anderer Objekte informiert werden, ohne selbst aktiv werden zu müssen. Die Objekte sollen dann selbst entscheiden, wie sie auf diese Zustandsänderungen reagieren.
- Wenn ein Objekt andere Objekte benachrichtigen soll, ohne Annahmen darüber treffen zu dürfen, wer bzw. genauer von welchem Typ diese anderen Objekte sind.

Beispiel(e)

Ein Verkehrsverwaltungssystem erhält von Sensoren kontinuierlich Informationen darüber, ob auf einer Autobahn ein Stau herrscht oder nicht. Autofahrer, die eine Autobahn benutzen wollen, sollen sich über ihr Handy Informationen über einen Stau auf der zu befahrenen Autobahn informieren können. Eine Lösungsmöglichkeit besteht darin, dass jedes Handy, eine öffentlich bekannte Methode in regelmäßigen oder unregelmäßigen Abständen aufruft, um sich über den Zustand der entsprechenden Autobahn zu informieren. Als Parameter wird in der Methode die Autobahnnummer angegeben. Der Nachteil dieses Verfahrens besteht darin, dass unter Umständen viele unnötige Aufrufe erfolgen, da sich in einem bestimmten Zeitabschnitt keine neuen Informationen ergeben haben.

Beispiel 1a
Staumelder

Problem/Kontext

Das oder die Objekte, die andere Objekte über ihre Zustandsänderung informieren sollen, sollen sich selbst nicht um die zu informierenden Objekte kümmern müssen.

Lösung

Die Objekte, die andere Objekte über ihre Zustandsänderung informieren sollen, stellen allgemein bekannte Methoden zum Registrieren und zum De-Registrieren zur Verfügung. Objekte, die informiert werden wollen – Beobachter (*observer*) genannt –, müssen sich mit der Registrieren-Methode anmelden. Allgemein bedeutet dies, dass es eine Schnittstelle geben muss, die diese Methoden festlegt. Klassen, die diese Methoden zur Verfügung stellen wollen, müssen die Schnittstelle implementieren. Damit Objekte, bei denen sich andere Objekte registriert haben, diese Objekte über Zustandsänderungen informieren können, müssen die zu informierenden Objekte ebenfalls eine allgemein bekannte Methode implementieren. Diese Methode ruft dann das Objekt, das eine Zustandsänderung erfährt, bei allen registrierten Objekten auf. Es gibt verschiedene Varianten des Beobachter-Musters. Zunächst wird gezeigt, wie in Java die Realisierung des Beobachter-Musters erfolgen kann.

I 8 Architektur- und Entwurfsmuster

Java In Java gibt es die Klasse `java.util.Observable`, die folgende Methoden bereitstellt:

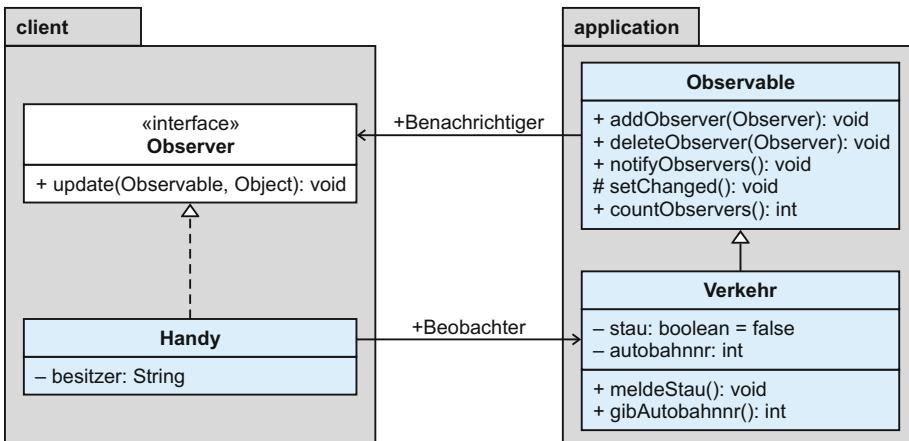
- Observable**
- `void addObserver(Observer o)`: Registriert einen Beobachter bei dem Objekt.
 - `void deleteObserver(Observer o)`: Löscht die Registrierung des Beobachters.
 - `boolean hasChanged()`: Prüft, ob sich das Objekt geändert hat.
 - `protected void clearChanged()`: Gibt an, dass sich dieses Objekt nicht verändert hat oder dass alle Beobachter über die letzte Änderung bereits informiert worden sind, sodass die Methode `hasChanged()` das Ergebnis `false` liefert.
 - `void notifyObservers()`: Gibt es in dem Objekt eine Änderung – wie durch die Methode `hasChanged()` angezeigt –, dann werden alle registrierten Beobachter informiert und die Methode `clearChanged()` aufgerufen, um anzudeuten, dass dieses Objekt keine weitere Änderung besitzt. Die Art der Zustandsänderung wird *nicht* mitgeteilt, sodass der Beobachter sich über die Zustandsänderung durch Aufruf von Methoden des beobachteten Objekts selbst informieren muss (sogenanntes *Pull*-Modell).
 - `void notifyObservers(Object arg)`: Analog wie die Methode `void notifyObservers()`, jedoch werden Informationen über das beobachtete Objekt mit über die Parameterliste übergeben. Der Beobachter wird so direkt über die Art der Änderung in dem beobachteten Objekt informiert (sogenanntes *Push*-Modell).
 - `protected void setChanged()`: Gibt an, dass dieses Objekt sich geändert hat. Die Methode `hasChanged()` gibt den Wert `true` zurück. Durch diese Methode ist es möglich, Benachrichtigungen flexibel zu steuern. Ändert sich ein Zustand sehr oft oder nur minimal, dann sollte diese Methode *nicht* aufgerufen werden. Eine Benachrichtigung der Beobachter erfolgt dann nicht.
 - `int countObservers()`: Gibt die Anzahl der registrierten Beobachter zurück.

Observer Die Schnittstelle `public interface Observer` muss von allen Beobachtern implementiert werden, wenn sie von Objekten über ihre Änderungen informiert werden wollen. Folgende Methode ist festgelegt:

- `void update(Observable o, Object arg)`: Diese Methode wird vom beobachteten Objekt aufgerufen, wenn sich sein Zustand ändert. Der Parameter `o` gibt das beobachtete Objekt an, der Parameter `arg` ist das Argument, das durch die Methode `notifyObservers(Object arg)` übergeben wird.

Beispiel 1b Staumelder Mithilfe des Beobachter-Musters lässt sich in Java das Verkehrsinformationssystem problemgerecht implementieren. Die Abb. 8.3-1 zeigt das entsprechende OOD-Diagramm.

8.3 Das Beobachter-Muster (*observer pattern*) I



Die Klasse **Verkehr** erhält von Sensoren Staumeldungen und verwaltet diese. Sie erbt von der Klasse **Observable**. Die Klasse **Handy** implementiert die Schnittstelle **Observer**. Objekte der Klasse **Handy** können sich über die Methode `addObserver()` bei einem Verkehrsobjekt als Beobachter registrieren. Ändert sich der Zustand eines Verkehrsobjekts, dann informiert dieses Verkehrsobjekt alle bei ihm registrierten Beobachter durch Aufruf der Methode `update()`. Alle Objekte der Klasse **Verkehr** müssen sich um ihre Umwelt nicht kümmern, sondern die Objekte, die Informationen haben wollen, müssen sich selbst registrieren. Dadurch ist eine Entkopplung zwischen den Nutzern und der Applikation **Verkehr** gegeben.

Die Java-Programme für dieses Beispiel können wie folgt aussehen:

```

import java.util.Observable;

public class Verkehr extends Observable
{
    private boolean stau = false;
    private int autobahnrr;

    public Verkehr (int autobahnrr)
    {
        this.autobahnrr = autobahnrr;
    }

    public void meldeStau()
    {
        if(Math.random() < 0.5f)
            stau = true;
        else
            stau = false;
        setChanged();
        notifyObservers(stau);
    }
}
  
```

Abb. 8.3-1: UML-Klassendiagramm des Beobachter-Musters für das Beispiel Verkehrsverwaltungssystem.

Java

Klasse **Verkehr**

I 8 Architektur- und Entwurfsmuster

```
public int gibAutobahnNr()
{
    return autobahnNr;
}

Klasse Handy import java.util.Observer;
import java.util.Observable;

public class Handy implements Observer
{
    private String besitzer;
    public Handy(String besitzer)
    {
        this.besitzer = besitzer;
    }

    public void update(Observable o, Object obj)
    {
        Verkehr einVerkehr = (Verkehr) o;
        System.out.println("Handy von " + besitzer +
            " meldet Stau auf der Autobahn " +
            einVerkehr.gibAutobahnNr() + ": " + obj);
    }
}
```

Klasse VerkehrsInfo Das folgende Programm simuliert zwei Handys und zwei Autobahnverwaltungen. Es zeigt die Registrierung als Beobachter. Ebenso werden Staumeldungen simuliert, die zu einer Meldung an die Beobachter führen.

```
import java.util.Observer;
import java.util.Observable;

public class VerkehrsInfo
{
    public static void main (String[] args)
    {
        System.out.println ("--- Start ---");
        //2 Handys erzeugen
        Handy helmutHandy = new Handy("Helmut");
        Handy frankHandy = new Handy("Frank");
        //Überwachungsmelder anlegen
        Verkehr autobahn1 = new Verkehr(1);
        Verkehr autobahn2 = new Verkehr(2);
        //Handys als Beobachter registrieren
        autobahn1.addObserver(helmutHandy);
        autobahn2.addObserver(helmutHandy);
        autobahn1.addObserver(frankHandy);
        //Anzahl der Beobachter abfragen
        System.out.println("Anzahl Beobachter - Autobahn 1: " +
            + autobahn1.countObservers());
        System.out.println("Anzahl Beobachter - Autobahn 2: " +
            + autobahn2.countObservers());
        //Staumeldung abfragen, unabhängig von den Handys
```

8.3 Das Beobachter-Muster (*observer pattern*) I

```
autobahn1.meldeStau();
autobahn2.meldeStau();
//1 Beobachter abmelden
autobahn2.deleteObserver(helmutHandy);
//Anzahl der Beobachter abfragen
System.out.println("Anzahl Beobachter - Autobahn 1: "
+ autobahn1.countObservers());
System.out.println("Anzahl Beobachter - Autobahn 2: "
+ autobahn2.countObservers());
//Staumeldung abfragen, unabhängig von den Handys
autobahn1.meldeStau();
autobahn2.meldeStau();
System.out.println ("--- Ende ---");
}
}
```

Eine Ausführung des Programms ergibt folgende Ausgaben:

```
--- Start ---
Anzahl Beobachter - Autobahn 1: 2
Anzahl Beobachter - Autobahn 2: 1
Handy von Frank meldet Stau auf der Autobahn 1: true
Handy von Helmut meldet Stau auf der Autobahn 1: true
Handy von Helmut meldet Stau auf der Autobahn 2: true
Anzahl Beobachter - Autobahn 1: 2
Anzahl Beobachter - Autobahn 2: 0
Handy von Frank meldet Stau auf der Autobahn 1: true
Handy von Helmut meldet Stau auf der Autobahn 1: true
--- Ende ---
```

Was müssen Sie tun, wenn innerhalb von Sekunden mehrere Staumeldungen gemeldet werden und Sie verhindern wollen, dass die registrierten Beobachter ständig mit Meldungen »überschüttet« werden.

Sie können den Aufruf der Methode `setChanged()` zum Beispiel nur alle 10 min erlauben.

Gehen Sie *nicht* davon aus, dass die Benachrichtigungen an die registrierten Beobachter in einer festgelegten Reihenfolge erfolgen.

Die in Java gewählte Form des Beobachter-Musters hat folgende Vor- und Nachteile:

- + Die Standardfunktionalität wird zur Verfügung gestellt, sodass man diese Funktionalität *nicht* programmieren muss.
- Bei `Observable` handelt es sich um eine Klasse, sodass Unterklassen gebildet werden müssen. Benötigt eine solche Unterklasse aber eine weitere Oberklasse, dann ist dies *nicht* möglich. Dadurch wird der Einsatz dieser Klasse stark eingeschränkt.
- Die Methode `setChanged()` hat die Sichtbarkeit `protected`, d.h. sie kann nur von einer Unterklasse aus aufgerufen werden. Dadurch ist es *nicht* möglich, eine Komposition anstelle der Vererbung zu verwenden (siehe »Schnittstellen, Fabriken und Komposition«, S. 503).

[Frage](#)

[Antwort](#)

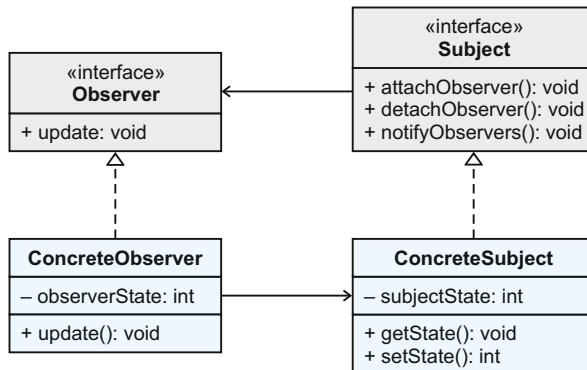


[Java](#)

I 8 Architektur- und Entwurfsmuster

Die Abb. 8.3-2 zeigt das Beobachter-Muster in verallgemeinerter Form.

Abb. 8.3-2: UML-Klassendiagramm des Beobachter-Musters.



- Dynamik
- Das Subjekt
 - kennt seine Beobachter. Eine beliebige Anzahl von Beobachtern kann sich bei einem Subjekt registrieren.
 - stellt eine Schnittstelle zum An- und Abmelden von Beobachtern zur Verfügung.
 - Ein Beobachter
 - stellt eine Aktualisierungsschnittstelle für seine Objekte zur Verfügung, damit sie über Änderungen eines Subjekts benachrichtigt werden können.
 - Ein konkretes Subjekt
 - speichert den für die konkreten Beobachter relevanten Zustand und
 - benachrichtigt alle seine Beobachter, wenn sich sein Zustand ändert.
 - Ein konkreter Beobachter
 - verwaltet Referenzen auf seine konkreten Subjekte,
 - speichert den Zustand, der mit dem des Subjekts in Einklang stehen soll, und
 - implementiert die Methode `update()` der Schnittstelle `Observer`, um seinen Zustand mit dem des Subjekts konsistent zu halten.
- Interaktionen
- Ein beobachtetes Objekt benachrichtigt seine Beobachter, wenn eine Zustandsänderung stattfindet.
 - Nach der Benachrichtigung über eine Änderung kann der Beobachter das beobachtete Objekt nach Informationen befragen. Der Beobachter verwendet diese Informationen, um seinen Zustand mit dem des beobachteten Objekts in Einklang zu bringen.
- Hinweise zur Implementierung
- Push- vs. Pull-Modell
 - Beim Push-Modell schickt das beobachtete Objekt den Beobachtern detaillierte Informationen über die Änderung mit, ob sie nun an ihnen interessiert sind oder nicht.

8.3 Das Beobachter-Muster (*observer pattern*) I

- Beim Pull-Modell sendet das beobachtete Objekt nur minimale Informationen mit. Die Beobachter müssen anschließend nachfragen, um die Details der Änderung in Erfahrung zu bringen.
- Das Push-Modell erschwert die Wiederverwendbarkeit von Beobachtern, da die beobachteten Objekte Annahmen über Beobachterklassen machen, die nicht immer zutreffen müssen. Das Pull-Modell kann dagegen ineffizient sein, da die Beobachter selbst feststellen müssen, was sich geändert hat.
- Registrieren mit Änderungswünschen
- Um die Effizienz von Aktualisierungen zu verbessern, kann die Registrierungsschnittstelle so erweitert werden, dass Beobachter angeben können, an welchen Zustandsänderungen sie interessiert sind. Tritt eine solche Zustandsänderung ein, dann werden nur jene Beobachter benachrichtigt, die sich für die entsprechende Änderung registriert haben.

Varianten

- Es können mehrere unterschiedlich parametrisierte `update()`-Methoden in der Beobachter-Schnittstelle für verschiedene Fälle definiert werden.
- Die `update()`-Methode kann als Parameter eine Referenz auf das beobachtete Objekt erhalten. Der Beobachter muss so keine Referenz auf das beobachtete Objekt halten und kann zudem bei mehreren zu beobachtenden Objekten registriert werden.
- Die Schnittstelle des Subjekts wird häufig reduziert – im Extremfall ist nur eine Anmeldung des Beobachters möglich.
- In Sonderfällen ist es sinnvoll, nur einen Beobachter zuzulassen.
- In Java-Swing gibt es eine Ereignis-Hierarchie. Der Beobachter erhält als Parameter ein Ereignis-Objekt, das Auskunft über die Quelle oder weitere Hinweise enthält.
- Die Ausführung der `update()`-Methode des Beobachters kann asynchron mit *Threading* oder *Message Queuing* erfolgen.
- Um die Anzahl der Benachrichtigungen in Grenzen zu halten, kann ein Update-Manager eingeführt werden [GHJ+96, S. 295 ff.].

In C# und .NET wird *nicht* das Beobachter-Muster, sondern das *Delegate*-Konzept verwendet, das eine andere Semantik für die Ereignisverarbeitung verwendet.

Hinweis

Vor- und Nachteile

Das Beobachter-Muster besitzt folgende Vorteile:

Vorteile

- + Die Subjekt- und Beobachterklassen können zu unterschiedlichen Abstraktionsschichten in einem System gehören. Ein Subjekt in einer tiefen Schicht kann einen Beobachter in einer höheren Schicht benachrichtigen, wobei das Schichtenmodell intakt bleibt (siehe »Das Schichten-Muster (*layers pattern*)«, S. 46).

I 8 Architektur- und Entwurfsmuster

| | |
|-----------|---|
| | <ul style="list-style-type: none">+ Das beobachtete Objekt muss seine Empfänger nicht kennen. Der Beobachter ist dafür zuständig, eine Benachrichtigung zu ignorieren oder zu bearbeiten.+ Die beobachteten Objekte und die Beobachter können unabhängig voneinander modifiziert werden. Beobachtete Objekte können wieder verwendet werden, ohne ihre Beobachter wieder verwenden zu müssen, und umgekehrt.+ Die beobachteten Objekte und die Beobachter sind lose miteinander gekoppelt. Die Beobachter müssen nur die Beobachter-Schnittstelle implementieren. Alles, was ein beobachtetes Objekt weiß, ist, dass es eine Liste von Beobachtern besitzt, von denen jeder die Beobachter-Schnittstelle realisiert.+ Es muss von vornherein <i>nicht</i> bekannt sein, wie viele Beobachter sich zur Laufzeit registrieren und welche das sind. |
| Nachteile | <p>Dem stehen folgende Nachteile gegenüber:</p> <ul style="list-style-type: none">- Änderungen an einem beobachteten Objekt führen bei einer großen Beobachteranzahl zu einem hohen Änderungsaufwand.- Die lose Kopplung führt dazu, dass oft nicht offensichtlich ist, welche Objekte wann benachrichtigt und aktualisiert werden. Es können Endlosschleifen entstehen, wenn ein Beobachter während der Bearbeitung einer gemeldeten Änderung wiederum Änderungsmethoden des beobachteten Objekts aufruft.- Oft werden Beobachter nicht wieder abgemeldet. Das kann zu Mehrfachanmeldung und Mehrfachbenachrichtigung führen.- Die <i>Garbage Collection</i> kann beeinträchtigt werden, falls der Beobachter selbst eine Referenz auf das beobachtete Objekt hat. |
| Literatur | <p>Zusammenhänge</p> <p>Das Beobachter-Muster unterstützt die nichtfunktionale Anforderung »Wartbarkeit« (siehe »Wartbarkeit«, S. 116) und das Prinzip der »Bindung und Kopplung« (siehe »Architekturprinzipien«, S. 29). Es beeinträchtigt die Leistung und Effizienz (siehe »Leistung und Effizienz«, S. 128).</p> <p>[GHJ+96, S. 287 ff.], [EiSt07, S. 50 ff.]</p> |

8.4 Das MVC-Muster (*model view controller pattern*)

Name(n)

Das **MVC-Muster** (*model view controller pattern*) ist ein objektbasiertes Verhaltensmuster (*behavioral pattern*). Es wird sowohl als Entwurfsmuster als auch als Architekturmuster verwendet. Bei dem MVC-Muster handelt es sich um eine Verfeinerung des Beobachter-Musters (siehe »Das Beobachter-Muster (*observer pattern*)«, S. 54).

8.4 Das MVC-Muster (*model view controller pattern*) I

Historisch gesehen war es umgekehrt: Das MVC-Muster wurde zuerst in der Programmiersprache Smalltalk 80 eingesetzt und später zum Beobachter-Muster vereinfacht.

Wenn Sie das Beobachter-Muster noch nicht kennen, dann sollten Sie es zuerst lesen (siehe »Das Beobachter-Muster (*observer pattern*)«, S. 54).

Hinweis

Grundidee

Die Grundidee des MVC-Musters ist die Aufteilung einer Anwendung in folgende Subsysteme oder Komponenten:

- **view**: Subsystem für die Benutzungsoberfläche.
- **model**: Subsystem für die Anwendungslogik, Geschäftslogik bzw. fachliche Logik.
- **controller**: Subsystem für die Steuerung der Anwendung; Abbildung von Ereignissen auf zugehörige Funktionen und Bindeglied zwischen den *view*- und *model*-Subsystemen.

Ein oder mehrere *controller* können den Zustand eines Objekts – *model* genannt – ändern. *views* stellen den Zustand eines Objektes, in der Regel auf der Benutzungsoberfläche, dar und registrieren sich als Beobachter bei dem Objekt. Über Zustandsänderungen werden sie informiert und passen dann ihre Darstellung an.

Anwendungsbereich(e)

Beim Entwurf grafischer Benutzungsoberflächen wird das MVC-Muster in vereinfachter Form als Beobachter-Muster eingesetzt. *view* und *controller* werden dabei zusammengefasst (siehe »Das Beobachter-Muster (*observer pattern*)«, S. 54).

In Web-Architekturen wird das MVC-Muster unverändert eingesetzt. Basiert eine Web-Architektur auf dem MVC-Muster, dann spricht man von einer **Modell-2-Architektur**. Das MVC-Muster ist bei vielen *Web-Frameworks* umgesetzt. Ein sehr bekanntes und verbreitetes Web-Framework, das auf dem MVC-Muster basiert, ist Struts von Apache.

Beispiel(e)

In dem Objekt `:model1` sind die Attribute `a`, `b` und `c` mit ihren Werten gespeichert (Abb. 8.4-1). Über das Objekt `:controller1` können zum Beispiel über eine Tastatur diese Werte geändert werden. Das Objekt `:view1` ist bei dem Objekt `:model1` als Beobachter registriert und wird über die Änderung informiert. Daraufhin aktualisiert das Objekt seine Darstellung, hier eine Tabellenansicht. Das Objekt `:view2` ist ebenfalls als Beobachter bei dem Objekt `:model1` registriert, wird ebenfalls über die Änderung informiert und aktualisiert hier seine

Beispiel 1

I 8 Architektur- und Entwurfsmuster

Kreisdiagrammdarstellung. Die Daten des Objekts `:model1` können auch über das Objekt `:controller2` geändert werden, zum Beispiel durch eine Eingabe über ein Handy.

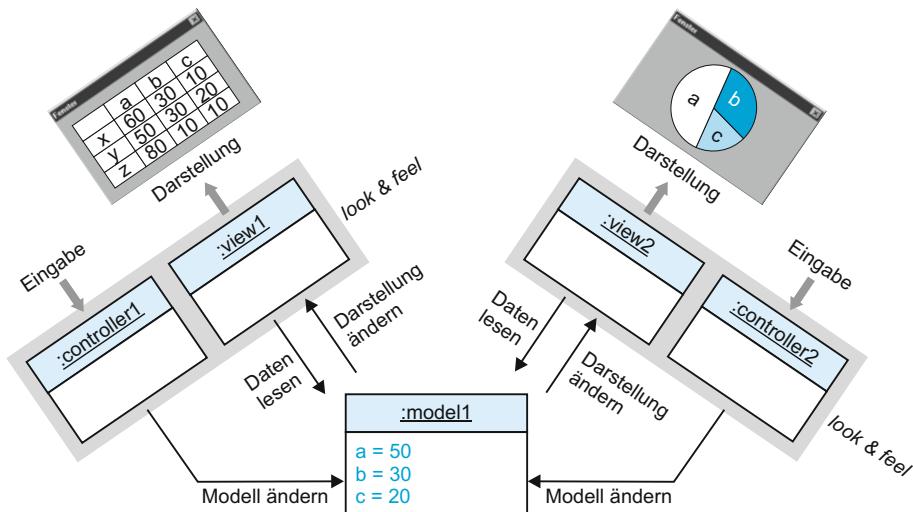


Abb. 8.4-1: Das Prinzip des MVC-Musters.

Problem/Kontext

Die fachlichen Daten einer Anwendung sollen durch verschiedene Eingabemedien unabhängig voneinander änderbar sein. Außerdem sollen die fachlichen Daten gegenüber dem Benutzer oder gegenüber anderen Systemen verschieden dargestellt werden können. Dabei sollen die Eingabemedien und die Ausgabemedien bzw. die verschiedenen Ausgabedarstellungen unabhängig voneinander sein.

Lösung

Das zu entwickelnde System wird in drei Subsysteme oder Komponenten gegliedert. Jedes Subsystem ist für eine Aufgabe verantwortlich (*separation of concerns*) (siehe »Architekturprinzip: Trennung von Zuständigkeiten«, S. 31). Das Subsystem *model* verwaltet die Anwendung, das Subsystem *view* ist für die Darstellung der Anwendungsdaten zuständig und das Subsystem *controller* steuert die Eingaben.

Struktur Die prinzipielle statische Struktur zeigt das UML-Diagramm der Abb. 8.4-2.

Bei der Initialisierung registrieren sich alle *views* bei dem *model*. Jede *view* erzeugt einen passenden *controller*. Es gibt eine 1:1-Beziehung zwischen *views* und *controller*. *views* bieten oft eine Funktionalität, die es dem *controller* erlauben, die Darstellung direkt zu ändern. Dies ist bei benutzergesteuerten Operationen nützlich, die nicht das *model* betreffen, z. B. das Scrollen.

8.4 Das MVC-Muster (*model view controller pattern*) I

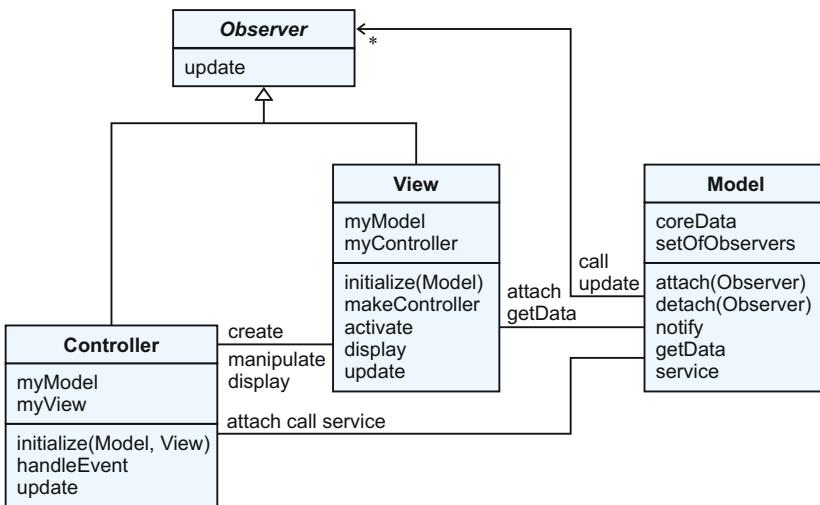


Abb. 8.4-2:
Klassenstruktur
des MVC-Musters
(in Anlehnung an
[BRM+96, S.129]).

Hängt das Verhalten des *controller* vom Zustand des *model* ab, dann registriert sich auch der *controller* beim *model* und implementiert eine Update-Funktion. Dies ist z.B. nötig, wenn eine Änderung des *model*-Zustands einen Menüeintrag aktiviert oder deaktiviert.

Den prinzipiellen dynamischen Ablauf bei der Verarbeitung einer Eingabe veranschaulicht das UML-Diagramm der Abb. 8.4-3.

Dynamik

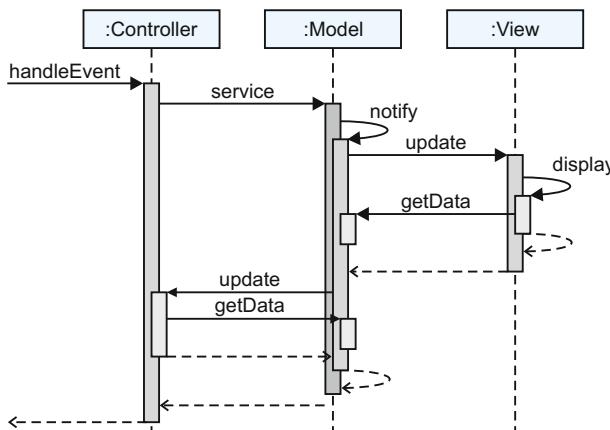


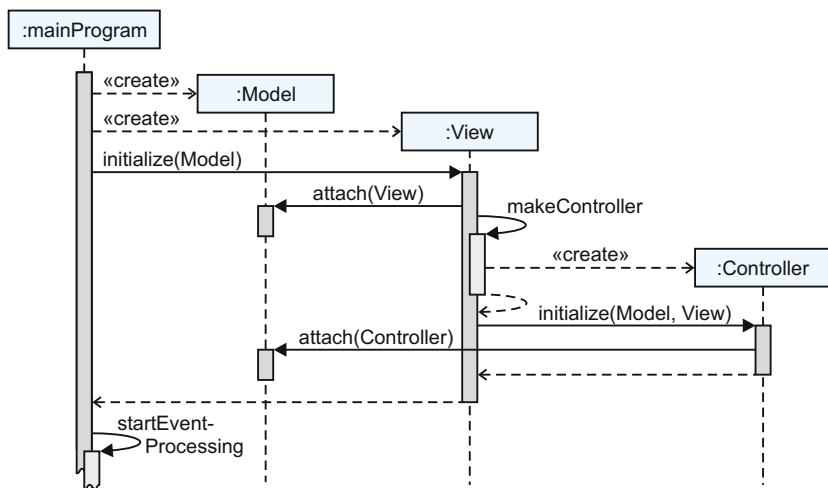
Abb. 8.4-3:
Dynamischer
Ablauf des MVC-
Musters bei einer
Eingabe [BRM+96,
S.130].

Den dynamischen Ablauf bei der Initialisierung des MVC-Musters zeigt die Abb. 8.4-4.

- Ein *model*-Objekt wird erzeugt und initialisiert seine internen Datenstrukturen.
- Ein *view*-Objekt wird erzeugt und erhält als Parameter bei der Initialisierung eine Referenz auf das *model*-Objekt.
- Das *view*-Objekt registriert sich bei dem *model*-Objekt.

I 8 Architektur- und Entwurfsmuster

Abb. 8.4-4:
Dynamischer
Ablauf des MVC-
Musters bei einer
Initialisierung
[BRM+96, S.131].



- Das *view*-Objekt erzeugt sein *controller*-Objekt und übergibt Referenzen auf das *model*-Objekt und sich selbst.
- Das *controller*-Objekt registriert sich beim *model*-Objekt.

Beispiel Das MVC-Muster – übertragen auf Web-Architekturen – zeigt die Abb. 8.4-5. Mit der Aufteilung der Anwendung in Subsysteme ist auch ein Kontrollfluss zur Abarbeitung einer Anfrage assoziiert.

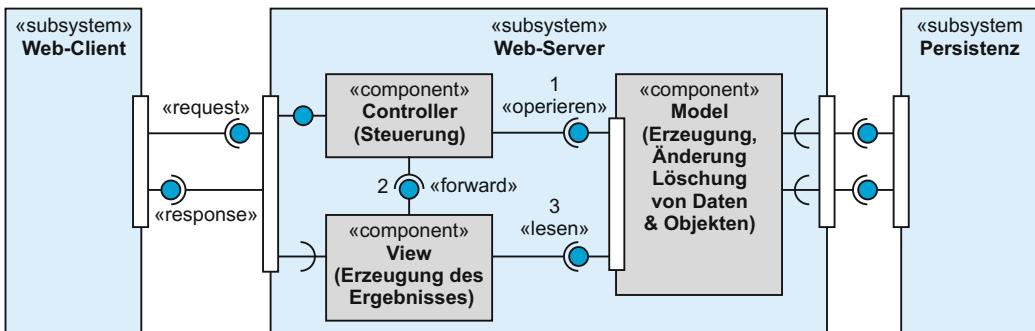


Abb. 8.4-5: Web-Architektur nach dem MVC-Muster.

Eine Anfrage wird stets zur Controller-Komponente geleitet. Der *controller* steuert die Bearbeitung der Anfrage, in dem er die angefragte URL und die Request-Parameter analysiert, dadurch die Ereignisse bestimmt und daraus die Funktionen, die auf der *model*-Komponente auszuführen sind, ableitet. Das Ausführen einer Funktion, d.h. das Operieren auf der *model*-Komponente, ist der erste Hauptschritt, den der *controller* durchführt.

Nachdem die Operationen auf dem *model* ausgeführt wurden, d.h. die mit der Anfrage assoziierte Funktion der Anwendung durchgeführt wurde, delegiert der *controller* im zweiten Schritt die Erzeugung eines Ergebnisdokuments an die *view*-Komponente. Dieser

8.4 Das MVC-Muster (*model view controller pattern*) I

zweite Schritt erfolgt in Abhängigkeit der Ergebnisse, die im Schritt 1 erzielt werden. Der *controller* leitet die weitere Bearbeitung der Anfrage abhängig von den Operationsergebnissen an verschiedene *view*-Teilsysteme weiter.

Die *view* hat die Aufgabe, ein **Ergebnisdokument** zu erzeugen und an die *Response*-Nachricht zu übergeben. Für die Erzeugung eines Ergebnisdokuments nutzt die *view* das *model*, indem dort Daten ausgelesen und ins Ergebnisdokument eingebaut werden. Das Ergebnisdokument wird meist spezifisch für den Client erzeugt, d.h. sowohl die Art der Client-Anwendung, z.B. welcher Webbrowser benutzt wird, als auch die Rolle des Benutzers, z.B. welche Informationen der Benutzer in Abhängigkeit seiner Rolle angezeigt bekommt, kann bei der Ergebniserzeugung berücksichtigt werden. Nachdem die *view* ihre Arbeit erledigt hat, ist die Bearbeitung einer Anfrage abgeschlossen. Insbesondere kehrt die Kontrolle *nicht* mehr an den *controller* zurück. Bei Web-Anwendungen ist *nicht* der Webbrowser die *view*-Komponente, wie dies bei klassischer Betrachtung des MVC-Musters der Fall ist. Die *view*-Komponente einer Web-Anwendung stellt nämlich *nicht* die Daten dar, wie im klassischen Fall, sondern erzeugt nur die Daten, die in einem anderen System – im Web-Umfeld im Webbrowser – angezeigt werden (siehe auch »Web-Architektur für das Subsystem Applikation«, S. 414).

Wenn abzusehen ist, dass sich die Funktionalität des *model* ändert oder wenn die *controller* wiederverwendbar sein sollen, sodass sie unabhängig von einer speziellen Schnittstelle sind, dann ist das Kommando-Muster anzuwenden, siehe »Das Kommando-Muster (*command pattern*)«, S. 75.

Wenn eine Klassenhierarchie von *views* und *controller* aufgebaut wird, dann ist das Fabrikmethoden-Muster anzuwenden, siehe »Fabrikmethoden-Muster (*factory method pattern*)«, S. 89. Es sollte eine Methode `makeController()` in den *view*-Klassen definiert werden. Jede *view*, die einen *controller* benötigt und sich von ihrer Oberklasse unterscheidet, redefiniert die Fabrikmethode.

Hinweise zur
Implemen-
tiering

Varianten

Der *controller* kann so implementiert werden, dass sein Verhalten austauschbar ist, siehe »Das Strategie-Muster (*strategy pattern*)«, S. 96.

Ist eine Benutzungsoberfläche sehr komplex, dann kann das MVC-Muster verfeinert werden. Die einzelnen *views* setzen sich meist aus kleineren fachlichen Sub-Komponenten zusammen, die eigene Verantwortlichkeiten in Bezug auf Darstellung, Ablauf und Fachlogik haben. Beispiele hierfür sind ein Hauptfenster der Anwendung mit Statuszeile und Menüs. Auch fachliche Fenster können aus wiederverwendbaren Komponenten für die Bearbeitung fachlich unteilba-

I 8 Architektur- und Entwurfsmuster

rer Einheiten, z. B. Adresse oder Bankverbindung, zusammengesetzt sein. Eine solche Hierarchie der GUI-Komponenten kann in eine korrespondierende Struktur von MVC-Tripel umgesetzt werden [Star05, S. 220 ff.].

Weitere Varianten des MVC-Musters sind das MVP-Muster, siehe »GUI-Entwurfsmuster MVP«, S. 457, und das MVVM-Muster, siehe »GUIs in der .NET-Plattform«, S. 467.

Vor- und Nachteile

| | |
|-----------|---|
| Vorteile | Das MVC-Muster bringt folgende Vorteile mit sich: <ul style="list-style-type: none">+ Lose gekoppelte, flexible Subsysteme bzw. Komponenten mit hohem Wiederverwendungspotenzial.+ Leichter Austausch von Subsystemen bzw. Komponenten, z. B. für Anpassung auf verschiedene Benutzeroberflächen des Clients (im Web-Umfeld: Webbrowser auf Einzelplatzrechner, Webbrowser auf PDA etc.).+ Gute Erweiterbarkeit der Funktionalität einer Anwendung.+ Verschiedene Darstellung desselben <i>model</i>.+ Synchronisierte Darstellung der verschiedenen Sichten.+ <i>view</i>- und <i>controller</i>-Objekte können während der Laufzeit ausgetauscht werden (<i>pluggable views and controller</i>).+ Leichte Aufteilung der Arbeitspakete in der Entwicklung. Projektmitarbeiter mit verschiedenem Know-how können spezifische Arbeitspakete erhalten, z. B. Web-Designer als Aufgabe die Entwicklung von Teilsystemen der <i>view</i>, Web-Programmierer als Aufgabe die Implementierung von Teilsystemen des <i>controllers</i> oder des <i>models</i>.+ Unterstützung der Wartbarkeit der Anwendung, da durch Änderungen im Umfeld hervorgerufene Wartungsarbeiten im Allgemeinen nur Teilsysteme aus einer Kategorie betreffen. |
| Nachteile | Dem stehen folgende Nachteile gegenüber: <ul style="list-style-type: none">- Der Entwurfs- und der Implementierungsaufwand erhöhen sich.- In verteilten Systemen müssen die Benachrichtigungen des <i>model</i> und die Anfragen an das <i>model</i> gut ausbalanciert werden, um unnötige Netzbelastungen zu vermeiden. Die Attribute eines <i>model</i> sind durch Proxy-Objekte oder Value-Objekte auf einen Schlag abzufragen, statt einzelne <i>get()</i>-Methodenaufrufe abzusetzen, siehe auch »Entwurfskonzepte für verteilte Anwendungen«, S. 306. |

Zusammenhänge

Das MVC-Muster unterstützt die nichtfunktionalen Anforderungen Wartbarkeit, Weiterentwickelbarkeit, Portabilität und Wiederverwendbarkeit (siehe »Nichtfunktionale Anforderungen«, S. 109), das Prinzip der Bindung und Kopplung (siehe »Architekturprinzipien«, S. 29) und das Architekturprinzip der Trennung von Zuständigkeiten (siehe »Architekturprinzip: Trennung von Zuständigkeiten«, S. 31).

8.5 Das Fassaden-Muster (*facade pattern*) I

Die erste Veröffentlichung zum MVC-Muster erfolgte in [Kras88]. [Literatur](#)
Weitere Literatur: [BMR+96, S. 125 ff.], [Wißm09, S. 259 ff.], [Star05,
S. 220 ff.]

8.5 Das Fassaden-Muster (*facade pattern*)

Name(n)

Das Fassaden-Muster (*facade pattern*) ist ein strukturelles Muster (*structural pattern*).

Grundidee

Es stellt einen exklusiven und/oder zusätzlichen, vereinfachten Zugriff auf ein komplexes Subsystem oder auf eine Menge zusammengehöriger Objekte bereit.

Anwendungsbereich(e)

Der Einsatz des Fassaden-Musters ist in folgenden Fällen sinnvoll:

- Es soll eine einfache, häufig benötigte Schnittstelle zu einem komplexen Subsystem angeboten werden. Nur Nutzer, die spezielle Methoden benötigen, müssen oder dürfen hinter die Fassade schauen.
- Bei Verwendung des Schichten-Musters (siehe »Das Schichten-Muster (*layers pattern*)«, S. 46) kann durch eine Fassade die Schnittstelle zu jeder Subsystemschicht definiert werden. Dadurch können die Abhängigkeiten zwischen den Subsystemen vereinfacht werden, wenn sie ausschließlich über ihre Fassaden miteinander kommunizieren (*Black Box*).
- Fassaden können auch weitere Aufgaben übernehmen, zum Beispiel die Zugangskontrolle, das Management von Transaktionen oder die Verwaltung von Zustandsinformationen für Clients oder Server.

Beispiel(e)

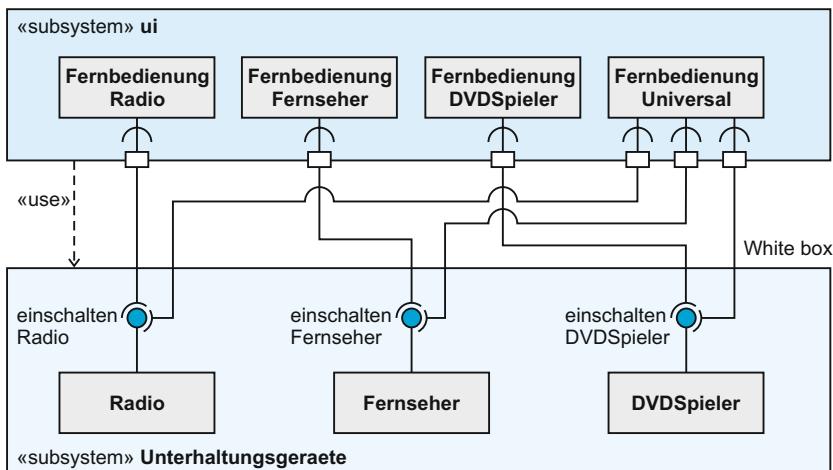
Sie verwalten Ihre Heimunterhaltungsgeräte Fernsehen, Radio mit Verstärker, DVD-Spieler in einem Subsystem Unterhaltungsgeräte. Der Fernseher ist mit dem Radio verbunden. Der Fernsehton ist nur über das Radio zu hören. Zu jedem Gerät gibt es eine eigene Fernbedienung. Zusätzlich gibt es eine universale Fernbedienung, die es ermöglicht das Radio einzuschalten, wenn man Radio hören will. Will man Fernsehen, dann schaltet die universale Fernbedienung sowohl das Radio als auch den Fernseher ein. Will man ein Video ansehen, dann schaltet die universale Fernbedienung sowohl das Radio als den Fernseher als auch den DVD-Spieler ein. Spezielle Einstellungen nehmen Sie mit den Fernbedienungen der einzelnen Geräte vor,

Beispiel 1a:
Unterhaltungs-
geräte

I 8 Architektur- und Entwurfsmuster

während die universale Fernbedienung nur die Voreinstellungen der Geräte verwendet. Die Abb. 8.5-1 zeigt das entsprechende UML-Diagramm.

Abb. 8.5-1: UML-Komponentendiagramm für das Beispiel Unterhaltungsgeräte.



Java In vereinfachter Form sehen die Java-Klassen wie folgt aus:

```
public class Fernseher
{
    public void einschaltenFernseher()
    {
        System.out.println("Fernseher ist eingeschaltet");
    }
}
public class Radio
{
    public void einschaltenRadio()
    {
        System.out.println("Radio ist eingeschaltet");
    }
}
public class DVDSpieler
{
    public void einschaltenDVDSpieler()
    {
        System.out.println("DVDSpieler ist eingeschaltet");
    }
}

import inout.Console;
public class FernbedienungRadio
{
    public static void main(String args[])
    {
        System.out.print
            ("Radio einschalten (e)oder ausschalten (a): ");
    }
}
```

8.5 Das Fassaden-Muster (*facade pattern*) I

```
char eingabe = Console.readChar();
Radio einRadio = new Radio();
if (eingabe == 'e')
    einRadio.einschaltenRadio();
}
}

import inout.Console;
public class FernbedienungFernseher
{
    public static void main(String args[])
    {
        System.out.print
            ("Fernseher einschalten (e)oder ausschalten (a): ");
        char eingabe = Console.readChar();
        Fernseher einFernseher = new Fernseher();
        if (eingabe == 'e')
            einFernseher.einschaltenFernseher();
    }
}

import inout.Console;
public class FernbedienungDVDSpieler
{
    public static void main(String args[])
    {
        System.out.print
            ("DVDSpieler einschalten (e)oder ausschalten (a): ");
        char eingabe = Console.readChar();
        DVDSpieler einDVDSpieler = new DVDSpieler();
        if (eingabe == 'e')
            einDVDSpieler.einschaltenDVDSpieler();
    }
}

import inout.Console;
public class FernbedienungUniversal
{
    public static void main(String args[])
    {
        System.out.print("Fernsehen sehen (f) oder " +
            "Radio hören (r) oder Video sehen (v): ");
        char eingabe = Console.readChar();
        Fernseher einFernseher = new Fernseher();
        Radio einRadio = new Radio();
        DVDSpieler einDVDSpieler = new DVDSpieler();
        switch (eingabe)
        {
            case 'f':
                einFernseher.einschaltenFernseher();
                einRadio.einschaltenRadio(); break;
            case 'r':
                einRadio.einschaltenRadio(); break;
            case 'v':
                einFernseher.einschaltenFernseher();
        }
    }
}
```

I 8 Architektur- und Entwurfsmuster

```
        einRadio.einschaltenRadio();
        einDVDSpieler.einschaltenDVDSpieler(); break;
    }
}
```

Problem/Kontext

Ein Subsystem besteht aus vielen miteinander kommunizierenden Klassen. Die Benutzung des Subsystems ist kompliziert, da die Nutzer die interne Struktur genau kennen müssen. Der Zugriff auf das Subsystem soll vereinfacht werden.

Lösung

Eine Fassade ist eine Klasse mit ausgewählten Methoden, die eine häufig benötigte Untermenge an Methoden des Subsystems umfasst. Dadurch wird der Umgang mit dem Subsystem für Standardfälle vereinfacht. Das Subsystem bleibt trotzdem ein *White Box*-Subsystem, da auch weiterhin auf alle Methoden des Subsystems direkt zugegriffen werden kann. Werden Klassen in Java *nicht* als public gekennzeichnet, dann kann das Subsystem auch zu einem *Black Box*-Subsystem oder einem *Grey Box*-Subsystem gemacht werden (siehe auch »Das Schichten-Muster (*layers pattern*)«, S. 46).

Beispiel 1b: Die Fassade kann wie folgt aussehen:

Unterhaltungs-

geräte

Java

```
public class Fassade
{
    Fernseher einFernseher = new Fernseher();
    Radio einRadio = new Radio();
    DVDSpieler einDVDSpieler = new DVDSpieler();

    public void radioEin()
    {
        einRadio.einschaltenRadio();
    }
    public void fernsehenEin()
    {
        einFernseher.einschaltenFernseher();
        einRadio.einschaltenRadio();
    }
    public void DVDEin()
    {
        einFernseher.einschaltenFernseher();
        einRadio.einschaltenRadio();
        einDVDSpieler.einschaltenDVDSpieler();
    }
}
```

Die FernbedienungUniversal lässt sich dadurch wie folgt vereinfachen:

8.5 Das Fassaden-Muster (*facade pattern*) I

```

import inout.Console;
public class FernbedienungUniversalFassade
{
    public static void main(String args[])
    {
        System.out.print("Fernsehen sehen (f) oder "
            "Radio hören (r) oder Video sehen (v): ");
        char eingabe = Console.readChar();
        Fassade eineFassade = new Fassade();
        switch (eingabe)
        {
            case 'f':
                eineFassade.fernsehenEin(); break;
            case 'r':
                eineFassade.radioEin(); break;
            case 'v':
                eineFassade.DVDEin(); break;
        }
    }
}

```

Das zugehörige UML-Diagramm zeigt die Abb. 8.5-2.

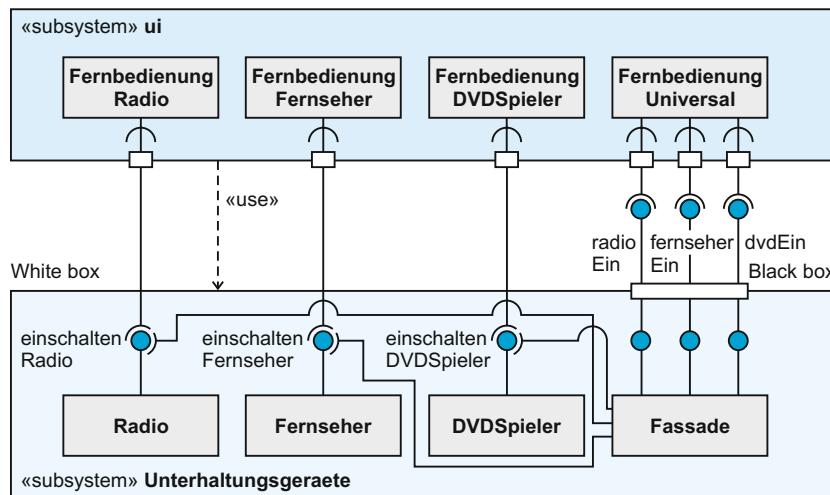


Abb. 8.5-2: UML-Komponentendiagramm für das Beispiel Unterhaltungsgeräte.

Die Fassade

Struktur

- weiß, welche Klassen des Subsystems für eine Anfrage zuständig sind,

- delegiert Anfragen an die entsprechenden Methoden.

Die Subsystemklassen

- stellen die Subsystemfunktionalität bereit,
- führen die von der Fassade aufgerufenen Methoden aus,
- wissen nichts von der Fassade, d. h. sie besitzen keine Referenzen auf sie.

I 8 Architektur- und Entwurfsmuster

Hinweise zur Implementierung In der Regel wird nur ein Fassadenobjekt benötigt, daher wird für die Fassaden-Klasse das **Singleton-Muster** verwendet.

Varianten

In [Fowl97, S. 257 ff.] und Application Facades (<http://martinfowler.com/apsupp/appfacades.pdf>) wird eine Erweiterung des Fassaden-Musters beschrieben. Eine Applikations-Fassade (*application facade*) wird als eigenständige Schicht zwischen die Benutzungsoberfläche und die Applikationsschicht gelegt. Die Benutzungsoberfläche sieht nicht die Applikationsschicht, sondern nur die Applikations-Fassade. Die Applikations-Fassade sieht nicht die Benutzungsoberfläche. Die Klassen der Applikations-Fassade sind dafür zuständig, mit der Applikationsschicht zu kommunizieren und die benötigten Informationen für die Benutzungsoberfläche in exakt der Form zu liefern, wie sie die Klassen der Benutzungsoberfläche benötigen. Dadurch müssen sich die Klassen der Benutzungsoberfläche nur um die Interaktion mit dem Benutzer kümmern und müssen nicht wissen, wie die Anwendungsklassen strukturiert sind.

Vor- und Nachteile

- Vorteile Der Einsatz des Fassaden-Musters bringt folgende Vorteile mit sich:
- + Das Fassaden-Muster fördert die lose Kopplung, weil mehrere, häufig benötigte Methoden in einer Fassaden-Klasse zusammengefasst werden, und dadurch die innere Struktur für den Zugriff nicht bekannt sein muss.
 - + Durch die lose Kopplung kann das Subsystem leichter erweitert werden.
 - + Die Anzahl der Objekte, die vom Nutzer gehandhabt werden müssen, werden reduziert.
 - + Das Subsystem kann durch die Fassade leichter benutzt werden.
- Nachteil Dem steht folgender Nachteil gegenüber:
- Werden während der Entwicklung häufig interne Schnittstellen geändert, dann muss die Fassade häufig angepasst werden.

Zusammenhänge

Das Fassaden-Muster unterstützt die nichtfunktionalen Anforderungen Weiterentwickelbarkeit (Erweiterbarkeit) (siehe »Weiterentwickelbarkeit«, S. 119) und Portabilität (siehe »Portabilität«, S. 132) eines Subsystems.

Es beeinträchtigt die Effizienz (siehe »Leistung und Effizienz«, S. 128).

Das Geheimnisprinzip und das Prinzip der Lokalität werden unterstützt (siehe »Architekturprinzipien«, S. 29).

Literatur

[GHJ+95]

8.6 Das Kommando-Muster (*command pattern*)

Name(n)

Das Kommando-Muster (*command pattern*) ist ein objektbasiertes Verhaltensmuster (*behavioral pattern*). Es ist auch als Befehls-, Aktions- oder Transaktions-Muster bekannt. Es ist eines der am meisten verwendeten Entwurfsmuster.

Grundidee

Es entkoppelt Sender (Auslöser, *invoker*) vom Empfänger (*receiver*). Ein Sender ist ein Objekt, das ein Kommando bzw. einen Befehl erteilt bzw. aufruft (Methodenaufruf), damit eine Anforderung ausgeführt wird. Ein Empfänger ist ein Objekt, das weiß, wie die Anforderung auszuführen ist. Entkopplung bedeutet, dass der Sender kein Wissen über die Schnittstelle des Empfängers hat. Das Kommando bzw. der Befehl wird als ein Objekt gekapselt. Dadurch ist es möglich zu variieren, wann und wie eine Anforderung erfüllt werden soll. Anforderungen können dadurch rückgängig (*undo*) gemacht oder wieder ausgeführt (*redo*) werden.

Anwendungsbereich(e)

Der Einsatz des Kommando-Musters ist in folgenden Fällen sinnvoll:

- Strukturierung eines Systems in abstrakte Operationen, die auf einfachen Operationen basieren.
- Entkopplung des Objekts, das eine Aktion aufruft, von dem Objekt, das die Aktion ausführt. Wird dieses Muster für diesen Zweck eingesetzt, dann wird es auch als Produzenten-Konsumenten-Muster bezeichnet.
- Es wird eine mehrfache Undo- und/oder Redo-Funktionalität benötigt.
- Anforderungen sollen *nicht* sofort, sondern zeitlich verzögert oder asynchron ausgeführt werden. Kommandos können beispielsweise in einer Warteschlange eingeordnet werden, obwohl der Aufrufer einen synchronen Methodenabrufl abgesetzt hat.
- Ausgeführte Kommandos sollen protokolliert werden (Logbuch).
- Transaktionen können modelliert werden. Kommandoobjekte besitzen eine gemeinsame Schnittstelle, die es ihnen ermöglicht, alle Transaktionen auf die gleiche Weise aufzurufen. Außerdem kann ein System um neue Transaktionen erweitert werden.
- Viele Kommandos können von unterschiedlichen Stellen aus aufgerufen werden. Bei einer Textverarbeitung können Kommandos zum Beispiel über einen Menüeintrag, einen Druckknopf, ein Kontext-Menü oder ein Tastaturkürzel aufgerufen werden. Durch die Kapselung eines Kommandos in einem Objekt kann ein und das-selbe Kommando von mehreren Objekten aus aufgerufen werden.

I 8 Architektur- und Entwurfsmuster

- Eine Makro-Aufzeichnung ist möglich, wenn alle Benutzeraktionen durch Kommando-Objekte repräsentiert werden.

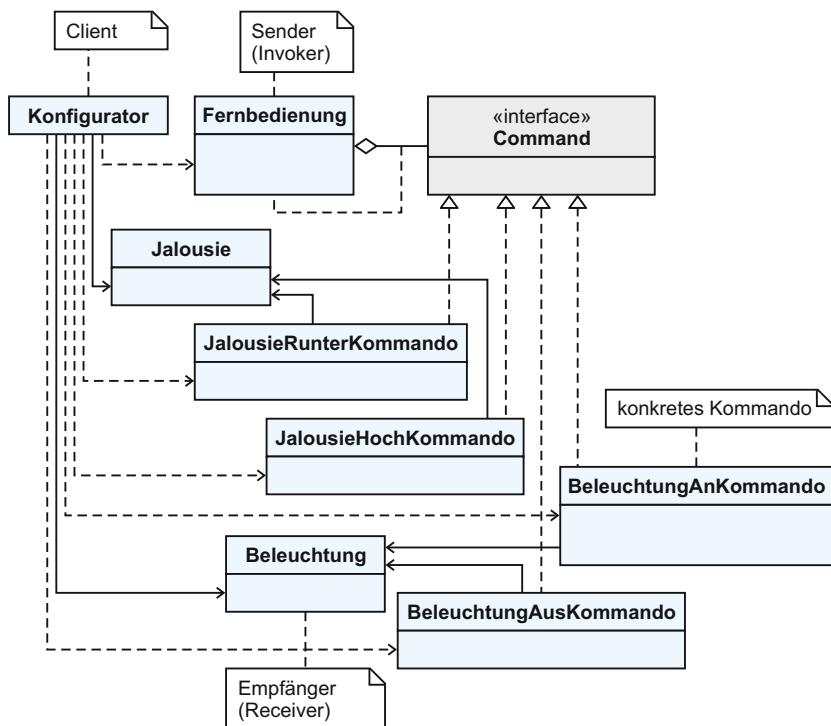
In den XUnit-Frameworks für Unit-Tests (z. B. JUnit) werden Testfälle als Kommandos gekapselt.

Das Paket javax.swing.undo bietet eine Sammlung von Klassen, die es ermöglichen, Java-Anwendungen mit Undo/Redo-Kommandos auszustatten.

Beispiel(e)

Beispiel 1:
Fernbedienung In einem Bürogebäude sollen in jedem Zimmer die Jalousien und die Beleuchtung über eine Fernbedienung gesteuert werden. Die Fernbedienung soll später um weitere Steuerungsfunktionen erweitert werden. Da die Jalousie-Steuerung und die Beleuchtungs-Steuerung von verschiedenen Herstellern stammen, ist die Ansteuerungsschnittstelle verschieden. Die Abb. 8.6-1 gibt einen Überblick über die benötigten Klassen.

Abb. 8.6-1: UML-Klassendiagramm für das Kommando-Muster-Beispiel Fernbedienung.



Java Die Klasse Jalousie stellt folgende Methoden zur Ansteuerung zur Verfügung:

```
class Jalousie //Empfänger (Receiver)
{
    public void herunterfahren()
```

8.6 Das Kommando-Muster (*command pattern*) I

```
{  
    System.out.println("Jalousie wird heruntergefahren");  
}  
public void hochfahren()  
{  
    System.out.println("Jalousie wird hochgefahren");  
}  
}
```

Die Klasse Beleuchtung stellt andere Methoden zur Ansteuerung zur Verfügung:

```
//Empfänger (Receiver)  
class Beleuchtung {  
    public void anschalten()  
    {  
        System.out.println("Beleuchtung wird angeschaltet");  
    }  
    public void ausschalten()  
    {  
        System.out.println("Beleuchtung wird ausgeschaltet");  
    }  
}
```

Die Schnittstelle Command spezifiziert eine abstrakte Methode execute():

```
//Schnittstelle Command  
public interface Command  
{  
    public void execute();  
}
```

Diese Schnittstelle wird von vier konkreten Klassen implementiert.

```
//Konkretes Kommando  
class JalousieHochKommando implements Command  
{  
    private Jalousie meineJalousie;  
    public JalousieHochKommando(Jalousie meineJalousie)  
    {  
        this.meineJalousie = meineJalousie;  
    }  
    public void execute()  
    {  
        meineJalousie.hochfahren();  
    }  
}
```

```
//Konkretes Kommando  
class JalousieRunterKommando implements Command  
{  
    private Jalousie meineJalousie;  
    public JalousieRunterKommando(Jalousie meineJalousie)  
    {  
        this.meineJalousie = meineJalousie;  
    }
```

I 8 Architektur- und Entwurfsmuster

```
public void execute( )
{
    meineJalousie.herunterfahren( );
}

//Konkretes Kommando
class BeleuchtungAnKommando implements Command
{
    private Beleuchtung meineBeleuchtung;
    public BeleuchtungAnKommando(Beleuchtung meineBeleuchtung)
    {
        this.meineBeleuchtung = meineBeleuchtung;
    }
    public void execute( )
    {
        meineBeleuchtung.anschalten( );
    }
}

//Konkretes Kommando
class BeleuchtungAusKommando implements Command
{
    private Beleuchtung meineBeleuchtung;
    public BeleuchtungAusKommando(Beleuchtung meineBeleuchtung)
    {
        this.meineBeleuchtung = meineBeleuchtung;
    }
    public void execute( )
    {
        meineBeleuchtung.ausschalten( );
    }
}
```

In diesen Klassen wird eine Beziehung zwischen einem konkreten Empfänger und seinen Methoden hergestellt.

In der Klasse Fernbedienung werden die Kommandos aufgerufen:

```
//Aufrufer des Kommandos (Invoker)
class Fernbedienung
{
    private Command einKommando, ausKommando;
    public Fernbedienung(Command einKommando, Command ausKommando)
    {
        //Ein konkretes Kommando registriert sich
        //selbst beim Aufrufer
        this.einKommando = einKommando;
        this.ausKommando = ausKommando;
    }
    void ein( )
    {
        //Der Aufrufer ruft das konkrete Kommando auf (call back),
        //das das Kommando beim Empfänger ausführt
        einKommando.execute( );
    }
}
```

8.6 Das Kommando-Muster (*command pattern*) I

```
void aus( )
{
    ausKommando.execute( );
}
```

In der Klasse Konfigurator (normalerweise Client genannt) erfolgt die Zuordnung der Fernbedienungsbelegung zu den Kommandos.

```
//Client
public class Konfigurator
{
    public static void main(String[] args)
    {
        //Mit Kommando-Muster
        Beleuchtung meinLicht = new Beleuchtung();
        //meinLicht wird dem BeleuchtungAnKommando zugeordnet
        BeleuchtungAnKommando meinLichtAn =
            new BeleuchtungAnKommando(meinLicht);
        BeleuchtungAusKommando meinLichtAus =
            new BeleuchtungAusKommando(meinLicht);
        //Zuordnung von meinLicht zu der Fernbedienung
        Fernbedienung meineFernbedienungKnopf1 =
            new Fernbedienung(meinLichtAn,meinLichtAus);
        meineFernbedienungKnopf1.ein();
        meineFernbedienungKnopf1.aus();

        Jalousie meineJalousie = new Jalousie();
        JalousieHochKommando meineJalousieHoch =
            new JalousieHochKommando(meineJalousie);
        JalousieRunterKommando meineJalousieRunter =
            new JalousieRunterKommando(meineJalousie);
        Fernbedienung meineFernbedienungKnopf2 =
            new Fernbedienung(meineJalousieHoch,
                meineJalousieRunter);
        meineFernbedienungKnopf2.ein();
        meineFernbedienungKnopf2.aus();

        //Ohne Kommando-Muster
        //((die Schnittstelle pro Gerät muss bekannt sein)
        meinLicht.anschalten();
        meinLicht.ausschalten();
        meineJalousie.herunterfahren();
        meineJalousie.hochfahren();
    }
}
```

I 8 Architektur- und Entwurfsmuster

Die Ausführung des Programms führt zu folgendem Ergebnis:

```
Beleuchtung wird angeschaltet  
Beleuchtung wird ausgeschaltet  
Jalousie wird hochgefahren  
Jalousie wird heruntergefahren  
Beleuchtung wird angeschaltet  
Beleuchtung wird ausgeschaltet  
Jalousie wird heruntergefahren  
Jalousie wird hochgefahren
```

Wie der Quellcode zeigt, entkoppelt das Kommando-Muster die Objekte, die Operationen aufrufen (hier Fernbedienung) von denen, die wissen, wie die Operationen auszuführen sind (hier Jalousie und Beleuchtung). Dadurch erhält man eine große Flexibilität. Die Objekte, die eine Anforderung auslösen, müssen nur wissen, wie sie die Anforderung auslösen (hier Drücken eines Fernbedienungsknopfes bzw. Aufruf der Methoden `ein()` oder `aus()`). Sie müssen aber nicht wissen, wie die Anforderung ausgeführt wird (hier `anschalten()`, `ausschalten()` oder `herunterfahren()`, `hochfahren()`).

Im E-Learning-Kurs zu diesem Buch finden Sie eine Animation zur Verdeutlichung dieses Sachverhalts.



Zeichnen Sie für das Beispiel ein UML-Sequenzdiagramm und ein UML-Objektdiagramm, um sich die dynamischen Vorgänge zu verdeutlichen.

Problem/Kontext

Der Aufruf eines Kommandos soll von der Ausführung des Kommandos entkoppelt werden.

Lösung

In einer Schnittstelle `Command` werden die Schnittstellen der Kommandos festgelegt, z. B. `execute()`, `undo()`, `redo()` (Abb. 8.6-2). Klassen, die diese Schnittstelle implementieren, legen ein Empfänger-Kommando-Paar fest, indem sie den Empfänger als ein Objektattribut speichern und das `execute()`-Kommando so implementieren, dass es ausgeführt werden kann. Empfänger wissen, wie ein Kommando auszuführen ist. Grundsätzlich kann jede beliebige Klasse ein Empfänger sein. Eine Klasse `ConcreteCommand` kann die Ausführung des Kommandos selbst implementieren, anstatt sie an Receiver-Klassen zu delegieren.

- Struktur
- Die Schnittstelle `Command` spezifiziert eine Schnittstelle zum Ausführen einer Methode.
 - Die Klasse `ConcreteCommand`
 - bindet einen Empfänger (Receiver) an eine Methode.
 - implementiert `execute()` durch Aufrufen der entsprechenden Methoden beim Empfänger.

8.6 Das Kommando-Muster (*command pattern*) I

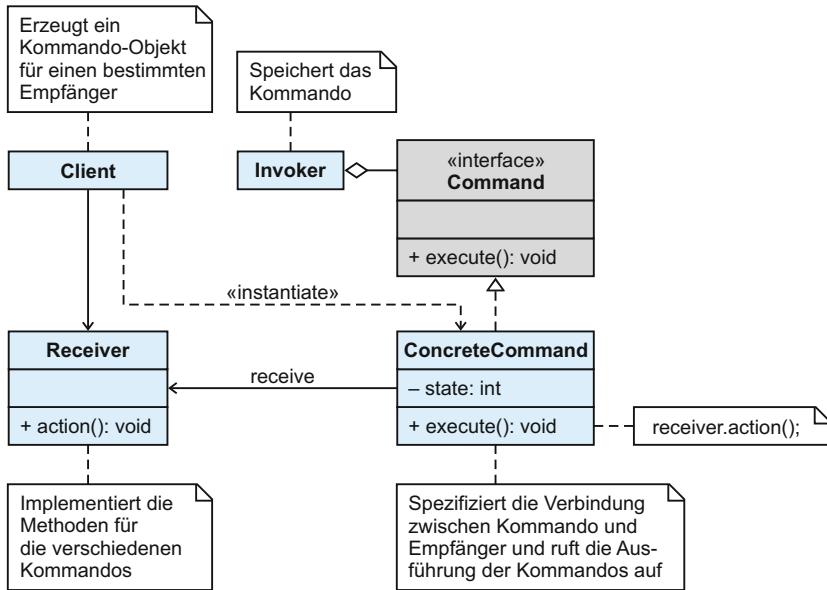


Abb. 8.6-2: UML-Klassendiagramm des Kommando-Musters.

Die Anwendung (Client) erzeugt ein Objekt der Klasse `ConcreteCommand` und übergibt ihm den Empfänger.

Der Aufrufer (Invoker) befiehlt dem Kommandoobjekt, das Kommando auszuführen.

Der Empfänger weiß, wie die an die Ausführung eines Kommandos gebundenen Methoden auszuführen sind.

Die Abb. 8.6-3 zeigt die Interaktionen zwischen den Objekten. Dynamik.

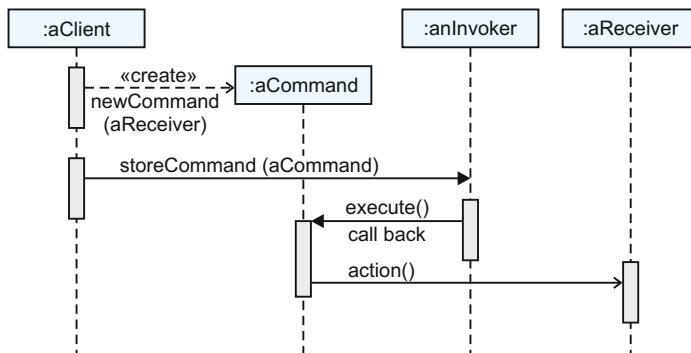


Abb. 8.6-3: UML-Sequenzdiagramm des Kommando-Musters.

In der Java-Swing-Bibliothek wird das Kommando-Muster ebenfalls Beispiel 2 eingesetzt. Die entsprechenden Schnittstellen und Klassen sind in der Tab. 8.6-1 gegenübergestellt.

I 8 Architektur- und Entwurfsmuster

Tab. 8.6-1:
Vergleich
Kommando-Muster
vs.
Swing-Bibliothek.

| Kommando-Muster | Swing-Bibliothek |
|-------------------------|--------------------------------|
| Command | AbstractAction |
| ConcreteCommand | ConcreteAction |
| Unterklasse von Command | Unterklasse von AbstractAction |
| execute() | actionPerformed() |

- Hinweise zur Implementierung
- Intelligenz von Kommandobjekten
 - Keine Intelligenz: Das Kommando stellt nur eine Verbindung zwischen einem Empfänger und den Methoden her, die das Kommando umsetzen.
 - Keine Delegation: Das Kommando implementiert alle Funktionalitäten selbst, ohne irgendetwas an den Empfänger zu delegieren. Dieser Sonderfall ist sinnvoll, wenn Kommandos spezifiziert werden sollen, die von existierenden Klassen unabhängig sind, wenn kein passender Empfänger existiert oder wenn ein Kommando seinem Empfänger nur implizit kennt.

Varianten

Das *Command Processor*-Muster trennt die Ausführung und das Management von *Command*-Objekten [EiSt07, S. 41 f.].

Vor- und Nachteile

| | |
|-----------|---|
| Vorteile | Der Einsatz des Kommando-Musters bringt folgende Vorteile mit sich: <ul style="list-style-type: none">+ Aufrufer müssen nicht wissen, welche Objekte die Kommandos letztendlich ausführen.+ Aufrufer sind nur von der Schnittstelle <i>Command</i> abhängig.+ In der Regel verwenden Klassen Objekt- oder Klassen-Attribute um über Methoden ihre Dienstleistungen zur Verfügung zu stellen. Bei Verwendung des Kommando-Musters werden die Kommandos losgelöst von Attributen ausgeführt.+ Komandoobjekte können manipuliert und erweitert werden – wie jedes andere Objekt auch.+ Komandoobjekte können leicht hinzugefügt werden, da keine existierenden Klassen geändert werden müssen. In diesem Fall kann das <i>Composite</i>-Muster verwendet werden, um existierende Kommandos zu einem neuen Kommando zu gruppieren.+ Kommandos können zu zusammengesetzten Kommandos zusammengefügt werden. |
| Nachteile | Dem stehen folgende Nachteile gegenüber: <ul style="list-style-type: none">- Die Systemstruktur ist schwieriger zu verstehen.- Es werden viele zusätzliche kleine Klassen benötigt. |

Zusammenhänge

Das Kommando-Muster unterstützt die nichtfunktionale Anforderung Weiterentwickelbarkeit, insbesondere das Teilmerkmal Erweiterbarkeit (siehe »Weiterentwickelbarkeit«, S. 119). Durch die Möglichkeit, Undo- und Redo-Funktionalität zu implementieren, unterstützt das Kommando-Muster auch die Benutzbarkeit (siehe »Benutzbarkeit«, S. 130).

Kommandoobjekte sind ein objektorientierter Ersatz für *Callbacks*, siehe »Exkurs: *Callback*«, S. 42.

Das Kommando-Muster fördert das Prinzip der Abstraktion (siehe »Architekturprinzipien«, S. 29), reduziert aber das Architekturprinzip Sichtbarkeit (siehe »Architekturprinzip: Sichtbarkeit«, S. 34).

[GHJ+95]

Literatur

8.7 Das Proxy-Muster (*proxy pattern*)

Name(n)

Das Proxy-Muster (*proxy pattern*) ist ein strukturelles Muster (*structural pattern*). Es ist auch unter dem Namen Stellvertreter-Muster (*placeholder pattern*, *surrogate pattern*) bekannt.

Grundidee

Der Zugriff auf ein Objekt soll mithilfe eines vorgelagerten Stellvertreterobjekts kontrolliert werden.

Anwendungsbereich(e)

Der Einsatz des Proxy-Musters ist in folgenden Fällen sinnvoll:

- In allen Fällen, in denen ein Client ein Objekt nicht direkt referenzieren darf oder kann, aber dennoch mit dem Objekt interagieren will.
- Besitzen unterschiedliche Zugriffsobjekte verschiedene Zugriffsrechte auf ein Objekt, dann kann ein **Schutz-Proxy** (*protection proxy*, *access proxy*) den Zugriff auf das Originalobjekt kontrollieren (siehe Beispiel 2).
- Befindet sich ein Objekt in einem anderen Adressraum, z. B. auf einem anderen Computersystem, dann repräsentiert ein **Remote-Proxy** einen lokalen Stellvertreter. Zugreifende Objekte werden bei verteilten Systemen vor der Netzwerk-Adressierung und der Prozess-Kommunikation abgeschirmt (siehe »Arten der Netzkomunikation«, S. 205). Ein Remote-Proxy kann Anfragen verschlüsseln.
- Sollen »teure« Methoden nur aufgerufen werden, wenn sie auch benötigt werden, dann kann ein **virtueller Proxy** benutzt werden. Beispielsweise sollten Bilder erst dann in einen Texteditor geladen werden, wenn sie in den sichtbaren Bereich gescrollt werden.

I 8 Architektur- und Entwurfsmuster

den. Eine Sonderform des virtuellen Proxy ist der Copy-on-Write-Proxy, der ein von ihm verwaltetes Objekt erst kopiert, wenn es modifiziert wurde.

- Ein Proxy kann auch dazu verwendet werden, das Verhalten eines Objektes zu erweitern oder zu verändern (*smart reference*) (siehe Beispiel 1). Beispielsweise können Referenzen gezählt werden oder es kann getestet werden, ob das eigentliche Objekt für den Zugriff frei ist. Methodenaufrufe können synchronisiert werden.
- Ein **Cache-Proxy** stellt einen temporären Speicher für aufwändige Methoden des Originalobjekts zur Verfügung, damit verschiedene Clients auf die Ergebnisse zugreifen können.
- Ein **Firewall-Proxy** schützt Objekte vor »bösen« Clients.
- Ein **Synchronisations-Proxy** ermöglicht den nebenläufigen Zugriff auf ein Objekt.

Beispiel(e)

Beispiel 1: Wetterstation Eine Wetterstation liefert auf Anfrage die aktuelle Temperatur in Fahrenheit. Auf die Wetterstation wird über eine Proxy-Klasse zugegriffen, die zusätzlich die gelieferte Temperatur in Celsius umrechnet.

Java Die gemeinsame Schnittstelle für den Proxy und die Wetterstation sieht wie folgt aus:

```
//Diese Schnittstelle stellt sicher, dass der Proxy
//dieselben Methoden zur Verfügung stellt, wie das reale Subjekt
public interface WetterI
{
    public double getTemperatur();
}
```

Die Klassen ProxyWetter und Wetter können folgendermaßen aussehen:

```
public class ProxyWetter implements WetterI
{
    Wetter eineWetterstation;
    public ProxyWetter()
    {
    }

    public double getTemperatur()
    {
        eineWetterstation = new Wetter();
        double eineTemperaturInFahrenheit =
            eineWetterstation.getTemperatur();
        return (eineTemperaturInFahrenheit - 32) * 5 / 9;
    }
}

public class Wetter implements WetterI
{
    public Wetter()
```

8.7 Das Proxy-Muster (*proxy pattern*) I

```
{  
}  
  
public double getTemperatur()  
{  
    return (Math.floor(Math.random() * 101));  
}  
}
```

Zur Simulation dient die Klasse ClientWetter:

```
public class ClientWetter  
{  
    public static void main(String[] args)  
    {  
        WetterI eineWetterstation = new ProxyWetter();  
        System.out.println("Aktuelle Temperatur 1: " +  
            eineWetterstation.getTemperatur());  
        System.out.println("Aktuelle Temperatur 2: " +  
            eineWetterstation.getTemperatur());  
    }  
}
```

Zeichnen Sie für das Beispiel 1 ein UML-Klassendiagramm.



Der Zugriff auf Personaldaten soll nur berechtigten Personen ermöglicht werden.

Beispiel 2:
Personaldaten

Java

```
package de.w3l.protect;  
//Schnittstelle zu den Personaldaten  
public interface PersonalI  
{  
    public String[] getListe();  
}  
  
package de.w3l.protect;  
  
public class ProxyPersonal implements PersonalI  
{  
    private Personal einePersonalliste;  
    private String einName, zugangName = "Sommer";  
    private String einPasswort, zugangPasswort = "geheim";  
    private String [] eineFehlermeldung =  
        {"Keine Zugangsberechtigung"};  
  
    public ProxyPersonal(String einName, String einPasswort)  
{  
        this.einName = einName;  
        this.einPasswort = einPasswort;  
        einePersonalliste = new Personal();  
    }  
  
    @Override  
    public String[] getListe()  
    {  
        if (einName.equals(zugangName)&&
```

I 8 Architektur- und Entwurfsmuster

```
        einPasswort.equals(zugangPasswort))
        return einePersonalliste.getListe();
    else
        return eineFehlermeldung;
}
}

package de.w3l.protect;

public class Personal implements PersonalI
{
    String [] gehalt = {"Meyer: 2500", "Herbst: 4800"};

    /*
     * Keine Angabe: Zugriff nur aus Klassen desselben Pakets
     * Selbst bei Vererbung kein Zugriff
     *
     * public: Von allen Klassen aller Pakete zugreifbar
     *
     * protected: Von allen Klassen dieses Pakets zugreifbar
     * Klassen von außerhalb können jedoch zugreifen, wenn sie
     * Unterklassen sind
     *
     * private: Nur von dieser Klasse aus zugreifbar
     */
    Personal() //Package private (default)
    {
    }

    @Override
    public String[] getListe()
    {
        return gehalt;
    }
}

package de.w3l.client;

import de.w3l.protect.*;

public class Client
{
    public static void main(String[] args)
    {
        PersonalI einProxy = new ProxyPersonal("Sommer", "geheim");
        String [] liste = einProxy.getListe();
        for (int i = 0; i < liste.length; i++)
            System.out.println("Gehaltsliste: " + liste[i]);
        einProxy = new ProxyPersonal("Herb", "geheim");
        liste = einProxy.getListe();
        for (int i = 0; i < liste.length; i++)
            System.out.println("Gehaltsliste: " + liste[i]);
        /*Umgehen
        Personal einPersonal = new Personal();
```

8.7 Das Proxy-Muster (*proxy pattern*) I

```
liste = einPersonal.getListe();
for (int i = 0; i < liste.length; i++)
    System.out.println("Gehaltsliste: " + liste[i]);*/
}
}
```

Folgendes wird ausgegeben:

```
Gehaltsliste: Meyer: 2500
Gehaltsliste: Herbst: 4800
Gehaltsliste: Keine Zugangsberechtigung
```

Damit der Proxy nicht umgangen werden kann, müssen die Klassen `ProxyPersonal` und `Personal` sich in einem Paket befinden. Der Konstruktor von der Klasse `Personal` muss `package private` sein, was er *by default* auch ist. Der Client darf *nicht* im selben Paket liegen.

Problem/Kontext

Der Zugriff auf ein Objekt kann schwierig sein, die Erzeugung kann aufwändig sein oder lange dauern, das Objekt kann in einem Netzwerk verteilt liegen oder das Objekt soll aus Sicherheitsgründen vor dem Zugriff von außen geschützt werden.

Lösung

Ein Proxy ist in seiner allgemeinsten Form eine Klasse, die als Schnittstelle zu einem anderen »Subjekt« auftritt. Dieses Subjekt kann beispielsweise ein großes Objekt im Speicher, ein Objekt auf einem anderen Computersystem, eine Datei oder eine andere Ressource sein. Als Stellvertreter dieses Objekts kann der Proxy die Erzeugung des Objektes sowie den Zugriff darauf kontrollieren.

Die Abb. 8.7-1 zeigt die allgemeine Struktur. Der Client stellt das Struktur Objekt dar, das durch den Proxy auf das reale Subjekt zugreift.

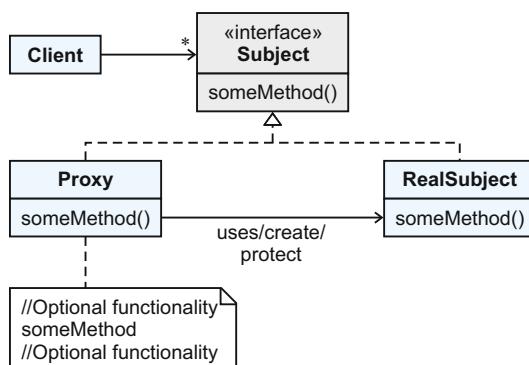


Abb. 8.7-1: UML-Klassendiagramm des Proxy-Musters.

Die Klasse `Proxy` bietet nach außen hin eine zur Klasse `RealSubject` identische Schnittstelle. Sie verwaltet eine Referenz auf `RealSubject` und ist eventuell verantwortlich für die Erzeugung und Löschung von Objekten (Ressourcenmanagement). Sie leitet Methodenaufrufe

I 8 Architektur- und Entwurfsmuster

fe an das RealSubject weiter (Delegation). Die Schnittstelle Subject spezifiziert die gemeinsame Schnittstelle der Klassen Proxy und RealSubject.

Varianten

Das Proxy-Musters hat den Nachteil, dass für jede Applikations-Klasse, jede Applikations-Methode und jede zusätzliche Framework-Funktionalität eine Stellvertreter-Klasse/Methode implementiert und gewartet werden muss. Wird eine Applikations-Klasse um eine Methode ergänzt, dann muss diese ebenfalls im Proxy implementiert werden. Eine Lösung für dieses Problem sind **dynamische Proxy**, die sich vom statischen Proxy dadurch unterscheiden, dass sie zur Laufzeit für ein Objekt dynamisch erzeugt und nicht bereits zur Implementierungszeit programmiert werden. Dynamische Proxies werden auch als »Abfänger« (*interceptor*) verwendet. Ein »Abfänger« fängt den Zugriff auf ein Objekt ab und kann vor und nach jedem Methodenaufruf bestimmte Aktionen ausführen.

In Java ist es möglich, mit der Schnittstelle InvocationHandler und der Klasse Proxy dynamische Proxy zu erzeugen.

Vor- und Nachteile

| | |
|-----------|---|
| Vorteile | Der Einsatz des Proxy-Musters bringt folgende Vorteile mit sich: <ul style="list-style-type: none">+ Die Funktionalität des echten Objekts kann durch den Proxy durch Sicherheitsabfragen, Protokollierung, <i>Pooling</i>, <i>Caching</i> oder Ähnliches geändert bzw. erweitert werden.+ Durch den Umweg (<i>indirection</i>) über einen Proxy wird eine geringere Kopplung erreicht.+ Der Client muss <i>nicht</i> wissen, wo das eigentliche Objekt liegt. Dadurch wird der Zugriff transparenter.+ Der Client muss sich nicht um das Speichermanagement kümmern. Dadurch wird die Verantwortung besser verteilt.+ Das Laden von großen Objekten wird optimiert. Das ergibt eine gesteigerte Effizienz. |
| Nachteile | Durch den Einsatz des Proxy-Musters ergeben sich folgende Nachteile: <ul style="list-style-type: none">- Der Umweg über einen Proxy erhöht die Komplexität.- Für jede einzelne Methode muss in der Proxy-Klasse eine Methode mit gleicher Signatur implementiert werden. Das ist aufwändig und führt u. U. zu einer unerwünschten Redundanz. Dieser Nachteil kann durch einen dynamischen Proxy vermieden werden, der das Proxy-Objekt zur Laufzeit erzeugt (siehe Abschnitt Varianten oben).- Beim Löschen des eigentlichen Objekts muss – synchronisiert dazu – auch das Proxy-Objekt gelöscht werden. Das führt zu einem komplexen Ressourcen-Management. |

8.8 Fabrikmethoden-Muster (*factory method pattern*) I

Zusammenhänge

Das Proxy-Muster unterstützt die nichtfunktionale Anforderung Sicherheit (siehe »Betriebssicherheit und Funktionssicherheit«, S. 121) und erschwert die Wartbarkeit (siehe »Wartbarkeit«, S. 116).

Das Prinzip der Bindung und Kopplung (siehe »Architekturprinzipien«, S. 29) wird unterstützt, ebenso das Architekturprinzip »Trennung von Zuständigkeiten« (siehe »Architekturprinzip: Trennung von Zuständigkeiten«, S. 31). Der Entwicklungsaufwand wird erhöht.

Die Tab. 8.7-1 zeigt die Eigenschaften von drei Mustern im Vergleich.

| Proxy | Dekorator | Adapter |
|---------------------------|-----------------------------------|--------------------------------|
| gleiche Schnittstelle | gleiche Schnittstelle | unterschiedliche Schnittstelle |
| kontrolliert Zugriff | zusätzliche / veränderte Funktion | ähnlich Schutz-Proxy |
| »Implementierungs-Detail« | »konzeptionelle Eigenschaft« | |

Das Proxy-Muster wird oft in Kombination mit dem Fabrikmethoden-Muster verwendet (siehe »Fabrikmethoden-Muster (*factory method pattern*)«, S. 89).

[GHJ+95]

Tab. 8.7-1:
Mustervergleich.

Literatur

8.8 Fabrikmethoden-Muster (*factory method pattern*)

Name(n)

Das Fabrikmethoden-Muster (*factory method pattern*) ist ein klassenbasiertes Erzeugungsmuster. Es ist auch als virtueller Konstruktor (*virtual constructor*) bekannt.

Grundidee

In der Regel wird durch den Aufruf eines Konstruktors einer Klasse ein neues Objekt erzeugt. Das hat verschiedene Nachteile:

- Der Name des Konstruktors ist identisch mit dem Klassennamen. Der problembezogene Kontext ist dadurch oft nicht sichtbar. Beispielsweise kann eine Klasse Komplex nicht zwei beschreibende Erzeugungs-Methoden `vonKartesischenKoordinaten(double real, double im)` und `vonPolarkoordinaten(double real, double im)` bereitstellen, sondern nur den Konstruktor `Komplex(double real, double im)`.
- Durch den Aufruf eines Konstruktors wird immer ein neues Objekt erzeugt. Ist bereits ein Objekt vorhanden, dann kann es nicht wieder verwendet werden.

I 8 Architektur- und Entwurfsmuster

Fabrikmethoden kapseln die Erzeugung von Objekten. Ist ein geeignetes Objekt bereits vorhanden, dann kann es wieder verwendet werden. Der Name der Erzeugungsmethode kann frei gewählt werden. Dadurch werden die beiden genannten Nachteile vermieden.

Das Fabrikmethoden-Muster geht jedoch noch weiter. Eine Fabrikmethoden-Klasse – oft Erzeuger, *Factory* oder *Creator* genannt – erzeugt die Objekte nicht selbst, sondern delegiert die Objekterzeugung an Unterklassen. Was zu erzeugen ist, wird also delegiert. Orthogonal wird mit dem Fabrikmethoden-Muster auch festgelegt, wie etwas zu erzeugen bzw. herzustellen ist. Dazu wird eine Produkt-hierarchie aufgebaut.

Anwendungsbereich(e)

Der Einsatz des Fabrikmethoden-Musters ist in folgenden Fällen sinnvoll:

- Wenn eine Klasse die von ihr zu erzeugenden Objekte nicht kennen kann bzw. soll, oder wenn Unterklassen bestimmen sollen, welche Objekte erzeugt werden.
- Wenn die Erzeugungsmethode einen beschreibenden Namen besitzen soll, anstelle des Klassennamens. Es kann mehrere Fabrikmethoden mit unterschiedlichen Namen und unterschiedlicher Semantik geben.
- Wenn die Erzeugungsmethode die Erzeugung von Objekten kapseln soll. Dies kann sinnvoll sein, wenn der Erzeugungsprozess sehr komplex ist, zum Beispiel wenn er von Konfigurationsdateien oder von Benutzereingaben abhängig ist.

Beispiel(e)

Beispiel 1: Der Dachverband der Sparkassen erhält die Aufgabe, für alle Sparkassen, die sich beteiligen wollen, eine Software zur Verfügung zu stellen, um Portfolios zu erstellen und auszugeben. Jedes Portfolio besteht aus einzelnen Produkten, die von der jeweiligen Sparkasse bestimmt werden können. Um diese gewünschte Flexibilität zu erreichen, wird eine abstrakte Oberklasse Sparkasse mit der abstrakten Methode `erzeugePortfolioObjekt()` sowie einer Methode zur Ausgabe aller Produkte eines Portfolios `getPortfolio()` erstellt. Jede Sparkasse, die dieses Angebot nutzen will, erstellt eine konkrete Unterklassse zur Klasse Sparkasse und redefiniert die Methode `erzeugePortfolioObjekt()`. Für das zu erstellende Portfolio wird die abstrakte Oberklasse Portfolio mit einer Methode `druckePortfolio()` für die Ausgabe aller Produkte erstellt. Die Produkte werden in einer Liste verwaltet und es wird festgelegt, dass die Methode `erstellePortfolio()` von der jeweiligen Unterklassse redefiniert werden muss. Für die konkreten Produkte wird eine Klasse Produkt mit vorgegebenen Attributen und Methoden bereitgestellt. Jede Sparkas-

8.8 Fabrikmethoden-Muster (*factory method pattern*) I

se erstellt eigene Portfoliounterklassen mit ihren individuellen Produkten. Die Programme, hier DemoSparkasse, die diese Software nutzen wollen, müssen nur Objekte der jeweiligen Sparkassen erstellen und die Methode `getPortfolio()` der Oberklasse aufrufen. Die Abb. 8.8-1 zeigt das UML-Klassendiagramm.

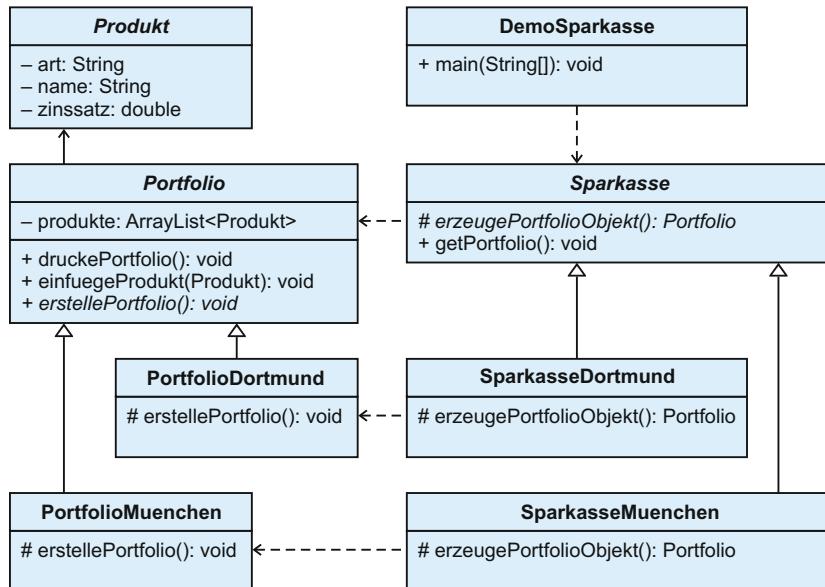


Abb. 8.8-1:
Klassendiagramm
für das Beispiel
Fabrikmethoden-
Muster.

Die einzelnen Klassen sind im Folgenden aufgeführt:

```

//Produkt eines Portfolios
public class Produkt
{
    private String name;
    private String art;
    private double zinssatz;

    public Produkt(String name, String art, double zinssatz)
    {
        this.name = name;
        this.art = art;
        this.zinssatz = zinssatz;
    }

    public String getName()
    {
        return name;
    }

    public String getArt()
    {
        return art;
    }
}
  
```

Java

DemoSparkasse

I 8 Architektur- und Entwurfsmuster

```
public double getZinssatz()
{
    return zinssatz;
}

//Oberklasse
public abstract class Sparkasse
{
/*
 * Die Festlegung, welche Portfolioklasse genutzt werden
 * soll, wird in der Unterkasse durchgefuehrt.
 */
protected abstract Portfolio erzeugePortfolioObjekt();

public void getPortfolio()
{
    Portfolio pf = erzeugePortfolioObjekt();
    pf.druckePortfolio();
}
}

public class SparkasseDortmund extends Sparkasse
{
protected Portfolio erzeugePortfolioObjekt()
{
    return new PortfolioDortmund();
}
}

public class SparkasseMuenchen extends Sparkasse
{
protected Portfolio erzeugePortfolioObjekt()
{
    return new PortfolioMuenchen();
}
}

//Oberklasse
public abstract class Portfolio
{
ArrayList<Produkt> produkte = new ArrayList<Produkt>();

/*
 * Die Festlegung, welche Produkte zum Portfolio gehoeren,
 * wird in der Unterkasse durchgefuehrt.
 */
protected abstract void erstellePortfolio();

public void einfuegeProdukt(Produkt einProdukt)
{
    produkte.add(einProdukt);
}

public void druckePortfolio()
```

8.8 Fabrikmethoden-Muster (*factory method pattern*) I

```
{  
    erstellePortfolio();  
    for (Produkt produkt: produkte)  
        System.out.println(  
            "Sparkasse " + produkt.getName() +  
            " gewährt auf " + produkt.getArt() +  
            " " + produkt.getZinssatz() + "% Zinsen."  
        );  
    }  
}  
  
public class PortfolioDortmund extends Portfolio  
{  
    @Override  
    protected void erstellePortfolio()  
    {  
        einfuegeProdukt(new Produkt("Dortmund",  
            "Kredit > 50.000 Euro", 5.25));  
        einfuegeProdukt(new Produkt("Dortmund",  
            "Kredit <= 50.000 Euro", 6.0));  
        einfuegeProdukt(new Produkt("Dortmund",  
            "Festgeld", 0.5));  
    }  
}  
  
public class PortfolioMuenchen extends Portfolio  
{  
    @Override  
    protected void erstellePortfolio()  
    {  
        einfuegeProdukt(new Produkt("München",  
            "Kredit", 5.0));  
        einfuegeProdukt(new Produkt("München",  
            "Festgeld", 0.75));  
    }  
}  
  
public class DemoSparkasse  
{  
    public static void main(String[] args)  
    {  
        Sparkasse dortmund = new SparkasseDortmund();  
        dortmund.getPortfolio();  
  
        Sparkasse muenchen = new SparkasseMuenchen();  
        muenchen.getPortfolio();  
    }  
}
```

Der Programmlauf ergibt folgendes Ergebnis:

```
Sparkasse Dortmund gewährt auf Kredit > 50.000 Euro 5.25% Zinsen.  
Sparkasse Dortmund gewährt auf Kredit <= 50.000 Euro 6.0% Zinsen.  
Sparkasse Dortmund gewährt auf Festgeld 0.5% Zinsen.  
Sparkasse München gewährt auf Kredit 5.0% Zinsen.  
Sparkasse München gewährt auf Festgeld 0.75% Zinsen.
```

I 8 Architektur- und Entwurfsmuster

Ein UML-Sequenzdiagramm zeigt die Abb. 8.8-2.

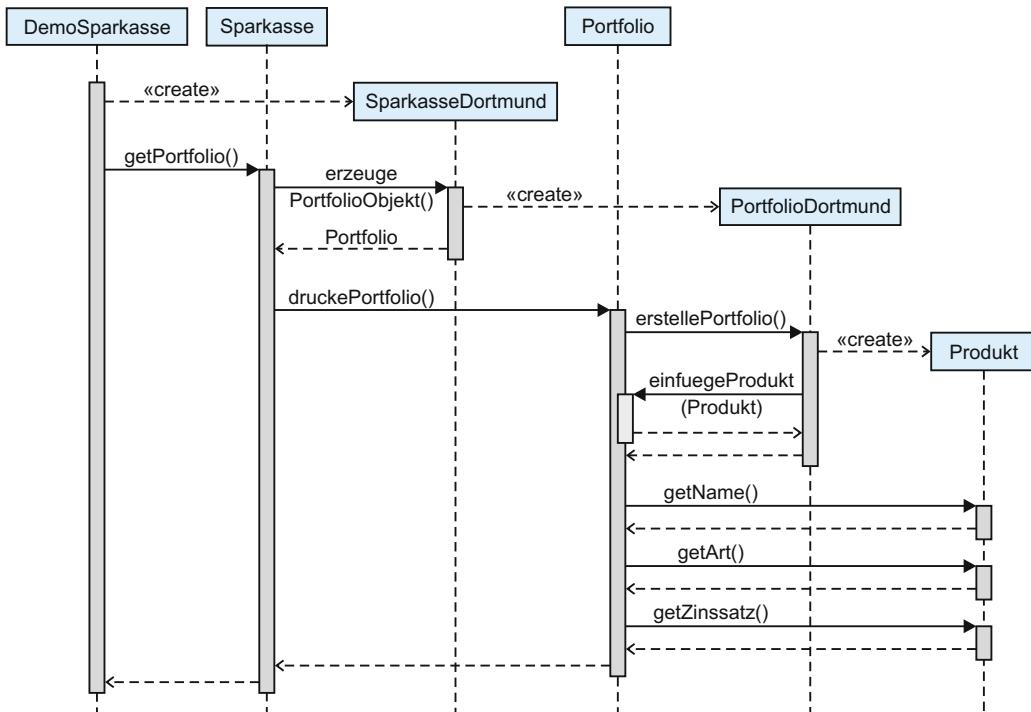


Abb. 8.8-2:
Sequenzdiagramm
für das Beispiel
Fabrikmethoden-
Muster.

Problem/Kontext

Eine Anwendung soll von einer Familie von Klassen getrennt werden, um bei einer Weiterentwicklung der Klassen die Anwendung nicht oder nur geringfügig ändern zu müssen. Es soll eine lose Kopplung entstehen.

Ein Objekt soll immer durch den Aufruf einer Methode erzeugt werden. Es ist nicht erlaubt, ein Objekt durch den direkten Aufruf eines Konstruktors zu instanzieren.

Lösung

Eine abstrakte Oberklasse spezifiziert das Standardverhalten und das generische Verhalten und delegiert die Erzeugungsdetails an die Unterklassen.

Struktur

Die Struktur des Fabrikmethoden-Musters zeigt die Abb. 8.8-3.

Die Struktur zeigt zwei parallele Klassenhierarchien. Beide haben abstrakte Klassen, die von den konkreten Klassen erweitert werden. Die abstrakte Klasse oder Schnittstelle `Product` spezifiziert das zu erzeugende Produkt. Die Klasse `Creator` deklariert die Fabrikmethode, um ein solches Produkt zu erzeugen und kann eine *Default*-Implementierung enthalten.

8.8 Fabrikmethoden-Muster (*factory method pattern*) I

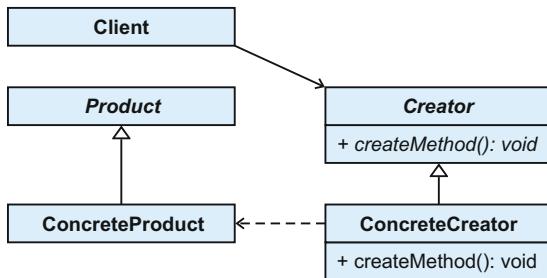


Abb. 8.8-3:
Allgemeine
Struktur des
Fabrikmethoden-
Musters.

- Wenn eine Vererbungshierarchie vorliegt, die Polymorphismus ermöglicht, dann wird eine polymorphe Fabrikmethode in der Oberklasse bereitgestellt.
- Es ist zu überlegen, ob ein interner »Objektpool« gebildet wird, der es erlaubt, Objekte wieder zu verwenden anstatt neue zu erzeugen.

Hinweise zur
Implementierung

Varianten

- Parametrisierte Fabrikmethoden: Die Fabrikmethode erhält Parameter, die die Art des erzeugten Objekts bestimmen.
- Die Klasse Creator muss nicht abstrakt sein.

Vor- und Nachteile

- | | |
|--|-----------|
| <ul style="list-style-type: none"> + Die Architektur ist besser anpassbar. + Die Abhängigkeiten von konkreten Klassen werden reduziert. + Der Aufrufer (Client) ist vollständig von den Implementierungsdetails der Unterklassen entkoppelt. Das ist essenziell, wenn Frameworks sich während der Lebenszeit einer Anwendung weiter entwickeln. Zu einem späteren Zeitpunkt können Objekte anderer Klassen erzeugt werden, ohne dass sich die Anwendung ändern muss. + Die Anforderung eines Objekts wird verkapselt. Ein Objekt muss nicht unbedingt neu erzeugt werden, sondern kann auch wieder verwendet werden. Im Gegensatz dazu wird beim <code>new</code>-Operator immer ein neues Objekt erzeugt. + Wichtige Aufgaben, z.B. die Abfrage der Systemkonfigurationen, können zentral in der Klasse Factory oder einer ihrer Unterklassen erfolgen. Dort kann auch ein interner Zustand verwaltet und bei der Erzeugung berücksichtigt werden. - Die Architektur wird komplexer. - Es muss eine eigene Klasse geben, die die Fabrikmethode aufnimmt. - Fabrikmethoden sollten nicht in zeitkritischen Anwendungsteilen verwendet werden, da ein Effizienzverlust gegenüber einer direkten Objekterzeugung eintritt. | Vorteile |
| | Nachteile |

I 8 Architektur- und Entwurfsmuster

Zusammenhänge

Ähnliche Muster sind das Strategie-Muster (siehe »Das Strategie-Muster (*strategy pattern*)«, S. 96) und das Besucher-Muster (*visitor pattern*). Das abstrakte Fabrikmuster erweitert das Fabrikmethoden-Muster auf Familien. Fabrikmethoden werden in der Regel innerhalb von Schablonen-Methoden (*template pattern*) aufgerufen. Bei Fabrikmethoden erfolgt die Erzeugung durch Vererbung, bei Prototypen erfolgt die Erzeugung durch Delegation. Das Fabrikmethoden-Muster unterstützt die nichtfunktionale Anforderung Weiterentwickelbarkeit (siehe »Weiterentwickelbarkeit«, S. 119). Es beeinträchtigt die Effizienz (siehe »Leistung und Effizienz«, S. 128). Das Prinzip der Bindung und Kopplung wird unterstützt, ebenso das Prinzip der Verbalisierung und das Prinzip der Abstraktion (siehe »Architekturprinzipien«, S. 29). Das Architekturprinzip Sichtbarkeit wird beeinträchtigt (siehe »Architekturprinzip: Sichtbarkeit«, S. 34).

Literatur

[GHJ+95]

8.9 Das Strategie-Muster (*strategy pattern*)

Name(n)

Das Strategie-Muster (*strategy pattern*) ist ein objektbasiertes Verhaltensmuster (*behavioral pattern*) – auch als *policy* bekannt.

Grundidee

Wenn es für Aufgaben unterschiedliche Algorithmen gibt, z. B. laufzeitoptimierte und speicheroptimierte Algorithmen, dann ermöglicht es das Strategie-Muster über eine einheitliche Schnittstelle zur Laufzeit den Algorithmus zu wählen, der im aktuellen Kontext am besten geeignet ist.

Anwendungsbereich(e)

Das Strategie-Muster eignet sich für folgende Fälle:

- Es soll ein Algorithmus unabhängig von den nutzenden Klassen ausgetauscht werden.
- Eine zu erledigende Aufgabe kann durch verschiedene Algorithmen gelöst werden, die aber eine einheitliche Schnittstelle besitzen.
- Für eine zu erledigende Aufgabe werden verschiedene Algorithmenvarianten benötigt.

8.9 Das Strategie-Muster (*strategy pattern*) I

Beispiel(e)

Für eine Problemlösung stehen die beiden einfachen Sortieralgorithmen »Sortieren durch Auswahl« (*selection sort*) und »Sortieren durch Austauschen« (*bubble sort*) zur Verfügung. In der Klasse Client erfolgt die Auswahl durch eine switch-Anweisung. Es wird ein Objekt der gewählten Sortierstrategie erstellt und dessen Methode `sortiere()` aufgerufen.

Die Java-Programme sehen wie folgt aus:

Beispiel:
Sortieren

Java

```
public class Client
{
    public static enum
        SORTIERUNG {AUSTAUSCHEN,AUSWAHL};
    public static SORTIERUNG art;

    public static void main(String[] args)
    {
        int[] arr1 = {7,2,5,3,9,8,1,4,6};
        int[] arr2 = {7,2,5,3,9,8,1,4,6};

        art=SORTIERUNG.AUSTAUSCHEN;
        sortiere(arr1);
        for(int elem : arr1)
            System.out.println(""+elem);

        art=SORTIERUNG.AUSWAHL;
        sortiere(arr2);
        for(int elem : arr2)
            System.out.println(""+elem);
    }

    public static void sortiere(int[] arr)
    {
        switch(art)
        {
            case AUSTAUSCHEN:
                StrategieAustauschen bubble =
                    new StrategieAustauschen();
                bubble.sortiere(arr); break;
            case AUSWAHL:
                StrategieAuswahl auswahl =
                    new StrategieAuswahl();
                auswahl.sortiere(arr); break;
            default:
        }
    }
}

public class StrategieAustauschen
{
    public void sortiere(int[] arr)
    {
        System.out.println("Austauschen");
    }
}
```

I 8 Architektur- und Entwurfsmuster

```
for(int i=0;i<arr.length;i++)
{
    for(int j=arr.length-1;j>i;j--)
    {
        if(arr[j]<arr[j-1])
        {
            int tmp=arr[j];
            arr[j]=arr[j-1];
            arr[j-1]=tmp;
        }
    }
}

public class StrategieAuswahl
{
    public void sortiere(int[] arr)
    {
        System.out.println("Auswahl");
        int posMin, pos;
        for(int i=0;i<arr.length;i++)
        {
            posMin=i; pos=arr[i];
            for(int j=i+1;j<arr.length;j++)
            {
                if(arr[j]<pos)
                {
                    posMin=j; pos=arr[posMin];
                }
            }
            int tmp=arr[i];
            arr[i]=arr[posMin];
            arr[posMin]=tmp;
        }
    }
}
```

Unter Verwendung des Strategie-Musters ergibt sich folgende Lösung:

```
Java public class Client
{
    public static void main(String[] args)
    {
        Kontext ctx;
        int[] arr1 = {7,2,5,3,9,8,1,4,6};
        int[] arr2 = {7,2,5,3,9,8,1,4,6};

        ctx = new Kontext(new StrategieAustauschen());
        ctx.ausfuehrenSortierung(arr1);
        for(int elem : arr1)
            System.out.println(""+elem);

        ctx = new Kontext(new StrategieAuswahl());
        ctx.ausfuehrenSortierung(arr2);
```

8.9 Das Strategie-Muster (*strategy pattern*) I

```
for(int elem : arr2)
    System.out.println(""+elem);
}

public class Kontext
{
    private Strategie meineStrategie;

    public Kontext(Strategie eineStrategie)
    {
        meineStrategie = eineStrategie;
    }

    public void ausfuehrenSortierung(int[] arr)
    {
        meineStrategie.sortiere(arr);
    }
}

public interface Strategie
{
    void sortiere(int[] arr);
}

public class StrategieAustauschen implements Strategie
{
    @Override
    public void sortiere(int[] arr)
    {
        ...
    }
}

public class StrategieAuswahl implements Strategie
{
    @Override
    public void sortiere(int[] arr)
    {
        ...
    }
}
```

In der Klasse Client wird ein Objekt der Klasse Kontext deklariert und je nach gewünschter Sortierstrategie mit einem Objekt der Klasse StrategieAuswahl oder StrategieAustauschen als Parameter initialisiert. Durch den Aufruf der Methode ausfuehrenSortierung() der Klasse Kontext wird die Methode sortiere() der in diesem Kontextobjekt gespeicherten Sortierstrategie aufgerufen. Die Klassen StrategieAuswahl und StrategieAustauschen implementieren die Schnittstelle Strategie.

Die Unterschiede zwischen beiden Lösungen zeigt die Abb. 8.9-1.

I 8 Architektur- und Entwurfsmuster

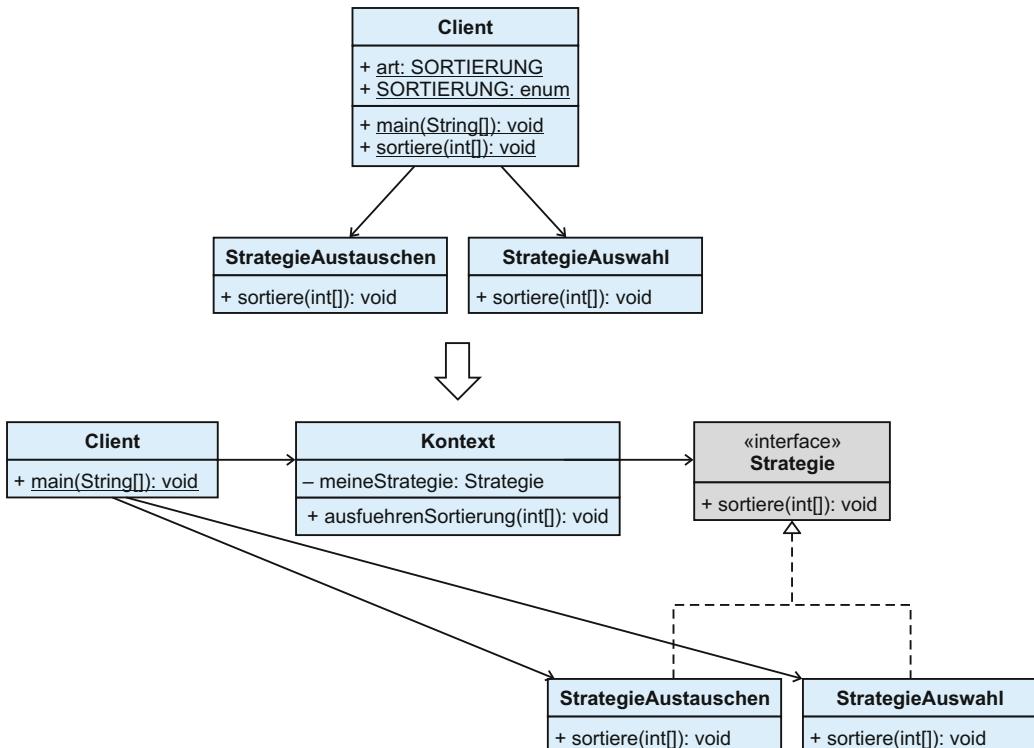


Abb. 8.9-1: Beispiel für die Transformation des algorithmus-abhängigen Verhaltens in separate Objekte mit polymorphen Schnittstellen.

Problem/Kontext

Eine Gruppe von Algorithmen, die dieselbe Schnittstelle besitzen und dieselbe Aufgabe lösen, soll zur Laufzeit ausgewählt und auch ausgetauscht werden können.

Lösung

Es wird für alle betroffenen Algorithmen eine einheitliche Schnittstelle definiert, die als **Strategie** bezeichnet wird. Jeder Algorithmus wird in Form einer Methode in einer Klasse implementiert, die wiederum die Schnittstelle **Strategie** implementiert – oft »konkrete Strategien« genannt. Die konkreten Strategien werden von einer Klasse **Kontext** aus verwendet. In dieser Klasse werden die Algorithmen jedoch nicht direkt, sondern über die Schnittstelle **Strategie** aufgerufen.

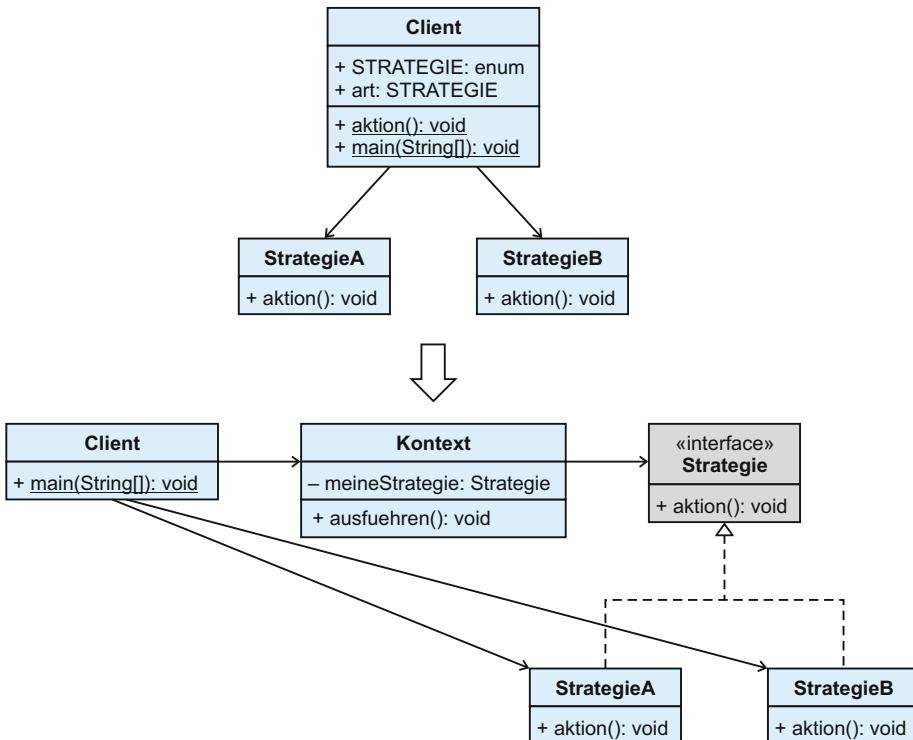
Die Abb. 8.9-2 zeigt die Struktur mit und ohne Anwendung des Strategie-Musters.

Code ohne Strategie-Muster:

```

public class Client
{
    public static enum STRATEGIE {A,B};
    public static STRATEGIE art;
  
```

8.9 Das Strategie-Muster (*strategy pattern*) I



```

public static void main(String[] args)
{
    art=STRATEGIE.A;
    aktion();

    art=STRATEGIE.B;
    aktion();
}

public static void aktion()
{
    switch(art)
    {
        case A:
            StrategieA a = new StrategieA();
            a.aktion(); break;
        case B:
            StrategieB b = new StrategieB();
            b.aktion(); break;
        default:
    }
}

public class StrategieA
{
```

*Abb. 8.9-2:
Transformation
des algorithmus-
abhängigen
Verhaltens in
separate Objekte
mit polymorphen
Schnittstellen.*

I 8 Architektur- und Entwurfsmuster

```
public void aktion()
{
    System.out.println("A");
}
}

public class StrategieB
{
    public void aktion()
    {
        System.out.println("B");
    }
}
```

Code mit Strategie-Muster:

```
public class Client
{
    public static void main(String[] args)
    {
        Kontext ctx;

        ctx = new Kontext(new StrategieA());
        ctx.ausfuehren();

        ctx = new Kontext(new StrategieB());
        ctx.ausfuehren();
    }
}

public class Kontext
{
    private Strategie meineStrategie;

    public Kontext(Strategie eineStrategie)
    {
        meineStrategie = eineStrategie;
    }

    public void ausfuehren()
    {
        meineStrategie.aktion();
    }
}

public interface Strategie
{
    void aktion();
}

public class StrategieA implements Strategie
{
    @Override
    public void aktion()
    {
        System.out.println("A");
    }
}
```

8.10 Das Brücken-Muster (*bridge pattern*) I

```
    }  
}  
  
public class StrategieB implements Strategie  
{  
    @Override  
    public void aktion()  
    {  
        System.out.println("B");  
    }  
}
```

Vor- und Nachteile

Der Einsatz des Strategie-Musters bringt folgende Vorteile mit sich: Vorteile

- + Es wird eine Komposition anstelle einer Vererbung verwendet (siehe »Schnittstellen, Fabriken und Komposition«, S. 503).
- + Die Wiederverwendung der einzelnen Strategien wird erleichtert.
- + Algorithmen können zur Laufzeit ausgetauscht werden.
- + Algorithmen, die dasselbe Problem lösen, werden zu Gruppen zusammengefasst.
- + switch-Anweisungen zur Auswahl von Algorithmen werden vermieden (siehe »Reengineering (Teil 3)«, S. 561).

Dem steht folgender Nachteil gegenüber:

- Es entstehen viele kleine Klassen.

Zusammenhänge

Das Strategie-Muster unterstützt die nichtfunktionalen Anforderungen Wartbarkeit (siehe »Wartbarkeit«, S. 116) und Weiterentwickelbarkeit (siehe »Weiterentwickelbarkeit«, S. 119). Es beeinträchtigt die Leistung (siehe »Leistung und Effizienz«, S. 128).

Das Prinzip der Bindung und Kopplung wird unterstützt, da die konkreten Strategieklassen nur jeweils einen Algorithmus enthalten (Stärkung der Bindung).

[GHJ+95, S. 315 ff.], [EiSt07, S. 47 f.], [DDN09, S. 295 ff.]

Literatur

8.10 Das Brücken-Muster (*bridge pattern*)

Name(n)

Das Brücken-Muster (*bridge pattern*) ist ein strukturelles Muster (*structural pattern*) – auch unter dem Namen *handle/body* bekannt.

Grundidee

Normalerweise stellt eine Klasse über ihre Methoden Dienstleistungen zur Verfügung, die in den Methoden oft auch direkt implementiert sind. Änderungen in der Methodensignatur erfordern in der Regel auch Änderungen an der Implementierung. Durch das Brücken-Muster werden die Schnittstellen der Dienstleistung, d. h. die Metho-

I 8 Architektur- und Entwurfsmuster

densignaturen, in eine eigenständige Klasse ausgelagert. Dadurch ist es auch möglich, abstraktere Schnittstellen anzubieten, die sich auf primitivere Methoden stützen. Von dieser eigenständigen Klasse wird dann auf die Klasse zugegriffen, die die Methoden implementieren. Dadurch wird eine Entkopplung zwischen der Schnittstelle und der Implementierung hergestellt. Implementierungen können unabhängig von der Schnittstelle geändert werden. Umgekehrt können Schnittstellen geändert werden, ohne die Implementierung zu tangieren.

Anwendungsbereich(e)

Das Brücken-Muster eignet sich für folgende Fälle:

- Die Implementierung einer Schnittstelle bzw. einer Abstraktion soll vollständig vor der benutzenden Klasse verborgen werden.
- Änderungen in der Implementierung einer Schnittstelle sollen keine Auswirkungen auf die benutzende Klasse haben, d. h. ihr Code soll nicht erneut übersetzt werden müssen.
- Sowohl die Schnittstelle bzw. die Abstraktion als auch ihre Implementierungen sollen durch Unterklassenbildung erweiterbar sein. Verschiedene Abstraktionen und Implementierungen sollen kombiniert und unabhängig voneinander erweitert werden können.
- Eine permanente Verbindung zwischen einer Abstraktion und ihrer Implementierung soll vermieden werden.

Beispiel(e)

Beispiel: Routenberechnung Es soll in simulierter Form eine Routenberechnung durchgeführt werden. Die Klasse ClientBruecke greift auf die Methode `ermittleRoute()` zu, um die Entfernung zwischen zwei Orten zu ermitteln. Die Klasse Route ermittelt die GPS-Koordinaten und delegiert die Berechnung an die Klasse RouteImpl:

```
Java public class ClientBruecke
{
    public static void main(String args[])
    {
        Route meineRoute = new Route();
        int km = meineRoute.ermittleRoute("DO", "M");
        System.out.println("Entfernung " + km);
    }
}
public class Route
{
    private RouteImpl link = null;
    public int ermittleRoute(String ortA, String ortB)
    {
        //Ermittle GPS-Koordinaten zu ortA
        int posA = 100;
        //Ermittle GPS-Koordinaten zu ortB
        int posB = 200;
```

8.10 Das Brücken-Muster (*bridge pattern*) I

```
link = new RouteImpl();
return link.gibEntfernung(posA, posB);
}
public class RouteImpl
{
    public int gibEntfernung (int posX, int posY)
    {
        return posY - posX;
    }
}
```

Für die Entfernungs berechnung gibt es in Zukunft zwei Möglichkeiten. Daher wird die Klasse `RouteImpl` in eine Schnittstelle `RouteI` umgewandelt, die die zwei Berechnungsklassen `RouteImpl1` und `RouteImpl2` implementieren. Außerdem gibt es Routen für Fußgänger und Autofahrer. Daher wird die Klasse `Route` in eine abstrakte Klasse mit einer abstrakten Methode verwandelt, die von den Klassen `RouteFussgaenger` und `RouteAutofahrer` implementiert wird. Es ergeben sich dadurch folgende Programme:

```
public class ClientBruecke
{
    public static void main(String args[])
    {
        Route meineRoute = new RouteFussgaenger();
        int km = meineRoute.ermitteRoute("D0", "M");
        System.out.println("Entfernung Fussgänger: " + km);
        meineRoute = new RouteAutofahrer();
        km = meineRoute.ermitteRoute("D0", "M");
        System.out.println("Entfernung Autofahrer: " + km);
    }
}
public abstract class Route
{
    protected RouteI link = null;
    public abstract int ermitteRoute
        (String ortA, String ortB);
}
public class RouteAutofahrer extends Route
{
    public int ermitteRoute(String ortA, String ortB)
    {
        //Ermittle GPS-Koordinaten zu ortA
        int posA = 100;
        //Ermittle GPS-Koordinaten zu ortB
        int posB = 200;
        link = new RouteImpl1();
        return link.gibEntfernung(posA, posB);
    }
}
public class RouteFussgaenger extends Route
{
    public int ermitteRoute(String ortA, String ortB)
```

I 8 Architektur- und Entwurfsmuster

```

    {
        //Ermittle GPS-Koordinaten zu ortA
        int posA = 100;
        //Ermittle GPS-Koordinaten zu ortB
        int posB = 200;
        link = new RouteImpl2();
        return link.gibEntfernung(posA, posB);
    }
}
public interface RouteI //Abstraktion
{
    public int gibEntfernung (int posX, int posY);
}
public class RouteImpl1 implements RouteI
{
    public int gibEntfernung (int posX, int posY)
    {
        return (posY - posX) * 2;
    }
}
public class RouteImpl2 implements RouteI
{
    public int gibEntfernung (int posX, int posY)
    {
        return posY - posX;
    }
}

```

Die Abb. 8.10-1 zeigt das zugehörige UML-Klassendiagramm.

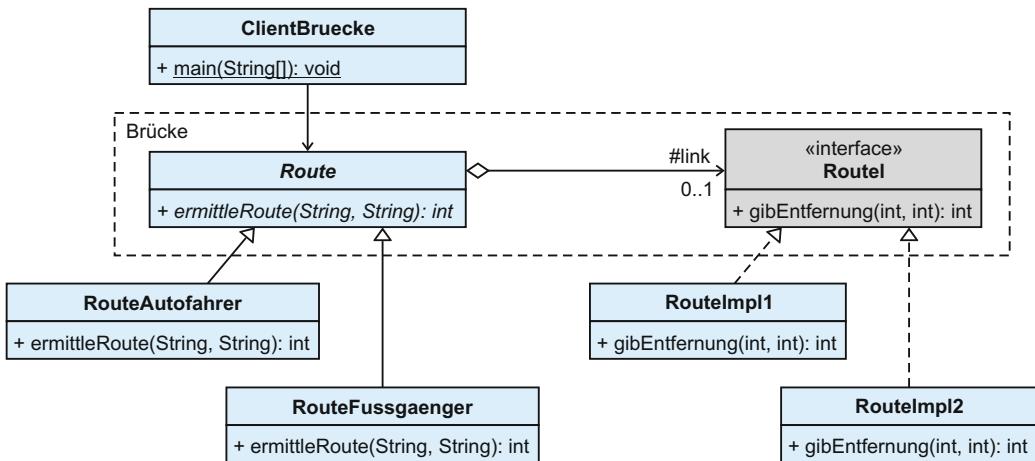


Abb. 8.10-1:
Beispiel für das
Brücken-Muster. Abstraktionen und zugehörige Implementierungen sollen getrennt werden, sodass sie unabhängig voneinander erweitert und modifiziert werden können.

8.10 Das Brücken-Muster (*bridge pattern*) I

Lösung

Die Abstraktionen und ihre Implementierungen werden in zwei unterschiedlichen Klassenhierarchien verwaltet. Dadurch werden die verschiedenen Abstraktionen von ihren verschiedenen spezifischen Implementierungen entkoppelt. Beide Hierarchien können unabhängig voneinander variiert werden. Die Abb. 8.10-2 zeigt das allgemeine Muster.

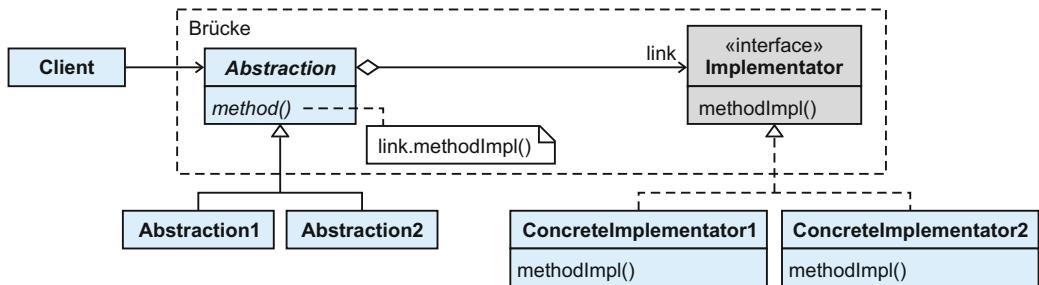


Abb. 8.10-2: Das Brücken-Muster.

Vor- und Nachteile

Das Brücken-Muster besitzt folgende Vorteile:

- + Zur Laufzeit kann die Implementierung einer Abstraktion konfiguriert oder gar ausgetauscht werden.
- + Die Client-Klasse und die Abstraktionsklasse müssen *nicht* neu übersetzt werden, wenn sich die Implementierungsklasse ändert.
- + Vor den benutzenden Klassen können die jeweiligen Implementationsdetails verborgen werden.
- + Änderungen verschiedener Aspekte einer Architektur können unabhängig voneinander vorgenommen werden.

Dem steht folgender Nachteil gegenüber:

- Höherer Entwicklungsaufwand.

Zusammenhänge

Das Brücken-Muster unterstützt die nichtfunktionalen Anforderungen Wartbarkeit (siehe »Wartbarkeit«, S. 116) und Weiterentwickelbarkeit (siehe »Weiterentwickelbarkeit«, S. 119).

Das Prinzip der Bindung und Kopplung wird unterstützt, da die Schnittstelle bzw. die Abstraktion von ihrer Implementierung entkoppelt wird. Die Implementierung muss nur temporär an eine bestimmte Schnittstelle gebunden werden.

[GHJ+95, S. 151 ff.], [EiSt07, S. 56 ff.]

Literatur

9 Nichtfunktionale Anforderungen

Ziel jeder Softwareentwicklung ist es, die Anforderungen des Auftraggebers bzw. des Marktes an das Softwareprodukt zu erfüllen. Es werden funktionale und nichtfunktionale Anforderungen unterschieden.

Funktionale Anforderungen (*functional requirements*, kurz FRs) spezifizieren, welche Funktionalität oder welches Verhalten das Softwareprodukt unter festgelegten Bedingungen besitzen bzw. erfüllen soll.

Nichtfunktionale Anforderungen (*nonfunctional requirements*, kurz NFRs), auch Technische Anforderungen genannt, beschreiben Aspekte, die typischerweise mehrere oder alle funktionalen Anforderungen betreffen bzw. überschneiden (*cross-cut*). Sie haben in der Regel einen Einfluss auf die gesamte Softwarearchitektur. Außerdem beeinflussen sich nichtfunktionale Anforderungen gegenseitig.

Sicherheit (*security*) steht häufig im Konflikt mit Benutzbarkeit, Beispiel Speichereffizienz ist meist gegenläufig zur Laufzeiteffizienz.

Für manche nichtfunktionale Anforderungen kann auch eine Dienstgüte – *Quality of Service* (QoS) genannt – festgelegt werden, z.B. bezogen auf Leistung und Effizienz.

Nichtfunktionale Anforderungen können sich auf verschiedene Bausteinarten beziehen.

Die Effizienz des Softwaresystems ergibt sich aus dem komplexen Zusammenspiel verschiedener Komponenten. Die Zuverlässigkeit und teilweise auch die Änderbarkeit werden durch die einzelnen Komponenten bestimmt.

Vereinfacht ausgedrückt, legen funktionale Anforderungen fest, was das Softwareprodukt tun soll, während die nichtfunktionalen Anforderungen spezifizieren, wie es arbeiten soll.

Die Abgrenzung zwischen funktionalen und nichtfunktionalen Anforderungen ist oft nicht einfach. Beispielsweise können Zeitanforderungen als Verhalten des Softwareproduktes betrachtet oder auch als nichtfunktionale Anforderung aufgefasst werden.

Oft hängt die Unterscheidung davon ab, wie die Anforderung formuliert ist [Glin07, S. 23]:

I 9 Nichtfunktionale Anforderungen

»Das System soll einen unautorisierten Zugriff auf die Kundendaten verhindern.« Diese Formulierung spricht für eine nichtfunktionale Anforderung.

»Die Datenbank soll den Zugriff auf die Kundendaten nur den Benutzern gestatten, die sich durch den Benutzernamen und das Passwort autorisiert haben.« Diese Formulierung spricht für eine funktionale Anforderung.

In [Glin07] wird daher vorgeschlagen, *nicht* zwischen funktionalen und nichtfunktionalen Anforderungen zu unterscheiden, sondern nach folgenden vier Aspekten zu spezifizieren:

- Beschreibungsart: Operational, qualitativ, quantitativ, deklarativ
- Art der Anforderung: Funktionen, Daten, Leistung, spezifische Qualität, Randbedingung
- Erfüllungsgrad: »Harte« oder »weiche« Anforderungen
- Rolle der Anforderung: präskriptiv, normativ, *assumptive*

ISO/IEC 9126 In dem häufig zitierten internationalen Standard ISO/IEC 9126 werden folgende Qualitätscharakteristika mit ihren Teilmerkmalen spezifiziert (ohne Rangfolge):

- **Funktionalität** (*functionality*)
 - Angemessenheit (*suitability*)
 - Genauigkeit (*accuracy*)
 - Interoperabilität (*interoperability*)
 - Sicherheit (*security*)
 - Konformität der Funktionalität (*functionality compliance*)
- **Zuverlässigkeit** (*reliability*)
 - Reife (*maturity*)
 - Fehlertoleranz (*fault tolerance*)
 - Wiederherstellbarkeit (*recoverability*)
 - Konformität der Zuverlässigkeit (*reliability compliance*)
- **Benutzbarkeit, Gebrauchstauglichkeit** (*usability*)
 - Verständlichkeit (*understandability*)
 - Erlernbarkeit (*learnability*)
 - Bedienbarkeit (*operability*)
 - Attraktivität (*attractiveness*)
 - Konformität der Benutzbarkeit (*usability compliance*)
- **Effizienz** (*efficiency*)
 - Zeitverhalten (*time behaviour*)
 - Verbrauchsverhalten (*resource utilisation*)
 - Konformität der Effizienz (*efficiency compliance*)
- **Wartbarkeit** (*Maintainability*)
 - Analysierbarkeit (*analyzability*)
 - Änderbarkeit (*changeability*)
 - Stabilität (*stability*)
 - Testbarkeit (*testability*)
 - Konformität der Wartbarkeit (*Maintainability compliance*)

■ Portabilität (*portability*)

- Anpassbarkeit (*adaptability*)
- Installierbarkeit (*installability*)
- Koexistenz (*co-existence*)
- Austauschbarkeit (*replaceability*)
- Konformität der Portabilität (*portability compliance*)

Obwohl seit über 30 Jahren nichtfunktionale Anforderungen postuliert werden, gibt es bis heute keinen Konsens über die wichtigsten nichtfunktionalen Anforderungen, ihre Definition und ihre Untergliederung. In [MZN10] wurden 182 Quellen der letzten 30 Jahre zu nichtfunktionalen Anforderungen ausgewertet. Es wurden 252 nichtfunktionale Anforderungen identifiziert, davon 114 mit einem Bezug zu Qualität. Die am meisten genannten nichtfunktionalen Anforderungen sind (Top 5):

1 Leistung (*performance*) (89 %)

Top 5

2 Zuverlässigkeit (*reliability*) (68 %)

3 Benutzbarkeit, Gebrauchstauglichkeit (*usability*) (62 %)

4 Sicherheit (*security*) (60 %)

5 Wartbarkeit (*maintainability*) (55 %)

Auch die Mehrbenutzerfähigkeit oder die Verteilbarkeit einer Anwendung auf mehrere, miteinander vernetzte Computersysteme könnte man als nichtfunktionale Anforderungen ansehen. Da diese Anforderungen in der Literatur aber *nicht* unter nichtfunktionalen Anforderungen aufgeführt werden, werden sie hier unter Einflussfaktoren beschrieben (siehe »Einflussfaktoren auf die Architektur«, S. 135).

In [MZN10, S. 315] werden die fünf Systemtypen Echtzeitsysteme, sicherheitskritische Systeme, Websysteme, Informationssysteme und Prozesssteuerungssysteme mit ihren relevanten nichtfunktionalen Anforderungen identifiziert (Abb. 9.0-1). In allen Systemtypen werden Leistung, Sicherheit und Benutzbarkeit als relevant angesehen.

NFRs & Systemtypen

Die Zuordnung von relevanten nichtfunktionalen Anforderungen zu Anwendungsdomänen zeigt die Tab. 9.0-1 [MZN10, S. 316].

Anhand der Abb. 9.0-1 und der Tab. 9.0-1 können Sie feststellen, welche nichtfunktionalen Anforderungen für Ihren Systemtyp und Ihre Anwendungsdomäne relevant sind.

Hinweis

Nichtfunktionale Anforderungen lassen sich nach verschiedenen Kriterien klassifizieren.

In [Glin07, S. 25] erfolgt eine Klassifizierung in Leistungs-Anforderungen, spezifische Qualitäts-Anforderungen und Randbedingungen (*constraints*):

- Leistungs-Anforderungen: Zeit- und Speicherbegrenzungen wie Klassifizierung 1
Zeit, Geschwindigkeit, Volumen, Durchsatz

I 9 Nichtfunktionale Anforderungen

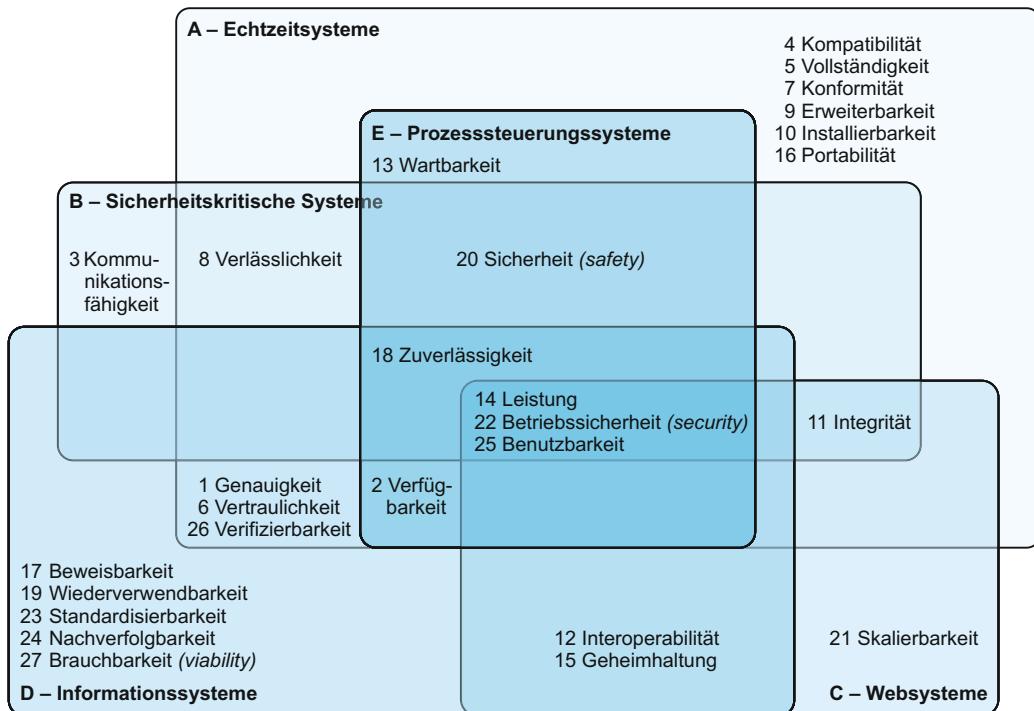


Abb. 9.0-1: Zuordnung von nichtfunktionalen Anforderungen zu Systemtypen.

Klassifizierung 2

- Qualitäts-Anforderungen: »-keiten« wie Zuverlässigkeit, Benutzbarkeit, Sicherheit, Wartbarkeit, Verfügbarkeit, Portabilität
- Randbedingungen: Physik, Recht, Kultur, Umgebung, Entwurfs- und Implementierungsschnittstellen usw.

In [SoAr] wird folgende Klassifizierung vorgenommen:

- Laufzeit-Anforderungen: Dazu zählen alle Anforderungen, die zur Laufzeit gemessen werden können wie Funktionalität, Leistungen, Sicherheit, Benutzbarkeit, Verfügbarkeit, Interoperabilität.
- Nicht-Laufzeit-Anforderungen: Dazu zählen alle Anforderungen, die nicht zur Laufzeit gemessen werden können wie Modifizierbarkeit, Portabilität, Wiederverwendbarkeit, Integrierbarkeit, Testbarkeit.
- Architektur-Anforderungen: Anforderungen, die spezifisch für die Architektur selbst sind wie konzeptionelle Integrität, Korrektheit.
- Domänenpezifische Anforderungen: Anforderungen für spezifische Domänen wie Sensitivität, Kalibrierbarkeit.

In [MaBr01] werden zwei Klassen unterschieden:

- Klassifizierung 3
- Laufzeit-Anforderungen: Benutzbarkeit, Leistung, Sicherheit, Konfigurierbarkeit, Korrektheit, Zuverlässigkeit, Verfügbarkeit, Skalierbarkeit.

9 Nichtfunktionale Anforderungen I

| Anwendungsdomäne | relevante NFRs |
|------------------------------|---|
| Banken und Finanzen | Genauigkeit, Geheimhaltung, Leistung, Sicherheit, Benutzbarkeit |
| Ausbildung | Interoperabilität, Leistung, Zuverlässigkeit, Skalierbarkeit, Sicherheit, Benutzbarkeit |
| Energiressourcen | Verfügbarkeit, Leistung, Zuverlässigkeit, Betriebssicherheit, Benutzbarkeit |
| Regierung und Militär | Genauigkeit, Vertraulichkeit, Leistung, Geheimhaltung, Beweisbarkeit, Wiederverwendbarkeit, Sicherheit, Standardisierbarkeit, Benutzbarkeit, Verifizierbarkeit, Brauchbarkeit |
| Versicherungen | Genauigkeit, Vertraulichkeit, Integrität, Interoperabilität, Sicherheit, Benutzbarkeit |
| Medizin, Gesundheitswesen | Kommunikationsfähigkeit, Vertraulichkeit, Integrität, Leistung, Geheimhaltung, Zuverlässigkeit, Sicherheit, Betriebssicherheit, Nachverfolgbarkeit, Benutzbarkeit |
| Telekommunikation | Kompatibilität, Konformität, Verlässlichkeit, Installierbarkeit, Wartbarkeit, Leistung, Portabilität, Zuverlässigkeit, Benutzbarkeit |
| Verkehrswesen | Genauigkeit, Verfügbarkeit, Kompatibilität, Vollständigkeit, Vertraulichkeit, Verlässlichkeit, Integrität, Leistung, Sicherheit, Betriebssicherheit, Verifizierbarkeit |

■ Entwicklungszeit-Anforderungen: Weiterentwickelbarkeit, Wiederverwendbarkeit, Modifizierbarkeit, Lokalisierbarkeit, Zusammensetzbarkeit.

Laufzeit-Anforderungen sind wertvoll für den Benutzer und unterstützen die kurzfristige Wettbewerbsfähigkeit. Entwicklungszeit-Anforderungen erhöhen in der Regel den Wert des Softwareprodukts und sorgen für die langfristige Wettbewerbsfähigkeit.

Eine weitere Unterscheidungsmöglichkeit ist eine Klassifizierung in quantitative und qualitative nichtfunktionale Anforderungen. Zu den quantitativen, d.h. messbaren Anforderungen gehören z.B. Laufzeit-Anforderungen. Zu den qualitativen Anforderungen gehören z.B. die Benutzbarkeit und die Wartbarkeit, die nicht oder nur sehr aufwändig zu messen sind (siehe hierzu auch [Glin08]).

Bezogen auf den Softwarelebenszyklus lassen sich nichtfunktionale Anforderungen nach Ansicht des Autors auch wie folgt klassifizieren:

- Relevant während der Entwicklungszeit:
 - Funktionalität
 - Benutzbarkeit
 - Sicherheit
- Relevant während der Installation:
 - Funktionalität – Interoperabilität
 - Portabilität – Anpassbarkeit
 - Portabilität – Instalierbarkeit
 - Portabilität-Koexistenz

Tab. 9.0-1: NFRs & Anwendungsdomänen.

Klassifizierung 4

Klassifizierung 5

I 9 Nichtfunktionale Anforderungen

- Relevant während des Betriebs:
- Leistung/Effizienz
- Zuverlässigkeit
- Benutzbarkeit
- Sicherheit
- Wartbarkeit
- Weiterentwickelbarkeit

Relevant während der Entwicklungszeit bedeutet, dass ein System nicht abgenommen wird, wenn die angegebenen Anforderungen nicht erfüllt sind. Beispielsweise kann geprüft werden, ob die geforderte Funktionalität, Benutzbarkeit und Sicherheit vorhanden sind.

Relevant während der Installation bedeutet, dass die nichtfunktionalen Anforderungen während der Installation sichtbar werden.

Relevant während des Betriebs bedeutet, dass diese nichtfunktionalen Anforderungen bei der Abnahme des Systems in der Regel *nicht* getestet werden können, sondern dass der Grad ihrer Erfüllung oft erst während des Betriebs überprüft werden kann.

trade-offs

Nichtfunktionale Anforderungen können sich gegenseitig verstärken, es kann aber auch zu Zielkonflikten (*trade-offs*) kommen. Zielkonflikte können aber auch zwischen funktionalen und nichtfunktionalen Anforderungen auftreten. Die Tab. 9.0-2 zeigt, bezogen auf die wichtigsten Anforderungen, die gegenseitigen Abhängigkeiten (siehe auch [EgGr04], [Wieg03]).

| ↓Anforderung / Effekt→ | Funktio- | Leistung / | Zuverläs- | Benutz- | Sicherheit | Wartbar- |
|------------------------|--------------|----------------|---------------|--------------|------------|-----------|
| | nalität + | Effizienz + | sigkeits + | barkeit + | + | keit + |
| Funktionalität | o | – | – | +– | – | – |
| Leistung / Effizienz | – | o | – | +– | – | – |
| Zuverlässigkeit | – | – | o | + | + | + |
| Benutzbarkeit | +– | +– | + | o | – | o |
| Sicherheit | – | – | + | – | o | + |
| Wartbarkeit | – | – | + | o | o | o |

Tab. 9.0-2: Die Tab. 9.0-2 ist wie folgt zu lesen: In den Zeilen stehen die Anforderungs-Attribute. Die Spalten repräsentieren die potenziellen Effekte auf die anderen Anforderungs-Attribute.

Fügt eine Anforderung eine zusätzliche Funktionalität hinzu, dann zeigt die erste Zeile, dass dies einen negativen Einfluss auf die Effizienz (–), Zuverlässigkeit (–), die Sicherheit (–) und die Wartbarkeit (–) hat. Ein positiver Aspekt ergibt sich für die Benutzbarkeit (+).

9 Nichtfunktionale Anforderungen I

Ein umgekehrter Effekt entsteht, wenn eine Anforderung entfällt oder der Grad einer Anforderung reduziert wird. Wird beispielsweise die Sicherheit reduziert (Zeile 5), dann erhöht sich die Leistung/Effizienz.

Oft ist es erforderlich, Anforderungs-Attribute weiter zu untergliedern. Beispielsweise kann das Anforderungs-Attribut Leistung/Effizienz unterteilt werden in Laufzeit-Effizienz und Speicher-Effizienz.

Die Tab. 9.0-2 gibt Anhaltspunkte für mögliche positive oder negative gegenteilige Effekte. Die Abhängigkeiten müssen mit den konkreten Anforderungen überprüft werden. Es kann durchaus sein, dass zunächst offensichtliche Konflikte bei näherem Betrachten nicht bestehen, da sich die Anforderungen auf verschiedene Subsysteme beziehen.

Zunächst wird die »Wartbarkeit« näher betrachtet, da sie die Softwarearchitektur stark beeinflusst:

- »Wartbarkeit«, S. 116

Für langlebige Softwaresysteme wird die »Weiterentwickelbarkeit« (*evolvability*) immer wichtiger. Sie wird daher nicht untergeordnet unter der Wartbarkeit behandelt, sondern separat:

- »Weiterentwickelbarkeit«, S. 119

Geforderte Sicherheitsaspekte haben ebenfalls einen wesentlichen Einfluss auf die Softwarearchitektur:

- »Betriebssicherheit und Funktionssicherheit«, S. 121

Die Sicherstellung der Zuverlässigkeit eines Softwaresystems erfordert besondere Maßnahmen:

- »Zuverlässigkeit«, S. 124

Da die nichtfunktionalen Anforderungen »Leistung« und »Effizienz« oft synonym oder gegenseitig untergeordnet klassifiziert werden, werden sie im Folgenden gemeinsam behandelt:

- »Leistung und Effizienz«, S. 128

Die Gebrauchstauglichkeit und Benutzbarkeit kann durch verschiedene architektonische Maßnahmen gefördert werden:

- »Benutzbarkeit«, S. 130

Für die Installierbarkeit und die Weiterentwickelbarkeit eines Softwareprodukts spielt die Portabilität eine wichtige Rolle. Sie wird daher als eigenständige, nichtfunktionale Anforderung behandelt:

- »Portabilität«, S. 132

[MZN10], [Glin08], [Glin07], [SoAr], [MaBr01], [EgGr04], [Wieg03]

Verfeinerung

Problem



Literatur

I 9 Nichtfunktionale Anforderungen

9.1 Wartbarkeit

Software ist auch nach der Inbetriebnahme ständigen Veränderungen unterworfen. Diese Veränderungen können sich durch die Umwelt des Softwaresystems ergeben, aber auch durch neue oder veränderte Anforderungen. Ziel beim Entwurf der Architektur muss es daher sein, Veränderungen auch in der Betriebsphase vorzunehmen, sodass die Einsatzbereitschaft des Softwaresystems gewährleistet bleibt. Änderungen sollen mit geringem Aufwand und in kurzer Zeit durchgeführt werden können. Wartungsaktivitäten umfassen meist kleine Änderungen und Fehlerbehebungen, weil Zeit- und Kostenanforderungen eine große Rolle spielen. Ziel der Wartbarkeit ist die Lebenserhaltung des Systems, *nicht* die Weiterentwickelbarkeit (siehe »Weiterentwickelbarkeit«, S. 119). Ein System, das *nicht* weiterentwickelt werden kann, wird zu einem Altsystem – in der Softwaretechnik *Legacy-System* genannt.

Die nichtfunktionale Anforderung **Wartbarkeit** (*Maintainability*) wird in der Norm ISO/IEC 9126-1 als Qualitätsmerkmal eingeordnet und wie folgt definiert:

Definition

Fähigkeit des Softwareprodukts änderungsfähig zu sein. Änderungen können Korrekturen, Verbesserungen oder Anpassungen der Software an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen einschließen.

Wartbarkeit wird untergliedert in folgende Teilmerkmale:

- **Analysierbarkeit** (*Analyzability*)
Fähigkeit des Softwareprodukts, Mängel oder Ursachen von Versagen zu diagnostizieren oder änderungsbedürftige Teile zu identifizieren.
- **Änderbarkeit** (*Changeability*)
Fähigkeit des Softwareprodukts, die Implementierung einer spezifizierten Änderung zu ermöglichen.
- **Stabilität** (*stability*)
Fähigkeit des Softwareprodukts, unerwartete Wirkungen von Änderungen der Software zu vermeiden.
- **Testbarkeit** (*testability*)
Fähigkeit des Softwareprodukts, die modifizierte Software zu validieren.

Zu dem Teilmerkmal Analysierbarkeit gehört auch die Verständlichkeit und die Rückverfolgbarkeit (*traceability*).

In [MZN10, S. 315] wird Wartbarkeit wie folgt definiert:

Anforderungen, die die Fähigkeit eines Softwareproduktes beschreiben, modifiziert zu werden. Modifizieren beinhaltet die Korrektur von Defekten oder die Vornahme von Verbesserungen oder Änderungen der Software.

Definition

Zu den Teilmerkmalen nach ISO/IEC 9126–1 kommen noch folgende Teilmerkmale hinzu:

- **Verständlichkeit** (*understandability*)
- **Modifizierbarkeit** (*modifiability*)

Die Wartbarkeit ist entscheidend in der Betriebsphase eines Systems.

Während der Betriebsphase durchgeführte Änderungen führen in der Regel zu einer Verschlechterung der Struktur – auch als Architekturerosion (*architectural decay, architectural drift*) bezeichnet. Dadurch werden weitere Änderungen erschwert oder verhindert. Durch *Reengineering* kann versucht werden, die Wartbarkeit zu verbessern.

Problem

| ↓Anforderung / Effekt→ | Funktionalität | Leistung / Effizienz | Zuverlässigkeit | Benutzbarkeit | Sicherheit + | Wartbarkeit + |
|------------------------|----------------|----------------------|-----------------|---------------|--------------|---------------|
| Wartbarkeit | – | – | + | o | o | o |

Wie die Tab. 9.1-1 zeigt, führt eine zunehmende Funktionalität und eine Steigerung der Leistung/Effizienz jeweils zu einer schlechteren Wartbarkeit. Wird die Zuverlässigkeit verbessert, dann verbessert sich auch die Wartbarkeit. Der Zusammenhang zwischen der Benutzbarkeit und der Wartbarkeit ist nicht eindeutig. Führt eine verbesserte Benutzbarkeit zu einem umfangreicheren Code, dann kann dies die Wartbarkeit verschlechtern. Reduziert sich dagegen durch eine verbesserte Benutzbarkeit der Quellcode, dann kann dies die Wartbarkeit verbessern. Analog gilt dies für den Gesichtspunkt der Sicherheit.

Tab. 9.1-1:
Gegenseitige
Abhängigkeiten.

Die Wartbarkeit steht außerdem im Zusammenhang mit der nicht-funktionalen Anforderung **Weiterentwickelbarkeit** (siehe »Weiterentwickelbarkeit«, S. 119). Während die Wartbarkeit sich auf eine überschaubare Lebenszeit eines Softwaresystems bezieht, berücksichtigt die Weiterentwickelbarkeit auch sehr lange Lebenszeiten eines Softwaresystems.

Was ist zu tun?

Die Wartbarkeit kann durch die Einhaltung von folgenden Regeln verbessert werden:

Regeln

- Identifizieren Sie die Aspekte in Ihrem Entwurf, die sich ändern können, und trennen Sie diese Aspekte von denen, die gleich bleiben.

I 9 Nichtfunktionale Anforderungen

- Programmieren Sie gegen Abstraktionen und *nicht* gegen konkrete Klassen. Dadurch wird die Abhängigkeit von konkreten Klassen reduziert. Man spricht vom Prinzip der Abhängigkeitsumkehr (*dependency inversion principle*).
- Programmieren Sie gegen Schnittstellen (*interfaces*). Durch den Polymorphismus funktioniert der Code auch mit jeder Klasse, die die entsprechende Schnittstelle implementiert (siehe »Schnittstellen, Fabriken und Komposition«, S. 503).

Prinzipien Die Einhaltung folgender Prinzipien (siehe »Architekturprinzipien«, S. 29) unterstützen die Wartbarkeit:

- **Prinzip der Strukturierung:** Dadurch bessere Verständlichkeit und Analysierbarkeit.
- **Prinzip der Bindung und Kopplung:** Dadurch Begrenzung der Auswirkungen von Änderungen, d. h. bessere Stabilität.
- **Prinzip der Modularisierung:** Dadurch wird eine weitgehende Kontextunabhängigkeit von der Umgebung erreicht. Die Testbarkeit wird verbessert, da Module für sich testbar sind.
- **Geheimnisprinzip:** Änderungen an den Datenstrukturen wirken sich *nicht* auf die Zugriffsmethoden aus. Dadurch wird eine lokale Änderbarkeit ermöglicht.
- **Prinzip der Lokalität:** Da sich alle benötigten Informationen lokal an einer Stelle befinden, wird die Verständlichkeit und Analysierbarkeit wesentlich verbessert.
- **Prinzip der Verbalisierung:** Dadurch bessere Verständlichkeit und Analysierbarkeit.
- **Architekturprinzip: Konzeptionelle Integrität:** Dadurch schnellere Einarbeitung (siehe »Architekturprinzip: Konzeptionelle Integrität«, S. 30).
- **Architekturprinzip: Trennung von Zuständigkeiten:** Dadurch schnellere Einarbeitung und leichtere Analysierbarkeit (siehe »Architekturprinzip: Trennung von Zuständigkeiten«, S. 31).
- **Architekturprinzip: Sichtbarkeit:** Dadurch bessere Verständlichkeit und Analysierbarkeit (siehe »Architekturprinzip: Sichtbarkeit«, S. 34).

Architektur- & Entwurfs-Muster Folgende Architektur- und Entwurfs-Muster unterstützen z. B. die Wartbarkeit:

- »Das Schichten-Muster (*layers pattern*)«, S. 46: Zur Entkopplung der Subsysteme.
- »Das Beobachter-Muster (*observer pattern*)«, S. 54: Zur Entkopplung des Subsystems Benutzungsoberfläche von dem Subsystem Applikation.
- »Das MVC-Muster (*model view controller pattern*)«, S. 62: Zur Entkopplung der Subsysteme.
- »Das Proxy-Muster (*proxy pattern*)«, S. 83: Zum transparenten Zugriff auf entfernte Subsysteme (Remote-Proxy).

9.2 Weiterentwickelbarkeit

Während man früher dachte, Softwaresysteme leben im Einsatz 10 bis 20 Jahre, muss man heute feststellen, dass viele eingesetzte Softwaresysteme wesentlich länger in Betrieb sind. Wenn man heute davon ausgeht, dass neu entwickelte Systeme auch in Zukunft wesentlich länger eingesetzt werden, dann muss bereits bei der Softwarearchitektur dafür gesorgt werden, dass diese Systeme auch langfristig nutzbar sind. Das bedeutet, dass strukturelle Änderungen und die Erhaltung der Architekturintegrität berücksichtigt werden müssen. Folgender Begriff hat sich für diese Anforderungen gebildet:

Die nichtfunktionale Anforderung **Weiterentwickelbarkeit** (*evolvability*) – besser **Nachhaltigkeit** genannt – bezeichnet die Fähigkeit eines Softwareprodukts sich langfristig an geänderte Anforderungen und Techniken, die einen Einfluss auf die Architektur und/oder die funktionalen Erweiterungen haben, anzupassen, ohne die architektonische Integrität zu verletzen.

Definition

Die Weiterentwickelbarkeit lässt sich in folgende Merkmale untergliedern [RiBo09, S. 341]:

- Analysierbarkeit
- Änderbarkeit
- Testbarkeit
- Portabilität
- Erweiterbarkeit
- Integrität der Architektur

Weiterentwickelbarkeit bedeutet also, die Wartbarkeit langfristig zu erhalten. Nur so kann vermieden werden, dass ein Softwaresystem zu einem Altsystem (*legacy system*) wird – oft architektonischer Verfall genannt.

Weiterentwickelbarkeit erlaubt zusätzlich zur Wartung auch Modifikationen struktureller Art. Neue Techniken und Plattformen sowie neue nichtfunktionale Anforderungen, wie die Steigerung von Skalierbarkeit und die Einführung von Mehrsprachigkeit, werden ermöglicht.

| ↓Anforderung / Effekt→ | Funktionalität | Leistung / Effizienz | Zuverlässigkeit | Benutzbarkeit | Sicherheit | Wartbarkeit |
|------------------------|----------------|----------------------|-----------------|---------------|------------|-------------|
| Weiterentwickelbarkeit | – | – | o | o | o | o |

Tab. 9.2-1:
Gegenseitige
Abhängigkeiten.

I 9 Nichtfunktionale Anforderungen

Die Weiterentwickelbarkeit steht im Zusammenhang mit der Wartbarkeit (siehe »Wartbarkeit«, S. 116). Eine Erhöhung der Weiterentwickelbarkeit verbessert auch immer die Wartbarkeit, was umgekehrt aber *nicht* gilt.

Zusätzliche Funktionalität und eine Erhöhung der Leistung/Effizienz wirken sich negativ auf die Weiterentwickelbarkeit aus (Tab. 9.2-1). Eine verbesserte Zuverlässigkeit, Benutzbarkeit und Sicherheit kann sich positiv oder negativ auf die Weiterentwickelbarkeit auswirken.

Frage Überlegen Sie sich Beispiele, die den Unterschied zwischen Weiterentwickelbarkeit und Wartbarkeit zeigen.

Antwort Wird in einem Softwaresystem eine Plugin-Schnittstelle von der Architektur her vorgesehen, dann ist der Austausch von Komponenten im Betrieb statt durch Weiterentwicklung durch Änderungen möglich.

Was ist zu tun?

Die Weiterentwickelbarkeit kann durch die Einhaltung von folgenden Regeln verbessert werden:

- Eine gute Weiterentwickelbarkeit kann u. a. durch Komponentenmodelle erreicht werden.
- Wichtig sind ebenfalls die Rückverfolgbarkeit (*traceability*) von Entwurfsentscheidungen sowie die explizite Darstellung von Abhängigkeiten.

Prinzipien Alle Prinzipien, die helfen, Komplexität zu beherrschen, unterstützen die Weiterentwickelbarkeit (siehe »Architekturprinzipien«, S. 29):

- **Prinzip der Strukturierung:** Dadurch bessere Verständlichkeit und Analysierbarkeit.
- **Prinzip der Hierarchisierung:** Dadurch bessere Verständlichkeit und Analysierbarkeit.
- **Prinzip der Bindung und Kopplung:** Dadurch Begrenzung der Auswirkungen von Änderungen, d. h. bessere Stabilität.
- **Prinzip der Modularisierung:** Dadurch wird eine weitgehende Kontextunabhängigkeit von der Umgebung erreicht. Die Testbarkeit wird verbessert, da Module für sich testbar sind.
- **Geheimnisprinzip:** Änderungen an den Datenstrukturen wirken sich *nicht* auf die Zugriffsmethoden aus. Dadurch wird eine lokale Änderbarkeit ermöglicht.
- **Prinzip der Lokalität:** Da sich alle benötigten Informationen lokal an einer Stelle befinden, wird die Verständlichkeit und Analysierbarkeit wesentlich verbessert.
- **Prinzip der Verbalisierung:** Dadurch bessere Verständlichkeit und Analysierbarkeit.

9.3 Betriebssicherheit und Funktionssicherheit I

- **Architekturprinzip: Konzeptionelle Integrität:** Dadurch schnellere Einarbeitung (siehe »Architekturprinzip: Konzeptionelle Integrität«, S. 30).
- **Architekturprinzip: Trennung von Zuständigkeiten:** Dadurch schnellere Einarbeitung und leichtere Analysierbarkeit (siehe »Architekturprinzip: Trennung von Zuständigkeiten«, S. 31).
- **Architekturprinzip: Sichtbarkeit:** Dadurch bessere Verständlichkeit und Analysierbarkeit (siehe »Architekturprinzip: Sichtbarkeit«, S. 34).
- **Architekturprinzip: Selbstorganisation:** Dadurch geringere Komplexität und weniger zentrale Steuerung (siehe »Architekturprinzip: Selbstorganisation«, S. 34).

Folgende Architektur- und Entwurfs-Muster unterstützen u.a. die Weiterentwickelbarkeit:

- »Fabrikmethoden-Muster (*factory method pattern*)«, S. 89: Ermöglicht eine schwache Kopplung zwischen Klassen. Erlaubt es mit geringen Änderungen am Anwendungsprogramm, eine Produktfamilie zu erweitern.
- »Das Strategie-Muster (*strategy pattern*)«, S. 96: Erlaubt den Austausch bzw. das Hinzufügen von Algorithmen.
- »Das Schichten-Muster (*layers pattern*)«, S. 46: Zur Entkopplung der Subsysteme.
- »Das Beobachter-Muster (*observer pattern*)«, S. 54: Zur Entkopplung des Subsystems Benutzungsoberfläche von dem Subsystem Applikation.
- »Das MVC-Muster (*model view controller pattern*)«, S. 62: Zur Entkopplung der Subsysteme.

[RiBo09], [BBR09]

Architektur- &
Entwurfs-Muster

Literatur

9.3 Betriebssicherheit und Funktionssicherheit

Die nichtfunktionale Anforderung Sicherheit muss in die Betriebssicherheit und die Funktionssicherheit unterteilt werden.

Betriebssicherheit

Die **Betriebssicherheit** (*security*) eines Softwaresystems ist eine fundamentale Anforderung. In der Norm ISO/IEC 9126-1 ist die Sicherheit ein Teilmerkmal des Qualitätsmerkmals Funktionalität und ist wie folgt definiert:

I 9 Nichtfunktionale Anforderungen

Definition Fähigkeit des Softwareprodukts, Informationen und Daten so zu schützen, dass nicht autorisierte Personen oder Systeme sie nicht lesen oder verändern können, und autorisierten Personen oder Systemen der Zugriff nicht verweigert wird.

In [MZN10, S. 315] wird **Betriebssicherheit** wie folgt definiert:

Definition Anforderungen, die sich darauf beziehen, unautorisierten Zugriff zu Systemen, Programmen und Daten zu verhindern.

Betriebssicherheit wird in folgende Teilmerkmale untergliedert:

- Vertraulichkeit (*confidentiality*)
- Integrität (*integrity*)
- Verfügbarkeit (*availability*)
- Zugriffssteuerung (*access control*)
- Authentifizierung (*authentication*)

Die Betriebssicherheit ist entscheidend in der Betriebsphase eines Systems.

| ↓Anforderung / Effekt→ | Funktionalität | Leistung / Effizienz | Zuverlässigkeit | Benutzbarkeit | Sicherheit | Wartbarkeit |
|------------------------|----------------|----------------------|-----------------|---------------|------------|-------------|
| (Betriebs-) Sicherheit | + | + | + | + | o | + |

Tab. 9.3-1: Gegenseitige Abhängigkeiten. Wie die Tab. 9.3-1 zeigt, führt eine zunehmende Funktionalität und eine Steigerung der Leistung/Effizienz jeweils zu einer reduzierten Sicherheit. Eine verbesserte Benutzbarkeit kann zu einer schlechteren Sicherheit führen, da sicherheitsrelevante Eingaben durch den Benutzer u.U. reduziert werden – um den Aufwand für den Benutzer zu reduzieren.

Eine erhöhte Zuverlässigkeit und eine verbesserte Wartbarkeit dürften sich positiv auf die Sicherheit auswirken.

Zusammenhänge Die Realisierung der Anforderung **Mehrbenutzerfähigkeit** (siehe »Authentifizierung und Autorisierung«, S. 153) erfordert eine Benutzerverwaltung und eine Rechteverwaltung einschließlich Authentifizierung.

Was ist zu tun?

Regeln Die Betriebssicherheit kann durch die Einhaltung von folgenden Regeln verbessert werden:

- Eine hohe Betriebssicherheit wird durch eine Authentisierung, Authentifizierung und Autorisierung erreicht (siehe »Authentifizierung und Autorisierung«, S. 153).

9.3 Betriebssicherheit und Funktionssicherheit I

- Durch die Beachtung folgender, spezieller Entwurfsprinzipien kann die Betriebssicherheit verbessert werden (siehe »Authentifizierung und Autorisierung«, S. 153):
 - Prinzip der ökonomischen Sicherheitsmechanismen
 - Prinzip der sicheren Voreinstellungen
 - Prinzip der vollständigen Zugriffsüberprüfung
 - Prinzip des offenen Entwurfs
 - Prinzip der Aufteilung von Privilegien
 - Prinzip des kleinsten Privilegs
 - Prinzip der psychologischen Akzeptanz
 - Prinzip des kleinsten allgemeinen Mechanismus

Die Einhaltung folgender Prinzipien (siehe »Architekturprinzipien«, Prinzipien S. 29) unterstützt die Sicherheit:

- **Prinzip der Bindung und Kopplung:** Dadurch Begrenzung der Auswirkungen von Sicherheitsmängeln.
- **Geheimnisprinzip:** Datenstrukturen können *nicht* direkt manipuliert werden.

Folgende Architektur- und Entwurfs-Muster unterstützen u. a. die Sicherheit:

- »Das Schichten-Muster (*layers pattern*)«, S. 46: Einführung von Zugriffsschichten.
- »Das Proxy-Muster (*proxy pattern*)«, S. 83: Verwendung von Schutz-Proxy und Firewall-Proxy.
- »Das Fassaden-Muster (*facade pattern*)«, S. 69: Verwendung einer Fassade zur Zugangskontrolle.
- »Das Kommando-Muster (*command pattern*)«, S. 75: Zur Protokollierung ausgeführter Kommandos und zur Modellierung von Transaktionen.

Architektur- & Entwurfs-Muster

Funktionssicherheit

Die **Funktionssicherheit** (*safety*) eines Systems ist ein Maß für die Fähigkeit einer Betrachtungseinheit, weder Menschen, Sachen noch die Umwelt zu gefährden [Biro97].

Definition

Während die Betriebssicherheit (*security*) den Schutz von (Software-)Systemen vor Einflüssen und insbesondere Bedrohungen aus der Umwelt fordert, verlangt die Funktionssicherheit den Schutz der Umwelt vor Systemen.

Der Unterschied liegt in der Wirkrichtung. Im Falle der Betriebssicherheit entsteht die Bedrohung außerhalb des betrachteten Systems und wirkt ggf. in das System hinein. Funktionssicherheit fragt nach Bedrohungen, die in einem betrachteten System entstehen und auf seine Umgebung wirken.

I 9 Nichtfunktionale Anforderungen

Analyse der Restrisiken Aus diesem Grund erfordert die Beurteilung der Funktionssicherheit stets eine Analyse der gesamten Wirkkette, da die Kritikalität von Systemen, Systemteilen oder Funktionen nur anhand der potenziellen Schadenshöhe in der Umgebung eingeschätzt werden kann. Dies bedeutet im Normalfall, dass Software sowie elektronische und mechanische Komponenten analysiert werden müssen. Der Standard IEC 61508 10 (Funktionale Sicherheit sicherheitsbezogener elektrischer/ elektronischer/ programmierbar elektronischer Systeme) definiert Sicherheit im Sinne von *safety* als die Abwesenheit unakzeptabler Risiken. Diese Definition unterstreicht, dass sichere Systeme durchaus Restrisiken enthalten können. Sie dürfen eine bestimmte Akzeptanzschwelle nicht übersteigen. Daher ist im Rahmen einer Zertifizierung eines sicherheitskritischen Systems im Kontext eines Zulassungsverfahrens die quantitative Bestimmung der Restrisiken erforderlich. Anschließend muss entschieden werden, ob diese Risiken akzeptabel sind oder Modifikationen des Systems erforderlich sind. Man bezeichnet diesen Vorgang als Risikoakzeptanz.

Unfall - verhütung & technische Sicherheit Die Funktionssicherheit kann in die Sicherheit des ausfallfreien Systems (Unfallverhütung) und die sogenannte technische Sicherheit des ausfallbehafteten Systems untergliedert werden. Systeme müssen so konstruiert sein, dass bei perfekter Funktion aller Komponenten keine unangemessenen Gefährdungen von ihnen ausgehen. Darüber hinaus wird oft verlangt, dass Systeme auch in Gegenwart bestimmter Ausfälle sicher bleiben. Dies kann z.B. durch redundante Komponenten erreicht werden, die die Funktion ausgefallener Komponenten übernehmen.

9.4 Zuverlässigkeit

Softwaretechnik In der Softwaretechnik sind folgende Definitionen des Begriffs Zuverlässigkeit gebräuchlich:
Die nichtfunktionale Anforderung **Zuverlässigkeit** (*reliability*) wird in der Norm ISO/IEC 9126-1 als Qualitätsmerkmal eingeordnet und wie folgt definiert:

Definition Fähigkeit des Softwareprodukts, ein spezifiziertes Leistungsniveau zu bewahren, wenn es unter festgelegten Bedingungen benutzt wird.

Zuverlässigkeit wird untergliedert in folgende Teilmerkmale:

■ **Reife** (*maturity*)

Fähigkeit des Softwareprodukts, trotz Fehlzuständen in der Software nicht zu versagen.

■ **Fehlertoleranz** (*fault tolerance*)

Fähigkeit des Softwareprodukts, ein spezifiziertes Leistungsniveau bei Software-Fehlern oder Nicht-Einhaltung ihrer spezifizierten Schnittstelle zu bewahren.

■ **Wiederherstellbarkeit** (*recoverability*)

Fähigkeit des Softwareprodukts, bei einem Versagen ein spezifiziertes Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen.

In [MZN10, S. 315] wird **Zuverlässigkeit** wie folgt definiert:

Anforderungen, die die Fähigkeit eines Softwareprodukts spezifizieren, ohne Fehler zu arbeiten und ein festgelegtes Leistungsniveau beizubehalten, wenn es unter spezifizierten normalen Einsatzbedingungen während einer gegebenen Zeitperiode betrieben wird.

Definition

Zuverlässigkeit wird in folgende Teilmerkmale untergliedert:

- Vollständigkeit (*completeness*)
- Genauigkeit (*accuracy*)
- Konsistenz (*consistency*)
- Verfügbarkeit (*availability*) einschl. 24/7-Betriebsfähigkeit
- Integrität (*integrity*)
- Korrektheit (*correctness*)
- Reife (*maturity*) (identisch mit ISO/IEC 9126-1)
- **Fehlertoleranz** (*fault tolerance*) (identisch mit ISO/IEC 9126-1)
- **Wiederherstellbarkeit** (*recoverability*) (identisch mit ISO/IEC 9126-1)
- Einhaltung gesetzlicher, unternehmensinterner und vertraglicher Regelungen (*compliance*)

Für Zuverlässigkeit bezogen auf eingebettete Systeme existieren mehrere genormte Definitionen und zahlreiche Definitionen in der Fachliteratur.

Eingebettete Systeme

Die Norm DIN 40041 / DIN 40041 90 (Zuverlässigkeit; Begriffe) definiert **Zuverlässigkeit** als Teil der Qualität im Hinblick auf das Verhalten der Einheit während oder nach vorgegebenen Zeitspannen bei vorgegebenen Anwendungsbedingungen.

Die DIN EN ISO 9000 / DIN EN ISO 9000 05 (Qualitätsmanagementsysteme – Grundlagen und Begriffe) definiert **Zuverlässigkeit** als Sammelbegriff zur Beschreibung der Verfügbarkeit und ihrer Einflussfaktoren Funktionsfähigkeit, Instandhaltbarkeit und Instandhaltungsbereitschaft. Der entsprechende Begriff im englischen Sprachraum für Zuverlässigkeit in diesem umfassenden Sinne ist *Dependability*.

[Biro97] verwendet eine eingeschränktere Definition des Zuverlässigsbegriffs. Diese Zuverlässigkeit im engeren Sinne wird mit dem Begriff *Reliability* übersetzt. Sie ist ein Maß für die Fähigkeit

I 9 Nichtfunktionale Anforderungen

einer Betrachtungseinheit, funktionstüchtig zu bleiben, ausgedrückt durch die Wahrscheinlichkeit, dass die geforderte Funktion unter vorgegebenen Arbeitsbedingungen während einer festgelegten Zeitdauer ausfallfrei ausgeführt wird. Diese Definition gestattet eine Beschreibung der Zuverlässigkeit mit Hilfe stochastischer Techniken. So kann der Erwartungswert für die Zeitspanne bis zum Ausfall eines Systems – die sogenannte *Mean Time to Failure* (MTTF) – angegeben und als Zuverlässigkeitsmaß verwendet werden. Bei reparierbaren Systemen verwendet man den Erwartungswert zwischen zwei aufeinander folgenden Ausfällen – die sogenannte *Mean Time between Failure* (MTBF).

Alternativ kann die Überlebenswahrscheinlichkeit $R(t)$ verwendet werden. Sie gibt die Wahrscheinlichkeit an, ein System, das zum Zeitpunkt Null in Betrieb genommen wurde, am Zeitpunkt t noch intakt anzutreffen.

Definition

Zuverlässigkeit im engeren Sinne bezeichnet die Wahrscheinlichkeit des ausfallfreien Funktionierens einer betrachteten Einheit über eine bestimmte Zeitspanne bei einer bestimmten Betriebsweise.

Die Zuverlässigkeit ist im Allgemeinen nicht konstant, sondern wird bei Software im Zuge der mit Fehlerkorrekturen einhergehenden Zunahme der Stabilität größer sowie durch neue Fehler im Rahmen von Funktionalitätserweiterungen kleiner. Bei Hardware-Komponenten sind Einflüsse des Verschleißes zu beachten. Die beobachtete Zuverlässigkeit wird ferner durch die Art der Benutzung eines Systems bestimmt. Es ist oft möglich, Zuverlässigkeitsprognosen mit statistischen Verfahren zu erzeugen (siehe [Ligg09]).

Definition

Verfügbarkeit ist ein Maß für die Fähigkeit einer Betrachtungseinheit, zu einem gegebenen Zeitpunkt funktionstüchtig zu sein.

Sie wird ausgedrückt durch die Wahrscheinlichkeit, dass die Betrachtungseinheit zu einem gegebenen Zeitpunkt die geforderte Funktion unter vorgegebenen Arbeitsbedingungen ausführt. Als Maß für Verfügbarkeit kann der Quotient $MTBF / (MTBF + MTTR)$ verwendet werden. Der Wert dieses Quotienten kann auf zwei Arten gesteigert werden. Man kann die MTBF vergrößern, also die Zuverlässigkeit erhöhen, oder die MTTR (*Mean Time to Repair*) verringern, also die Stillstandszeitintervalle nach Ausfällen verkürzen.

Während die Steigerung der MTBF auch auf die Zuverlässigkeit wirkt, beeinflusst die Verringerung der MTTR nur die Verfügbarkeit. Die Zuverlässigkeit ist entscheidend in der Betriebsphase eines Systems.

9.4 Zuverlässigkeit I

| ↓Anforderung / Effekt→ | Funktionalität | Leistung / Effizienz | Zuverlässigkeit | Benutzbarkeit | Sicherheit + | Wartbarkeit + |
|------------------------|----------------|----------------------|-----------------|---------------|--------------|---------------|
| Zuverlässigkeit | + | + | + | + | + | + |

Wie die Tab. 9.4-1 zeigt, führt eine zunehmende Funktionalität und eine Steigerung der Leistung/Effizienz jeweils zu einer reduzierten Zuverlässigkeit. Werden die Sicherheit und die Wartbarkeit verbessert, dann kann dies einen positiven Einfluss auf die Zuverlässigkeit haben. Eine Verbesserung der Benutzbarkeit kann sich sowohl positiv als auch negativ auf die Zuverlässigkeit auswirken. Werden beispielsweise alle Benutzereingaben auf Korrektheit überprüft, dann erhöht dies die Zuverlässigkeit. Werden andererseits Bedienungsvarianten für den Benutzer erhöht, dann kann die Zuverlässigkeit dadurch sinken, dass nicht alle Varianten getestet werden können.

Tab. 9.4-1:
Gegenseitige
Abhängigkeiten.

Was ist zu tun?

Die Zuverlässigkeit kann durch die Einhaltung von folgenden Regeln verbessert werden:

- Identifikation der Subsysteme, von deren Zuverlässigkeit die Funktionsweise des Systems maßgeblich abhängt. Festlegung zusätzlicher Qualitäts- und Testmaßnahmen zur Sicherstellung der geforderten Zuverlässigkeit.
- Alle Benutzereingaben und alle Informationen, die andere Systeme liefern, auf Korrektheit überprüfen (defensives Programmieren).
- Mögliche Fehlerquellen in der Programmierung durch Ausnahmeregeln »abfangen« (In Java: try-catch-finally).
- Bereitstellung von Programmen, um bei einem Systemabsturz die Wiederherstellung der Betriebsbereitschaft und der Daten zu ermöglichen (siehe »Transaktionen«, S. 177).
- Mehrere Operationen, die als Gesamtheit ausgeführt werden müssen, durch Transaktionen absichern (siehe »Transaktionen«, S. 177).

Die Einhaltung folgender Prinzipien (siehe »Architekturprinzipien«, S. 29) unterstützt die Zuverlässigkeit:

- **Prinzip der Strukturierung** und **Prinzip der Modularisierung**: Dadurch Begrenzung auf einzelne Subsysteme bei Ausfall dieser Subsysteme.
- **Geheimnisprinzip**: Dadurch, dass andere Systeme die Interna eines Subsystems nicht kennen, wirken sich Ausfälle auf andere Systeme nur aus, wenn die Schnittstellen betroffen sind.
- **Prinzip der Bindung und Kopplung**: Dadurch Begrenzung der Auswirkungen von unzuverlässigen Subsystemen.

I 9 Nichtfunktionale Anforderungen

Architektur- & Entwurfs-Muster Folgende Architektur- und Entwurfs-Muster unterstützen u.a. die Zuverlässigkeit:

- »Das Schichten-Muster (*layers pattern*)«, S. 46: Die Un-Zuverlässigkeit einer Subsystem-Schicht wirkt sich nur begrenzt auf andere Schichten aus.
- »Das Kommando-Muster (*command pattern*)«, S. 75: Zur Protokollierung ausgeführter Kommandos, zur Modellierung von Transaktionen und zur Wiederholung von fehlgeschlagenen Transaktionen (Wiederherstellbarkeit).
- »Das Fassaden-Muster (*facade pattern*)«, S. 69: Dadurch, dass andere Systeme die Interna eines Subsystems nicht kennen (bei Black-Box), wirken sich Ausfälle auf andere Systeme nur aus, wenn die Fassade betroffen ist.

9.5 Leistung und Effizienz

Die nichtfunktionale Anforderung **Effizienz** (*efficiency*) wird in der Norm ISO/IEC 9126-1 als Qualitätsmerkmal eingeordnet und wie folgt definiert:

Definition Fähigkeit des Softwareprodukts, ein angemessenes Leistungs niveau bezogen auf die eingesetzten Ressourcen unter festgelegten Bedingungen bereitzustellen.

Effizienz wird untergliedert in folgende Teilmerkmale:

- **Zeitverhalten** (*time behaviour*)
Fähigkeit des Softwareprodukts, angemessene Antwort- und Verarbeitungszeiten sowie Durchsatz bei der Funktionsausführung unter festgelegten Bedingungen sicherzustellen.
- **Verbrauchsverhalten** (*resource utilisation*)
Fähigkeit des Softwareprodukts, eine angemessene Anzahl und angemessene Typen von Ressourcen zu verwenden, wenn die Software ihre Funktionen unter festgelegten Bedingungen ausführt.

In [MZN10, S. 315] wird **Leistung** (*performance*) wie folgt definiert:

Definition Anforderungen, die die Fähigkeit eines Softwareprodukts spezifizieren, eine angemessene Leistung relativ zu den benötigten Ressourcen zu erbringen, wobei die volle Funktionalität unter festgelegten Bedingungen bereitgestellt wird.

Leistung wird in folgende Teilmerkmale untergliedert:

- Antwortzeit (*response time*)
- Speicherplatz (*space*)

9.5 Leistung und Effizienz I

- Kapazität (*capacity*)
- Wartezeit (*latency*)
- Durchsatz (*throughput*)
- Rechenzeit (*computation*)
- Ausführungsgeschwindigkeit (*execution speed*)
- Übergangsverzögerung (*transit delay*)
- Auslastung (*workload*)
- **Verbrauchsverhalten** (*resource utilisation*) (identisch mit ISO/IEC 9126-1)
- Speicher-Inanspruchnahme (*memory usage*)
- **Genauigkeit** (*accuracy*) (in ISO/IEC 9126-1 unter Funktionalität subsumiert)
- Modi (*modes*)
- Verzögerung (*delay*)
- Fehlerraten (*miss rates*)
- Datenverlust (*data loss*)
- Nebenläufige Transaktionsverarbeitung (*concurrent transaction processing*)

Die Leistung/Effizienz ist entscheidend in der Betriebsphase eines Systems.

| ↓Anforderung / Effekt→ | Funktionalität + | Leistung / Effizienz + | Zuverlässigkeit + | Benutzbarkeit + | Sicherheit + | Wartbarkeit + |
|------------------------|------------------|------------------------|-------------------|-----------------|--------------|---------------|
| Leistung / Effizienz | – | ● | – | +– | – | – |

Wie die Tab. 9.5-1 zeigt, führen zusätzliche Funktionalität, verbesserte Zuverlässigkeit, erhöhte Sicherheit und bessere Wartbarkeit zu einem negativen Effekt auf die Leistung/Effizienz. Eine verbesserte Benutzbarkeit kann sich positiv oder negativ auf die Leistung/Effizienz auswirken.

Tab. 9.5-1:
Gegenseitige
Abhängigkeiten.

Was ist zu tun?

Die Leistung/Effizienz kann durch die Einhaltung von folgenden Regeln verbessert werden:

- Die einzusetzenden Algorithmen sind so auszuwählen, dass sie für das geforderte Effizienzkriterium optimal sind.
- Identifikation der Subsysteme, die entscheidend für die Leistungsfähigkeit des Systems sind. Gezielte Optimierung dieser Subsysteme entsprechend dem geforderten Effizienzkriterium.

Die Einhaltung folgender Prinzipien (siehe »Architekturprinzipien«, S. 29) unterstützt die Leistung/Effizienz:

- **Prinzip der Bindung und Kopplung:** Der Informationsfluss findet lokal in Komponenten und nicht zwischen Komponenten statt. Dadurch wird die Effizienz verbessert.

I 9 Nichtfunktionale Anforderungen

- **Architekturprinzip Ökonomie:** Dadurch wird sichergestellt, dass nur das realisiert wird, was unbedingt benötigt wird. Die Effizienz wird nicht durch unnötige Funktionalität oder Architekturkonzepte reduziert.
- **Architekturprinzip Trennung von Zuständigkeiten:** Dadurch, dass Zuständigkeiten an jeweils einer Stelle erledigt werden, wird ein unnötiger Informationsaustausch mit anderen Subsystemen vermieden.
- **Architekturprinzip Selbstorganisation:** Die Selbstorganisation von Komponenten reduziert zentrale Steuerungsmechanismen und verbessert dadurch die Effizienz.

Architektur- & Entwurfs-Muster

- Folgendes Entwurfs-Muster unterstützt u. a. die Leistung/Effizienz:
- »Das Proxy-Muster (*proxy pattern*)«, S. 83: Einsatz eines **virtuellen Proxy**, um »teure« Methoden nur aufzurufen, wenn sie auch benötigt werden. Einsatz eines **Cache-Proxy** für aufwändige Methoden, damit verschiedene Clients auf die Ergebnisse zugreifen können.

9.6 Benutzbarkeit

Die nichtfunktionale Anforderung **Benutzbarkeit** (*usability*) – oft auch **Gebrauchstauglichkeit** genannt – wird in der Norm ISO/IEC 9126-1 als Qualitätsmerkmal eingeordnet und wie folgt definiert:

Definition

Fähigkeit des Softwareprodukts, vom Benutzer verstanden und benutzt zu werden sowie für den Benutzer erlernbar und »attraktiv« zu sein, wenn es unter den festgelegten Bedingungen benutzt wird.

Die Benutzbarkeit wird in folgende Teilmerkmale untergliedert:

- **Verständlichkeit** (*understandability*)
Fähigkeit des Softwareprodukts, den Benutzer zu befähigen, zu prüfen, ob die Software angemessen ist und wie sie für bestimmte Aufgaben und Nutzungsbedingungen verwendet werden kann.
- **Erlernbarkeit** (*learnability*)
Fähigkeit des Softwareprodukts, den Benutzer zu befähigen, die Anwendung zu erlernen.
- **Bedienbarkeit** (*operability*)
Fähigkeit des Softwareprodukts, den Benutzer zu befähigen, die Anwendung zu bedienen und zu steuern.
- **Attraktivität** (*attractiveness*)
Fähigkeit des Softwareprodukts für den Benutzer attraktiv zu sein, z. B. durch die Verwendung von Farbe oder die Art des grafischen Designs.

■ Konformität der Benutzbarkeit (*usability compliance*)

Fähigkeit des Softwareprodukts, Standards, Konventionen, Stilvorgaben (*style guides*) oder Vorschriften bezogen auf die Benutzbarkeit einzuhalten.

In [MZN10, S. 315] wird Benutzbarkeit wie folgt definiert:

Anforderungen, die die Benutzer-Interaktionen mit dem System und den benötigten Aufwand, um das System zu erlernen, zu benutzen, die Eingaben in das System vorzubereiten und die Ausgabe des Systems zu interpretieren, festlegen.

Definition

Neben den identischen Teilmerkmalen der Norm ISO/IEC 9126-1 werden in [MZN10, S. 315] folgende weitere Teilmerkmale aufgeführt:

- Bedienungskomfort (*ease of use*)
- Ausführungsgeschwindigkeit (*execution speed*)
- Ergonomie (*human engineering*)
- Benutzungsfreundlichkeit (*user friendliness*)
- Einprägsamkeit (*memorability*)
- Effizienz (*efficiency*)
- Benutzerproduktivität (*user productivity*)
- Bedienungskomfort (*ease of use*)
- Ausführungsgeschwindigkeit (*execution speed*)
- Ergonomie (*human engineering*)
- Brauchbarkeit (*usefulness*)
- Erwartungskonformität (*likeability*)
- Benutzerreaktionszeit (*user reaction time*)

Die Benutzbarkeit ist entscheidend in der Entwicklungs- und der Betriebsphase eines Systems.

| ↓Anforderung / Effekt→ | Funktionalität | Leistung / Effizienz | Zuverlässigkeit | Benutzbarkeit | Sicherheit + | Wartbarkeit + |
|------------------------|----------------|----------------------|-----------------|---------------|--------------|---------------|
| Benutzbarkeit | +− | +− | + | o | − | o |

Wie die Tab. 9.6-1 zeigt, kann zusätzliche Funktionalität und verbesserte Leistung/Effizienz die Benutzbarkeit verbessern aber auch verschlechtern. Eine erhöhte Zuverlässigkeit führt in der Regel auch zu einer verbesserten Benutzbarkeit. Eine verstärkte Sicherheit kann sich auf die Bequemlichkeit der Benutzbarkeit negativ auswirken. Eine verbesserte Wartbarkeit dürfte sich neutral auf die Benutzbarkeit auswirken.

Tab. 9.6-1:
Gegenseitige
Abhängigkeiten.

Was ist zu tun?

Die Benutzbarkeit kann durch die Einhaltung von folgenden Regeln verbessert werden:

I 9 Nichtfunktionale Anforderungen

- Alle Aktivitäten, die die Benutzerinteraktion betreffen, werden in einem Subsystem angeordnet.
- Das Subsystem Benutzungsoberfläche greift nur auf das Subsystem Applikation zu, nicht aber auf das Subsystem Persistenz.

Prinzipien Die Einhaltung folgender Prinzipien (siehe »Architekturprinzipien«, S. 29) unterstützt die Benutzbarkeit:

- **Prinzip der Verbalisierung:** Durch die domänengerechte Wahl der Begriffe auf der Benutzungsoberfläche werden das Verständnis, die Einarbeitung und die Bedienung für den Benutzer erleichtert.
- **Architekturprinzip: Ökonomie:** Es wird nur die Funktionalität zur Verfügung gestellt, die der Benutzer zur Erledigung seiner Aufgaben benötigt. Dadurch wird die Benutzbarkeit verbessert.
- **Architekturprinzip: Trennung von Zuständigkeiten:** Alle Aufgaben, die mit der Benutzerinteraktion zu tun haben, werden in einem Subsystemen gekapselt.

Architektur- & Entwurfs-Muster Folgende Architektur- und Entwurfs-Muster unterstützen die Benutzbarkeit:

- »Das Schichten-Muster (*layers pattern*)«, S. 46: Ermöglicht es, die Benutzungsoberfläche in einer Schicht anzuhören.
- »Das Kommando-Muster (*command pattern*)«, S. 75: Zur Realisierung von Undo- und Redo-Kommandos so wie die Ermöglichung einer Makro-Aufzeichnung.
- »Das MVC-Muster (*model view controller pattern*)«, S. 62: Unterstützt die leichte Erweiterung um zusätzliche Ansichten (*views*) und Eingaben über verschiedene Geräte (*controls*).

9.7 Portabilität

Die nichtfunktionale Anforderung **Portabilität** (*portability*) wird in der Norm ISO/IEC 9126-1 als Qualitätsmerkmal eingeordnet und wie folgt definiert:

Definition

Fähigkeit des Softwareprodukts, von einer Umgebung in eine andere übertragen zu werden. Umgebung kann organisatorische Umgebung, Hardware- oder Software-Umgebung einschließen.

Portabilität wird in folgende Teilmerkmale untergliedert:

■ Anpassbarkeit (*adaptability*)

Fähigkeit des Softwareprodukts, die Software an verschiedene, festgelegte Umgebungen anzupassen, wobei nur Aktionen oder Mittel eingesetzt werden, die für diesen Zweck für die betrachtete Software vorgesehen sind.

■ **Installierbarkeit** (*installability*)

Fähigkeit des Softwareprodukts, in einer festgelegten Umgebung installiert zu werden.

■ **Koexistenz** (*co-existence*)

Fähigkeit des Softwareprodukts, mit anderen unabhängigen Softwareprodukten in einer gemeinsamen Umgebung gemeinsame Ressourcen zu teilen.

■ **Austauschbarkeit** (*replaceability*)

Fähigkeit des Softwareprodukts, diese Software anstelle einer spezifizierten anderen in der Umgebung jener Software für denselben Zweck zu verwenden.

■ **Konformität der Portabilität** (*portability compliance*)

Fähigkeit des Softwareprodukts, Standards oder Konventionen bezogen auf die Portabilität einzuhalten.

Nach [MZN10] gehört Portabilität nicht zu den am häufigsten genannten nichtfunktionalen Anforderungen. Sie wird jedoch bei Echtzeitsystemen als eine wichtige Anforderung aufgeführt.

Was ist zu tun?

Die Portabilität kann durch die Einhaltung von folgender Regel verbessert werden:

- Die von einer möglichen Portabilität betroffenen Teile sind zu identifizieren und in eigene Subsysteme zu kapseln.

Die Einhaltung folgender Prinzipien (siehe »Architekturprinzipien«, S. 29) unterstützen die Portabilität:

■ **Prinzip der Modularisierung:** Die zu portierenden Teile sind in Module gekapselt.

■ **Architekturprinzip Trennung von Zuständigkeiten:** Die von der Portabilität betroffenen Teile sind in Subsystemen isoliert.

■ **Architekturprinzip Sichtbarkeit:** Die zu portierenden Teile sind in der Architektur gut sichtbar.

Folgende Architektur- und Entwurfs-Muster unterstützen die Portabilität:

- »Das Schichten-Muster (*layers pattern*)«, S. 46: Oft muss nur die unterste Schicht modifiziert werden, um ein System auf eine andere Hardware- oder Software-Umgebung zu übertragen. Soll die Benutzungsoberfläche für einen anderen Gerätetyp geändert werden, dann muss oft nur die oberste Schicht angepasst werden.
- »Das MVC-Muster (*model view controller pattern*)«, S. 62: Unterstützt die Portabilität auf andere Ein- und Ausgabegeräte.
- »Das Fassaden-Muster (*facade pattern*)«, S. 69: Ermöglicht die Anpassung an Schnittstellen umgebender Systeme.

Architektur- & Entwurfs-Muster

10 Einflussfaktoren auf die Architektur

Eine Softwarearchitektur zu entwerfen ist schwierig, da verschiedene Einflussfaktoren zu berücksichtigen sind, die sich zudem auch noch gegenseitig beeinflussen können. Die meisten dieser Einflussfaktoren sollten bereits bei der Spezifikation festgelegt worden sein. Ist dies nicht der Fall, dann muss dies vor der Entwicklung der logischen Architektur nachgeholt werden.

Es lassen sich folgende Faktorengruppen unterscheiden:

- **Funktionale Anforderungen**
- **Nichtfunktionale Anforderungen einschl. Qualitätsanforderungen** (siehe »Nichtfunktionale Anforderungen«, S. 109)
- **Anwendungsart**
- **Verteilungsart** (siehe »Verteilte Architekturen«, S. 191)
- **Kontext des Anwendungssystems**
- **Art der softwaretechnischen Infrastruktur** (siehe »Softwaretechnische Infrastrukturen«, S. 319)

Anwendungsart

- **Einzelplatz-Anwendung** (auch Desktop-Anwendung genannt): Das zu entwickelnde Softwaresystem wird von einem Benutzer oder von mehreren Benutzern nacheinander auf einem Computersystem benutzt. Soll das System vor unberechtigten Zugriffen geschützt werden, dann ist eine Authentifizierung und Autorisierung erforderlich. Haben die Benutzer unterschiedliche Rechte, dann wird zusätzlich eine Rechteverwaltung benötigt. Beispiele für solche Systeme sind: Fakturierungs-Systeme, Buchhaltungssysteme für Selbstständige und Handwerksbetriebe, Vereinsverwaltung, Apps für Smartphones.
- **Mehrplatz-Anwendung**: Die Anwendung wird von vielen Benutzern *nebenläufig* auf unterschiedlichen Computersystemen benutzt, die miteinander vernetzt sind. In der Regel werden für solche Systeme eine Benutzerverwaltung und auch eine Rechteverwaltung benötigt. Zusätzlich muss das Softwaresystem in der Lage sein, parallele Zugriffe zu verwalten. Wesentlich bei einer Mehrplatz-Anwendung ist, ob die maximale Anzahl der Benutzer im Voraus feststeht oder nicht.

I 10 Einflussfaktoren auf die Architektur

- **Unternehmenslösung:** Anwendungen, die zur Unterstützung der Geschäftsabläufe in einem Unternehmen eingesetzt werden. Eine Unternehmenslösung ist immer eine Mehrplatz-Anwendung (siehe »Anforderungen an Unternehmensanwendungen«, S. 319).
- **Stand-alone-Anwendung:** Die Anwendung ist unabhängig von anderen Anwendungen einsetzbar, z.B. ein Zeichenprogramm. Es handelt sich dabei oft um eine Einzelplatz-Anwendung.
- **Stapelverarbeitungs-Anwendung** (*Batch*-Anwendung, automatisierte Anwendung): Anwendung, die nach dem Start automatisch ohne weitere Interaktionen abläuft, z.B. Lohnabrechnungs-Anwendungen. Es handelt sich dabei oft um eine *Standalone*-Anwendung.
- **Produktfamilie/Produktlinie:** Anwendungen, die auf einer gemeinsamen Grundlage basieren, und es ermöglichen, für bestimmte Branchen oder Produktfamilien Anwendungsvarianten zu erzeugen, z.B. unterschiedliche Funktionalitäten für verschiedene Handy-Modelle.

Verteilungsart

Einzelplatz-Anwendungen können, Mehrplatz-Anwendungen und Unternehmenslösungen müssen auf mehrere Computersysteme verteilt werden. Der Benutzer bzw. die Benutzer interagieren mit der Software auf einem Client-Computer, während sich die zentralen Teile der Anwendungssoftware auf einem oder mehreren Server-Computern befinden. Es lassen sich folgende wichtige Verteilungsarchitekturen unterscheiden (siehe im Detail »Verteilte Architekturen«, S. 191):

- **Client-Server-Architektur:** Ein Teil der Software wird auf den Client-Computern der Benutzer installiert. Die Client-Computer stehen in der Regel über eine Netzwerkverbindung permanent mit einem Server-Computer in Verbindung (siehe »Client-Server-Architektur«, S. 193).
- **Web-Architektur:** Die Benutzer verwenden auf den Client-Computern einen Standard-Webbrowser um über eine temporäre Netzwerkverbindung mit der Anwendung auf dem Server-Computer zu arbeiten (siehe »Web-Architektur«, S. 196).
- Eine Kombination beider Architekturen ist möglich.
- Bei den Clients kann es sich auch um mobile Clients handeln (Handys, PDAs, Smartphones).
- **Serviceorientierte Architektur** (SOA): Die beteiligten Systeme und Komponenten stellen ihre Funktionalitäten in Form von Services zur Verfügung (siehe »SOA – Serviceorientierte Architekturen«, S. 201).

10 Einflussfaktoren auf die Architektur I

- Ein Sonderfall stellt eine **Peer-to-Peer-Architektur** dar, bei der gleichberechtigte Computersysteme miteinander kommunizieren und jeder gleichzeitig Server und Client ist.

Kontext des Anwendungssystems

In der Regel steht ein neues Softwaresystem *nicht* für sich allein, sondern es interagiert mit anderen Softwaresystemen. Folgende Gesichtspunkte sind zu beachten:

- Welche Schnittstellen anderer Systeme stehen zur Verfügung?
- Welche Schnittstellen müssen anderen Systemen zur Verfügung gestellt werden?
- Auf welche Dienstleistungen anderer Systeme ist das Softwaresystem permanent oder temporär angewiesen, um seine Leistungen zur Verfügung zu stellen?
- Auf welche Dienstleistungen des Softwaresystems sind andere Systeme permanent oder temporär angewiesen?
- Welche anderen Softwaresysteme verwenden die Benutzer des neuen Systems permanent oder temporär?

Art der softwaretechnischen Infrastruktur

Die softwaretechnische Infrastruktur, die auf dem Zielsystemen zur Verfügung steht oder zur Verfügung gestellt werden muss, hat Einfluss auf die Softwarearchitektur. Sie entscheidet darüber, welche Dienstleistungen in Anspruch genommen werden können bzw. welche Dienstleistungen selbst programmiert werden müssen. Die notwendige softwaretechnische Infrastruktur hängt natürlich von der Anwendungsart und der Verteilungsart ab. Außerdem muss die softwaretechnische Infrastruktur auf der physischen Infrastruktur installierbar sein. Es lassen sich grob folgende softwaretechnische Infrastrukturen unterscheiden:

- Plattformspezifische Entwicklungs- und Laufzeitumgebungen, z. B. .NET.
- Plattformübergreifende Entwicklungs- und Laufzeitumgebungen, z. B. Java.
- Heterogene Entwicklungs- und Laufzeitumgebungen, z. B. wenn von mobilen Endgeräten auf Unternehmenslösungen zugegriffen werden soll.
- *Middleware*-Entwicklungs- und Laufzeitumgebungen, die komplexe Dienstleistungen zur Verfügung stellen, z. B. Java EE und .NET Framework.

I 10 Einflussfaktoren auf die Architektur

Wechselwirkungen

Wie die Abb. 10.0-1 zeigt, stehen die Einflussfaktoren in einer gegenseitigen Wechselwirkung.

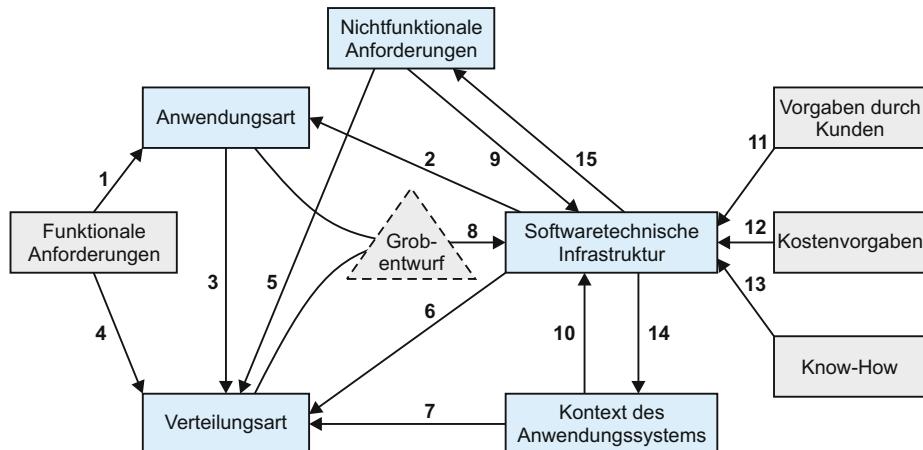


Abb. 10.0-1:
Wechselwirkungen
zwischen
Einflussfaktoren
(in Anlehnung an
[Opit10, S. 22]).

Die in der Abb. 10.0-1 dargestellten Wechselwirkungen sind durchnummiert und werden im Folgenden erklärt (in Anlehnung an [Opit10, S.22 ff.]).

Einflussfaktoren für die Anwendungsart

Funktionale Anforderungen (1): Die Anwendungsart ergibt sich in der Regel bei der Spezifikation der funktionalen Anforderungen.

Beispiel Aus der funktionalen Anforderung »Alle Mitarbeiter aus der Kundenbetreuung sollen neue Kunden erfassen und vorhandene Kunden bearbeiten können« lässt sich ableiten, dass es sich um eine Mehrplatz-Anwendung handeln wird.

Softwaretechnische Infrastruktur (2, 6): Ist bereits vorgegeben, welche softwaretechnische Infrastruktur genutzt werden soll, kann dies die Auswahl der zur Verfügung stehenden Anwendungs- und Verteilungsarten beeinflussen.

Einflussfaktoren für die Verteilungsart

Anwendungsart (3): Die Verteilungsart leitet sich aus der gewählten Anwendungsart unter Berücksichtigung der funktionalen und nichtfunktionalen Anforderungen ab (4, 5).

Funktionale Anforderungen (4): Aus bestimmten funktionalen Anforderungen lässt sich nicht nur die Anwendungsart, sondern auch direkt eine Verteilungsart ableiten.

10 Einflussfaktoren auf die Architektur I

Die funktionale Anforderung »Die Mitarbeiter der Kundenbetreuung sollen von beliebigen mobilen Geräten Zugriff auf die Kundendaten haben« kann die Festlegung auf eine Web-Architektur zur Folge haben. Beispiel

Nichtfunktionale Anforderungen (5): Bestimmte nichtfunktionale Anforderungen können die Festlegung der Verteilungsart beeinflussen.

Soll eine flexible, skalierbare und gut erweiterbare Unternehmenslösung entwickelt werden, deren einzelne Komponenten unterschiedliche fachliche Aufgaben unterstützen, könnte eine serviceorientierte Architektur die geeignete Verteilungsart darstellen. Beispiel

Softwaretechnische Infrastruktur (2, 6): Ist bereits vorgegeben, welche softwaretechnische Infrastruktur genutzt werden soll, kann dies die Auswahl der zur Verfügung stehenden Anwendungs- und Verteilungsarten einschränken.

Kontext des Anwendungssystems (7): Die vorhandenen Softwaresysteme können die Auswahl einer Verteilungsart beeinflussen.

Handelt es sich bei allen bisher beim Kunden eingesetzten Softwaresystemen um Webanwendungen und kommen bei der zu entwickelnden Anwendung sowohl eine Client-Server-Architektur als auch eine Web-Architektur in Frage, könnte dies die Wahl einer Web-Architektur zur Folge haben. Beispiel

Häufig geschieht die Festlegung von Anwendungsart und Verteilungsart auch in einem Schritt. Wird beispielsweise direkt festgelegt, dass eine Webanwendung entwickelt werden soll, entscheidet man sich damit indirekt für eine Mehrplatz-Anwendung (Anwendungsart) mit einer Web-Architektur (Verteilungsart).

Einflussfaktoren bei der Wahl der softwaretechnischen Infrastruktur

Anwendungsart und Verteilungsart (8): Nach der Festlegung der Anwendungsart und der Verteilungsart lässt sich bereits eine grobe Architektur des zu entwickelnden Systems definieren. Daraus lässt sich ableiten, welche Funktionen die softwaretechnische Infrastruktur zur Verfügung stellen muss.

Wurde festgelegt, dass eine Webanwendung entwickelt werden soll, werden Techniken für die Weboberfläche, die Anwendungslogik und die Datenhaltung benötigt. Anwendungs- und Verteilungsart beeinflussen somit stark die Auswahl der softwaretechnischen Infrastruktur. Beispiel

I 10 Einflussfaktoren auf die Architektur

Nichtfunktionale Anforderungen (9): Stehen bei der Auswahl der softwaretechnischen Infrastruktur mehrere Alternativen zur Verfügung, die allesamt für die Realisierung der funktionalen Anforderungen geeignet sind, sollte untersucht werden, in welchem Maße die verschiedenen zur Auswahl stehenden Techniken die geforderten nichtfunktionalen Anforderungen unterstützen.

Beispiel Wird viel Wert auf die Portabilität der zu entwickelnden Software gelegt, ist es sinnvoll eine plattformunabhängige Infrastruktur, wie z.B. Java, zu wählen.

Kontext des Anwendungssystems (10): Auch die Softwaresysteme, mit denen die zu entwickelnde Anwendung interagieren soll, können Einfluss auf die Auswahl der softwaretechnischen Infrastruktur haben.

Beispiel Die zu entwickelnde Anwendung soll intensiv mit dem Office-Paket von Microsoft zusammenarbeiten. Dies könnte die Auswahl der .NET-Laufzeitumgebung zur Folge haben, weil dafür Bibliotheken für die Interaktion mit Office-Produkten verfügbar sind.

Vorgaben durch den Kunden (11): In bestimmten Fällen kann die softwaretechnische Infrastruktur bereits durch den Kunden vorgegeben sein.

- Beispiele
- Ein Kunde nutzt bereits mehrere Java EE-Anwendungen. Aus diesem Grund könnte er die »Java Enterprise Edition« als Infrastruktur vorschreiben, weil er bereits dafür optimierte Hardware oder Lizizenzen für kommerzielle Java EE-Anwendungsserver besitzt.
 - Ein Kunde benutzt z.B. eine Oracle-Datenbank mit allen ihren Möglichkeiten und möchte daher nicht auf eine Microsoft-SQL-Datenbank wechseln.

Kosten (12): Auch Kostenvorgaben können Einfluss auf die softwaretechnische Infrastruktur haben. Ist ein bestimmter Kostenrahmen vorgegeben, könnten Produkte, deren Lizenzkosten den Rahmen sprengen würden, bereits im Voraus ausgeschlossen werden.

Vorhandenes Know-How (13): Die im entwickelnden Unternehmen bereits vorhandenen Erfahrungen mit bestimmten Produkten können eine entscheidende Rolle bei der Auswahl der softwaretechnischen Infrastruktur spielen.

Beispiel Ein Unternehmen entwickelt schon seit Jahren mit Java und besitzt viele Experten auf diesem Gebiet. Bei der Entwicklung neuer Anwendungen wird es in vielen Fällen nicht auf alternative Produkte ausweichen.

Einfluss auf den Kontext des Anwendungssystems

Softwaretechnische Infrastruktur (14): Nach der Auswahl einer softwaretechnischen Infrastruktur müssen sowohl die Schnittstellen der zu entwickelnden Anwendung, als auch die der vorhandenen Systeme auf eine Interaktion miteinander ausgelegt werden.

Einfluss auf nichtfunktionale Anforderungen

Softwaretechnische Infrastruktur (15): Zwar sollte bei der Auswahl der softwaretechnischen Infrastruktur darauf geachtet werden, dass sich mit dieser die geforderten nichtfunktionalen Anforderungen erfüllen lassen. Es kann aber durchaus vorkommen, dass sogar mit der bestmöglichen Infrastruktur nicht alle Anforderungen wie gewünscht realisiert werden können. In diesem Fall ist es unter Umständen nötig, Zugeständnisse bei den nichtfunktionalen Zielvorgaben zu machen.

11 Globalisierung von Software

Softwaresysteme werden heute in der Regel nicht nur in einem Land, sondern oft in mehreren Ländern oder sogar weltweit eingesetzt. Bereits bei der Architektur einer Software ist der mögliche Einsatz in verschiedenen Ländern zu berücksichtigen.

Was verbinden Sie mit dem Begriff »Globalisierung von Software«? [Frage](#)

Unter dem Begriff **Globalisierung** (*globalization*) von Software – abgekürzt **g11n** für die 11 Buchstaben zwischen g und n im englischen Wort *globalization* – versteht man die Internationalisierung und Lokalisierung von Anwendungen. [Antwort](#)

Internationalisierung (*internationalization*) – abgekürzt **i18n** – bedeutet, eine Software so zu konzipieren und zu implementieren, dass sie leicht an andere Sprachen und Kulturen angepasst werden kann, ohne den Quellcode zu ändern und ohne eine neue Compilierung vorzunehmen. Außerdem darf die Software keine eingebauten Annahmen über die Sprache und kulturelle Konventionen enthalten.

Lokalisierung (*localization*) – abgekürzt **L10N** (groß geschrieben) – bedeutet, eine internationalisierte Software an eine spezifische Region oder Sprache durch das Hinzufügen lokal-spezifischer Komponenten und übersetzter Texte anzupassen. Lokalisierungen müssen durch Personen vorgenommen werden können, die auf Übersetzungen spezialisiert sind, aber keine Programmierer sind. Dennoch muss jede lokalisierte Version getestet werden, auch wenn der Test eingeschränkter ist als ein vollständiger Test der Anwendung.

Die Internationalisierung wird pro Anwendungssystem einmal durchgeführt, während die Lokalisierung für jede Kombination von Anwendungssystem und Nutzungsgebiet durchzuführen ist. In der Regel erfolgt eine Lokalisierung zunächst für eine Sprache und später für weitere Sprachen.

Was ist bei der Konzeption einer internationalisierten Software zu beachten? [Frage](#)

Ziel muss es sein, alle möglichen lokalen Besonderheiten von vornherein aus der Software herauszuhalten und die Besonderheiten in Dateien und Bibliotheken auszulagern. Die Besonderheiten beinhalten insbesondere folgende Punkte: [Antwort](#)

- Unabhängigkeit der Implementierung von einer **Sprache**:
- Keine Texte oder Bezeichnungen für Interaktionselemente wie Menüs und Druckknöpfe, die auf der Benutzeroberfläche angezeigt werden, fest in der Implementierung programmieren.
- Keine Meldungstexte an den Benutzer direkt im Quellcode.

I 11 Globalisierung von Software

- Verwendung des Unicode-Standards, um Zeichencodierungsprobleme zu vermeiden.
- Keine Festlegung auf eine Schreibrichtung (z. B. von links nach rechts) auf der Benutzungsoberfläche bei Eingaben.
- Keine Grafiken, die Texte enthalten.
- Keine Untertitelung von Filmen und Videoaufnahmen.
- Keine gesprochenen Texte (Audio).
- Keine Festlegung auf **Schreibkonventionen**:**
- Datum/Zeit-Format einschließlich der Benutzung verschiedener Kalender.
- Zeitzonen (UTC in internationalen Umgebungen).
- Formatierung von Zahlen (Dezimaltrennung, Zahlengruppierung).
- Keine Festlegung auf **kulturelle Besonderheiten**:**
- Telefonnummern, Adressen und internationale Postleitzahlen.
- Währungen (Symbole, Position der Währungsangabe).
- Gewichte und Maße.
- Papiergrößen.
- Von der Regierung vergebene Nummern (Steueridentifikationsnummer in Deutschland, *Social Security*-Nummer in den USA, *National Insurance*-Nummer in Großbritannien).
- Namen und Titel.
- Bilder und Farben, die in verschiedenen Kulturen unterschiedlich interpretiert werden können.

Lokalisierung Bei der Lokalisierung müssen die jeweiligen sprachlichen Informationen bereitgestellt sowie die Schreibkonventionen und die kulturellen Besonderheiten eingestellt werden. Üblicherweise werden textuelle Informationen in externe Dateien (*resources files*) und Bibliotheken ausgelagert. Auf diese Dateien und Bibliotheken greift die Anwendung während der Laufzeit zu.

Hinweise Um eine Lokalisierung einfach vornehmen zu können, sollten folgende Hinweise beachtet werden [Vine10]:

- Nichts sollte »hart« codiert werden:** Konstanten, Literale und ähnliche Dinge sollten in einem eigenen Definitionsabschnitt des Programms definiert werden.
- Die Schriftgröße (*font size*) sollte nicht beschränkt werden,** da asiatische Zeichen oft mehr Raum benötigen.
- Alle Dinge, die durch den Benutzer betrachtet, erzeugt, geändert oder benutzt werden können,** sollten in separate Ressourcendateien ausgelagert werden. Dazu gehören Bezeichnungen von Komponenten, Meldungen (Status, Fehler, Hilfe), Fenstertitel und Kommandokürzel (*shortcuts, hotkeys*).
- Feldlängen dürfen nicht im Quellcode festgelegt werden,** da Zeichenketten und Währungswerte durch die Lokalisierung größer werden können.

11.1 Globalisierung in Java I

- Mitteilungen müssen als ganze Sätze gespeichert und dürfen nicht aus Teilen zusammengesetzt werden. Für Werte, die vom System geliefert werden, z.B. Dateinamen, verfügbarer Plattenplatz, können Platzhalter verwendet werden, solange sie umgeordnet werden können.

Es ist leichter, ein Softwaresystem von vornherein zu internationalisieren, als später ein für einen lokalen Markt entwickeltes Softwaresystem nachträglich zu internationalisieren.

Merke

In Java gibt es eine ganze Reihe von Klassen, die die Internationalisierung und Lokalisierung unterstützen:

- »Globalisierung in Java«, S. 145

Anhand der Fallstudie wird gezeigt, wie aufwändig es ist, von vornherein eine Anwendung zu internationalisieren:

- »Fallstudie: KV – Globalisiert«, S. 151

11.1 Globalisierung in Java

In Java gibt es im Wesentlichen folgende Klassen, die die Internationalisierung und Lokalisierung unterstützen:

- ResourceBundle: Stellt die Zuordnung zwischen gewählter Sprache und gewähltem Land zu der entsprechenden Datei her.
 - Locale: Repräsentiert Sprache und Land sowie herstellerabhängige Varianten.
 - MessageFormat: Erlaubt es, zusammengesetzte Mitteilungen an den Benutzer in einer sprachneutralen Weise zu behandeln.
 - NumberFormat: Abstrakte Oberklasse für alle Zahlenformate. Er ermöglicht es zu bestimmen, welche Lokalisierungen welche Zahlenformate haben und wie ihre Namen lauten.
 - DateFormat: Abstrakte Oberklasse für alle Zeit- und Datumsformate. Erlaubt es sowohl ein Datum als auch eine Zeit sprachabhängig zu parsen.
 - Collator: Abstrakte Oberklasse, die es erlaubt, String-Vergleiche in Abhängigkeit von der jeweiligen Lokalisierung vorzunehmen.
- Einige Klassen werden im Folgenden näher behandelt.

ResourceBundle und Locale

Landessprachliche Mitteilungen und Texte werden in Java im Quellcode durch symbolische Namen gekennzeichnet. Pro Sprache und Land werden den symbolischen Namen die landessprachlichen Texte in einer Datei zugeordnet.

I 11 Globalisierung von Software

.properties Diese Dateien haben die Endung .properties und besitzen folgenden Dateinamenaufbau:

- Bezeichnung der Datei, z. B. Mitteilungen,
- _Sprache, z. B. de für Deutsch,
- _Land, z. B. DE für Deutschland oder CH für die Schweiz,
- _Variante, beschreibt herstellerabhängige Angaben, z. B. WIN für Windows-Betriebssysteme.

Die Landes- und Variantenangaben sind optional. In der jeweiligen Datei – es handelt sich um reine Textdateien – wird pro Zeile nach dem symbolischen Namen der anzuseigende Text aufgeführt.

Beispiel 1a: Den symbolischen Namen Start, Frage, Ende sollen Texte in Deutsch für Deutschland und für die Schweiz sowie für Englisch in den USA zugeordnet werden. Zusätzlich soll es eine default-Datei geben, die verwendet wird, wenn eine landesspezifische Datei *nicht* gefunden wird. Folgende Dateien sind zu erstellen:

```
Java Datei: Mitteilungen.properties
Inhalt:
start = Hallo
frage = Wie geht's?
ende = Auf Wiedersehen!
Datei: Mitteilungen_de_DE.properties
Inhalt:
start = Hallo
ende = Tschüß
frage = Wie geht's?
Datei: Mitteilungen_de_CH.properties
Inhalt:
start = Grüezi
frage = Mir geits guet u dir?
ende = Auf Wiedersehen!
Datei: Mitteilungen_en_US.properties
Inhalt:
start = Hello
frage = How are you?
ende = Good bye
```

In Java werden die Dateien in nachfolgender Suchreihenfolge durchlaufen:

```
Bezeichnung der Datei_Sprache_Land_Variante
Bezeichnung der Datei_Sprache_Land
Bezeichnung der Datei_Sprache
Bezeichnung der Datei_defaultSprache_defaultLand_defaultVariante
Bezeichnung der Datei_defaultSprache_defaultLand
Bezeichnung der Datei_defaultSprache
Bezeichnung der Datei
```

Es können mehrere Dateien angegeben werden, die nacheinander durchlaufen werden. Werte können sich dadurch kumulieren und auch überschreiben.

11.1 Globalisierung in Java I

Mithilfe der Fabrikmethode `public static final ResourceBundle getBundle(String baseName, Locale locale)` der abstrakten Klasse `ResourceBundle` wird die Verbindung zwischen den symbolischen Namen im Programm und den lokalen Übersetzungen hergestellt. `baseName` ist der voll qualifizierte Klassenname der entsprechenden Ressourcendatei. `locale` gibt an, um welche Sprache, welches Land und welche Variante es sich handelt.

Mithilfe der Methode `public final String getString(String key)` erhält man zum symbolischen Namen die landesspezifische Übersetzung.

Mithilfe der Klasse `Locale` werden sprach-, landes- und varianten- spezifische Objekte erzeugt:

```
public Locale(String language, String country, String variant)
```

Die Sprache wird durch einen klein geschriebenen Zwei-Buchstabencode nach ISO 693 angegeben. Das Land wird durch einen groß geschriebenen Zwei-Buchstabencode nach ISO 3166 spezifiziert. Die Landes- und Variantenangabe sind optional.

Das folgende Programm demonstriert die Verwendung dieser Klassen:

```
package de.w3l.anw;

import java.util.Locale;
import java.util.ResourceBundle;

public class HelloGlobal
{
    static public void main(String[] args)
    {
        String sprache;
        String land;

        sprache = new String("de");
        land = new String("DE");

        Locale aktuelleLokalisierung;
        ResourceBundle mitteilungen;

        aktuelleLokalisierung = new Locale(sprache, land);

        mitteilungen =
            ResourceBundle.getBundle("de.w3l.anw.res.Mitteilungen",
            aktuelleLokalisierung);

        System.out.println(mitteilungen.getString("start"));
        System.out.println(mitteilungen.getString("frage"));
        System.out.println(mitteilungen.getString("ende"));

        sprache = new String("en");
        land = new String("US");
```

Beispiel 1b:
HelloGlobal

Java

I 11 Globalisierung von Software

```
aktuelleLokalisierung = new Locale(sprache, land);

mitteilungen =
    ResourceBundle.getBundle("de.w3l.anw.res.Mitteilungen",
    aktuelleLokalisierung);

System.out.println(mitteilungen.getString("start"));
System.out.println(mitteilungen.getString("frage"));
System.out.println(mitteilungen.getString("ende"));

sprache = new String("de");
land = new String("CH");

aktuelleLokalisierung = new Locale(sprache, land);

mitteilungen =
    ResourceBundle.getBundle("de.w3l.anw.res.Mitteilungen",
    aktuelleLokalisierung);

System.out.println(mitteilungen.getString("start"));
System.out.println(mitteilungen.getString("frage"));
System.out.println(mitteilungen.getString("ende"));
}
}
```

Es werden folgende Ausgaben erzeugt:

```
Hallo
Wie geht's?
Tschüß
Hello
How are you?
Good bye
Grüezi
Mir geits guet u dir?
Auf Wiedersehen!
```

Empfehlung

Um den Quellcode besser lesbar und leichter wartbar zu machen, sollten in den .properties-Dateien Kategorien von Texten zusammengefasst werden, z. B. alle GUI-Texte für ein Fenster. Dadurch werden die Dateien nicht zu lang und können schnell geladen werden.

MessageFormat

Mithilfe der Klasse MessageFormat ist es möglich, zusammengesetzte Mitteilungen mit variablen Anteilen sprachneutral zu verwalten.

- Beispiel 2a: Die folgende Mitteilung enthält vier variable Teile:
MitteilungGlobal Am 26. Juli 2010 haben wir um 11:55 Uhr 10 Bücher bei der Buchhandlung "TopBuch" bestellt.

11.1 Globalisierung in Java I

Das Datum ist variabel, die Uhrzeit, die Buchanzahl und der Name der Buchhandlung.

Für diese Mitteilung wird eine `.properties`-Datei mit folgender Schablone angelegt:

`schablone = Am {2,date,long} haben wir um {2,time,short} Uhr {1,number,integer} Bücher bei der Buchhandlung "{0}" bestellt.`

Jeder variable Anteil wird als Argument in {} eingeschlossen. Jedes Argument beginnt mit einer Zahl, die den Index eines Elements in einem Object-Array angibt, in dem sich der zugehörige Argumentwert befindet. Die Argumentzahlen müssen keine bestimmte Reihenfolge einhalten. Die Argumentangaben in diesem Beispiel haben folgende Bedeutung:

- {2,date,long}: Datumsteil eines Date-Objekts.
- {2,time,short}: Zeitteil eines Date-Objekts. Dasselbe Datenobjekt wird sowohl für das Datum als auch für die Zeitangabe verwendet. In dem Object-Array befindet sich das Date-Objekt im Index 2.
- {1,number,integer}: Ein Number-Objekt, dargestellt als integer.
- {0}: Der String in der `.properties`-Datei, der zu dem Schlüssel geschaefbt gehört.

Folgende Methoden der Klasse `MessageFormat` sind wichtig:

`MessageFormat`

- `public void setLocale(Locale locale)`: Legt fest, welche Lokalisierung verwendet werden soll.
- `public void applyPattern(String pattern)`: Diese Methode parst das angegebene Muster und erzeugt eine Liste von Unterformaten für die Formatelemente, die in ihm enthalten sind.
- `public final String format(Object obj)`: Formatiert ein Feld von Objekten.

Das folgende Java-Programm demonstriert die beschriebenen Möglichkeiten:

`Beispiel 2b:
MitteilungGlobal`

`package de.w3l.anw;`

`Java`

```
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.Date;
import java.text.MessageFormat;

public class MitteilungGlobal
{
    static void anzeigenMitteilung(Locale aktuelleLokalisierung)
    {
        System.out.println("Sprache und Land: " +
            aktuelleLokalisierung.toString());
        System.out.println();

        ResourceBundle mitteilungen =
            ResourceBundle.getBundle("de.w3l.anw.res.MitteilungSchablone",
                aktuelleLokalisierung);
```

I 11 Globalisierung von Software

```
Object[] argumenteDerMitteilung =
{
    mitteilungen.getString("geschaeft"), new Integer(10),
    new Date()
};

MessageFormat format = new MessageFormat("");
format.setLocale(aktuelleLokalisierung);

format.applyPattern(mitteilungen.getString("schablone"));
String output = format.format(argumenteDerMitteilung);
System.out.println(output);
}

static public void main(String[] args)
{
    anzeigenMitteilung(new Locale("de", "DE"));
    System.out.println();
    anzeigenMitteilung(new Locale("en", "US"));
}
}
```

Folgende Dateien liegen vor:

```
Datei: MitteilungSchablone_de_DE.properties
Inhalt:
geschaeft = TopBuch
schablone = Am {2,date,long} haben wir um {2,time,short} Uhr
{1,number,integer} Bücher bei der Buchhandlung "{0}" bestellt.
Datei: MitteilungSchablone_en_US.properties
Inhalt:
geschaeft = BestBook
schablone = At {2,time,short} on {2,date,long}, we ordered
{1,number,integer} books at the bookshop "{0}".
```

Das Programm erzeugt folgende Ausgaben:

Sprache und Land: de_DE

Am 26. Juli 2010 haben wir um 12:46 Uhr 10 Bücher bei der
Buchhandlung "TopBuch" bestellt.

Sprache und Land: en_US

At 12:46 PM on July 26, 2010, we ordered 10 books at the
bookshop "BestBook".

Tipp

Mithilfe des kostenlosen Eclipse-Plug-Ins JIntro (<http://www.guh-software.de/eclipse>) ist es möglich, *Resource Bundles* komfortabel zu bearbeiten und zu pflegen. Insbesondere können alle Sprachversionen parallel nebeneinander bearbeitet werden. Ein Codeassistent hilft beim Auswählen von Schlüsseln und beim Anlegen neuer Schlüssel.

11.2 Fallstudie: KV – Globalisiert I

Anwendung des Fassaden-Musters

Da es in Java eine ganze Reihe von Klassen für die Globalisierung gibt, ist es für die praktische Anwendung sinnvoll, diese Klassen hinter einer Fassade (siehe »Das Fassaden-Muster (*facade pattern*)«, S. 69) zu »verstecken«. Die Abb. 11.1-1 zeigt die Klassenstruktur einer möglichen i18n-Fassade.

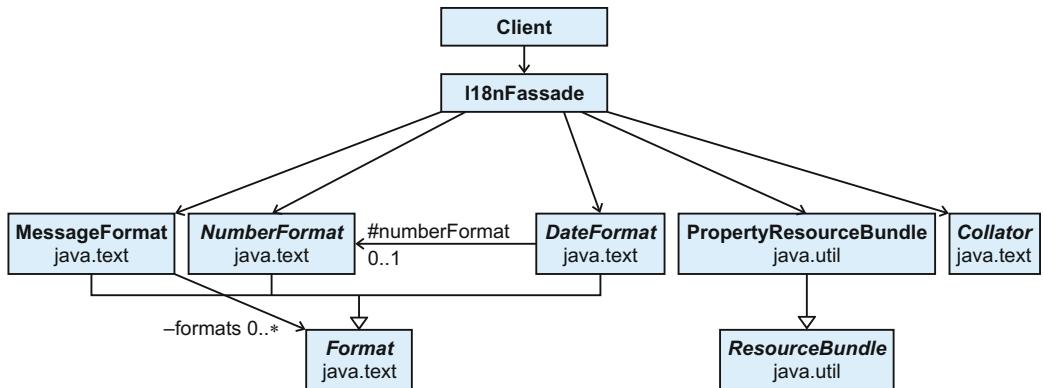


Abb. 11.1-1:
Vorschlag für ein
Fassaden-Muster
i18n.

11.2 Fallstudie: KV – Globalisiert

Die Fallstudie »Kundenverwaltung-Mini« kann durch geeignete Maßnahmen globalisiert werden.

Ein zusätzliches Fenster dient dazu, die Sprache auszuwählen (Abb. 11.2-1).



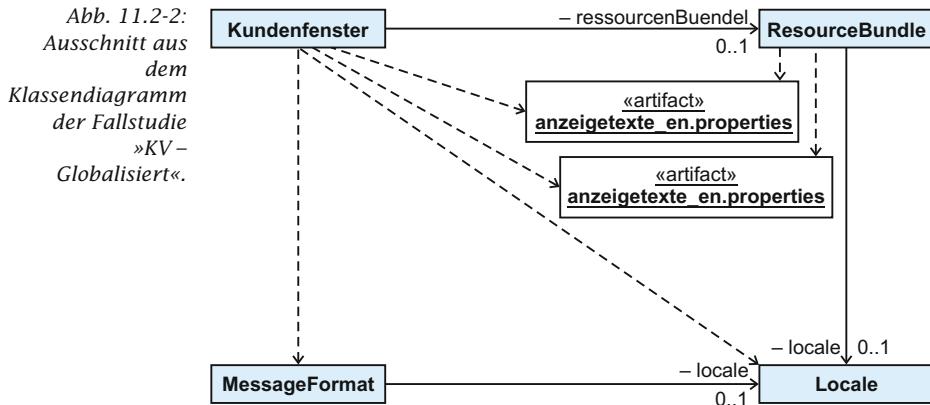
Abb. 11.2-1:
Startfenster zur
Fallstudie »KV –
Globalisiert«.

Die Klasse Kundenfenster wird um mehrere Klassen erweitert (Abb. 11.2-2).

Die Datei anzeigengetexte_de.properties hat folgenden Inhalt:

```
newCustomer = neuer Kunde
title = Kundenverwaltung
searchFunction = Suchfunktion
customerNumber = Kundennr.
search = Suchen
save = Speichern
name = Name
message = Mitteilung
```

I 11 Globalisierung von Software



```

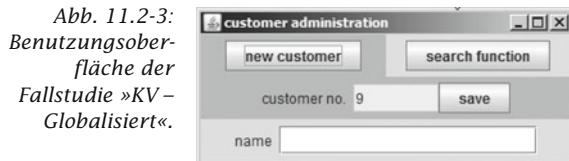
requiredFieldsNotFilledMsg =
  Es sind Mussfelder nicht ausgefuellt!
invalidInput = Fehlerhafte Eingabe!
customerNotFoundMsg =
  Kunde mit Nr. {0} konnte nicht gefunden werden
  
```

Die Datei anzeigetexte_en.properties hat folgenden Inhalt:

```

newCustomer = new customer
title = customer administration
searchFunction = search function
customerNumber = customer no.
search = search
save = save
name = name
message = Message
requiredFieldsNotFilledMsg =
  Some required fields are not filled
invalidInput = Invalid input!
customerNotFoundMsg =
  customer with no. {0} could not be found!
  
```

Das englischsprachliche Fenster zeigt die Abb. 11.2-3.



OOP Das vollständige Programm finden Sie im kostenlosen E-Learning-Kurs zu diesem Buch.

12 Authentifizierung und Autorisierung

In der Regel werden Anwendungen heute von mehreren Benutzern benutzt. Das kann bedeuten, dass bei einer Einzelplatz-Anwendung mehrere Benutzer nacheinander die Anwendung verwenden. Oder dass bei einer verteilten Anwendung mehrere Benutzer nebenläufig oder parallel auf die Anwendung, die auf einem Server installiert ist, zugreifen.

Mehrbenutzerfähigkeit

Überlegen Sie, was eine Mehrbenutzerfähigkeit für die Softwarearchitektur bedeutet. Frage

Soll ein Softwaresystem von mehreren Benutzern benutzt werden, dann hat dies zunächst zur Konsequenz, dass das Softwaresystem über eine **Benutzerverwaltung** verfügen muss. Benutzerverwaltung bedeutet, dass Benutzer sich für das Softwaresystem authentifizieren müssen, z. B. durch ein Login mit Benutzernamen (oder Pseudonym oder E-Mail-Adresse) und Passwort. In dem Subsystem »Benutzungsoberfläche« muss die Authentifizierung beispielsweise durch ein Log-in und eine Passworteingabe ermöglicht werden. Diese Informationen müssen verwaltet und durch einen privilegierten Benutzer (z. B. Administrator) geändert und gelöscht werden können. Da es sich dabei um sehr sensible Daten handelt, müssen sie verschlüsselt gespeichert und gegen Hacker-Angriffe geschützt werden.

Befindet sich das Softwaresystem in einem Kontext, in dem die weitgehend gleichen Benutzer verschiedene Softwaresysteme benutzen, dann ist es dem einzelnen Benutzer nicht zuzumuten, sich für jedes benutzte System einzeln zu authentifizieren. Eine Lösung dafür ist das sogenannte **Single Sign-On** (SSO) – auch Einmalanmeldung genannt.

Der Benutzer authentifiziert sich einmal und kann dann ohne erneute Anmeldung auf alle Systeme zugreifen, für die er eine Berechtigung besitzt.

Gibt es in einem Unternehmen bereits eine SSO-Verwaltung, dann muss das neue System die Fähigkeit besitzen, mit dieser SSO-Verwaltung geeignet zu kommunizieren.

Darf jeder Benutzer in einem Softwaresystem alle Funktionen benutzen? Frage

I 12 Authentifizierung und Autorisierung

| | |
|----------------|---|
| Antwort | In der Regel nein. Je nach Arbeitsbereich und Kompetenz dürfen Benutzer mehr oder weniger Funktionen benutzen bzw. auf Daten zugreifen. Beispielsweise dürfen Gehälter nur von wenigen Mitarbeitern der Personalabteilung eingesehen werden. Eine Mehrbenutzerfähigkeit erfordert daher i. Allg. eine Rechteverwaltung, die es ermöglicht, Rollen definierte Berechtigungsprofile zuzuordnen. |
| Fazit | Wird eine Mehrbenutzerfähigkeit gefordert, dann muss geprüft werden, ob eine Benutzerverwaltung und eine Rechteverwaltung selbst erstellt oder erworben werden können oder in der Infrastruktur bereits vorhanden sind. Ist ein Single Sign-On erforderlich, dann muss eine Schnittstelle dafür bereitgestellt werden. Die Benutzerverwaltung erfordert eine Authentifizierung und die Rechteverwaltung eine Autorisierung des jeweiligen Benutzers. |
| Abhängigkeiten | Die Rechteverwaltung steht im Zusammenhang mit Sicherheitssystemen zur Verwaltung von Zugriffsrechten (siehe auch »Betriebssicherheit und Funktionssicherheit«, S. 121). |

Authentisierung, Authentifizierung und Autorisierung

| | |
|-----------------|---|
| Authentisierung | Will ein Benutzer ein Anwendungssystem benutzen, dann muss er sich authentisieren . Dies geschieht in der Regel durch die Angabe eines Benutzernamens (<i>user id</i>) und eines Passwortes. Das Anwendungssystem führt nach der Authentisierung durch den Benutzer eine Authentifizierung durch, indem es prüft, ob der Benutzernname und das Passwort mit entsprechenden gespeicherten Daten übereinstimmen. Nach der Authentisierung, d. h. nachdem der Benutzer glaubwürdig seine Identität angegeben hat, folgt also anschließend die Authentifizierung, d. h. die Überprüfung der Echtheit der Identität. Beide Begriffe werden oft nicht auseinandergehalten. Im Englischen gibt es dafür nur den Begriff <i>Authentification</i> . |
| Autorisierung | In der Regel erhalten Benutzer, die authentifiziert wurden, nicht alle die gleichen Zugriffsrechte innerhalb eines Anwendungssystems. Um nicht für jeden einzelnen Benutzer individuelle Zugriffsrechte in einem Anwendungssystem festlegen zu müssen, werden Benutzern Rollen zugeordnet. Eine Rolle fasst dabei eine Reihe von Zugriffsrechten zu einer Einheit zusammen. Benutzer, die einer bestimmten Rolle zugeordnet sind, sind dann autorisiert , entsprechende Zugriffe durchzuführen. |

Entwurfsprinzipien

Für die **Autorisierung** (*authorization*) werden eine Reihe von Prinzipien vorgeschlagen [SaSc75, S. 1282 f.]:

- **Prinzip der ökonomischen Sicherheitsmechanismen** (*Economy of Mechanism*): Die Sicherheitsmechanismen sollen so einfach und übersichtlich sein, wie möglich. Dadurch gibt es weniger Möglichkeiten für Fehler und die Qualitätssicherung wird vereinfacht.
- **Prinzip der sicheren Voreinstellungen** (*Fail-Safe Defaults*): Der Benutzer hat nur dann ein Zugriffsrecht, wenn er dieses Zugriffsrecht ausdrücklich zugewiesen bekommen hat. Ohne Rechtezuweisung besitzt er keine Rechte.
- **Prinzip der vollständigen Zugriffsüberprüfung** (*Complete Mediation*): Alle Zugriffe müssen bei jedem Zugriffsversuch geprüft werden. Ein Zwischenspeichern (*caching*) von Rechten ist zu vermeiden. Nach einer Rechteänderung wird sonst noch nach der alten Rechtevergabe verfahren.
- **Prinzip des offenen Entwurfs** (*Open Design*): Der Sicherheitsmechanismus darf nicht von der Geheimhaltung des Entwurfs abhängen. Unberührt hiervon sind jedoch geheime Schlüssel. Erfahrungen haben gezeigt, dass keine Sicherheit dadurch entsteht, dass über den Sicherheitsmechanismus Unklarheit besteht (*security through obscurity*). Angreifer sind schnell in der Lage, sich Klarheit über den Sicherheitsmechanismus zu verschaffen.
- **Prinzip der Aufteilung von Privilegien** (*Separation of Privilege*): Ein Schutzmechanismus, der zwei Schlüssel zum Aufschließen benötigt, ist robuster und flexibler als ein Schutzmechanismus, der nur einen einfachen Schlüssel benötigt. Dieses Prinzip wird oft bei Bankschließfächern benutzt. In Softwaresystemen sollten getrennte Schlüssel immer dann verwendet werden, wenn mehr als zwei Bedingungen erfüllt sein müssen, um einen Zugriff zu erlauben.
- **Prinzip des kleinsten Privilegs** (*Least Privilege*): Der Benutzer soll nur über die Rechte verfügen, die für die Erfüllung seiner Aufgabe tatsächlich zwingend notwendig sind.
- **Prinzip der psychologischen Akzeptanz** (*Psychological Acceptability*): Die Bedienung soll durch die Zugangskontrolle nicht komplizierter werden, als sie es ohne Zugangskontrolle wäre.
- **Prinzip des kleinsten allgemeinen Mechanismus** (*Least Common Mechanism*): Jeder Mechanismus, der für alle Benutzer gilt, ist schwerer zu realisieren als derjenige, der nur für einen oder einige Benutzer gilt.

Für die Authentifizierung und Autorisierung gibt es eine ganze Reihe von Entwurfsmustern:

- »A & A- Entwurfsmuster«, S. 156

Java unterstützt die Authentifizierung und Autorisierung durch ein eigenes Framework:

- »JAAS«, S. 161

I 12 Authentifizierung und Autorisierung

Die Fallstudie »Kundenverwaltung« zeigt, welcher Aufwand notwendig ist, um eine Authentifizierung und Autorisierung vorzunehmen:
■ »Fallstudie: KV – JAAS«, S. 167

12.1 A & A-Entwurfsmuster

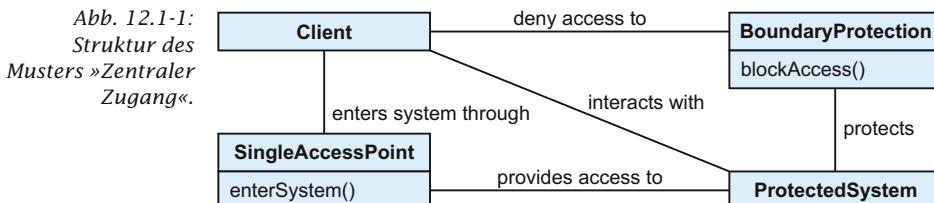
Die Entwurfsmuster zur Authentifizierung und Autorisierung gehören zur Gruppe der *Security Patterns*. Über 20 Muster werden in [SNL06, S. 479 ff.] vorgestellt, über 40 Muster in [SFH+06]. Im Folgenden werden einige dieser Muster beschrieben:

- Das zentrale Zugangsmuster (*Single Access Point Pattern*)
- Das Checkpoint-Muster (*Check Point*)
- Das rollenbasierte Zugangssteuerungs-Muster (*Role-Based Access Control*)
- Das Referenzmonitor-Muster (*Reference Monitor*)

Das zentrale Zugangsmuster (*Single Access Point Pattern*)

Name(n): Auch als *Single Point of Entry*, *One Way In* oder in konkreten Ausprägungen *Login Window*, *Guard Door* oder *Validation Screen* bezeichnet.

Grundidee: Alle Benutzer, die ein Anwendungssystem benutzen wollen, müssen eine zentrale Zugangspforte (*single access point*) benutzen, die den Zugang gestattet oder ablehnt. Die zentrale Zugangspforte stößt die Authentifizierung an bzw. delegiert diese. Die Abb. 12.1-1 zeigt das zugehörige UML-Klassendiagramm.



BoundaryProtection sorgt dafür, dass die zentrale Zugangspforte zu der Anwendung *nicht* umgangen werden kann. Einen regulären Zugang zu einer Anwendung beschreibt das Sequenzdiagramm der Abb. 12.1-2.

Der Benutzer – oder ein anderes Softwaresystem – führt ein Login bei der zentralen Zugangspforte durch und betritt die geschützte Anwendung. Es muss sicher sein, dass es keine »Hintertüren« (*back doors*) gibt, um die Anwendung zu betreten. Z. B. dürfen bei einem Firewall nur Ports freigegeben werden, die aktuell benötigt werden.

12.1 A & A-Entwurfsmuster I

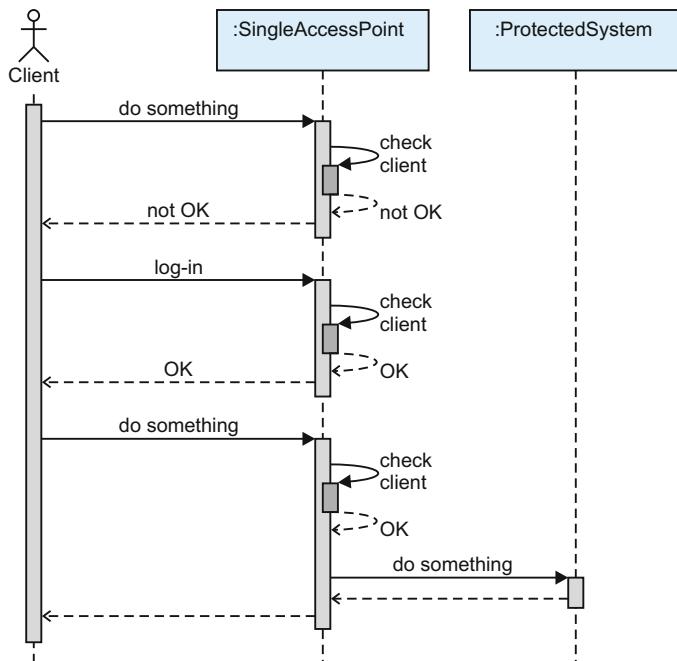


Abb. 12.1-2:
Sequenzdiagramm
zum »Zentrales
Zugangsmuster«.

Varianten: Das *Front Controller Pattern* sorgt in einer Webanwendung dafür, dass alle Anfragen an eine einzige, zentrale Komponente geleitet werden.

Zusammenhänge: Die innere Systemstruktur wird einfacher, da eine wiederholte Autorisierung vermieden wird. Dadurch wird die Wartbarkeit erleichtert (siehe »Wartbarkeit«, S. 116). Die Sicherheit wird erhöht, da nur ein Zugang zum System möglich ist (siehe »Betriebssicherheit und Funktionssicherheit«, S. 121). Ist die zentrale Zugangspforte jedoch fehlerhaft, dann kann das System nicht benutzt werden. Das A & A-Entwurfsmuster *Checkpoint* zeigt, wie ein zentraler Zugang flexibel und effektiv implementiert werden kann.

[?, S. 279 ff.]

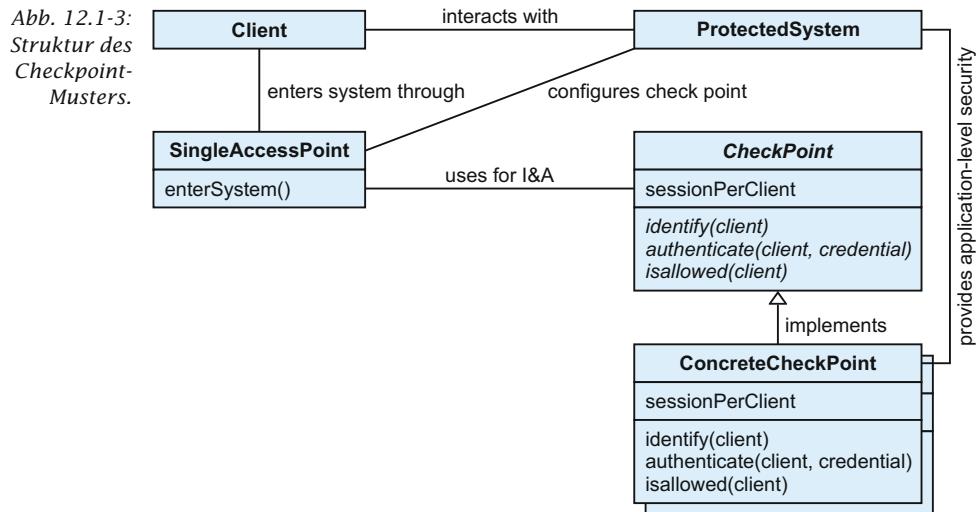
Literatur

Das Checkpoint-Muster (*Check Point*)

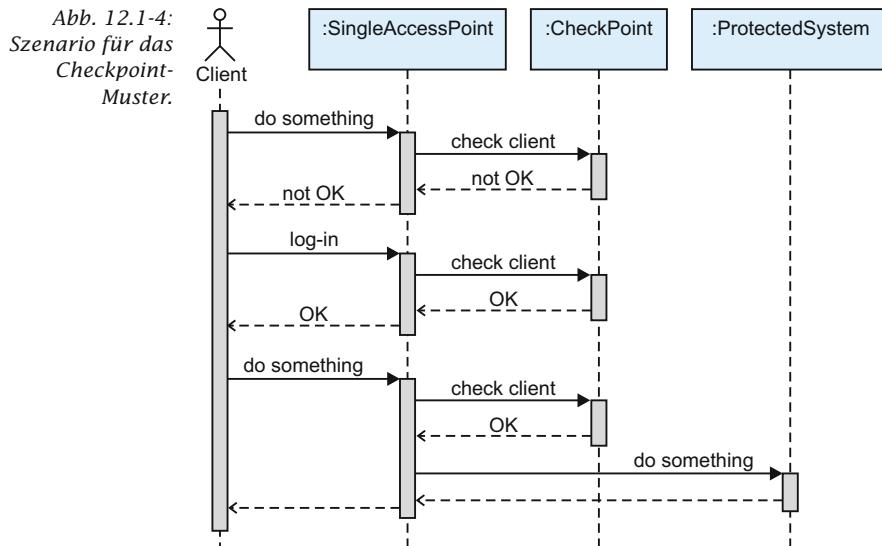
Name(n): Pluggable Authentication, Policy Definition Point (PDP), Policy Enforcement Point (PEP), Access Verification, Holding off hackers, Validation and Penalization.

Grundidee: An einer einzigen Stelle soll eine Authentifizierung erfolgen. Die Verfahren zur Authentifizierung sollen dabei flexibel steuerbar und austauschbar sein. Die Abb. 12.1-3 zeigt das zugehörige UML-Klassendiagramm.

I 12 Authentifizierung und Autorisierung



Das Sequenzdiagramm der Abb. 12.1-4 zeigt, wie eine zentrale Zugangspforte (siehe zentrales Zugangsmuster) eine Checkpoint-Implementierung benutzt, um den Benutzer zu identifizieren und zu authentifizieren.



Zusammenhänge: Das rollenbasierte Zugangssteuerungs-Muster wird oft benutzt, um die Checkpoint-Sicherheitsüberprüfungen durchzuführen (siehe unten).

Das *Framework JAAS* in Java (»JAAS«, S. 161) erlaubt es, verschiedene Module zur Realisierung unterschiedlicher Authentifizierung-Strategien flexibel und »steckbar« einzusetzen – PAM (*Pluggable Authentication Modules*) genannt.

Die Sicherheit des Systems wird erhöht (siehe auch »Betriebssicherheit und Funktionssicherheit«, S. 121), außerdem die Weiterentwickelbarkeit durch die Realisierung des PAM-Konzepts (siehe auch »Weiterentwickelbarkeit«, S. 119).

[?, S. 287 ff.]

Literatur

Das rollenbasierte Zugangssteuerungs-Muster (*Role-Based Access Control*)

Name(n): Für den englischen Begriff wird oft die Abkürzung RBAC verwendet.

Grundidee: Nicht jeder Benutzer erhält individuelle Zugriffsrechte, sondern Benutzer, die dieselben Aufgaben zu erledigen haben, werden zu Rollen zusammengefasst. Den Rollen werden Zugriffsrechte zugeordnet. Dadurch wird die Verwaltung der Zugriffsrechte wesentlich vereinfacht. Die Abb. 12.1-5 zeigt das zugehörige UML-Klassendiagramm.

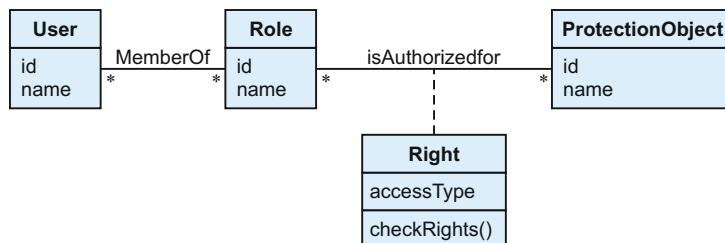


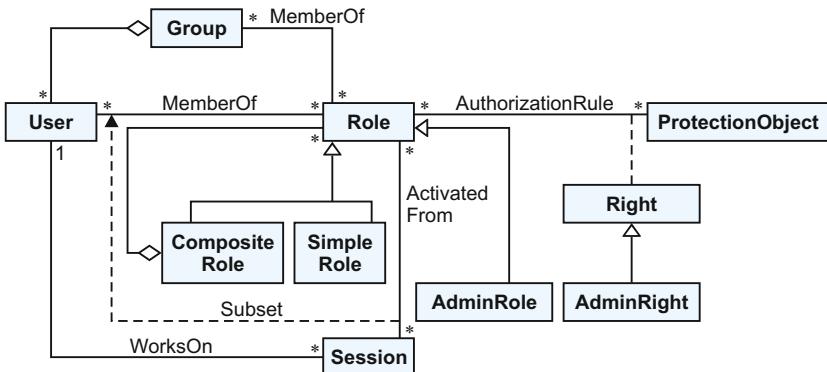
Abb. 12.1-5:
Struktur des
Musters
»Rollenbasierte
Zugangssteue-
rung«.

Die Klassen *User* und *Role* beschreiben die registrierten Benutzer und die vordefinierten Rollen. Benutzer werden Rollen zugeordnet. Die Rollen erhalten Zugriffsrechte in Abhängigkeit von ihren Funktionen. Die assoziative Klasse *Right* definiert die Zugriffstypen, die einen Benutzer innerhalb einer Rolle dazu autorisiert, ein geschütztes Objekt (*ProtectionObject*) zu nutzen.

Varianten: Die Abb. 12.1-6 zeigt einen erweiterten Grundmodell mit zusammengesetzten Rollen und der Abtrennung der Administration von den anderen Rechten (Trennung der Zuständigkeiten). Der Administrator hat das Recht, Rollen zu Gruppen und Benutzern zuzuordnen. Er ist ein spezieller Benutzer der Autorisierungsregeln für Rollen definieren, Benutzergruppen anlegen und löschen so wie Benutzer zu Rollen zuordnen kann. Zusätzlich werden Benutzer noch Sitzungen (Session) zugeordnet.

I 12 Authentifizierung und Autorisierung

Abb. 12.1-6:
Erweitertes Muster
der
rollenbasierten Zu-
gangssteuerung.



Zusammenhänge: Das RBAC-Muster verbessert die Sicherheit (siehe auch »Betriebssicherheit und Funktionssicherheit«, S. 121), er schwert aber die Wartbarkeit, da die konzeptionelle Komplexität erhöht wird (siehe auch »Wartbarkeit«, S. 116).

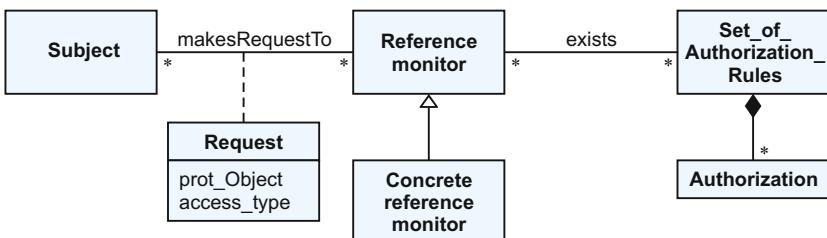
Literatur [?, S. 249 ff.]

Das Referenzmonitor-Muster (Reference Monitor)

Name(n): Policy Enforcement Point.

Grundidee: Die Einhaltung spezifizierter Autorisierungsregeln muss erzwungen werden. Dies kann dadurch erfolgen, dass alle Anfragen für die Benutzung von Ressourcen abgefangen und auf Übereinstimmung mit den Autorisierungen überprüft werden. Die Abb. 12.1-7 zeigt das zugehörige UML-Klassendiagramm.

Abb. 12.1-7:
Struktur des
Musters »Referenz-
monitor«.



Das Sequenzdiagramm der Abb. 12.1-8 zeigt, wie eine Anfrage von einem Prozess geprüft wird. Der Referenzmonitor prüft, ob eine Regel vorhanden ist, die die Anfrage autorisiert. Wenn eine existiert, dann wird es der Anfrage erlaubt fortzufahren.

Zusammenhänge: Dieses Muster ist ein Spezialfall des Checkpoint-Musters (siehe oben).

Der Java Security Manager ist ein Beispiel, für die Implementierung dieses Konzepts (siehe »JAAS«, S. 161).

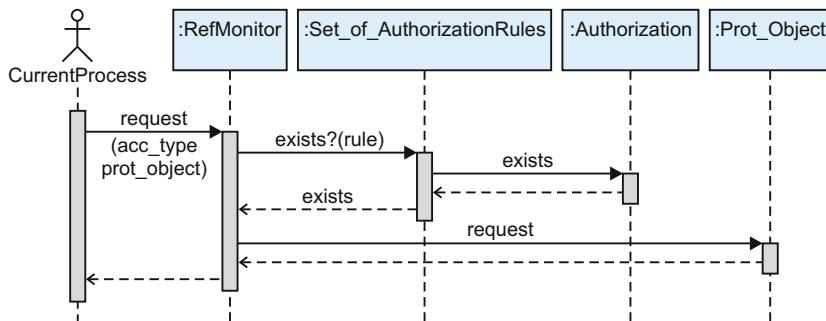


Abb. 12.1-8:
Sequenzdiagramm
zum Muster »Refe-
renzmonitor«.

Die Überprüfung jeder Anfrage kann zu einem Leistungs- und Effizienzverlust führen (siehe auch »Leistung und Effizienz«, S. 128). Einige Überprüfungen sollten daher zur Übersetzungszeit erfolgen und *nicht* jedes Mal zur Laufzeit wiederholt werden.

[?, S. 256 ff.]

Literatur

12.2 JAAS

Mit JAAS (*Java Authentication and Authorization Service*) stellt Java in dem Paket `javax.security.auth` ein *Framework* für die Authentifizierung und Autorisierung zur Verfügung.

Java

Dem Benutzer oder anderen Systemen, die eine Anwendung benutzen (in JAAS *Subject* genannt), werden eine oder mehrere authentifizierte Subjektkennungen (in JAAS als *Principals* bezeichnet) zugeordnet.

Subject &
Principals

Das *Subject* Franz Meyer kann zwei *Principals* haben: Die Nummer des Personalausweises 5677949880D und die Nummer des Führerscheins J48999VPX62. Beide Nummern werden an das *Subject* Franz Meyer gebunden, d. h. beide *Principals* referenzieren dasselbe *Subject*, obwohl sie unterschiedlich sind.

Beispiel

Ein Subjekt kann also eine Menge von *Principals* haben:

```

public final class Subject
{
    //Liefert die Menge der Principals,
    //die dem Subject zugeordnet sind
    public Set getPrincipals() { }
}
  
```

Principals können einem *Subject* nach einer erfolgreichen Authentifizierung zugeordnet werden.

Ein *Subject* kann zusätzlich sicherheitsrelevante Attribute besitzen, in Java *Credentials* (Empfehlungsschreiben, Beglaubigungsschreiben, Zeugnisse) genannt. Ein *Credential* kann Informationen enthalten, die dazu dienen, ein Subjekt für einen neuen Service zu

Credentials

I 12 Authentifizierung und Autorisierung

authentifizieren. Solche *Credentials* können Passwörter und öffentliche Schlüsselzertifikate enthalten. Sie werden in Umgebungen benutzt, die *Single Sign-On* einsetzen.

Die wichtigsten Klassen und Schnittstellen von JAAS zeigt die Abb. 12.2-1.

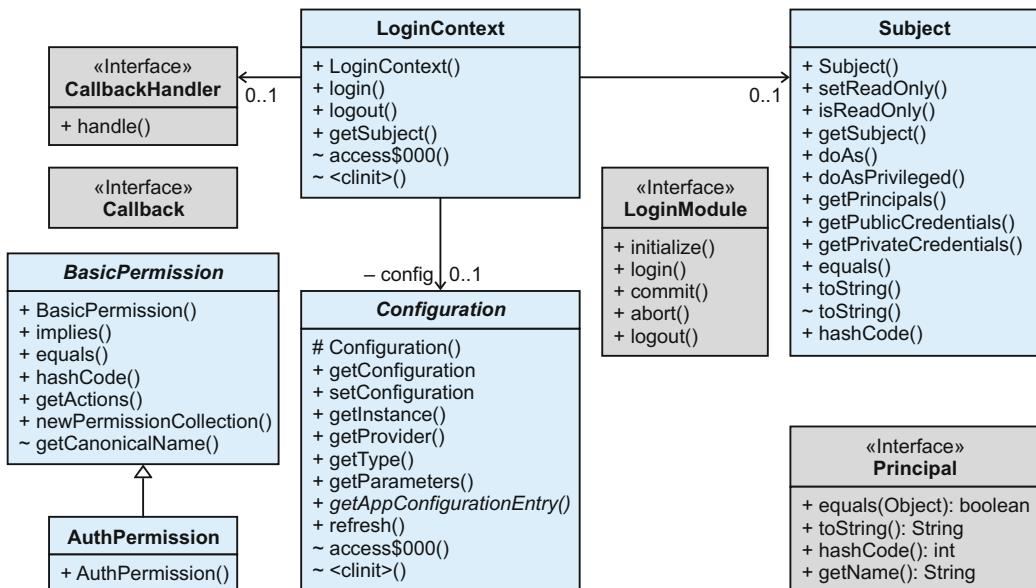


Abb. 12.2-1: JAAS-Framework-Klassen und -Schnittstellen.

Authentifizierung

JAAS ermöglicht eine flexible Authentifizierung über auswechselbare und kaskadierbare Authentifizierungsmodule (Login-Module) nach dem PAM-Prinzip (*Pluggable Authentication Modules*).

In einer JAAS-Konfigurationsdatei werden die Login-Module aufgeführt. Die Konfigurationsdatei hat folgenden Aufbau:

```

<entry name>
{
    <LoginModule> <flag> <LoginModule options>;
    <LoginModule> <flag> <LoginModule options>;
}
  
```

- <entry name> ist der Name der für die Authentifizierung zu wendenden Konfiguration. Sie definiert ein oder mehrere Login-Module, die für die Authentifizierung verwendet werden sollen. Enthält eine Konfiguration mehrerer Login-Module, dann entscheidet die Reihenfolge über den Ablauf der gesamten Authentifizierung.

- <LoginModule> ist der voll qualifizierte Klassenname des jeweiligen Login-Moduls, der für die Authentifizierung verwendet werden soll und der die Schnittstelle javax.security.auth.spi.LoginModule implementiert.
- <flag> gibt an, was passieren soll, falls die Authentifizierung fehlgeschlägt.
 - required legt fest, dass das betreffende Login-Modul erfolgreich authentifiziert werden muss, damit die gesamte Authentifizierung der gewählten Konfiguration erfolgreich ist. Der Authentifizierungsprozess wird für die anderen Login-Module jedoch fortgesetzt.
 - requisite spezifiziert, dass das betreffende Login-Modul erfolgreich authentifiziert werden muss, damit die gesamte Authentifizierung erfolgreich ist. Die Authentifizierung endet jedoch im Fehlerfall.
 - sufficient legt fest, dass die gesamte Authentifizierung beendet werden kann, wenn die Authentifizierung des betreffenden Login-Moduls erfolgreich war. Vorausgehende required- und requisite-Module müssen im Erfolgsfall für die gesamte Konfiguration allerdings auch erfolgreich gewesen sein.
 - optional gibt an, dass ein solcher Modul für die Authentifizierung nicht entscheidend ist. Mindestens ein optional- oder sufficient-Modul muss erfolgreich gewesen sein, wenn keine required- oder requisite-Module vorhanden sind.
- Mit <LoginModule options> wird eine Werteliste (name=value-Paare) an das Modul durchgereicht.

Damit die Konfigurationsdatei berücksichtigt werden kann, muss sie der Anwendung bekannt gemacht werden. Dafür gibt es folgende Möglichkeiten:

- Sie liegt in JAVA_HOME/jre/lib/security.
- Angabe der Konfigurationsdatei über die Kommandozeile beim Start der Anwendung: java -D java.security.auth.login.config==jaas.config LoginContext
- Der Pfadname der Konfigurationsdatei wird in die Anwendung eingelesen.

Das »Steckbare« (*pluggable*) in JAAS besteht in der Angabe der jeweiligen Konfigurationsdatei beim Initialisieren der Klasse LoginContext (siehe unten).

```
WindowsLogin
{
    //Für Windows:
    com.sun.security.auth.module.NTLoginModule required debug=false;
};
```

Beispiel:

I 12 Authentifizierung und Autorisierung

In diesem Beispiel wird ein Login-Modul eingebunden, das standardmäßig in Java enthalten ist, und Berechtigungen für die Windows-ID ausstellt, unter der das Programm ausgeführt wird.

Wie das Beispiel zeigt, wird ein *Single Sign-On* dadurch unterstützt, dass die Kennung des Betriebssystems-Benutzers ohne weitere Interaktion an die Anwendung »durchgereicht« werden kann.

LoginContext

Die Klasse LoginContext aus dem Paket javax.security.auth.login stellt die Verbindung zwischen JAAS und der Anwendung, die JAAS benutzt, her. Über diese Klasse kann später auf den authentifizierten Benutzer (getSubject()) zugegriffen werden. Das Codegerüst sieht wie folgt aus:

```
public static void main(String[] args)
{
    LoginContext lc = null;
    try
    {
        lc = new LoginContext("<entry name>",
            new MeinCallbackHandler());
        lc.login();
    }
    catch (LoginException)
    {
        //Authentifizierung fehlgeschlagen
    }
    //Authentifizierung erfolgreich
    //Es kann auf den zurückgegebenen Benutzer (Subject)
    //zugegriffen werden, um Autorisierungen auszuführen
    Subject benutzer = lc.getSubject();
    Subject.doAs(benutzer, new MyPrivilegedAction());
}
```

<entry name> gibt den Namen einer Konfiguration an (siehe oben).

Callback Handler

MeinCallbackHandler ist eine Klasse, die die Schnittstelle CallbackHandler implementiert. Ein CallbackHandler dient dazu, die Authentifizierungsdaten aufzusammeln oder zu verarbeiten. Er entkoppelt die Eingabe der Benutzerdaten vom eigentlichen LoginModul. Innerhalb der Methode handle() müssen die entsprechenden Daten abgefragt werden.

LoginModule

Die Klasse, die die Authentifizierung durchführt, muss die Schnittstelle LoginModule implementieren. Die Schnittstelle besteht aus folgenden Methoden:

- **initialize()**: Durchführung von Initialisierungsarbeiten.
- **login()**: Login-Informationen werden ermittelt, eventuell werden Benutzereingaben über *Callbacks* abgefragt.

- `abort()` und `commit()`: In `commit()` müssen dem Benutzer (*subject*) ein oder mehrere Berechtigungs-Exemplare hinzugefügt werden (*principals*). `abort()` wird für Aufräumarbeiten, wie dem Zurücksetzen von Datenelementen, verwendet. Schlägt ein Login fehl, dann wird `abort()` aufgerufen.
- `logout()`: Die zugeteilten Berechtigungs-Exemplare werden wieder entzogen.

Nach der Implementierung kann die Klasse in die JAAS-Konfigurationsdatei eingebunden werden.

Folgende Schritte sind notwendig, um eine Authentifizierung Was ist zu tun? durchzuführen:

- 1 Erstellen oder verwenden einer oder mehrerer Login-Module.
- 2 Eintrag der Login-Module in eine JAAS-Konfigurationsdatei.
- 3 Erstellung eines Objekts vom Typ `LoginContext`.
- 4 Erstellung einer Klasse `MeinCallbackhandler`, der die Schnittstelle `Callbackhandler` implementiert.

Der dynamische Ablauf sieht wie folgt aus:

- 1 Der `LoginContext` sucht bei der Initialisierung nach der Konfigurationsdatei, um die zu ladenden `LoginModule` und ihre Optionen zu bestimmen.
- 2 Beim eigentlichen Login-Vorgang ruft der `LoginContext` die `login()`-Methode jedes `LoginModules` auf.
- 3 Jede `login()`-Methode führt die Authentifizierung durch, u. U. unter Nutzung des `CallbackHandler`.
- 4 Der `CallbackHandler` benutzt ein oder mehrere *Callbacks*, um vom Benutzer die notwendigen Daten zu erhalten.
- 5 Ein neues Subject-Exemplar wird mit den Daten über die Authentifizierung gefüllt.

Autorisierung

Wurde die Authentifizierung erfolgreich durchgeführt, dann stellt JAAS Verfahren zur Verfügung, um eine Zugriffssteuerung anhand der *Principals*, die mit dem authentifiziertem Subjekt assoziiert sind, durchzuführen.

In JAAS können Rollen als benannte *Principals* verwendet werden. Rollen

Der Rolle »Administrator« kann erlaubt werden, Benutzerpasswörter Beispiel zuzuordnen:

```
grant Principal foo.Role "administrator"
{
    permission java.io.FilePermission
    "/passwords/-", "read, write";
}
```

I 12 Authentifizierung und Autorisierung

Zugriffskontrolle Über die Klasse SecurityManager kontrolliert das Java-Laufzeitsystem die Ausführung von Methoden. Der SecurityManager delegiert die Verantwortung an die Klasse AccessController. Diese Klasse wiederum besorgt sich über die Klasse AccessControlContext den aktuellen Kontext und prüft, ob ausreichende Erlaubnisse für die Durchführung der Methode vorhanden sind.

JAAS ergänzt diese Architektur durch die Methode doAs() der Klasse Subject, um dynamisch ein authentifiziertes Subjekt, z.B. einen Benutzer, mit dem aktuellen Zugriffskontext zu verknüpfen. Von hier an kann der AccessController seine Entscheidungen entsprechend dem auszuführenden Code selbst oder anhand der *Principals*, die mit dem Subjekt assoziiert sind, treffen:

```
public final class Subject
{
    ...
    //Das Subjekt mit dem aktuellen AccessControlContext
    //verknüpfen und die Aktion ausführen
    public static Object doAs(Subject s,
        java.security.PrivilegedAction action) {}
}
```

Die doAs()-Methode stellt die Verbindung des Subjekts mit dem aktuellen AccessControlContext her und führt dann die Methode aus. Wenn die Sicherheitsüberprüfungen während der Ausführung vorgenommen werden, dann fragt der SecurityManager die JAAS-Richtlinien (*policies*) ab, aktualisiert den AccessControlContext mit den Erlaubnissen, die dem Subjekt und dem auszuführenden Quellcode zugeordnet sind, und führt dann die regulären Erlaubnisüberprüfungen durch. Wenn die Methode beendet ist, dann entfernt die doAs()-Methode das Subjekt aus dem aktuellen AccessControlContext und gibt das Ergebnis an den Aufrufer zurück.

Vor- und Nachteile

- + Flexible Zuordnung von Login-Modulen in der Konfigurationsdatei ohne neue Übersetzung von Java-Programmen (*Pluggable Authentication Modules*).
- Hat ein Benutzer Zugriff auf die *Class-* und *Property*-Dateien eines JAAS-Programms, dann kann er die verwendeten Provider-Module leicht austauschen und so von einem Programm vorgesehene Sicherheitsbeschränkungen unterlaufen. Daher wird JAAS heute im Wesentlichen auf Servern verwendet.

Literatur [Berg04], [LGK+99]

12.3 Fallstudie: KV – JAAS

Um die Fallstudie »Kundenverwaltung-Mini« um eine Authentifizierung und Autorisierung zu ergänzen, ist ein erheblicher Mehraufwand nötig. Ein zusätzliches Fenster ist für die Eingabe des Benutzernamens und das Passworts erforderlich (Abb. 12.3-1).

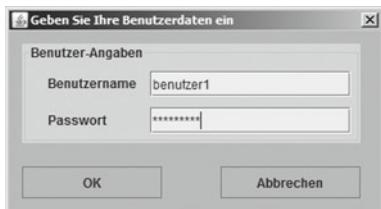
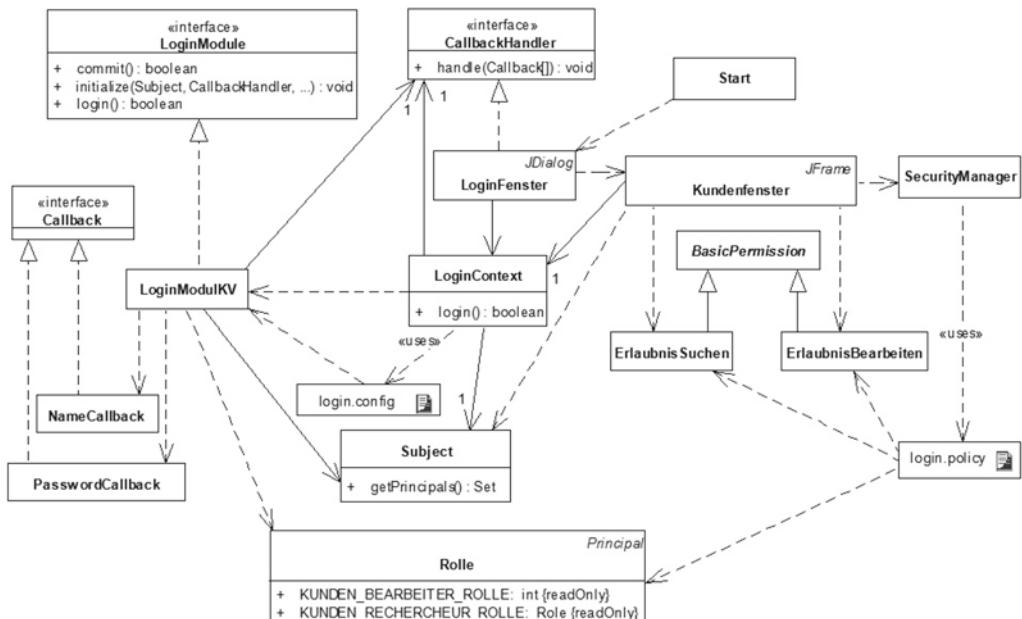


Abb. 12.3-1: Login-Fenster bei der Fallstudie »KV – JAAS«.

Das Klassendiagramm mit den wichtigsten benötigten Klassen zeigt die Abb. 12.3-2.



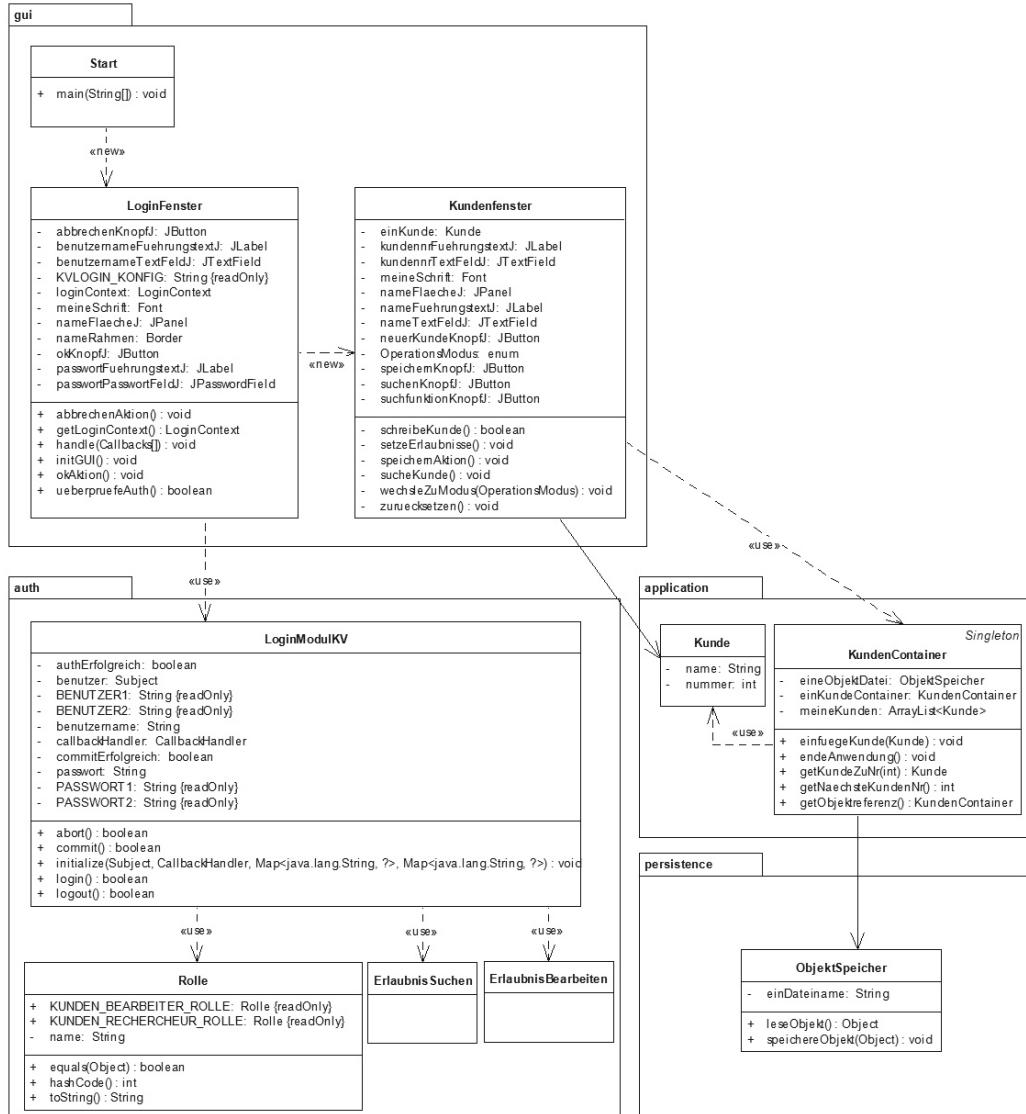
Das gesamte OOD-Modell zeigt die Abb. 12.3-3. Die Klassen **Kunde**, **KundenContainer** und **ObjektSpeicher** bleiben unverändert.

Die Dynamik zeigt die Abb. 12.3-4 anhand eines Sequenzdiagramms.

Nach dem Start der Anwendung wird aus der Klasse **Start** heraus ein neues Fenster für die Authentifizierungsangaben (Benutzernamen, Passwort, Abb. 12.3-1) erstellt:

Abb. 12.3-2:
Klassendiagramm mit den wichtigsten Klassen der Fallstudie »KV – JAAS«.

I 12 Authentifizierung und Autorisierung

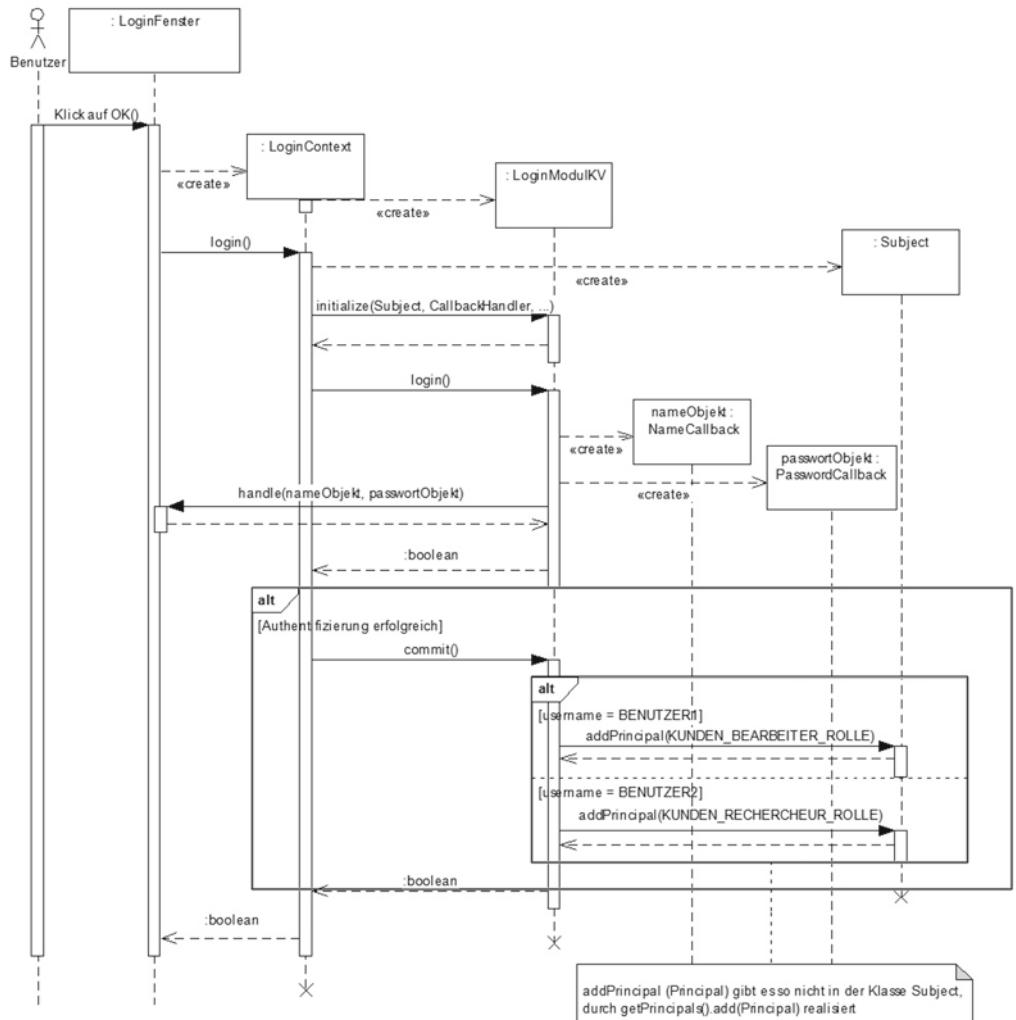


```

Abb. 12.3-3: OOD- //Konstruktor
Klassendiagramm public Start()
der Fallstudie {
    »KV-JAAS«. SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        {
            new LoginFenster().setVisible(true);
        }
    });
}

```

12.3 Fallstudie: KV – JAAS I



Klickt der Benutzer nach Eingabe seiner Daten auf den OK-Druckknopf (Abb. 12.3-4), dann werden die Angaben in der Klasse LoginFenster überprüft (ueberpruefeAuth()). Es wird ein Objekt der Klasse LoginContext erzeugt. An den Konstruktor dieser Klasse wird der Name der Login-Konfiguration übergeben, die in der Datei login.config spezifiziert ist.

Abb. 12.3-4:
Sequenzdiagramm
zur Fallstudie
»KV – JAAS«.

```
private boolean ueberpruefeAuth()
{
    try
    {
        loginContext = new LoginContext(KVLOGIN_KONFIG, this);
        loginContext.login();
        return true; //nur hier, wenn keine Exception auftrat
```

I 12 Authentifizierung und Autorisierung

```
        }
        //... LoginException und SecurityException abfangen ...
    }
```

Die Datei login.config hat folgenden Inhalt:

```
KVLoginKonfig {auth.LoginModulKV required debug=true;};
```

LoginContext liest aus dieser Datei die Angaben zu der Login-Konfiguration und erzeugt ein Objekt der Klasse LoginModulKV. Auch übergibt LoginFenster dem LoginContext-Konstruktor als zweiten Parameter sich selbst als CallbackHandler. LoginFenster ruft auf dem nun erzeugten LoginContext die login()-Methode auf (nicht zu verwechseln mit der login()-Methode der Klasse LoginModulKV). In dieser Methode wird zunächst ein Subject-Objekt erzeugt. Dieses Objekt steht für die Person, für die die Anmeldung durchgeführt wird. Es wird dann die initialize()-Methode von der Klasse LoginModulKV aufgerufen. Die ersten beiden Parameter dabei sind das erzeugte Subject-Objekt und der CallbackHandler, das ist das LoginFenster:

```
//Klasse: LoginModulKV
public void initialize(Subject benutzer,
    CallbackHandler callbackHandler, Map<java.lang.String, ?>
    sharedState, Map<java.lang.String, ?> options)
{
    this.benutzer = benutzer;
    this.callbackHandler = callbackHandler;
}
```

Danach ruft LoginContext die login()-Methode der Klasse LoginModulKV auf. Hier werden zwei *Callbacks* erzeugt, ein NameCallback und ein PasswordCallback. Diese dienen als Ablage für Name und Passwort des Benutzers:

```
//Klasse: LoginModulKV
public boolean login() throws LoginException
{
    //Frage nach Benutzername und Passwort
    if (callbackHandler == null)
        throw new LoginException("Fehler:
            Kein CallbackHandler verfügbar " +
            "zum Abfragen von Authentifizierungs-Info vom Benutzer");

    Callback[] callbacks = new Callback[2];
    callbacks[0] = new NameCallback("user name: ");
    callbacks[1] = new PasswordCallback("password: ", false);

    try
    {
        callbackHandler.handle(callbacks);
        benutzername = ((NameCallback)callbacks[0]).getName();
        char[] tmpPasswort =
            ((PasswordCallback)callbacks[1]).getPassword();
    }
}
```

12.3 Fallstudie: KV – JAAS I

```
if (tmpPasswort == null)
{
    //Behandle NULL-Passwort wie ein leeres Passwort
    tmpPasswort = new char[0];
}
passwort = new String(tmpPasswort);
((PasswordCallback)callbacks[1]).clearPassword();
}
//...IOException & UnsupportedCallbackException abfangen...

//Verifiziere benutzername/passwort
boolean benutzernameKorrekt = false;
if (benutzername.equals(BENUTZER1) ||
    benutzername.equals(BENUTZER2) )
benutzernameKorrekt = true;

if (benutzernameKorrekt &&
    ((BENUTZER1.equals(benutzername) &&
    PASSWORT1.equals(passwort)) ||
     (BENUTZER2.equals(benutzername) &&
    PASSWORT2.equals(passwort))))
{
    //Authentifizierung erfolgreich!
    authErfolgreich = true;
    return true;
}
else
{
    //Authentifizierung gescheitert - setze Zustand zurueck
    authErfolgreich = false;
    benutzername = null;
    passwort = null;
    if (!benutzernameKorrekt)
        throw new FailedLoginException("Benutzername falsch");
    else
        throw new FailedLoginException("Passwort falsch");
}
}
```

Diese beiden *Callbacks* (*NameCallback* und *PasswordCallback*) übergibt LoginModulKV über die *handle()*-Methode an seinen eingetragenen *CallbackHandler*, das ist das LoginFenster. Der *CallbackHandler* trägt die im Eingabefenster angegebenen Login-Daten des Benutzers in diese *Callbacks* ein:

```
//Klasse: LoginFenster
@Override
public void handle(Callback[] callbacks) throws IOException,
    UnsupportedCallbackException
{
    for (int i = 0; i < callbacks.length; i++)
    {
        if (callbacks[i] instanceof TextOutputCallback)
        {
            //Zeige dem Typ gemaess die Nachricht an
```

I 12 Authentifizierung und Autorisierung

```
    TextOutputCallback toc =
        (TextOutputCallback) callbacks[i];
    switch (toc.getMessageType())
    {
        //Abfrage "INFORMATION", "ERROR"
        //oder "WARNING"
    }
}
else if (callbacks[i] instanceof NameCallback)
{
    //Setze den Namen des Benutzers
    NameCallback nc = (NameCallback) callbacks[i];
    nc.setName(benutzernameTextFieldJ.getText());
}
else if (callbacks[i] instanceof PasswordCallback)
{
    //Setze Passwort des Benutzers
    PasswordCallback pc = (PasswordCallback) callbacks[i];
    pc.setPassword(passwordPasswordFieldJ.getPassword());
}
else
{
    throw new UnsupportedCallbackException(callbacks[i],
        "Unidentifizierbares Callback");
}
}
```

Dann überprüft LoginModulKV, ob die eingegebene Kombination Benutzername/Passwort gültig ist, und gibt an LoginContext zurück, ob die Authentifizierung erfolgreich war oder nicht. Wenn nicht, meldet LoginContext false (also Misserfolg) an LoginFenster zurück, ansonsten ruft LoginContext commit() auf LoginModulKV auf. Beim commit() werden die Rollen des Benutzers gesetzt: Der Benutzer mit dem Benutzernamen Benutzer1 bekommt die Kundenbearbeiter-Rolle und der mit dem Benutzernamen Benutzer2 die Kunden-Rechercheur-Rolle:

```
//Klasse: LoginModulKV
public boolean commit() throws LoginException
{
    if (authErfolgreich == false)
        return false;
    else
    {
        if (benutzername.equals(BENUTZER1))
            //Benutzer 1 ist Kundenbearbeiter
        {
            if (!benutzer.getPrincipals().contains(
                Rolle.KUNDEN_BEARBEITER_ROLLE))
                benutzer.getPrincipals().add(
                    Rolle.KUNDEN_BEARBEITER_ROLLE);
        }
        else if(benutzername.equals(BENUTZER2))
```

12.3 Fallstudie: KV – JAAS I

```

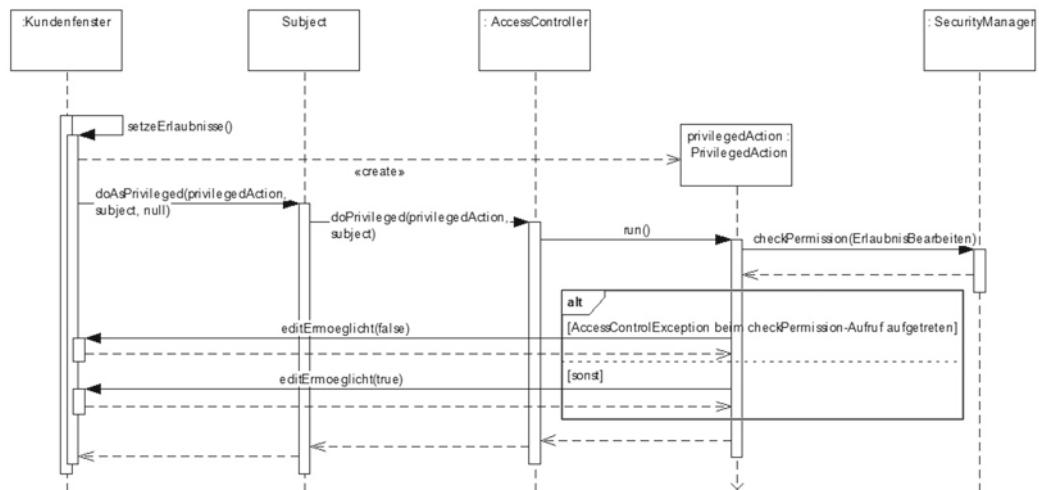
//Benutzer 2 ist 'nur' Kunden-Rechercheur
{
    if (!benutzer.getPrincipals().contains(
        Rolle.KUNDEN_RECHERCHEUR_ROLLE))
        benutzer.getPrincipals().add(
            Rolle.KUNDEN_RECHERCHEUR_ROLLE);
}

//Setze Zustand zurueck
benutzername = null;
passwort = null;
commitErfolgreich = true;
return true;
}
}

```

Dann wird LoginModulKV beendet, LoginContext auch. LoginFenster bekommt von der login()-Methode als Rückgabewert einen booleschen Wert, der anzeigen, ob die Anmeldung erfolgreich war oder nicht und kann sich dementsprechend schließen und an die aufrufende Stelle zurückgeben (Erfolg) oder dem Benutzer eine Fehlermeldung anzeigen (Misserfolg).

Nach erfolgreicher Authentifizierung wird aus der Klasse LoginFenster das Kundenfenster gestartet (Abb. 12.3-5).



Der aktuelle LoginContext wird dem Konstruktor der Klasse Kundenfenster übergeben und es wird durch Aufruf der Methode `setzeErlaubnisse()` zunächst festgestellt, welche Rolle der authentizierte Benutzer hat:

Abb. 12.3-5:
Sequenzdiagramm
für den Start des
Kundenfensters
der Fallstudie
»KV – JAAS«.

I 12 Authentifizierung und Autorisierung

```
//Klasse: Kundenfenster
private void setzeErlaubnisse()
{
    Subject subj = loginContext.getSubject();
    //Bearbeiten erlaubt?
    Subject.doAsPrivileged(subj, new PrivilegedAction<Object>()
    {
        @Override
        public Object run()
        {
            try
            {
                System.getSecurityManager().
                checkPermission(new ErlaubnisBearbeiten<Object>());
                editErmoeglicht = true;
            }
            catch(java.security.AccessControlException ace)
            {
                System.out.println("edit nicht erlaubt");
                editErmoeglicht = false;
            }
            return null;
        }
    }, null);
}

//Analog: Suchen erlaubt?
}
```

Innerhalb dieser Methode wird die Klassenmethode `doAsPrivileged()` der Klasse `Subject` aufgerufen. Dabei werden folgende Parameter übergeben:

- Das `Subject`-Objekt, in dem alle Informationen des Benutzers hinterlegt sind.
- Die auszuführende `PrivilegedAction` (also `Bearbeiten` oder `Suchen`).
- Der Parameter für `AccessControlContext` wird nicht genutzt und besitzt deshalb den Wert `null`. Dies ist bei einer Serverumgebung notwendig, damit mehrere eingehende Anfragen unabhängig voneinander authentifiziert werden.

`Subject` ruft nun die Methode `doPrivileged()` der Klasse `AccessController` auf und leitet somit den Aufruf weiter. Die Klasse `AccessController` ruft die `run()`-Methode der übergebenen `PrivilegedAction` auf. Hier wird nun ein `ErlaubnisBearbeiten` erzeugt und diese Erlaubnis wird als Parameter der `checkPermission()`-Methode der Klasse `SecurityManager` gesetzt. Die Klasse `SecurityManager` überprüft nun, ob entsprechend der Sicherheitspolitik (*policy*), die in der `login.policy`-Datei spezifiziert ist, diese Erlaubnis vorliegt. Die `login.policy`-Datei für die Kundenverwaltung-Mini sieht wie folgt aus (Auszug):

12.3 Fallstudie: KV – JAAS I

```
// Principal(Rollen)-bezogene Erlaubnisse  
grant Principal auth.Rolle "Kunden-Rechercheur"  
{  
    permission auth.ErlaubnisSuchen;  
};  
grant Principal auth.Rolle "Kunden-Bearbeiter"  
{  
    permission auth.ErlaubnisSuchen;  
    permission auth.ErlaubnisBearbeiten;  
};
```

Die Kunden-Bearbeiter-Rolle besitzt die angefragte ErlaubnisBearbeiten. Wenn der angemeldete Benutzer diese Rolle bekommen hat, wird checkPermission(), ohne eine Ausnahme zu erzeugen, zurückkehren. Sonst wird eine AccessControlException erzeugt. Je nachdem, ob eine Ausnahme erzeugt wurde oder nicht, setzt die PrivilegedAction dann den editErmoeglicht-Konfigurationsparameter des Kundenfensters. In Abhängigkeit von der Rolle wird dann das Kundenfenster angezeigt. Dabei sind nur diejenigen Funktionen sichtbar, die die Rolle erlaubt.

Der Ablauf zum Setzen des sucheErmoeglicht-Konfigurationsparameters verläuft analog.

Das vollständige Programm finden Sie im kostenlosen E-Learning- OOP Kurs zu diesem Buch.

Lassen Sie die Programme auf Ihrem Computersystem ablaufen und gehen Sie sie Schritt für Schritt durch. Führen Sie das Programm einmal mit benutzername1 und password1 und einmal mit benutzername2 und password2 aus.



13 Transaktionen

Wenn eine Bank eine Buchung vornimmt, so sind davon in der Regel zwei Konten betroffen. Dem einen Konto wird ein bestimmter Betrag belastet, dem anderen der exakt gleiche Betrag gutgeschrieben. Es wäre fatal, wenn eine Buchung, z. B. durch einen Hardware-Ausfall, zu einem Zeitpunkt abgebrochen würde, an dem zwar das eine Konto belastet, der Betrag dem anderen Konto aber noch nicht gutgeschrieben wurde.

Beispiel 1

Was muss man tun, um eine solche Situation zu verhindern?

Frage

Sowohl die Belastung des einen Kontos als auch die Gutschrift auf das andere Konto müssen als eine Einheit betrachtet werden. Nur wenn beide erfolgreich durchgeführt wurden, ist der Vorgang erfolgreich abgeschlossen (*commit*). Schlägt eine Buchung fehl, dann muss auch die bereits durchgeföhrte Buchung rückgängig gemacht werden (*rollback*).

Antwort

In der Softwaretechnik fasst man Aktionen, die alle erfolgreich durchgeführt werden müssen, zu sogenannten **Transaktionen** (*transactions*) zusammen. Für das obige Beispiel könnte das wie folgt aussehen:

```
begin of transaction
belaste das Konto X mit dem Betrag Z;
schreibe dem Konto Y den Betrag Z gut;
end of transaction
```

Eine Transaktion stellt sicher, dass entweder beide Aktionen stattfinden oder keine von beiden. Die Aktionen innerhalb einer Transaktion sind geordnet, d. h. ihre Reihenfolge darf *nicht* verändert werden.

Was müssen Sie tun, wenn in dem Beispiel 1 die Gutschrift auf dem anderen Konto fehlschlägt?

Frage

In diesem Fall müssen Sie die Abbuchung vom ersten Konto wieder rückgängig machen, d. h. dem Konto den abgebuchten Betrag wieder gutschreiben.

Antwort

Transaktionen müssen bestimmte Eigenschaften besitzen und werden von Transaktionssystemen verarbeitet:

i

- »Transaktionsverarbeitung«, S. 178

Transaktionen stammen ursprünglich aus der Datenbank-Welt. Es kann aber durchaus notwendig sein, Transaktionen auch zu verwenden, wenn keine Datenbank im Einsatz ist:

- »Fallstudie: KV – JTA«, S. 180

I 13 Transaktionen

.NET unterstützt ebenfalls Transaktionen:

- »Transaktionen in .NET«, S. 186

Zusammenhänge Der Einsatz einer Transaktionsverarbeitung trägt wesentlich zur Verbesserung der Zuverlässigkeit (siehe »Zuverlässigkeit«, S. 124) eines Softwaresystems bei, insbesondere wird die Wiederherstellbarkeit unterstützt.

13.1 Transaktionsverarbeitung

Transaktionen stammen ursprünglich aus der Datenbank-Welt. Eine **Transaktion** ist eine Folge von Aktionen, die zu einer logischen Einheit zusammengefasst werden.

Eine Transaktion kann nur vollständig durchgeführt oder gar nicht ausgeführt werden. Zur Abarbeitung von Transaktionen werden **Transaktionssysteme** eingesetzt. Sie müssen bei der Ausführung von Transaktionen die sogenannten ACID-Prinzipien garantieren:

ACID-Prinzipien

- **Atomicity** (Unteilbarkeit¹): Es werden entweder alle Aktionen einer Transaktion ausgeführt oder keine Aktion. Eine Transaktion wird niemals nach der Ausführung eines Teils der Aktionen beendet. Wird eine Transaktion abgebrochen, dann ist das Anwendungssystem unverändert.
- **Consistency** (Konsistenz): Vor Beginn einer Transaktion und nach ihrer Beendigung befindet sich der Datenbestand in einem konsistenten Zustand. Solange eine Transaktion läuft, kann sich der Datenbestand in einem inkonsistenten Zustand befinden.
- **Isolation** (Isolation): Wenn mehrere Transaktionen gleichzeitig auf demselben Datenbestand ausgeführt werden, so ist das Ergebnis ein Datenbestand, wie ihn eine sequenzielle Ausführung der Transaktionen in irgendeiner Reihenfolge ergeben hätte. Insbesondere sind inkonsistente Zustände während der Abarbeitung einer Transaktion für andere Transaktionen unsichtbar.
- **Durability** (Dauerhaftigkeit): Die Änderungen, die eine Transaktion an einem Datenbestand vornimmt, werden persistent gespeichert, d.h. sie werden auf ein nicht-flüchtiges Speichermedium geschrieben.

Transaktionssysteme

Zur Abwicklung und Verwaltung von Transaktionen werden Transaktionssysteme (*transaction processing systems*) verwendet. Sie sind z.B. Teil vieler Datenbank-Management-Systeme (DBMS). Zu Beginn einer Aktionsfolge, die als Transaktion zu handhaben ist, initiiert eine Anwendung mit Hilfe des Transaktionssystems eine neue Transaktion (*begin of transaction*).

¹Die oft verwendete unreflektierte deutsche Übersetzung »Atomarität« ist kein deutsches Wort.

13.1 Transaktionsverarbeitung I

Nach Beendigung der letzten Aktion erfolgt der Abschluss der Transaktion (*commit* oder *end of transaction*). Transaktionssysteme speichern Start- und Zwischenzustände der Daten und verwenden diese Informationen, um die Datenhaltung in den Ursprungszustand zurückzuversetzen, wenn die Transaktion nicht vollständig durchgeführt werden kann. Beispielsweise werden Kopien der Daten vor der Modifikation durch eine Transaktion erzeugt (manchmal *before image* genannt). Diese Kopien werden dann dazu verwendet, im Fehlerfall die Datenhaltung wieder in den ursprünglichen Zustand zu versetzen.

Kann eine Transaktion wegen eines Fehlers nicht korrekt beendet werden, so erfolgt ein Abbruch (*abort*). Das Transaktionssystem sorgt dann für eine Rückabwicklung (*rollback*). Im Rahmen der Rückabwicklung werden alle bisher am Datenbestand vorgenommenen Änderungen wieder rückgängig gemacht, sodass ein konsistenter Zustand erreicht wird. Die im Rahmen einer Transaktion vorgenommenen Änderungen müssen also genau protokolliert und/oder zwischengespeichert werden, um das ACID-Prinzip garantieren zu können.

Wenn der zu ändernde Datenbestand oder die Abarbeitung der einzelnen Aktionen über mehrere Computersysteme verteilt erfolgt, so spricht man von verteilten Transaktionen. Eine **verteilte Transaktion** wird auf einem Computersystem initiiert (*master*) und von dort werden – z. B. zur Steigerung der Leistung (*performance*) – einzelne Teilaufgaben auf andere Computersysteme (*slaves*) verteilt, die dort als Teil-Transaktionen ablaufen. Zur Koordination verteilter Transaktionen wurde das sogenannte **Zwei-Phasen-Commit-Protokoll** entwickelt. Es stellt sicher, dass eine verteilte Transaktion nur dann erfolgreich abgeschlossen wird, wenn zuvor alle Teil-Transaktionen erfolgreich abgeschlossen wurden. Schlug nur eine Teil-Transaktion fehl, so finden für alle Teil-Transaktionen Rückabwicklungen (*rollbacks*) statt. Die gesamte verteilte Transaktion ist dann fehlgeschlagen (siehe [Duck95]).

Unternehmensanwendungen benötigen in der Regel Transaktions-sicherheit. Viele Geschäftsprozesse müssen nach dem ACID-Prinzip abgewickelt werden. *Middleware*-Entwicklungs- und Laufzeitumgebungen, wie z. B. Java EE (siehe »Die Java EE-Plattform«, S. 321), unterstützen zu diesem Zweck verteilte Transaktionen. Transaktionssysteme sind Bestandteil der jeweiligen Plattform. Im Hintergrund stützen sich viele Transaktionssysteme dabei auf die Dienste eines Datenbank-Managementsystems.

Middleware-Plattformen erlauben es zusätzlich, das Transaktions-verhalten unabhängig von der Implementierung einer Komponente zu spezifizieren. Der Entwickler implementiert die Operationen

Verteilte
Transaktionen

Transaktions-
sicherheit

I 13 Transaktionen

einer Komponente (Geschäftsprozesse sind technisch gesehen Operationen) und braucht sich um Transaktionen nicht zu kümmern. Dies vereinfacht die Implementierung der Anwendungslogik.

Installation

Erst bei der Installation der Komponente wird festgelegt, welche Operationen/Geschäftsprozesse durch Transaktionen zu sichern sind und welche nicht. Das Laufzeitsystem der Plattform kann dann beim Aufruf einer entsprechend markierten Operation automatisch eine Transaktion starten. Ruft die Operation andere Operationen von anderen Objekten auf, so verfolgt das Laufzeitsystem diese Aufrufe und bettet auch die aufgerufenen Objekte in die Transaktion ein. Nach erfolgreicher Beendigung der Operation wird ein *commit* durchgeführt. Schlug die Ausführung fehl, wird eine Rückabwicklung veranlasst.

13.2 Fallstudie: KV – JTA

Transaktionen werden in der Regel im Zusammenhang mit Datenbanken eingesetzt. Es gibt aber auch eine ganze Reihe von Fällen, in denen Transaktionen auch ohne Datenbankeinsatz sinnvoll sind. Die Datenhaltung in der Fallstudie »Kundenverwaltung-Mini« kann durch eine indexbasierte Dateiorganisation erfolgen.

Indexbasiert

Bei einer indexbasierten Dateiorganisation wird die Position, ab der eine Information in einer Datei gespeichert ist, in einer Indextabelle aufbewahrt. Die Indextabelle stellt den Zusammenhang zwischen dem Schlüssel, z. B. der Kundennummer, und der Speicherposition her. Die Indextabelle wird ebenfalls in einer Datei gespeichert.

Beispiel

Die Abb. 13.2-1 zeigt ein Beispiel zur Veranschaulichung des Konzepts. Der Schlüssel der Indextabelle sei beispielsweise die Kundennummer. Die Indextabelle wird zunächst mit -1 initialisiert. Dadurch wird angegeben, dass zu dieser Kundennummer noch kein Datensatz existiert. Wird nun ein Datensatz zur Kundennummer 2 erfasst, dann wird dieser Datensatz in der Hauptdatei ab der Position 0 gespeichert. Diese Position wird in die Indextabelle an der Stelle 2 eingetragen. Es wird angenommen, dass jeder Datensatz eine Länge von 100 Bytes hat. Wird als nächstes ein Datensatz zur Kundennummer 5 erfasst, dann wird er ab der Position 100 gespeichert und die Nummer des Datensatzes als 1 in der Indextabelle gespeichert.

Soll der Datensatz zur Kundennummer 5 gelesen werden, dann wird in der Indextabelle beim Index 5 nachgesehen. Dort steht die Datensatznummer 1, die mit der Datensatzlänge 100 multipliziert (= 100) die Position in der Hauptdatei angibt (alternativ hätte man in der Indextabelle auch direkt die Datensatzposition speichern können). Ein Zeiger wird in der Hauptdatei auf die Position 100 gesetzt und die nächsten 100 Bytes dort ausgelesen.

13.2 Fallstudie: KV – JTA I

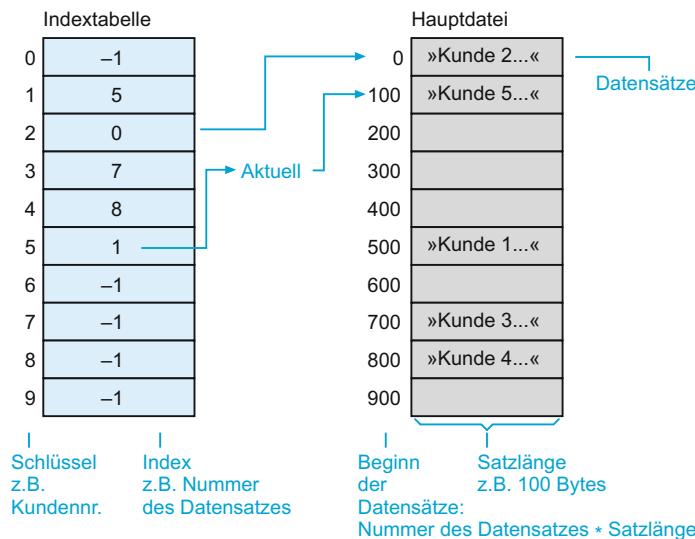


Abb. 13.2-1:
Beispiel einer
Indexverwaltung.

Bei der indexbasierten Dateiorganisation ergibt sich beim Schreiben eines Datensatzes das Problem, dass zwei Dateien geändert werden müssen. Schlägt eine dieser Änderungen fehl, dann muss die andere Änderung – wenn bereits vorgenommen – rückgängig gemacht werden. Das ist ein typischer Fall für den Einsatz eines Transaktionsmanagements. Der Vorgang des Schreibens in beide Dateien wird als ein einziger Vorgang – eine Transaktion – programmiert. Das Transaktionsmanagement sorgt dafür, dass die Transaktion entweder ganz durchgeführt ist (beide Dateien geändert) oder gar nicht durchgeführt ist (beide Dateien unverändert).

In Java unterstützt **JTA** (*Java Transaction API*) die Programmierung von verteilten Transaktionen. JTA spezifiziert dabei die Schnittstelle, über die Java-Programme mit Transaktionsmanagern kommunizieren können. Mithilfe des X/Open XA-Standards (kurz **XA**) können verschiedene Ressourcen, wie Datenbanken, Anwendungsserver und Nachrichtensysteme, innerhalb einer Transaktion angesprochen werden, wobei die ACID-Eigenschaften von Transaktionen eingehalten werden. JTA unterstützt diesen Standard. Damit JTA eingesetzt werden kann, müssen alle Ressourcen die Schnittstelle `javax.transaction.xa.XAResource` implementieren.

Java

JTA ist Bestandteil von Java EE, siehe »Die Java EE-Plattform«, S. 321. Bei der Verwendung von Java SE muss ein JTA-fähiger Transaktionsmanager hinzugezogen werden. Ein solcher Transaktionsmanager ist beispielsweise **TransactionsEssentials** von Atomikos, siehe Website Atomikos (<http://www.atomikos.com/Main/TransactionsEssentials>).

Java SE vs. Java EE

I 13 Transaktionen

Voraussetzungen Von dieser Website muss die JTA-Implementierung »Atomikos TransactionEssentials« herunter geladen und entpackt werden. Die folgenden Bibliotheken müssen dem Build-Path des Projekts hinzugefügt werden:

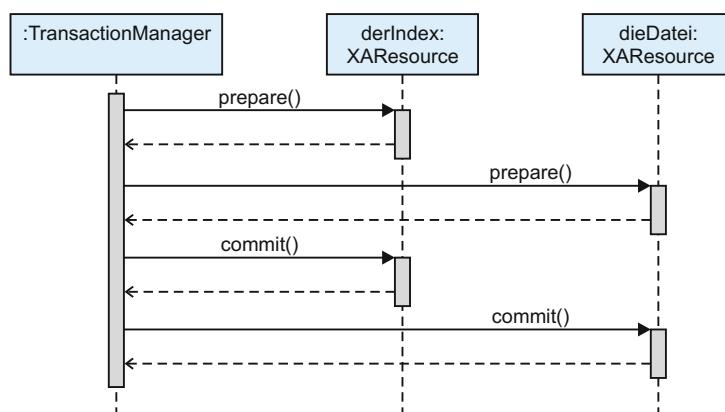
- /dist/transactions-essentials-all.jar
- /lib/jta.jar

Implementierungen der Schnittstelle XAResource

In JTA definiert XAResource die Schnittstelle zu einem Ressourcen-Manager. Diese Schnittstelle wird dann von dem Transaktionsmanager genutzt. In der Fallstudie KV-Mini müssen die beiden Klassen Index und Datei diese Schnittstelle implementieren. Von den 10 zu implementierenden Methoden der Schnittstelle XAResource sind drei Methoden besonders relevant (der XID-Parameter ist dabei die ID für die gerade durchgeführte Transaktion):

- `public int prepare(Xid xid) throws XAException`: Mit dieser Methode wird dem Ressourcen-Manager mitgeteilt, sich auf ein commit vorzubereiten. In dieser Methode bereitet sich der Ressourcen-Manager auch auf ein eventuelles rollback vor, d. h. ein *undo-log* wird geschrieben oder es werden Sicherungskopien angelegt. Es ist der erste Schritt im **Zwei-Phasen-Commit-Protokoll**. Der Transaktionsmanager geht erst in die zweite (*commit*-Phase) über, wenn kein Ressourcen-Manager beim `prepare()`-Aufruf eine `XAException` geworfen hat. Ein Ausschnitt aus der Interaktion zwischen Transaktions- und Ressourcen-Manager während des Zwei-Phasen-Commit-Protokolls ist in den Abbildungen dargestellt, einmal für den Erfolgsfall (Abb. 13.2-2) und einmal für den Misserfolgsfall (Abb. 13.2-3). Zu beachten ist: Die `prepare()`-Methode wird beim 2-Phasen-Protokoll bei jedem Teilnehmer aufgerufen.

Abb. 13.2-2:
Sequenzdiagramm
für den Commit-
Fall für die
Fallstudie »KV-
JTA«.



13.2 Fallstudie: KV – JTA I

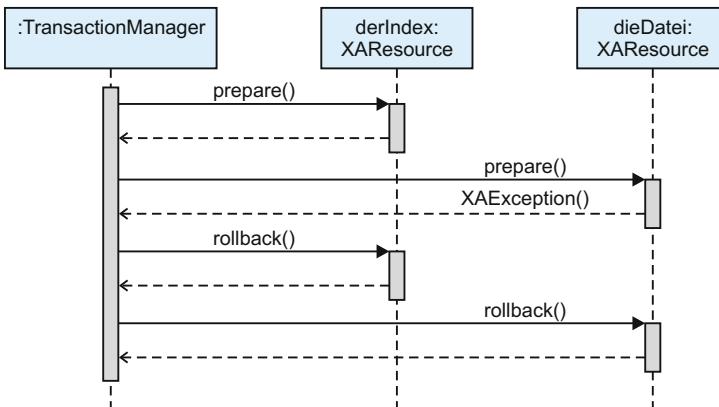


Abb. 13.2-3:
Sequenzdiagramm
für den Rollback-
Fall für die
Fallstudie »KV-
JTA«.

- `public void commit(Xid xid, boolean onePhase)`
throws `XAException`: Mit dieser Methode wird der Ressourcen-Manager aufgefordert, die Transaktion abzuschließen und ggfs. gesperrte Ressourcen freizugeben. `onePhase` zeigt an, ob nach dem Ein-Phasen- oder dem Zwei-Phasen-Protokoll gearbeitet wird. Beim Ein-Phasen-Protokoll fehlt die Vorbereitungsphase (keine `prepare()`-Aufrufe). Dieses Protokoll kann nur genutzt werden, wenn nur eine Ressource beteiligt ist. Für die Fallstudie KV-Mini ist es daher nicht relevant.
- `public void rollback(Xid xid) throws XAException`: Beim *Rollback* soll der Ressourcen-Manager alle während der Transaktion durchgeföhrten Änderungen wieder rückgängig machen. Hierfür werden die Sicherungen genutzt, die während der `prepare()`-Phase angelegt worden sind.

Das gesamte OOD-Modell zeigt die Abb. 13.2-4. Die Klassen `Kunde`, `KundenContainer`, `Start` und `Kundenfenster` bleiben unverändert.

Laden Sie die Programme dieser Fallstudie auf Ihr Computersystem und sehen Sie sich die Programme parallel zu den folgenden Erklärungen an.

Die Klasse `Datei` soll während der Transaktion folgende Methode ausführen: `speichereSatz(datensatz, index);`

Dazu werden die drei oben erklärten `XAResource`-Methoden wie folgt implementiert:

- In der `prepare()`-Methode wird zunächst für einen eventuellen Rollback eine Sicherheitskopie der verwendeten Datei angelegt. Danach wird die Operation `speichereSatz()` durchgeführt.
- In der `commit()`-Methode kann die Sicherheitskopie gelöscht werden.
- In der `rollback()`-Methode wird die ursprüngliche Datei wiederhergestellt, indem die verwendete Datei durch die Sicherheitskopie überschrieben wird.



I 13 Transaktionen

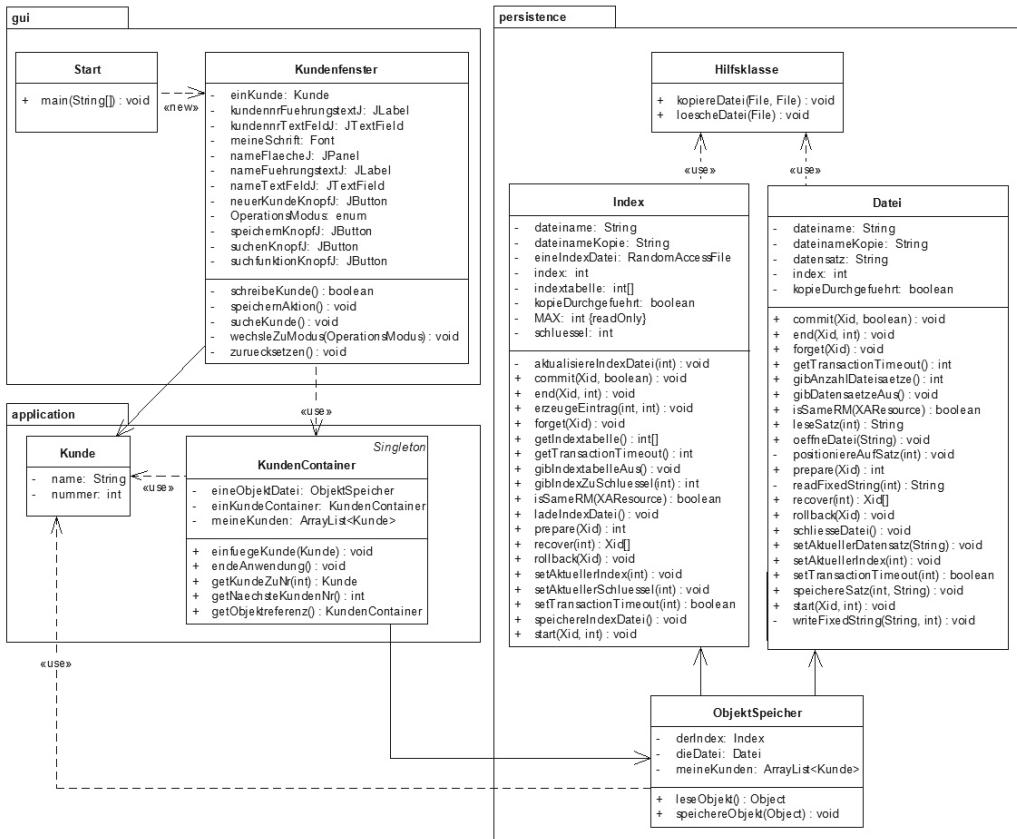


Abb. 13.2-4: OOD Klassendiagramm der Fallstudie »KV – JTA«.

In allen drei Methoden wird bei einer Ausnahme, die beim Schreiben in die Datei aufgetreten ist, diese als XAException weiter zum Aufrufer (Transaktionsmanager) gegeben:

```

...
catch (IOException e)
{
    e.printStackTrace();
    throw new XAException();
}

```

Zu beachten ist, dass auch beim Anlegen der Sicherheitskopie eine Ausnahme auftreten kann. `prepare()` meldet dies als XAException. Dementsprechend muss `rollback()` vor dem Wiederherstellen überprüfen, ob das Kopieren erfolgreich stattgefunden hat. `rollback()` kann dies am Attribut `kopieDurchgefuehrt` erkennen, welches in `prepare()` gesetzt wird.

Klasse Index Die Klasse Index soll während der Transaktion folgende Methode ausführen: `erzeugeEintrag(schluessel, index);`

Die XAResource-Methoden werden analog wie bei der Klasse Datei implementiert. Auch hier wird in der Methode `prepare()` eine Sicherheitskopie der verwendeten Datei angelegt, die dann in der Methode `rollback()` verwendet werden kann. Die eigentliche Operation `erzeugeEintrag(schlüssel, index);` erfolgt in `prepare()` nach dem Anlegen der Sicherungskopie.

Aufruf des Transaktionsmanagers

Der Aufruf des Transaktionsmanagers erfolgt in der Methode `speichereObjekt()` der Klasse ObjektSpeicher und zwar in dem Fall, dass der Kunde neu angelegt werden muss. Der entsprechende Code-Block sieht wie folgt aus:

```
TransactionManager tm = new UserTransactionManager();
try
{
    tm.begin();
    Transaction t = tm.getTransaction();
    derIndex.setAktuellerSchlüssel(schlüssel);
    derIndex.setAktuellerIndex(index);
    t.enlistResource(derIndex);
    dieDatei.setAktuellerDatensatz(datensatz);
    dieDatei.setAktuellerIndex(index);
    t.enlistResource(dieDatei);
    tm.commit();
}
catch (NotSupportedException e1)
{
    e1.printStackTrace();
}
...
(weitere Exception-Behandlungen)
...
```

Als erstes wird ein `TransactionManager` erzeugt. Die Implementierung dafür ist nicht in Java SE vorhanden, sondern kommt vom externen Paket `com.atomikos.icatch.jta`.

Auf dem `TransactionManager` wird `begin()` aufgerufen. Das führt dazu, dass eine neue Transaktion erzeugt und mit dem aktuellen `Thread` verbunden wird. Das erzeugte `Transaction`-Objekt wird vom `TransactionManager` geholt, und die beiden `XAResource`-Objekte werden ihr hinzugefügt (`enlistResource()`). Das sind das Objekt `derIndex` und das Objekt `dieDatei`. Vor dem Einfügen müssen noch deren Parameter gesetzt werden, die sie für die Abarbeitung der Transaktion benötigen.

Am Ende wird `commit()` auf dem `TransactionManager` aufgerufen. Dadurch wird die Transaktion ausgeführt (`prepare()`-, `commit()`-, `rollback()`-Aufrufe auf den Ressourcen-Managern). Nach dem Ausführen von `commit()` beim `TransactionManager` ist die Verbindung des `TransactionManager` mit der `Transaction` gelöst.

I 13 Transaktionen

Resümee Wie die Fallstudie zeigt, kann es für Anwendungen durchaus sinnvoll sein, das Verhalten von Transaktionen selbst festzulegen. In der Fallstudie werden dazu Dateikopien angelegt und im Falle eines Abbruchs einer Transaktion diese Dateikopien wieder als Originale verwendet. In einer realen Anwendung würde man natürlich niemals ganze Dateien kopieren, sondern beispielsweise über einen Delta-Mechanismus nur vorgenommene Änderungen speichern.

Literatur [Duck95], [JTA10]

13.3 Transaktionen in .NET

.NET unterstützt Transaktionen sowohl über eine Datenbank als auch über mehrere Datenbanken (lokale Transaktion) in einem Datenbankserver und mehrere Datenbanken in verschiedenen Datenbankservern (verteilte Transaktionen).

DbTransaction

Für lokale Transaktionen steht die Klasse System.Data.DbTransaction (jeweils mit Abteilungen in den einzelnen Datenbanktreibern, z.B. OracleTransaction und SqlTransaction) zur Verfügung. DbTransaction bezieht sich immer auf eine bestehende Datenbankverbindung.

```
Beispiel //Modell instanziieren unter Verwendung der
//Verbindungszeichenfolge aus der Konfigurationsdatei
using (WwWings6Entities modell = new WwWings6Entities())
{
    //Verbindung explizit öffnen und Transaktion beginnen
    modell.Connection.Open();
    DbTransaction transaction =
        modell.Connection.BeginTransaction();

    //... Verschiedene Aktionen auf der Datenbank

    //Transaktion erfolgreich oder nicht erfolgreich?
    if (Erfolg)
        transaction.Commit();
    else
        transaction.Rollback();
    modell.Connection.Close();
} //Ende using-Block -> Dispose() wird aufgerufen
```

Die Speicher-Methode des ADO.NET Entity Frameworks (SaveChanges()) erzeugt automatisch eine Transaktion über alle Änderungen.

System.Transactions

Eleganter und flexibler ist die Nutzung der .NET-Bibliothek `System.Transactions` (vergleichbar mit JTA in Java, siehe »Fallstudie: KV – JTA«, S. 180). Hiermit sind Transaktionen über beliebige transaktionsfähige Ressourcen möglich. Transaktionsfähige Ressourcen besitzen einen Ressourcenmanager, der entweder den XA-Standard der *The Open Group for distributed transaction processing* (DTP) oder das Microsoft-proprietäre Protokoll *OLE Transactions* unterstützt. Moderne Datenbankmanagementsysteme wie Oracle und Microsoft SQL Server bieten einen solchen Ressourcenmanager, aber auch das Windows-Betriebssystem (ab Version 6.0) bietet einen Transaktionsmanager für das NTFS-Dateisystem und die Registrierungsdatenbank.

Bei der Verwendung von `System.Transactions` ist eine Instanz von `System.Transactions.TransactionScope` zu erzeugen. Die Transaktion gilt als erfolgreich abgeschlossen (*Commit*), wenn die `Complete()`-Methode aufgerufen wird. Wird das `TransactionScope` vernichtet, ohne den Aufruf von `Complete()`, z. B. weil es zu einem Laufzeitfehler gekommen ist, gilt die Transaktion als nicht erfolgreich (*Abort*). Alle Aktionen innerhalb – zwischen der Erzeugung von `TransactionScope` und dem *Commit/Abort* – werden automatisch Teil der Transaktion, sofern es für diese Aktion einen Transaktionsmanager gibt. Die Verwaltung der Transaktion obliegt dem *Microsoft Distributed Transaction Coordinator* (MSDTC), einem Systemdienst von Windows, der gestartet sein muss.

Dieses Beispiel zeigt die elegante Verwendung von `TransactionScope` über einen `using{}`-Block in C# in einer Methode der Geschäftslogik. Innerhalb des `using{}`-Blocks werden zwei Methoden der Datenzugriffsschicht aufgerufen (`ReduzierePlatzAnzahl()` und `ErzeugeBuchung()` für eine Flugbuchung). Außerdem erfolgt das Erstellen einer Protokolldatei im NTFS-Dateisystem. Nur wenn alle drei Aktionen erfolgreich waren, kommt es zum Aufruf vom `Complete()`. Nur dann werden die Veränderungen in der Datenbank und die erzeugte Datei im Dateisystem bestandskräftig.

Beispiel:
Flugbuchung

```
//<summary>
//Flugbuchung erstellen
//</summary>
public string NewBuchung(int FlugNummer, int PassagierNummer)
{
    try
    {
        string Buchungscode =
            FlugNummer.ToString() + "-" +
            PassagierNummer.ToString();

        //Transaktion, nur erfolgreich
```

I 13 Transaktionen

```
//wenn Platzanzahl reduziert und Buchung erstellt!
using (System.Transactions.TransactionScope t =
new System.Transactions.TransactionScope())
{
    //hier erfolgen Änderungen in Datenbanken über
    //zwei Methoden der Datenzugriffsschicht
    if (!FlugDZS.ReduzierePlatzAnzahl(FlugNummer, 1))
        return "Kein Platz auf diesem Flug vorhanden!";
    if (!BuchungsDZS.ErzeugeBuchung(
        PassagierNummer, FlugNummer))
        return "Buchung nicht möglich!";

    //Protokolldatei im NTFS-Dateisystem schreiben
    //(als Teil der Transaktion)
    string Path = @"t:\buchungen\" + Buchungscode + ".txt";
    FileStream fs = TransactedFile.Open(Path,
        System.IO.FileMode.CreateNew,
        System.IO.FileAccess.Write,
        System.IO.FileShare.None
    );
    StreamWriter sw = new StreamWriter(fs);
    sw.WriteLine(DateTime.Now + ";" + Buchungscode + ";"
        + FlugNummer + ";" + PassagierNummer);
    sw.Close();
    fs.Close();

    //Transaktion erfolgreich abschließen
    t.Complete();

    //Buchungscode zurückgeben
    return Buchungscode;
}
catch (Exception ex)
{
    return "Fehler: " + ex.Message;
}
```

Mit der Klasse TransactionScope können Transaktionen auch verschachtelt werden. Wenn innerhalb eines Blocks, der eine TransactionScope-Instanz besitzt, eine weitere Instanz von TransactionScope erzeugt wird, muss durch den Parameter TransactionScopeOption definiert werden, ob es sich dabei um eine eigenständige Transaktion oder eine untergeordnete Transaktion (Teiltransaktion der übergeordneten Transaktion) handeln soll.

System.Transactions ermöglicht es auch, eine .NET-Klasse in eine Transaktion einzubinden bzw. eine Transaktion in einer .NET-Klasse zu beginnen und erst später in den MSDTC aufzunehmen. .NET-Klassen bilden sogenannte volatile Ressourcen, die einen Neustart des Prozesses nicht überstehen. Datenbanktransaktionen sind im Ge-

13.3 Transaktionen in .NET I

gensatz dazu dauerhafte Transaktionen (*Durable Transactions*). Eine .NET-Klasse, die eine volatile Ressource im Rahmen einer Transaktion bilden möchte, muss die Schnittstelle `IEnlistmentNotification` mit den Methoden `Prepare()`, `Commit()`, `Rollback()` und `InDoubt()` unterstützen.

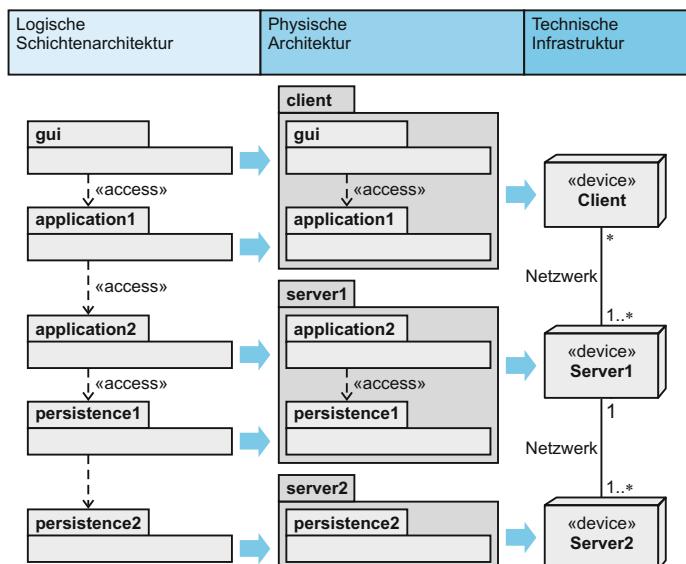
14 Verteilte Architekturen

Viele Softwaresysteme sind heute auf ein Netz miteinander verbundener Computersysteme verteilt. Um ein Softwaresystem verteilen zu können, muss es in Subsysteme strukturiert sein. Dabei sollten die Subsysteme in sich stark gebunden sein. Zwischen den Subsystemen sollte eine geringe Kopplung bestehen (siehe Prinzip der Bindung und Kopplung, »Architekturprinzipien«, S. 29).

Die Software-Subsysteme bezeichnet man auch als logische Subsysteme.

Während logische Subsysteme die Strukturierungseinheiten eines Softwaresystems darstellen, repräsentiert eine physische Architektur die verfügbaren Computersysteme (Arbeitsplatzcomputer, Abteilungs-Server, Datenbankserver, Zentralcomputer), auf die logische Subsysteme verteilt werden können.

Logische Subsysteme können mithilfe von Verteilungsmustern auf physische Architekturen abgebildet werden (Abb. 14.0-1).



Logische
Subsysteme

Physische
Architektur

Verteilungs-
muster

Abb. 14.0-1:
Beispiel für die
Verteilung von
logischen
Schichten auf eine
physische
Architektur.

Die Abb. 14.0-1 zeigt als Beispiel, wie eine nach dem Schichten-Muster (siehe »Das Schichten-Muster (*layers pattern*)«, S. 46) in fünf logische Schichten gegliederte Anwendung auf eine physische Architektur, bestehend aus vielen Clients und mehreren Servern, verteilt werden kann. Die Clients und Server sind dabei über ein Netzwerk miteinander verbunden. Das Beispiel beschreibt eine mehrstufige

Beispiel

I 14 Verteilte Architekturen

Client-Server-Architektur, da der Server 1 sowohl in der Rolle eines Servers gegenüber den Clients auftritt, aus Sicht des Servers 2 aber auch die Rolle eines Clients einnimmt.

Die Verteilungsmuster hängen von der Architekturart ab.

Folgende wichtige verteilte Architekturen lassen sich unterscheiden:

- »Client-Server-Architektur«, S. 193
- »Web-Architektur«, S. 196
- »SOA – Serviceorientierte Architekturen«, S. 201

Die ersten beiden Architekturen können auch kombiniert verwendet werden. Dies ist jedoch sorgfältig zu überlegen, da auf jeden Fall Mehraufwände bei der Architektur, bei der Implementierung, bei der Installation und bei der Wartung entstehen.

Eine weitere verteilte Architektur ist die **Peer-to-Peer-Architektur** – abgekürzt P2P-Architektur. Bei dieser Architektur sind alle beteiligten Computer gleichberechtigt und jeder Computer kann Dienste bereitstellen und in Anspruch nehmen, d.h. jeder Computer ist sowohl Client als auch Server. Bei einer reinen Peer-to-Peer-Architektur ist jeder Computer mit jedem anderen Computer verbunden (keine zentrale Infrastruktur), d.h. es gibt keinen verwaltenden Server. Das Peer-to-Peer-Netz muss sich selbst organisieren. Moderne P2P-Netzwerke verfügen meist über einen internes Overlay-Netzwerk, das die Organisation der anderen Computer sowie die Bereitstellung einer Suchfunktion übernimmt. Mit JXTA entsteht z.Z. ein Architektur-Standard für P2P-Anwendungen.

P2P-Architekturen werden im Wesentlichen für folgende Anwendungen eingesetzt:

- Austausch von Dateien (d.h. Kopieren) zwischen Computern (*File Sharing*) – bekannt geworden ist die Dateiauszbörse Napster.
- *Distributed Computing*: Einzelne Teilaufgaben einer komplexen Gesamtaufgabe werden auf Computer eines Netzwerkes verlagert, die freie Kapazitäten haben.
- Kommunikation: Anwendungen aus den Bereichen **Instant Messaging, VoIP und Chats**.

Da P2P-Architekturen nur für spezielle Anwendungsbereiche geeignet sind, werden sie hier nicht weiter vertieft.

Hinweis

Die Begriffe Client und Server sind mehrdeutig. Unter Client und Server kann man die jeweiligen hardwaremäßigen Computersysteme verstehen. Client und Server können aber auch als Softwaresysteme verstanden werden, die auf einem hardwaremäßigen Computersystem ablaufen. Technisch ist es heute möglich, dass mehrere Client-Softwaresysteme und Server-Softwaresysteme auf einem hardwaremäßigen Computersystem ablaufen. Dies wird oft der Fall sein, wenn Sie ein verteiltes System testen wol-

len. In der Regel befinden sich Client-Softwaresysteme und Server-Softwaresysteme jedoch auf unterschiedlichen, miteinander vernetzten, hardwaremäßigen Computersystemen. Solange keine Verwechslungsgefahr besteht, wird allgemein von Client und Server gesprochen, wobei davon ausgegangen wird, dass es sich um Softwaresysteme auf unterschiedlichen hardwaremäßigen Computersystemen handelt.

14.1 Client-Server-Architektur

Die **Client-Server-Architektur** (C/S-Architektur) ist die traditionelle Architektur für verteilte Systeme. Die (logischen) Subsysteme einer Anwendung werden auf Clients (*Frontend*) und einen oder mehrere Server (*Backend*) verteilt. Erfolgt eine Verteilung auf mehrere Server, dann spricht man von einer *mehrstufigen* Client-Server-Architektur.

Eine Client-Server-Architektur besitzt folgende Charakteristika:

- Die Anwendung wird auf mindestens einen Client und einen Server verteilt.
- Die Nutzung der Dienste eines Servers geht immer vom Client aus. Der Server selbst entscheidet, welche Anfragen er in welcher Reihenfolge bearbeitet.
- Der auf dem Client befindliche Teil des Softwaresystems ist nach der Anmeldung beim serverseitigen Teil des Softwaresystems in der Regel bis zur Abmeldung *permanent* mit dem Server verbunden.
- Die maximale Anzahl der nebenläufigen Benutzer liegt als Anforderung zur Entwurfszeit fest.
- Die Client-Server-Umgebung ist in der Regel bekannt und kann beeinflusst werden, zum Beispiel verwendete Betriebssysteme, Datenbanken usw.
- Die Entwickler sind sowohl für den Client- als auch für den Server-Teil verantwortlich.
- Auf den Clients muss ein Teil der Anwendungssoftware installiert (*deployed*) werden – zumindest initial, danach kann eine automatische Aktualisierung über das Netz erfolgen (siehe »Verteilung und Installation«, S. 521).

Charakteristika

Überlegen Sie, unter welchen Randbedingungen – unter Berücksichtigung dieser Charakteristika – eine Client-Server-Architektur *nicht* geeignet ist.

Frage

Soll ein Anwendungssystem unternehmensübergreifend, zum Beispiel in Zusammenarbeit mit Systemen von Kunden und Lieferanten, eingesetzt werden, dann kann die Client-Server-Umgebung vom eigenen Unternehmen nicht mehr ausreichend beeinflusst werden.

Antwort

I 14 Verteilte Architekturen

Entweder müssen dann Schnittstellen mit den Kunden- und Lieferanten-Systemen abgestimmt werden, oder es wird eine Architektur gewählt, die für heterogene, unternehmensübergreifende Anwendungsumgebungen ausgelegt ist.

Art der Verteilung

Die Art der Verteilung stellt eine wichtige Entwurfsentscheidung dar, da sie Rückwirkungen auf die logische Subsystem-Struktur sowie auf die Robustheit und Sicherheit der Anwendung als auch auf die Flexibilität, spätere Modifikationen oder Portierungen haben kann.

Clients

In Abhängigkeit davon, welche Aufgaben einer Anwendung auf dem Client ausgeführt werden, unterscheidet man folgende Client-Arten:

■ **Fat Client** (manchmal auch als *Thick Client* bezeichnet): In der Regel befinden sich die GUI-Schicht und die Applikationsschicht auf dem Client und müssen auf der dortigen Plattform installiert werden.

Vorteile

- + Viele Funktionen der Anwendung können *ohne* eine Netzwerkverbindung ausgeführt werden. Es muss jedoch periodisch eine Verbindung zum Server hergestellt werden.
- + Die GUI-Schicht kann alle Möglichkeiten des Clients einschließlich schneller Antwortzeiten und hoher Prozessorleistung nutzen.
- + Dialoge können nahtlos und verzögerungsfrei angepasst werden, Drag & Drop ist möglich.
- + Gut geeignet für multimediale Anwendungen (z.B. Videospiele, Animationen), die sonst eine hohe Übertragungsbandbreite zum Server benötigen würden.
- + Die Anwendungslogik kann effizient objektorientiert implementiert werden.
- + Es steht eine leistungsfähige Laufzeitumgebung zur Verfügung.
- + Der Server wird von Aufgaben entlastet.

Nachteile

- Die Aktualisierung und explizite lokale Installation neuer Versionen der Anwendung ist aufwändig.
- Sind die Plattformen der Clients unterschiedlich, dann müssen verschiedene Versionen vorgehalten werden.

■ **Rich Client** (manchmal auch als *Smart Client* bezeichnet): In der Regel befinden sich die GUI-Schicht und Teile der Applikationsschicht (zum Beispiel die Geschäftslogik) auf dem Client. Die Vorteile und Nachteile des *Fat Client* gelten weitgehend auch für den *Rich Client*.

■ **Thin Client** (manchmal auch als *Lean Client* oder *Slim Client* bezeichnet): In der Regel befindet sich nur die GUI-Schicht auf dem Client. Anwendungen können daher ohne Netzwerkverbindung *nicht* genutzt werden. Es wird die Plattform bzw. das Betriebssystem des Clients benutzt, um die Verbindung zum Server herzustellen.

14.1 Client-Server-Architektur I

- + Die Sicherheit einer Anwendung ist leichter sicherzustellen, da Vorteile nur die Server abgesichert werden müssen (*Single Point of Failure*).
- Die Hardware für den Client ist billiger.
- Ohne eine Netzwerkverbindung kann die Anwendung *nicht* ausgeführt werden. Nachteile
- Die Leistung der Server muss größer sein.
- Grafiklastige Anwendungen lassen sich nicht sinnvoll ausführen, wenn das Netz die Datenmenge nicht schnell genug verarbeiten kann.

■ **Ultra-Thin Client** (manchmal auch als *Zero Client* bezeichnet):

Im Gegensatz zum *Thin Client* wird kein vollständiges Betriebssystem benötigt. Stattdessen initialisiert einen Systemkern das Netzwerk, managt das Netzwerkprotokoll und die Ausgabe des Servers auf dem Client.

Auf der Serverseite ist zu überlegen, ob die Persistenz-Schicht auf einen weiteren Server ausgelagert werden soll oder nicht. Eine Auslagerung führt zu einer besseren Skalierung und Anpassbarkeit, verschlechtert aber die Leistung und erhöht den Programmieraufwand. Um die Anpassbarkeit zu verbessern, wird in der Regel zwischen die Anwendungs-Schicht und die Persistenz-Schicht eine Anpassungs-Schicht gelegt, wobei oft das DAO-Muster eingesetzt wird (siehe »Vom Direktzugriff bis zum JPA«, S. 422).

Die Abb. 14.1-1 zeigt einige Verteilungsalternativen bei einem zweistufigen Client-Server-Konzept, wenn die logischen Subsysteme nach dem Schichten-Muster strukturiert sind.

Eine Client-Server-Architektur besitzt folgende Vor- und Nachteile:

- + Da der Server pro Client eine Verbindung verwaltet, ist es einfach, Vorteile eine Benutzer-Sitzung zu verfolgen.
- + Durch den Einfluss auf die Laufzeitumgebung kann die Systemkomplexität reduziert werden.
- + Die bekannte Anzahl der maximalen nebenläufigen Benutzer erleichtert den Entwurf.
- Kaum skalierbar, d. h. nur aufwändig auf andere Benutzerzahlen Nachteile einzustellen.
- Aufwändige Verteilung der Client-Software, da sie auf jedem Client installiert werden muss.
- Plattformänderungen führen u.U. zur Neuprogrammierung von Anwendungsteilen.
- Eine Nutzung im Internet/Extranet ist nicht möglich oder nicht praktikabel.

Client-Server-Architekturen unterstützen das Prinzip der Bindung und Kopplung, da sie für eine lose Kopplung zwischen Clients und Server sorgen (siehe »Architekturprinzipien«, S. 29). Zusammenhänge

I 14 Verteilte Architekturen

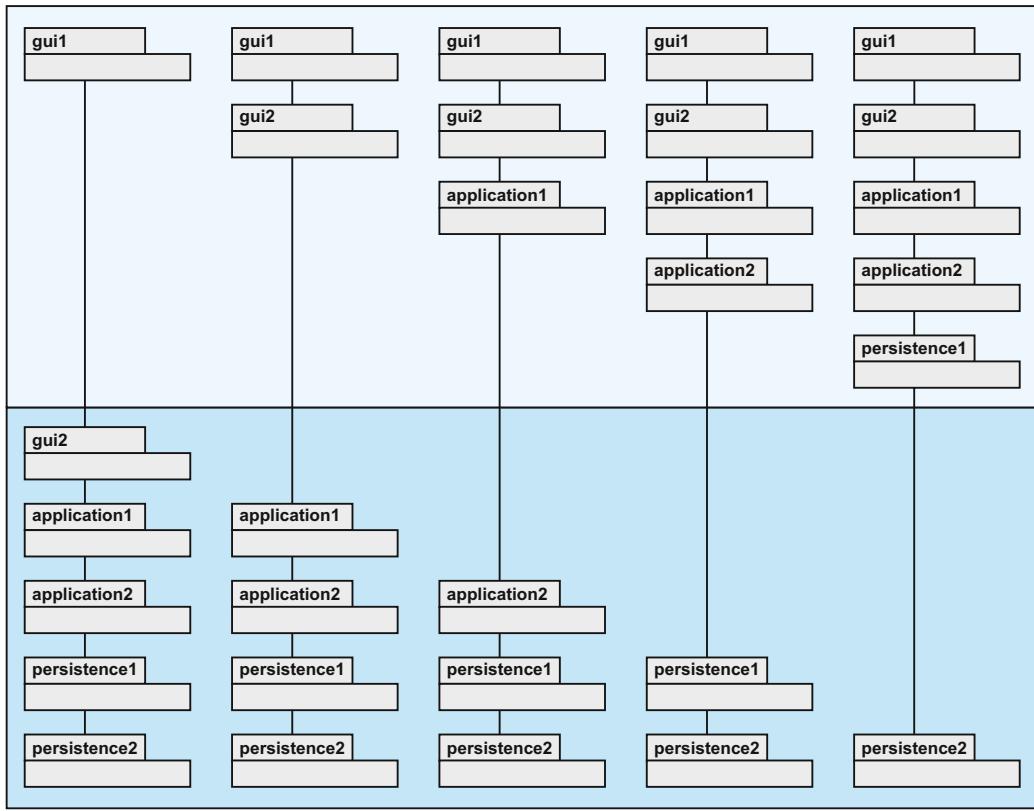


Abb. 14.1-1:
Verteilungsalternativen bei einem
zweistufigen
Client-Server-
Konzept.

Die nichtfunktionalen Anforderungen Installierbarkeit und Skalierbarkeit werden durch Client-Server-Architekturen *nicht* erfüllt, da auf jedem Client zusätzliche Programme installiert werden müssen und Client-Server-Architekturen nur aufwändig an wachsende Leistungsanforderungen angepasst werden können.

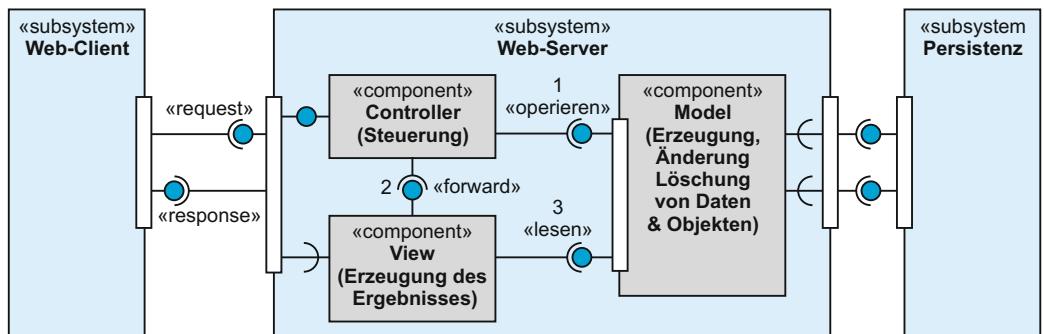
14.2 Web-Architektur

Bei einer **Web-Architektur** werden – ebenso wie bei einer Client-Server-Architektur (siehe »Client-Server-Architektur«, S. 193) – logische Schichten bzw. Subsysteme auf Clients und ein oder mehrere Server verteilt, wobei Web-Techniken eingesetzt werden.

Die klassische Web-Architektur

Auf den Clients befinden sich in der Regel Teile der GUI-Schicht. Diese Teile einer Anwendung werden in einem Webbrower ausgeführt. Der **Webbrower** ist sozusagen das Betriebssystem bzw. die Laufzeitumgebung des **Web-Clients**.

Auf der Serverseite nimmt ein **Web-Server** die Anfragen von Web-Clients entgegen und beantwortet sie. Zur Kommunikation wird das HTTP-Protokoll eingesetzt. Der Web-Server selbst enthält Teile der GUI-Schicht und eventuell der Applikations-Schicht. Es ist aber auch möglich, dass sich die Applikations-Schicht auf einem Applicationserver befindet. Die Persistenz-Schicht wiederum kann sich auf einem eigenen Server befinden. Den grundsätzlichen Aufbau einer Web-Architektur nach dem MVC-Muster (siehe »Das MVC-Muster (*model view controller pattern*)«, S. 62) zeigt die Abb. 14.2-1.



Eine Web-Architektur besitzt folgende Charakteristika:

- Durch das HTTP-Protokoll wird bei jeder Benutzer-Anfrage einer Web-Seite eine TCP-Verbindung mit dem Webbrower aufgebaut, eine Anfrage gesendet, vom Server bearbeitet und nach Rücksendung der Antwort die TCP-Verbindung wieder beendet. Es gibt also *keine* permanente Verbindung zwischen Web-Client und Web-Server.
 - Web-Anwendungen haben eine potenziell unbegrenzte Anzahl von Benutzern – mit Ausnahme von Intranet- und Extranet-Anwendungen.
 - Auf die Laufzeitumgebung des Web-Clients haben die Entwickler – außer bei Intranet- und Extranet-Anwendungen – *keinen* Einfluss.
- Welche Nachteile besitzt diese Web-Architektur?

Bei dieser Web-Architektur erfolgt die gesamte Verarbeitung auf dem Server. Auch die Vorbereitung der Ausgabe geschieht serverseitig. Dadurch wird der Server zum Engpass. Das *Rendering* von Daten, d.h. die Aufbereitung für die HTML-Ausgabe, bzw. deren Gruppierung und Filterung kann genauso gut durch den Web-Client erfolgen.

Abb. 14.2-1: Web-Architektur nach dem MVC-Muster.

Frage
Antwort

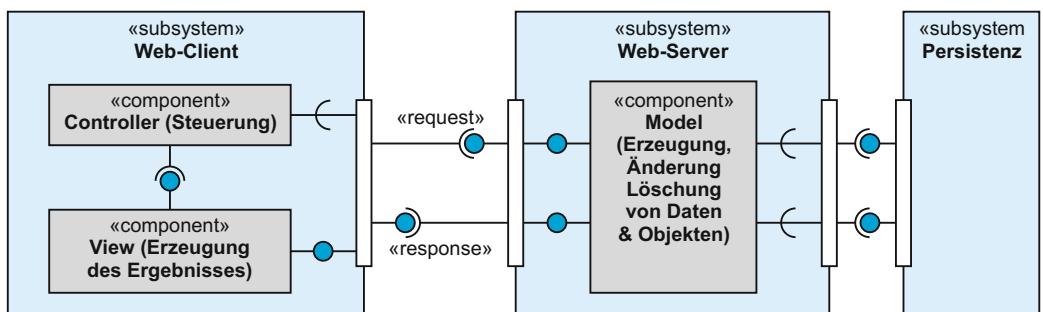
I 14 Verteilte Architekturen

RIA-Web-Architekturen

- Client Während in den Anfangszeiten des Web einen Webbrowser nur HTML-Seiten anzeigen musste bzw. konnte, gibt es heute eine Vielzahl von Möglichkeiten, den Web-Client mit zusätzlichen Funktionalitäten auszustatten:
- **HTML5**: Neue Möglichkeiten z.B. für die Einbettung von Multi-media-Inhalten, Unterstützung von neuen Interaktionselementen in Formularen, semantische Auszeichnung von Webseiten, Offline arbeiten, Zeichnen in Webseiten.
 - **CSS (Cascading Style Sheets)**: Gestaltung der Benutzeroberfläche.
 - **JavaScript**: Dynamische Veränderung von Webseiten, Überprüfung von Formulareingaben.
 - **AJAX (Asynchronous JavaScript and XML)**: Technik, die es erlaubt, Daten mit dem Web-Server auszutauschen, ohne eine HTML-Seite neu zu laden. Realisiert wird dies mit asynchronen JavaScript-Aufrufen, die für den Benutzer unsichtbar sind.
 - **Java-Applets**: Ausführung von Java-Programmen im Webbrowser.
 - **ActiveX**: Ausführung von Programmen im Internet Explorer von Microsoft.
 - Verwendung von im Web eingebetteten Ablaufumgebungen (*plugins*), z.B. **Adobe Flash**, Adobe Flex, Microsoft Silverlight (siehe »GUIs in der .NET-Plattform«, S. 467), JavaFX, OpenLazlo, Curl.
- Server Für die Realisierung des Web-Servers gibt es ebenfalls eine Reihe von Techniken:
- **JSP (JavaServer Pages)** und/oder **Servlets**
 - **JavaServer Faces** (JSF)
 - ActiveServer Pages (ASP) bzw. ActiveServer Pages.NET (ASP.NET), siehe »GUIs in der .NET-Plattform«, S. 467
 - Java EE (siehe »Die Java EE-Plattform«, S. 321)
- RIA Im Zusammenhang dieser neuen Techniken hat Jeremy Allaire im Jahr 2002 den neu erfundenen Begriff *Rich Internet Application (RIA)* eingeführt [Alla02]. Folgende Anforderungen lassen sich an *Rich Internet Applications* formulieren:
- Es handelt sich um Client-Server-Anwendungen. Dabei sind Client und Server über das Internet verbunden.
 - Der Client übernimmt einen größeren Teil der Verarbeitungsleistung als bei klassischen Web-Anwendungen. Dabei stellt der Client eine Laufzeitumgebung zur Verfügung, die die Darstellung von Inhalten, die Ausführung von Programmcode und die Kommunikation im Internet zusammenführt.

- Das HTML-Seitenkonzept von klassischen Web-Anwendungen wird »aufgeweicht«. Es findet eine fortlaufende Aktualisierung der Benutzeroberfläche – basierend auf einem asynchronen Datenaustausch – statt. Im Web vorhandene Datenquellen und Webservices können direkt vom Client genutzt werden.
- Eine RIA ist meist auf allen gängigen Betriebssystem- und Browser-Plattformen ohne sichtbare Unterschiede verfügbar. Das *Deployment* auf verschiedene Plattformen und Geräte muss einfach möglich sein.
- Die Benutzeroberfläche ist ähnlich den Standard-Desktop-Anwendungen, z. B. werden Menüs und Drag&Drop unterstützt.
- Die Zustandsverwaltung wandert vom Server auf den Client. Die Kommunikation zwischen Client und Server kann daher zustandslos erfolgen.
- Der Client kann gegebenenfalls auch offline, d. h. ohne Verbindung zum Server, arbeiten. Eine lokale Datenhaltung, die sich bei einer online-Verbindung mit dem Server synchronisiert, ist möglich.

Die Abb. 14.2-2 zeigt eine RIA-Web-Architektur.



Mit dem Einsatz von AJAX ist es möglich, einen Teil der Funktionalität auf den Client zu verlagern. Bei modernen Web-Clients werden keine gerenderten Inhalte mehr zwischen Client und Server ausgetauscht, sondern es werden Entitäten mittels XML oder JSON (*JavaScript Object Notation*) übertragen. Durch JavaScript oder XSLT werden die Daten zur Darstellung im Browser aufbereitet.

Bei den Web-Architekturen SOFEA (*Service-Oriented Front-End Architecture*) und SOUI (*Service-Oriented User Interface*) – gesprochen *soo-wee!* – greift der Browser über REST (siehe »REST«, S. 286) und Webservices (siehe »SOAP«, S. 262) direkt auf den Webserver zu. Daraus ergibt sich, dass die meisten Zustandsdaten der Anwendung sich im Browser befinden. Dadurch skaliert die Webanwendung besser, da durch den Verzicht von Zustandsdaten auf dem Server viele Skalierungsprobleme gelöst werden.

[NeSc10]

Abb. 14.2-2:
Beispiel für eine
RIA-Web-
Architektur.

SOFEA, SOUI

Literatur

I 14 Verteilte Architekturen

Entscheidungen

Für die Softwarearchitektur muss entschieden werden, welche dieser Möglichkeiten eingesetzt werden sollen. Insbesondere ist auf Inkompatibilitäten der Techniken zu achten. Außerdem ist zu entscheiden, wie die Kommunikation mit dem Applikations-Server und dem Persistenz-Server geschehen soll.

Web-Architekturen besitzen folgende Vor- und Nachteile:

| | |
|---------------|--|
| Vorteile | <ul style="list-style-type: none">+ Einsatz im Internet, Extranet und Intranet ohne Architekturwechsel möglich.+ Erlaubt in der Regel eine hohe Benutzeranzahl.+ Gute Skalierbarkeit und gute Wartbarkeit.+ Keine Verteilungsprobleme, da keine anwendungsspezifische Software auf dem Web-Client installiert werden muss. |
| Nachteile | <ul style="list-style-type: none">- Schlechte Antwortzeiten des GUI und der Anwendung. Der Benutzer nimmt Verzögerungen wahr.- Eingeschränkte GUI-Möglichkeiten, z. B. kein volles <i>Drag & Drop</i>, keine Nutzung von Tastaturkürzeln (<i>shortcuts</i>).- Integration von Desktop-Anwendungen schwierig möglich.- Um Inhalte, wie z. B. PDF-Dokumente, anzeigen zu können, muss der Webbrowser über entsprechende Plug-ins verfügen.- Manche Browser-Erweiterungen sind nicht nutzbar, da der Benutzer Sicherheitseinschränkungen aktiviert hat, z. B. werden keine Pop-Up-Fenster angezeigt.- Manche Webbrowser sperren häufig verwendete Plug-ins, z. B. wird auf dem iPad und dem iPhone von Apple kein Flash angezeigt.- Durch das verbindungslose und sitzungslose HTTP/1.0-Protokoll ist es aufwändig, den Zustand während und zwischen Sitzungen zu speichern und zu verfolgen.- Es müssen in der Regel mehrere unterschiedliche Webbrowser unterstützt werden, die sehr häufigen Versionswechseln unterworfen sind. |
| Zusammenhänge | <p>Web-Architekturen unterstützen das Prinzip der Bindung und Kopp lung, da sie für eine lose Kopplung zwischen Clients und Server sor gen (siehe »Architekturprinzipien«, S. 29).</p> <p>Die nichtfunktionalen Anforderungen Installierbarkeit und Skali erbarkeit werden durch Web-Architekturen erfüllt, da auf den Cli ents keine zusätzlichen Programme installiert werden müssen und Web-Architekturen gut an wachsende Leistungsanforderungen ange passt werden können (siehe »Nichtfunktionale Anforderungen«, S. 109).</p> |

14.3 SOA – Serviceorientierte Architekturen

In großen Unternehmen gibt es oft vielfältige Anwendungen, die auf unterschiedlichen Plattformen laufen und nach unterschiedlichen Architekturparadigmen entwickelt wurden. Die Zusammenarbeit zwischen solchen Anwendungen sicherzustellen ist schwierig. Mangelnde Flexibilität erschwert die Unterstützung sich ändernder Geschäftsprozesse. Die Kommunikation mit unternehmensexternen Anwendungen ist aufwändig herzustellen.

Die Antwort auf diese Probleme stellt das Architekturkonzept **SOA** (serviceorientierte Architektur) dar – bisweilen auch als Architekturprinzip, Architekturstil, Architekturparadigma oder Architekturmuster bezeichnet.

Die Idee von SOA besteht darin, sinnvoll zusammengehörende Anwendungsfunktionalitäten zu einer Einheit zusammen zu fassen und als Dienst(leistung) – **(SOA-)Service** genannt – zur Verfügung zu stellen. Durch das (flexible) Zusammensetzen bzw. Kombinieren mehrerer Services – **Orchestrierung** genannt – können dann komplexe Anwendungen realisiert werden.

Ein Ziel von SOA ist es, dem Nutzer die Möglichkeit zu geben, große Funktions-»Brocken« zusammenzufassen, um ad hoc-Anwendungen aus fast vollständig vorhandenen Services zu erstellen. Neue Geschäftsanforderungen können durch das »Einstöpseln« (*plug-in*) neuer Services oder durch das Upgraden existierender Services in einer granularen Art und Weise realisiert werden. Ein weiteres Ziel besteht darin, Alt-Anwendungen (*legacy applications*) als Services verpackt weiterhin zu nutzen und dadurch die existierenden IT-Investitionen abzusichern.

Schwierig ist es, den richtigen Grad der Granularität für die Funktionalität der Services zu finden. Je mehr Funktionalität ein Service zur Verfügung stellt, desto weniger Schnittstellen werden benötigt, um eine Anwendung zu realisieren. Sehr umfangreiche Funktionalitäten sind jedoch für eine leichte Wiederverwendung *nicht* gut geeignet. Auf der anderen Seite führt jede Schnittstelle zu einem Verarbeitungs-Overhead, sodass Effizienzgesichtspunkte bei der Granularität der Services eine Rolle spielen. Damit ein solches Konzept funktionieren kann, muss eine Architektur folgende Eigenschaften besitzen (Abb. 14.3-1):

- Jeder **SOA-Service** besitzt eine plattformunabhängige, standardisierte und selbstbeschreibende Schnittstelle, die angibt, welche Eingaben erforderlich sind und welcher Art das Ergebnis ist. Die Art und Weise, wie der Service seine Leistung erbringt, ist dem Nutzer nicht bekannt. Der Service kann in einer beliebigen

Probleme

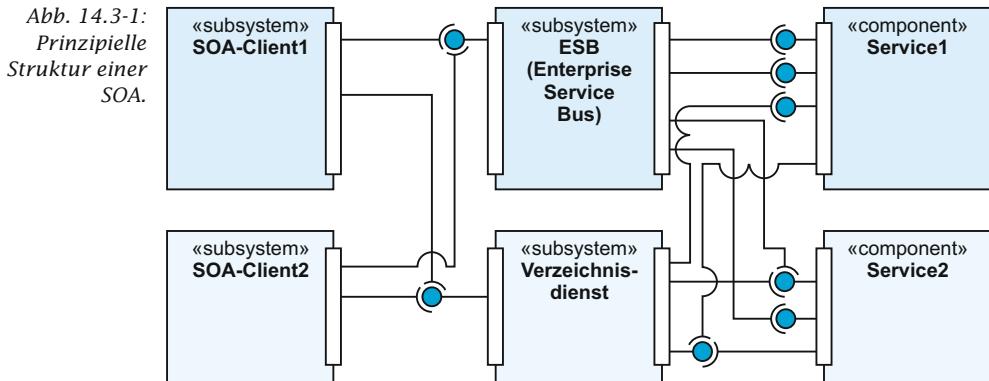
SOA-Konzept

Ziele

Granularität

SOA(rchitektur)

I 14 Verteilte Architekturen



Programmiersprache auf einer beliebigen Plattform realisiert sein bzw. werden. Jeder SOA-Service beschreibt seine Dienstleistung in einem **Verzeichnisdienst** (*registry*).

- Ein SOA-Nutzer bzw. ein SOA-Client kann über einen Verzeichnisdienst in Erfahrung bringen, welche SOA-Services zur Verfügung stehen. Über den Verzeichnisdienst erhält er eine Referenz auf den gewünschten Service.
- Die Kommunikation zwischen SOA-Nutzer und SOA-Service(s) kann über beliebige Netzwerkprotokolle erfolgen.
- Die Integration mehrerer Services kann über Punkt-zu-Punkt-Verbindungen realisiert werden. Bei einer großen Anzahl von Kommunikationswegen sollte ein Vermittler – auch Message-Broker genannt – zwischengeschaltet werden, der Protokollumwandlungen, Filteraufgaben und das Umleiten (*routing*) von Nachrichten übernimmt. Ist der Vermittler beliebig erweiterbar und protokollunabhängig aufgebaut, dann spricht man von einem **ESB** (*Enterprise Service Bus*). Services werden flexibel und dynamisch auf der Basis eines Nachrichtenkonzepts asynchron – ohne zusätzliche Programmierung – gekoppelt.
- Für die Orchestrierung von Services wird eine geschäftsprozessorientierte Sprache verwendet.

Das konzeptionelle Modell einer SOA-Architektur zeigt die Abb. 14.3-2. Der Service-Konsument erhält die Service-Beschreibung vom Service-Broker oder direkt vom Provider (der Broker ist optional).

Notation Mithilfe der grafischen Modellierungssprache **SoaML** kann man SOA-Services mittels eines erweiterten UML-Profilen beschreiben [OMG09].

Beispiel Ein Unternehmen sucht Mitarbeiter und möchte Stellenanzeigen in Zeitungen und Zeitschriften aufgeben. Über einen Verzeichnisdienst wird ermittelt, in welchen Zeitungen Stellenanzeigen möglich sind.

14.3 SOA – Serviceorientierte Architekturen I

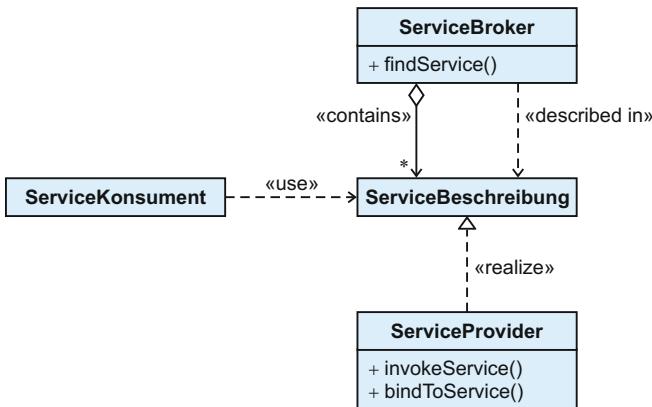


Abb. 14.3-2:
Konzeptionelles
Modell einer SOA-
Architektur.

Ein Service soll idealerweise folgende Eigenschaften besitzen – in der Praxis werden nicht alle Eigenschaften erfüllt [BKM07, S. 15]:

- Ein Service ist selbsterklärend, hochgradig modular und kann unabhängig installiert werden.
- Ein Service ist eine verteilte Komponente, d. h. er ist über ein Netzwerk verfügbar. Über einen Namen oder über eine Suche kann er lokalisiert werden.
- Ein Service besitzt eine veröffentlichte Schnittstelle (Vertrag). Nutzer des Services müssen nur die Schnittstelle zur Kenntnis nehmen.
- Ein Service ist plattformunabhängig. Service-Nutzer und Service-Anbieter können unterschiedliche Implementierungssprachen und Plattformen verwenden.
- Ein Service ist auffindbar. Er kann sich bei einem Verzeichnisdienst registrieren, sodass Nutzer ihn finden.
- Ein Service ist dynamisch gebunden. Bei der Erstellung einer Anwendung muss der Service noch nicht vorhanden sein. Er wird erst bei der Ausführung lokalisiert und eingebunden.

Eine SOA(rchitektur) besitzt folgende Vor- und Nachteile:

- | | |
|--|----------|
| <ul style="list-style-type: none"> + Ermöglicht es, unterschiedliche Anwendungen in einem Unternehmen zu integrieren. + Externe Services können flexibel und zuverlässig integriert werden. + Die Kosten der Softwareentwicklung werden gesenkt, da ab einem bestimmten Zeitpunkt alle notwendigen Services bereits vorhanden sind und diese nur noch kombiniert werden müssen. + Durch die Wiederverwendung bestehender Services wird eine hohe Flexibilität der Geschäftsprozesse ermöglicht. + Es ist eine Wiederverwendung auf der Makro-(Service-)Ebene möglich, im Gegensatz zur Mikro-(Klassen-)Ebene. + Ein Service ist autonom, Zustandslos, besitzt voll dokumentierte Schnittstellen und ist getrennt von quer schneidenden Belangen. | Vorteile |
|--|----------|

I 14 Verteilte Architekturen

| | |
|-------------------|--|
| Nachteile | <ul style="list-style-type: none">— Hoher Anfangsaufwand, da Einsparungen erst auftreten, wenn grundlegende Services bereits existieren und in mehreren Anwendungsbereichen genutzt werden können.— Aufwändig, Services zu entkoppeln.— Durch komplexere Abläufe schwierige Fehlersuche und aufwändige Tests.— Schlechtere Laufzeiteffizienz durch Verwendung standardisierter Protokolle, die einen Zusatzaufwand erfordern.— In einer Anwendung eingebaute Sicherheitskonzepte funktionieren nicht in SOA. |
| SOA-Infrastruktur | <p>Um eine SOA(rchitektur) zu realisieren, wird eine SOA-Infrastruktur benötigt, die wie folgt aussehen kann:</p> <ul style="list-style-type: none">■ Format des zu übertragenden Inhalts: XML■ Schnittstellenbeschreibung eines Services: WSDL (siehe »SOAP-Nachrichten, Webservices und WSDL«, S. 263).■ Kommunikation zwischen den Services (Kommunikationsprotokoll): SOAP (siehe »SOAP«, S. 262), XML-RPC (siehe »XML-RPC«, S. 253), REST (siehe »REST«, S. 286).■ Netzwerkprotokolle: IIOP, DCOM.■ Übertragungsprotokolle (Transportprotokolle): HTTP, SMTP, FTP■ Verzeichnisdienst: UDDI (siehe »UDDI«, S. 280).■ Orchestrierung: BPEL■ Services werden in der Regel als Webservices realisiert (siehe »SOAP«, S. 262). |
| Prozessmodelle | Ist es das Ziel, eine serviceorientierte Architektur zu entwickeln, dann hat dies Auswirkungen auf das Prozessmodell. Zu den SOA-spezifischen Prozessmerkmalen gehören die Serviceidentifikation, die Serviceselektion, die Servicekomposition und die Servicesteuierung. In [TLS10] werden neun verschiedene Prozessmodelle im Quervergleich dargestellt. In [WHS11] wird ein Reifegradmodell für SOA analog zum CMMI-Modell vorgestellt. |
| GUI | Ein Problem in SOA ist, wie eine einheitliche Benutzeroberfläche für die verschiedenen Services bereitgestellt werden kann. In der Literatur gibt es dafür verschiedene Vorschläge: [Nand11], [Stec10] und [HoLa10]. |
| Muster | Architektur- und Entwurfs-Muster sind erst im Entstehen. Vorschläge dazu gibt es zum Beispiel in [Stal06] und [SaBh08]. |
| Zusammenhänge | SOA unterstützt das Prinzip der Bindung und Kopplung, da sie für eine lose Kopplung zwischen den Services sorgt, und das Prinzip der Modularisierung, da jeder Service die Eigenschaften eines Moduls besitzt (siehe »Architekturprinzipien«, S. 29). Die nichtfunktionale Anforderung Weiterentwickelbarkeit wird durch SOA ermöglicht, da Änderungen von Service-Implementierungen die Nutzer von Services nicht beeinträchtigen (siehe »Weiterentwickelbarkeit«, S. 119). |

15 Arten der Netzkomunikation

Bei den meisten Softwaresystemen handelt es sich heute um verteilte Systeme, d.h. die einzelnen Subsysteme sind auf zwei oder mehrere Computersysteme verteilt. Die beteiligten Computersysteme sind miteinander vernetzt (leitungsgebunden, z.B. LAN, oder leitungsungebunden, z.B. WLAN, UTMS). Für die Softwareentwicklung stellt sich die Frage, wie die Kommunikation zwischen den einzelnen verteilten Subsystemen hergestellt werden kann.

Um die Softwareentwicklung zu erleichtern, soll es für den Softwareentwickler *keine* Rolle spielen, wo sich ein Subsystem physisch befindet. Der Zugriff auf andere Subsysteme soll daher ortstransparent erfolgen. Ein Client muss den physischen Standort eines Server-Objekts *nicht* kennen. Das Objekt kann auf demselben Computersystem liegen, auf einem Unternehmens-Server oder irgendwo im Internet. Wenn der Server nicht auf demselben Computersystem liegt wie der Client, dann spricht man von einem entfernten Objekt, sonst von einem lokalen.

Orts-
transparenz

Die **Ortstransparenz** bietet auf zwei Ebenen Vorteile:

- **Implementierung:** Bei der Implementierung einer Operation bzw. einer Methode in einer Programmiersprache können entfernte Objekte im Wesentlichen genauso behandelt werden wie lokale Objekte. Die Bibliotheken und das Laufzeitsystem sorgen dafür, dass Operationaufrufe, die über ein Netzwerk übertragen werden müssen, auf der Client-Seite genauso durchgeführt werden können, wie Aufrufe eines lokalen Objekts. Auf der anderen Seite braucht ein Server sich *nicht* darum zu kümmern, ob ein Aufruf »vom Objekt nebenan« kommt oder von einem anderen Kontinent. Ein Programmierer kann sich so gut auf die zu realisierende Geschäftslogik konzentrieren und wird nicht mit Details der verteilten Kommunikation aufgehalten.
- **Administration:** Skalierbarkeit und Verfügbarkeit (siehe »Zuverlässigkeit«, S. 124) sind wichtige Anforderungen an verteilte Softwaresysteme. Da der Aufenthaltsort eines Subsystems transparent ist, kann es leicht von einem Server auf einen anderen verschoben werden, ohne dass Änderungen an seinen Clients nötig wären. Auf diese Weise kann die Verfügbarkeit erhöht oder die Leistung (siehe »Leistung und Effizienz«, S. 128) durch das Hinzufügen weiterer oder leistungsfähigerer Server bei Bedarf relativ leicht verbessert werden (Abb. 15.0-1).

I 15 Arten der Netzkommunikation

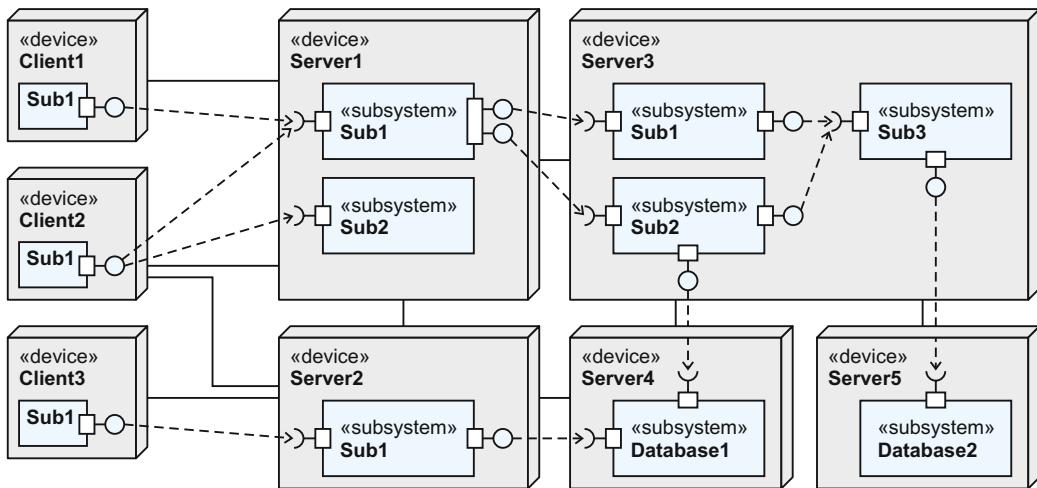


Abb. 15.0-1:

Lastverteilung durch Verteilung von Subsystemen auf mehrere Server.

Wenn ein Server ausfällt, kann ein anderer Server automatisch einspringen. Datenobjekte müssen dann nicht auf dem neuen Server erzeugt werden (*recovery*). Die Aufrufe von Clients können jedoch von der Plattform einfach an die neue Adresse geleitet werden, so dass der Ausfall unbemerkt bleibt.

Ein anderes Beispiel betrifft die automatische Lastenverteilung. Wenn auf einem Server so viele Prozessobjekte mit der Ausführung von Geschäftsprozessen beschäftigt sind, dass er ausgelastet ist, können weitere Prozessobjekte auf Anfrage von Clients auf einem anderen Server angelegt werden.

Namensdienst

In der Regel wird ein Namensdienst benötigt, der die Verbindung zu entfernten Objekten herstellt. Dieser Dienst liefert einem Client eine Referenz zu einem bestimmten Objekt oder legt ein Objekt einer bestimmten Komponente »irgendwo« an und liefert eine Referenz darauf.

In Abhängigkeit von der verwendeten Architektur, der oder den verwendeten Plattformen und der oder den verwendeten Programmiersprachen gibt es verschiedene Möglichkeiten der Netz-Kommunikation, die jeweils an einer Fallstudie demonstriert werden:

- »Sockets«, S. 207
- »RMI«, S. 220
- »CORBA«, S. 241
- »XML-RPC«, S. 253
- »SOAP«, S. 262
- »REST«, S. 286
- »Netzkommunikation in .NET«, S. 301

Da bei einer verteilten Anwendung über eine langsame Netzverbindung kommuniziert werden muss, sind einige Einschränkungen beim Entwurf zu beachten:

- »Entwurfskonzepte für verteilte Anwendungen«, S. 306

Die Charakteristika und Anwendungsgebiete werden noch einmal vergleichend dargestellt:

- »Vergleich der Konzepte«, S. 315

15.1 Sockets

Eine plattformunabhängige Möglichkeit der Client-Server-Kommunikation besteht in der Verwendung von sogenannten *Sockets*. Überetzt bedeutet der englische Begriff *Socket* Steckdose oder Sockel.

Für die Kommunikation werden Übertragungsprotokolle wie TCP (*transmission control protocol*) oder UDP (*user datagram protocol*) verwendet.

Sockets verbergen die Details der Netzwerkkommunikation. Anwendungen können dadurch unabhängig von der Programmiersprache, dem Betriebssystem und der Netzwerk-Schnittstelle funktionieren. Die Programmierschnittstelle (API) für *Sockets* wird von allen gängigen Betriebssystemen unterstützt. Aus der Sicht einer Anwendung ist ein *Socket* ein Zugang zu einem Netzwerk.

Das *Sockets*-API ermöglicht eine verbindungsorientierte (über TCP) Kommunikation (*stream sockets*) oder eine verbindungslose (über UDP) Kommunikation (*datagram sockets*).

Damit der Client den Server kontaktieren kann, muss der Server durch eine IP-Adresse und der eigentliche Service bzw. die Ziel-Anwendung über eine Port-Nummer identifizierbar sein. Die IP-Adresse bestimmt eindeutig den Server, die Port-Nummer gibt den bereitgestellten Dienst des Servers an. Jeder Dienst, den ein Server anderen zur Verfügung stellt, läuft auf einem anderen Port. Die Port-Nummern decken den Bereich von 0 bis 65535 ab. Die Nummern 0 bis 1023 sind für spezielle Dienste reserviert (*Well-Known-System-Ports, Contact Ports*). Zu vermeiden sind ebenfalls Ports, die von verbreiteten Programmen genutzt werden, z. B. Port 8080.

Sockets können auch dazu verwendet werden, um zwischen Programmen auf demselben Computersystem Daten auszutauschen (Interprozesskommunikation).

- + Leicht, aber aufwändig zu programmieren. Sprachen-, plattform- und betriebssystemunabhängig. Vorteile
- + Schnelle Einarbeitung möglich.
- + Kleine Anwendungen können mit wenig Aufwand erstellt werden, da keine vorgefertigten Strukturen beachtet oder Konventionen eingehalten werden müssen.

I 15 Arten der Netzkommunikation

- + Sehr anpassungsfähig.
- + Geringer Overhead.
- Nachteile
 - Für den Entwickler *nicht transparent*.
 - *Nicht* objektorientiert, d.h. der Entwickler muss sich ein eigenes Protokoll überlegen, um anwendungsorientiert auf den Server zugreifen zu können. Es werden Zeichen-Datenströme (bei *stream sockets*) oder einzelne Nachrichten (bei *datagram sockets*) versandt, aber keine Objekte.
 - Jedes Detail muss ausprogrammiert werden (Verbindungsaufbau und -abbau, Nebenläufigkeit, Verschlüsselung uvm.).
 - Die zu übertragenden Daten müssen für die Übertragung vor- und nachbereitet werden.
 - Für Methodenaufrufe usw. wird eine Zwischenschicht benötigt.
- | Zunächst werden TCP-Sockets näher betrachtet und an einem Beispiel erläutert:
 - »TCP-Sockets«, S. 208
- Anschließend wird die Funktionsweise von UDP-Sockets an einem Beispiel erklärt:
 - »UDP-Sockets«, S. 213
- Die Fallstudie »Kundenverwaltung-Mini« wird mit TCP-Sockets realisiert:
 - »Fallstudie: KV – Sockets«, S. 216

15.1.1 TCP-Sockets

- Zur Erstellung von TCP-Sockets sind folgende Schritte notwendig:
- Serverseite Auf der Serverseite:
- 1 *Socket* erstellen und an eine Adresse (Port) binden, über die Anfragen akzeptiert werden sollen.
 - 2 Ein Serverprozess »lauscht« dann an dem Port auf Verbindungswünsche (*listen*).
 - 3 Bei einem Verbindungswunsch erzeugt er einen Kommunikationssocket (*accept*). Der ursprüngliche *Socket* bleibt erhalten und nimmt weiterhin Verbindungswünsche entgegen.
 - 4 Der Kommunikationssocket erhält eine andere Nummer als der Verbindungswunschsocket und wird nur für die Kommunikation mit einem Client verwendet. Die Verbindung bleibt so lange bestehen, bis die Verbindung von einer der beiden Seiten beendet wird.
- Clientseite Auf der Clientseite:
- 1 *Socket* erstellen und mit der Server-Adresse verbinden, von der Daten angefordert werden sollen (*connect*).
 - 2 Der Client verbindet sich mit dem Kommunikationssocket.
 - 3 Senden und Empfangen von Daten.
 - 4 Trennen der Verbindung und Schließen des *Socket*.

15.1 Sockets I

Java unterstützt die Socket-Programmierung durch die Klassen im Paket `java.net`. Die Implementierung des *Sockets* für verschiedene Plattformen erfolgt in der Klassenbibliothek der virtuellen Maschine.

Java

Die Klasse `ServerSocket` implementiert *Server Sockets*.

ServerSocket

- `public ServerSocket(int port) throws IOException:` Erzeugt ein *Server Socket* und bindet es an den spezifizierten Port. Eine Portnummer 0 erzeugt ein *Socket* und bindet es an einen freien Port.
- `public Socket accept() throws IOException:` Wartet auf Verbindungswünsche und akzeptiert sie. Die Methode blockiert, bis eine Verbindung hergestellt ist.
- `public void close() throws IOException:` Schließt die Verbindung.

Konstruktor
Wichtige Methoden

Die Klasse `Socket` implementiert *Client Sockets*.

Socket

- `public Socket(String host, int port) throws UnknownHostException, IOException:` Erzeugt einen *Stream Socket* und verbindet es mit der spezifizierten Port-Nummer auf dem angegebenen *Host*.
- `public InputStream getInputStream() throws IOException:` Gibt einen *Input Stream* für diesen *Socket* zurück.
- `public OutputStream getOutputStream() throws IOException:` Liefert einen *Output Stream* für diesen *Socket*.
- `public void close() throws IOException:` Schließt die Verbindung.

Konstruktor
Wichtige Methoden

Dieses Beispiel zeigt eine einfache Client-Server-Kommunikation. Das Programm auf der Server-Seite sieht folgendermaßen aus:

Beispiel:
`DemoSocketTCP`

`package de.w3l.server;`

Java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class DemoSocketServer
{
    private ServerSocket listenSocket;

    public DemoSocketServer(int port) throws IOException
    {
        //Socket reservieren
        listenSocket = new ServerSocket(port);
    }

    private void starteDienstleistung()
    {
        System.out.println("Start des Servers");
        while ( true )
        {
            Socket commSocket = null;
```

I 15 Arten der Netzkomunikation

```
try
{
    //Auf Verbindungswunsch warten
    commSocket = listenSocket.accept();
    kommunizieren(commSocket);
}
catch (IOException e)
{
    System.out.println("Accept fehlgeschlagen");
    System.exit(-1);
}
finally
{
    if (commSocket != null)
        try
        {
            commSocket.close();
            System.out.println("Ende des Servers");
        }
        catch ( IOException e )
        {
            e.printStackTrace();
        }
} //end finally
} //end while
}

private void kommunizieren(Socket commSocket)
throws IOException
{
    //Ströme vorbereiten
    BufferedReader in =
        new BufferedReader(new InputStreamReader(
            commSocket.getInputStream()));
    PrintWriter out = new PrintWriter(
        commSocket.getOutputStream(), true);
    //Eine Zeile lesen
    String nachricht1 = in.readLine();
    System.out.println(
        "Server - Erhielt Nachricht: " + nachricht1);
    //Weitere Zeile lesen
    String nachricht2 = in.readLine();
    System.out.println(
        "Server - Erhielt Nachricht: " + nachricht2);
    //Zeile schreiben
    out.println( nachricht1 + " Server " + nachricht2 );
    System.out.println(
        "Server - Sendete Nachricht: " + nachricht1 +
        " Server " + nachricht2 );
    out.close();
    in.close();
}

public static void main( String[] args ) throws IOException
{
```

15.1 Sockets I

```
DemoSocketServer server = new DemoSocketServer( 4445 );
server.starteDienstleistung();
}
}
```

Ausgabe auf der Server-Konsole:

```
Start des Servers
Server - Erhielt Nachricht: Hallo
Server - Erhielt Nachricht: World
Server - Sendete Nachricht: Hallo Server World
Ende des Servers
```

Das Programm auf dem Client sieht wie folgt aus:

```
package de.w3l.client; Java

import java.io.*;
import java.net.*;

public class DemoSocketClient
{
    public static void main( String[] args )
        throws java.io.IOException
    {
        Socket server = null;
        BufferedReader in = null;
        PrintWriter out = null;
        try
        {
            //Neues Socket-Objekt erzeugen
            server = new Socket("localhost", 4445);
            in = new BufferedReader(new InputStreamReader(
                server.getInputStream()));
            out = new PrintWriter(server.getOutputStream(),true);
            System.out.println("Start des Client");
            //Daten an den Server übertragen
            out.println("Hallo");
            out.println("World");
            //Ergebnis vom Server empfangen
            String ergebnis = in.readLine();
            System.out.println("Ergebnis vom Server: "
                + ergebnis );
        }
        catch (UnknownHostException e)
        {
            System.err.println("Der Server ist unbekannt");
            System.exit(1);
        }
        catch (IOException e)
        {
            System.err.println(
                "Keine Verbindung zum Server herstellbar");
            System.exit(1);
        }
    }
}
```

I 15 Arten der Netzkomunikation

```
finally
{
    if ( server != null )
    try
    {
        out.close();
        in.close();
        server.close();
        System.out.println("Ende des Client");
    }
    catch ( IOException e )
    {
        e.printStackTrace();
    }
}
}
```

Ausgabe auf der Client-Konsole:

```
Start des Client
Ergebnis vom Server: Hallo Server World
Ende des Client
```

In diesem Beispiel befinden sich Client und Server auf einem Computersystem. Daher ist als *Host* "localhost" angegeben. Als Port-Nummer wurde eine beliebige Zahl gewählt.



Wenn Sie über zwei Computersysteme verfügen, die miteinander vernetzt sind, dann installieren Sie auf einem Computersystem das Client-Programm und auf dem anderen Computersystem das Server-Programm. Tragen Sie in dem Client-Programm als *Host* den Computernamen Ihres Server-Computers ein. Wenn Sie den Computernamen nicht mehr wissen, dann finden Sie ihn im Windows-Betriebssystem unter Systemsteuerung -> System -> Computername.

Unterstützung mehrerer Clients

Die Anfragen mehrerer Clients können am selben Port ankommen, d. h. am selben `ServerSocket`. Verbindungsanfragen von Clients werden im Port in eine Warteschlange eingeordnet. Der Server muss die Anfragen sequenziell akzeptieren (`accept()`). Er kann die Anfragen aber simultan durch die Benutzung von *Threads* bedienen – ein *Thread* pro Client-Verbindung. Effizienter ist es, nicht pro Anfrage einen *Thread* zu erzeugen, sondern einen *Thread-Pool* zur Verwaltung zu verwenden. Dies ist in Java z. B. mithilfe der Klassen im Paket `java.util.concurrent` zu realisieren.

- Vorteile
- + Es wird eine dauerhafte, gleichberechtigte Verbindung zwischen zwei Computersystemen aufgebaut. Clients und Server können senden und empfangen. Die Verbindung bleibt für die Dauer der Übertragung bestehen.

- + Pakete werden geordnet und zuverlässig über eine Verbindung transportiert.
- + Mehrere Anfragen können nebenläufig verarbeitet werden.
- + Gut geeignet für den reinen Datentransfer, wie Updates, Backups, Nachladen von Modulen.
- + Gut geeignet für Kleinst-Aufgaben wie Überprüfung, ob der Server online ist, Zeitsynchronisation, Logging.
- Es werden reine Daten versandt, die vom Empfänger verarbeitet werden müssen. Ein Aufruf von Funktionen ist nur möglich, wenn ein eigenes Protokoll dazu entwickelt wird.
- Langsamere Übertragung verglichen mit UDP.
- Bei Netzwerkproblemen kann die Verbindung zwischen Client und Server abreißen.

15.1.2 UDP-Sockets

UDP-Sockets verwenden das verbindungslose UDP-Protokoll (*user datagram protocol*), das auf dem Internet-Protokoll aufsetzt und eine ungesicherte Übertragung erlaubt. Analog wie TCP werden Dienstleistungen vom Server über einen Port angeboten. Mehrere Server können an unterschiedlichen Ports ihre Dienste anbieten. UDP-Sockets stellen *keine* feste Verbindung zum Server her. Jedes *Datagram* wird einzeln verschickt und kann auf unterschiedlichen Wegen und in verschiedener Reihenfolge am Client ankommen. Die Größe eines *Datagram* ist auf 64 KB begrenzt.

In Java können UDP-Sockets mit der Klasse `DatagramSocket` erzeugt werden. Sowohl Client als auch Server verwenden diese Klasse (anders als bei TCP).

- `public DatagramSocket() throws SocketException:` Ein UDP-Socket wird erzeugt. Diese Methode wird in der Regel von Clients verwendet.
- `public DatagramSocket(int port) throws SocketException:` Ein UDP-Socket wird erzeugt und an den spezifizierten Port des lokalen Computers gebunden. Diese Methode wird in der Regel vom Server verwendet.
- `public void send(DatagramPacket p) throws IOException:` Sendet Methoden von dem Socket ein *datagramm packet*.
- `public void receive(DatagramPacket p) throws IOException:` Empfängt von dem Socket ein *datagramm packet*.

Die Klasse `DatagramPacket` repräsentiert ein *datagram packet*:

- `public DatagramPacket(byte[] buf, int length):` Erzeugt ein Konstruktoren `DatagramPacket`, um ein Paket der Länge `length` zu empfangen.

I 15 Arten der Netzkommunikation

- `public DatagramPacket(byte[] buf, int length, InetAddress address, int port):` Erzeugt ein `DatagramPacket`, um ein Paket der Länge `length` über die angegebene Portnummer zu dem spezifizierten Computer zu senden.
- Methoden
 - `public int getPort():` Gibt die Portnummer des entfernten Computers wieder, zu dem das `datagram` gesendet wurde oder von dem das `datagram` empfangen wurde.
 - `public InetAddress getAddress():` Liefert die IP-Adresse, zu dem das `datagram` gesendet wurde oder von dem das `datagram` empfangen wurde.
 - `public byte[] getData():` Gibt den Datenpuffer der gesendeten oder empfangenen Daten zurück.
 - `public int getLength():` Gibt die Länge der Daten, die gesendet oder die empfangen worden, zurück.

Beispiel:
`DemoSocketUDP` In diesem Beispiel wartet der Server auf ein Paket mit einer Zeichenkette und antwortet anschließend mit einer anderen Zeichenkette. Das Programm auf der Server-Seite sieht wie folgt aus:

```
package de.w3l.server;

import java.net.*;

public class DemoSocketServerUDP
{
    DemoSocketServerUDP(int portNr)
        throws java.io.IOException
    {
        //Speicherplatz für Pakete vorbereiten
        byte[] empfangenDaten = new byte[1024];
        byte[] sendenDaten = new byte[1024];
        String botschaft;
        //Socket binden
        DatagramSocket socket = new DatagramSocket(portNr);
        System.out.println("Start des Servers");
        while (true)
        {
            //Ein Paket empfangen
            DatagramPacket empfangen = new DatagramPacket(
                empfangenDaten,empfangenDaten.length);
            socket.receive(empfangen);
            //Infos ermitteln & ausgeben
            InetAddress senderIP = empfangen.getAddress();
            int senderPort = empfangen.getPort();
            botschaft = new String(
                empfangen.getData(),0,empfangen.getLength());
            System.out.println("Erhalten: " + botschaft + " von " +
                senderIP + " Port: " + senderPort);
            //Antwort erzeugen
            sendenDaten = "Hello Server World".getBytes();
            DatagramPacket senden = new DatagramPacket(
                sendenDaten,sendenDaten.length,
```

```
        senderIP, senderPort);
    socket.send(senden); //Antwort senden
}
}

public static void main (String args[])
    throws java.io.IOException
{
    new DemoSocketServerUDP(10015);
}
}
```

Ausgabe auf der Server-Konsole:

Start des Servers

Erhalten: Hello von /192.168.178.44 Port: 51684

Das Programm auf dem Client sieht wie folgt aus:

```
package de.w3l.client;
```

```
import java.net.*;

public class DemoSocketClientUDP
{
    DemoSocketClientUDP(int portNr)
        throws java.io.IOException
    {
        byte[] empfangenDaten = new byte[1024];
        byte[] sendenDaten = new byte[1024];
        String botschaft;
        //Socket erzeugen
        DatagramSocket socket = new DatagramSocket();
        //Paket erzeugen & adressieren
        InetAddress serverIP = InetAddress.getByName("hb-PC");
        sendenDaten = "Hello".getBytes();
        DatagramPacket senden = new DatagramPacket(
            sendenDaten,sendenDaten.length,serverIP,portNr);
        //Paket senden
        socket.send(senden);
        //Antwort empfangen & ausgeben.
        DatagramPacket empfangen = new DatagramPacket(
            empfangenDaten,empfangenDaten.length);
        socket.receive(empfangen);
        botschaft = new String(
            empfangen.getData(),0,empfangen.getLength());
        System.out.println("Erhalten: " + botschaft);
        //Socket schliessen
        socket.close();
    }

    public static void main (String args[])
        throws java.io.IOException
    {
        new DemoSocketClientUDP (10010);
    }
}
```

I 15 Arten der Netzkomunikation

Ausgabe auf der Client-Konsole:

Erhalten: Hello Server World

- Vorteile
- + Die Übertragung über UDP ist schnell.
 - + Gut geeignet für kleine, häufig übermittelte Daten, die *keine* Empfangsbestätigung benötigen, z.B. Wetter- und Börsendaten, *Voice over IP, Live Streams*.

- Nachteile
- Ungesicherte Übertragung.
 - Keine Empfangsbestätigung, d. h. keine Kontrolle darüber, ob ein Paket angekommen ist.
 - Spielt die Reihenfolge der eingegangenen Pakete eine Rolle, dann muss auf dem Client die richtige Reihenfolge wieder hergestellt werden.

 Lassen Sie das Programm auf Ihrem Computersystem – oder noch besser auf zwei Computersystemen – ablaufen.

15.1.3 Fallstudie: KV – Sockets

Soll eine Anwendung auf Clients und Server verteilt werden, dann kann die Netzkomunikation über Sockets erfolgen. Im Gegensatz zu den transparenten Kommunikationsverfahren Java-RMI (siehe »RMI«, S. 220) und CORBA (siehe »CORBA«, S. 241) muss der Softwarearchitekt bei der Verwendung von Sockets selbst überlegen, wie auf Klassen zugegriffen werden soll, die sich auf anderen Computersystemen befinden. Die Abb. 15.1-1 zeigt eine entsprechende Möglichkeit, dies zu realisieren. Die Klassen ClientStart, Kundenfenster, Kunde, KundenContainerEinfach und ObjektSpeicher sowie die Schnittstelle KundenContainerEinfachI bleiben unverändert.

ServerProxy Die Klasse ServerProxy ist aus der Sicht des GUI-Clients der Stellvertreter für die Klassen auf dem Server (sie wird auf dem Client installiert). Sie realisiert das Proxy-Entwurfsmuster (siehe »Das Proxy-Muster (*proxy pattern*)«, S. 83). Die Abb. 15.1-2 zeigt, dass die Klasse ServerProxy die Methoden zur Verfügung stellt, die die Klasse Kundenfenster benötigt. Die GUI kommuniziert also über den ServerProxy mit dem Server. Sie muss nichts über die Details der Socket-Kommunikation wissen. Das bedeutet, dass die GUI-Klassen nicht geändert werden müssen, wenn sich an der Kommunikation mit dem Server Änderungen ergeben.

ClientStart Die Klasse ClientStart baut die GUI auf. Die Verbindung zum Server und die weitere Kommunikation werden über die Klasse ServerProxy realisiert.

ServerStart Das Gegenstück zu ClientStart auf der Seite des Servers ist ServerStart. In dieser Klasse wird ein Socket geöffnet, und auf Anfragen gewartet. Wenn eine Anfrage ankommt, wird diese an die Klasse ServerProtokoll geleitet. Die Antwort, die die Klasse ServerProtokoll

15.1 Sockets I

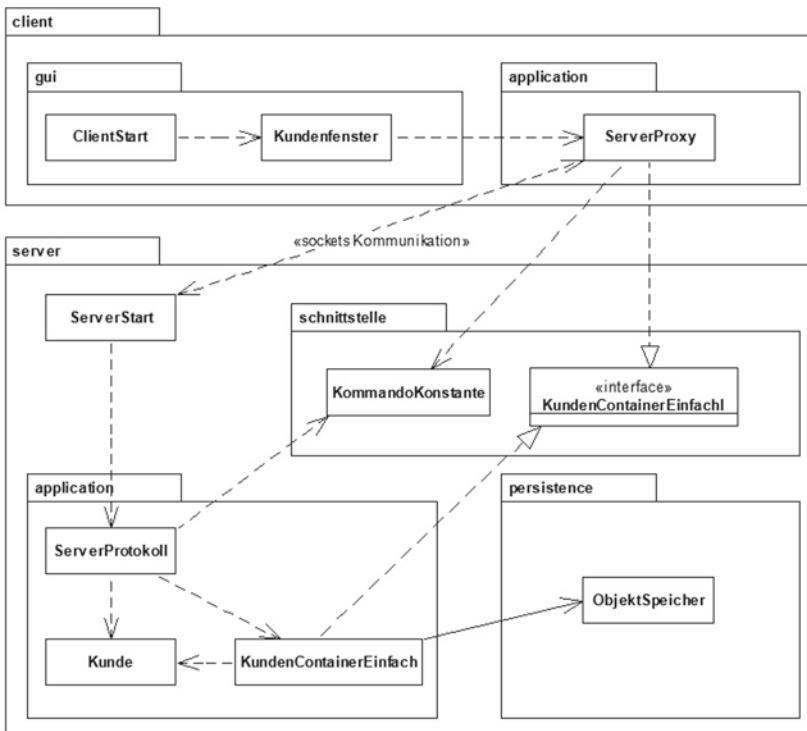


Abb. 15.1-1:
Paketdiagramm
der Fallstudie
»KV – Sockets«.

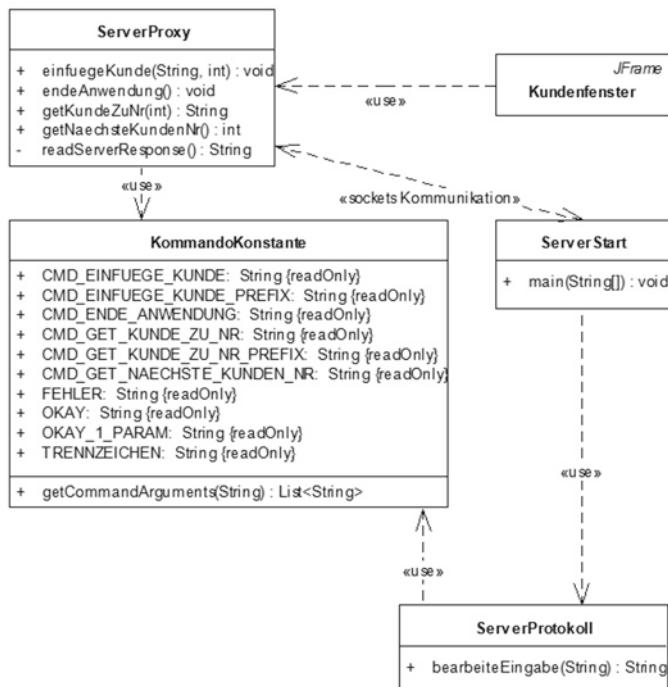
gibt, wird dann als Antwort zum Client gesendet. In der Klasse ServerProtokoll wird somit das Protokoll definiert, über das Client und Server kommunizieren. Die Kommunikation kann zustandsbehaftet oder zustandslos sein. Für die Fallstudie »Kundenverwaltung-Mini« ist sie zustandslos.

Die Kommunikation erfolgt auf Byte-Ebene, wobei die Bytes als Zeichenketten interpretiert werden. Der Client schickt ein Kommando (z.B. » gib den Namen des Kunden Nr. 2 «) und bekommt einen Rückgabewert (den Namen des Kunden als String). Damit Client und Server die gleiche Sprache »sprechen«, gibt es die Klasse KommandoKonstante. Diese Klasse ist sowohl den Clients als auch dem Server bekannt. Der Client benutzt die hier definierten Konstanten, wenn er ein Kommando übersendet. Somit wird sichergestellt, dass der Server dies versteht, und auch, dass der Client die Antwort des Servers interpretieren kann. Es ist auch möglich, Objekte über Sockets zu verschicken. Die Klassen ObjectOutputStream und ObjectInputStream können zu diesem Zweck eingesetzt werden. In der Kundenverwaltung könnte man bei der Anfrage nach einem Kunden das entsprechende Kunden-Objekt über die Socket-Verbindung senden.

Kommando
Konstante

I 15 Arten der Netzkommunikation

Abb. 15.1-2:
Klassendiagramm
für die
Kommunikation
mit dem Server
der Fallstudie
»KV – Sockets«.



Hinweis

Das setzt voraus, dass auch der Client die Definition der Klasse Kunde hat. Dies ist bei einer Architektur mit »sauber« voneinander getrennten Schichten unzulässig. Dagegen wird die Schichtentrennung *nicht* betroffen, wenn Klassen, die in der Java-API definiert sind, gesendet werden: Beispielsweise könnte der Client nach dem Geburtsdatum eines Kunden anfragen, und der Server könnte die Antwort (Objekt vom Typ Date) serialisieren und als Antwort zurücksenden.

OOP

Eine entsprechende Implementierung der »Kundenverwaltung-Mini« können Sie im E-Learning-Kurs herunterladen.

Ablauf

Das Sequenzdiagramm der Abb. 15.1-3 zeigt den Programmablauf beim Speichern der Daten eines Kunden.

Nach dem Drücken des Speichern-Druckknopfs wird die Methode speichernAktion() in der Klasse Kundenfenster aufgerufen. Die beiden Parameter nummer und name werden an die Methode einfuegeKunde() der Klasse ServerProxy weitergegeben. ServerProxy baut aus den Daten einen Kommandotext zusammen, z. B. einfuegeKunde#13#mueller. Dazu werden die Konstanten aus der Klasse KommandoKonstante benutzt. Die Zeichenkette wird dann über die Socket-Verbindung an den Server gesendet. Nach dem Senden wartet ServerProxy auf die Antwort des Servers.

15.1 Sockets I

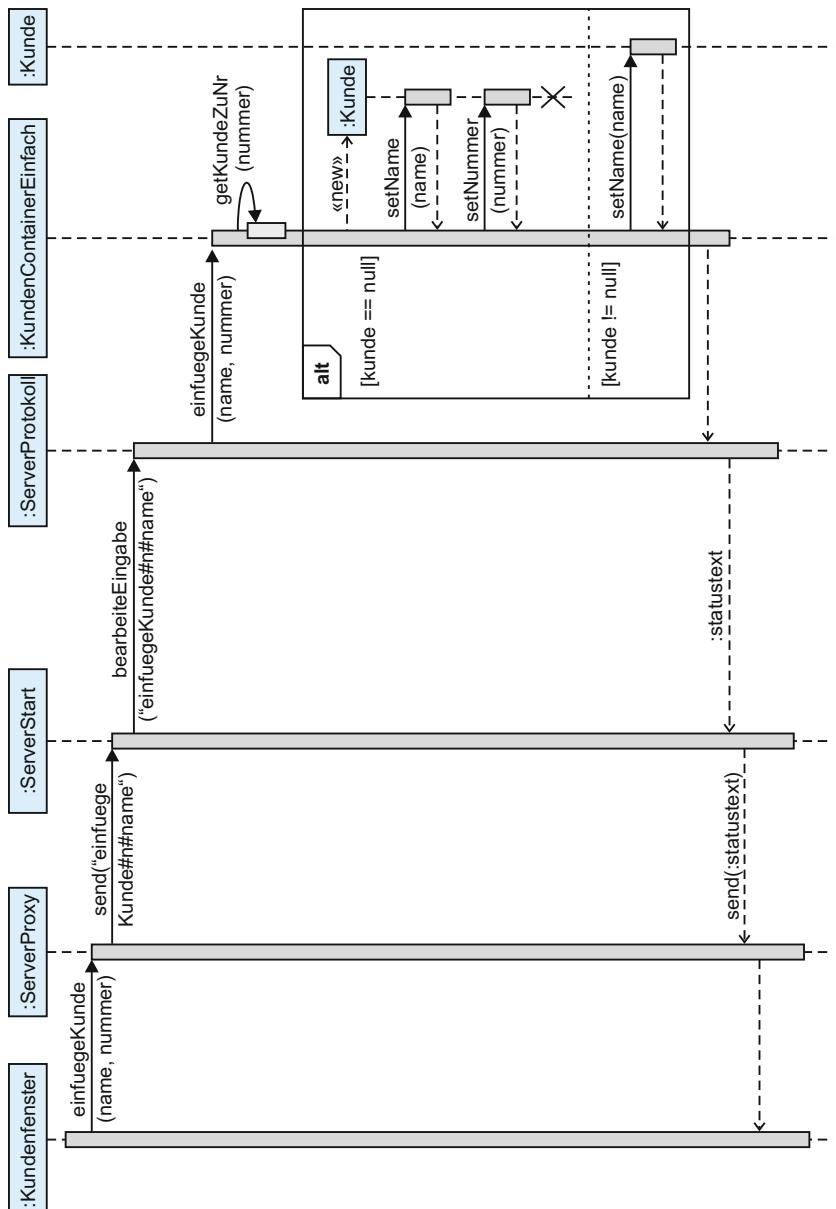


Abb. 15.1-3:
Ablauf beim Speichern der Daten eines Kunden in der Fallstudie »KV-Sockets«.

ServerStart empfängt die Zeichenkette auf der Serverseite und leitet sie unverändert an ServerProtokoll weiter. Diese Klasse wertet die Zeichenkette mithilfe der Klasse KommandoKonstante aus und entnimmt ihr den Befehl (einfügeKunde) und die Parameter dazu (nummer und name des Kunden). Es wird in der Klasse KundenContainerEinfach nachgesehen, ob es bereits einen Kunden mit dieser Nummer gibt. Falls es keinen gibt, wird ein neues Objekt Kunde erzeugt, anderen-

I 15 Arten der Netzkommunikation

falls wird ein bestehender Kunde bearbeitet. Die Parameter des Objekts Kunde werden gesetzt. Handelt es sich um einen neuen Kunden, dann wird das Objekt dem KundenContainerEinfach hinzugefügt.

Das Protokoll gibt einen Statustext (im Erfolgsfall `Okay`) an `ServerStart` zurück. Dieser sendet den Statustext über die Socket-Verbindung zurück an den Client.

`ServerProxy` empfängt diesen Text, und weiß somit, ob das Speichern erfolgreich war oder nicht. `ServerProxy` wirft im Fehlerfall eine `Exception`. Damit ist der Vorgang abgeschlossen.

Die Abb. 15.1-4 zeigt den Entwicklungsprozess beim Einsatz von Sockets.

Die Programme zu dieser Fallstudie können Sie im E-Learning-Kurs herunterladen.

15.2 RMI

Eine einfache Infrastruktur zur Realisierung verteilter Softwaresysteme stellt das RMI-Konzept (*remote method invocation*) von Java dar. Es handelt sich um mehrere Pakete des *Java Standard-APIs* und ist damit auf jeder Java-Plattform verfügbar. Java-RMI ermöglicht den Aufruf von Methoden für Objekte auf einem anderen Computersystem. Ereignisse, Transaktionen (siehe »Transaktionen«, S. 177) oder eine Benutzerverwaltung sind in RMI *nicht* vorgesehen. Abgesehen von einigen Besonderheiten können dank RMI entfernte Methoden genau so aufgerufen werden wie lokale. RMI ist implizit auch in der *Middleware-Plattform Java EE* enthalten (siehe »Die Java EE-Plattform«, S. 321).

 Eine kurze Einführung in RMI soll ein Gefühl für verteilte Anwendungen vermitteln. Begonnen wird mit einem »Hello World«-Beispiel, das auf zwei JVMs läuft:

- »Hello World mit Java-RMI«, S. 222

Als nächstes werden wichtige Begriffe erklärt:

- »RMI-Begriffe«, S. 229

RMI setzt für die Kommunikation mit entfernten Objekten lokale Stellvertreter-Objekte ein. Die Verteilung der Anwendung wird so vor dem Entwickler verborgen:

- »Stummel-Objekte (stubs)«, S. 231

Die Fallstudie »Kundenverwaltung-Mini« wird mit RMI implementiert:

- »Fallstudie: KV – RMI«, S. 236

Literatur Eine ausführliche Einführung in RMI enthält das Buch [ZiAr10], zu dem es auch einen E-Learning-Kurs gibt.

15.2 RMI I

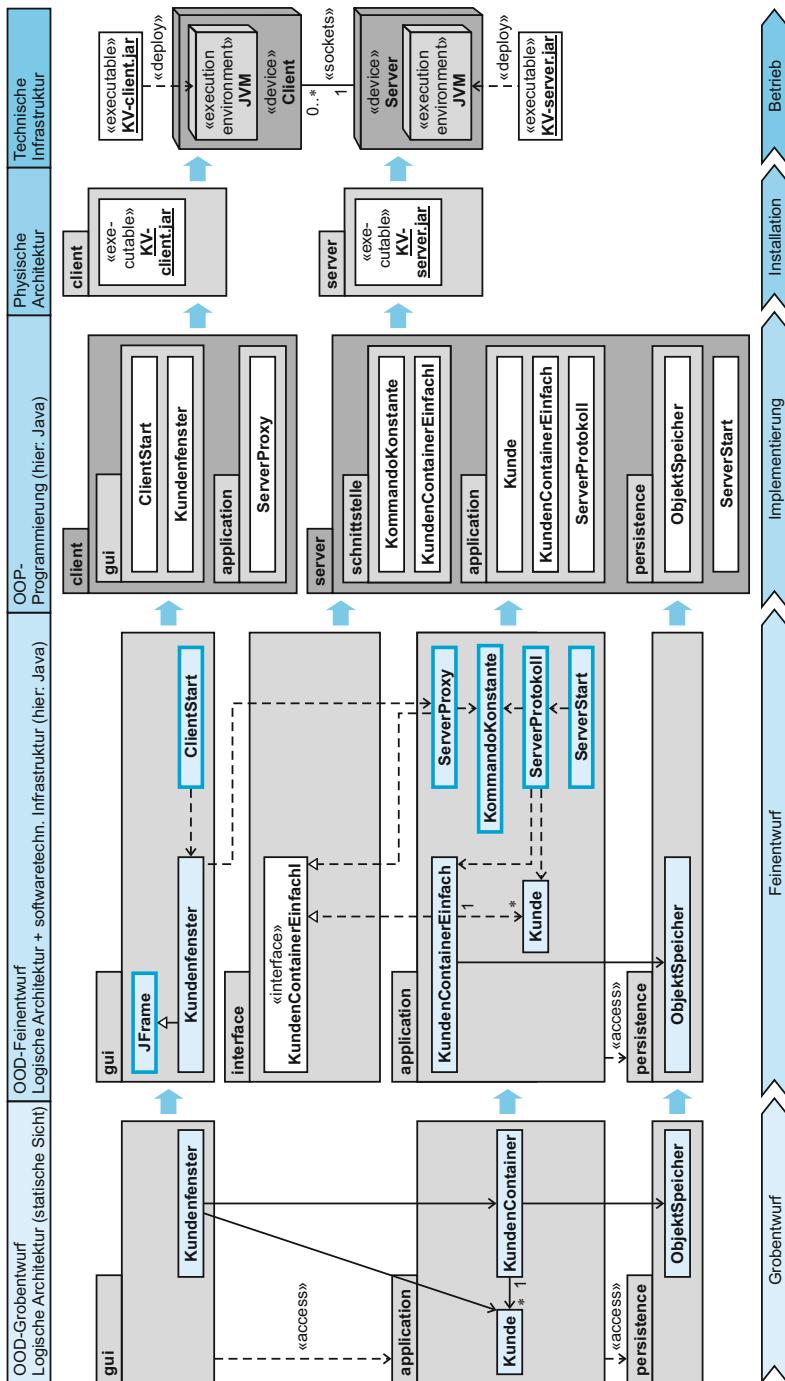


Abb. 15.1-4: Entwicklungsprozess der Fallstudie »KV – Sockets«.

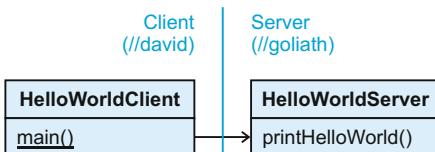
I 15 Arten der Netzkomunikation

15.2.1 »Hello World« mit Java-RMI

Beim Erlernen einer neuen Programmiersprache schreibt man traditionell als erstes ein »Hello World«.¹ RMI ist zwar keine neue Programmiersprache, besitzt aber einige zusätzliche Konzepte, mit denen sich ein Java-Programmierer zunächst vertraut machen muss. Diese Konzepte werden mit einem »Hello World«-Programm vorgestellt.

Zielsetzung Die `main()`-Methode einer Java-Klasse soll die Methode `printHelloWorld()` eines Objekts aufrufen, das sich auf einem anderen Computersystem befindet, und `Hello World` auf dem Bildschirm dieses anderen Computersystems ausgibt (Abb. 15.2-1).

Abb. 15.2-1: Die einfachste Client-Server-Anwendung.



David & Goliath

Für das verteilte »Hello World« soll der Server-Computer goliath heißen, der Client david.

5 Schritte

Das »Hello World«-Programm mit RMI wird in folgenden Schritten entwickelt:

- 1 Entwurf der Schnittstelle für das Server-Objekt
- 2 Implementierung der Server-Klasse
- 3 Entwicklung einer Server-Anwendung zur Aufnahme des Server-Objekts
- 4 Entwicklung einer Client-Anwendung zum Aufruf von Methoden des Server-Objekts
- 5 Anwendung übersetzen und starten



Programmieren Sie die folgenden Klassen und Schnittstellen auf Ihrem Computer nach und probieren Sie das Programm aus.

1 Entwurf der Schnittstelle für das Server-Objekt

In einer Schnittstelle müssen die Methoden, die von anderen JVMs aus nutzbar sein sollen, zusammengefasst werden. Diese Schnittstelle muss `java.rmi.Remote` erweitern. Alle Methoden (für »Hello World« nur eine) müssen die Ausnahme `java.rmi.RemoteException` auslösen können.

Beispiel 1a:
HelloWorld.java

```
import java.rmi.*;  
public interface HelloWorld extends Remote  
{
```

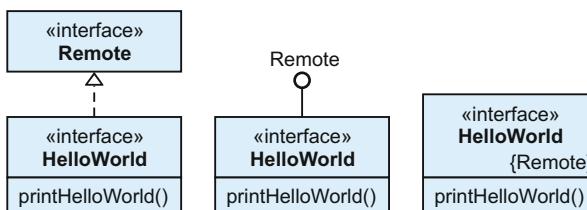
¹Teile dieses Textes und der Grafiken wurden mit freundlicher Genehmigung des W3L-Verlags aus [ZiAr10, S. 265 ff.] übernommen.

```
void printHelloWorld() throws RemoteException;
}
```

HelloWorld wird als *remote-Schnittstelle* bezeichnet, da sie die Schnittstelle Remote erweitert.

Bei der Schnittstelle Remote handelt es sich um eine Markierungs-Schnittstelle (*marker interface*). Sie enthält keine Methoden, die später in einer Klasse implementiert werden müssten. Sie zeigt lediglich an, dass Objekte mit dieser Schnittstelle von anderen JVMs aus nutzbar sein sollen.

In UML-Klassendiagrammen ist die explizite Darstellung der Schnittstelle Remote häufig überflüssig. Daher wird die Remote-Eigenschaft einer Schnittstelle besser durch einen *tagged value* ausgedrückt (Abb. 15.2-2). Die Schnittstelle Remote wird nur angegeben, wenn sie von besonderer Bedeutung für das Diagramm ist. In allen anderen Fällen gilt für eine mit {remote} markierte Schnittstelle implizit, dass sie von Remote erbt.



Markierungs-Schnittstelle

Abb. 15.2-2:
Darstellung von
Remote-
Schnittstellen in
der UML (Rechts
als tagged value).

2 Implementierung der Server-Klasse

Eine Klasse, deren Objekte von anderen JVMs aus nutzbar sein sollen, muss

- von `java.rmi.server.UnicastRemoteObject` erben,
- eine Remote-Schnittstelle implementieren und
- einen Konstruktor haben, der eine `RemoteException` auslösen kann.

```

import java.rmi.*;
import java.rmi.server.*;

public class HelloWorldImpl extends UnicastRemoteObject
    implements HelloWord
{
    public HelloWorldImpl() throws RemoteException
    {
        //nichts zu tun
        //nur für RemoteException
    }

    public void printHelloWorld()
    {
        System.out.println( "Hello World" );
    }
}
  
```

Beispiel 1b:
HelloWorldImpl.
.java
Java

I 15 Arten der Netzkommunikation

```
    }
```

Namenskonvention:
Postfix für Klassen

Konstruktor

Im Beispiel 1a wird die Remote-Schnittstelle wie eine gewöhnliche Klasse benannt. Da Clients vorwiegend mit dieser Schnittstelle arbeiten, ergibt sich für sie kein Unterschied gegenüber »gewöhnlichen« Klassen. Die Implementierungs-Klasse heißt wie die Schnittstelle, zusätzlich erhält sie das Postfix `Impl`. Diese Namenskonvention lehnt sich an die Regeln bei *Enterprise JavaBeans* an. In anderen Büchern werden andere oder gar keine Konventionen verwendet.

Der Konstruktor von `HelloWorldImpl` enthält keinen Code. Er ist dennoch erforderlich, denn der (parameterlose) Standard-Konstruktor der Oberklasse `UnicastRemoteObject` deklariert die Ausnahme `RemoteException`. Da immer ein Konstruktor der Oberklasse aufgerufen wird, sei es explizit oder automatisch vom Compiler eingefügt, muss auch der Konstruktor von `HelloWorldImpl` die `RemoteException` deklarieren. Würde kein solcher Konstruktor angegeben, d. h. würde der Java-Compiler automatisch einen leeren Standard-Konstruktor generieren, käme es zu Fehlermeldungen. Durch die Deklaration der Ausnahme muss jede Erzeugung von `HelloWorldImpl`-Objekten mit `new` in einem try-catch-Block stehen.

Für die Methode `printHelloWorld()` braucht nicht explizit angegeben zu werden, dass sie die Ausnahme `RemoteException` auslösen kann. Das ergibt sich automatisch aus der Deklaration in der Schnittstelle, die `HelloWorldImpl` implementiert.

3 Entwicklung einer Server-Anwendung zur Aufnahme des Server-Objekts

Die Server-Anwendung hat die Aufgabe, ein Objekt von `HelloWorldImpl` zu erzeugen und es unter einem eindeutigen Namen beim RMI-**Namensdienst** zu registrieren. Die Server-Anwendung muss *nicht* in einer neuen Klasse implementiert werden.

Beispiel 1c:
`HelloWorldImpl.java` kann um die folgende `main()`-Methode ergänzt werden:

```
public class HelloWorldImpl ...  
{  
    ...  
    public static void main( String[] args )  
    {  
        try  
        {  
            HelloWorldImpl serverObjekt = new HelloWorldImpl();  
            java.rmi.Naming.rebind(  
                "//localhost/HelloWorld", serverObjekt );  
            System.out.println(  
                "Objekt erzeugt, erwartet Aufrufe" );  
        }  
    }  
}
```

```
        }  
    catch( Exception e )  
    {  
        System.out.println( e );  
    }  
}
```

In der `main()`-Methode wird zunächst ein neues Objekt vom Typ `HelloWorldImpl` erzeugt. Anschließend wird es beim RMI-Namensdienst registriert.

Ein Namensdienst ist im Prinzip ein Verzeichnis (*Dictionary*, im **Java Collection Framework** auch als *Map* bezeichnet), das eine Referenz (serverObjekt) an einen Namen (»HelloWorld«) bindet. Über diesen Namen erhält man durch Aufruf der `lookup()`-Methode eine Referenz auf das Server-Objekt, unabhängig davon, in welcher JVM es liegt.

Normalerweise erfolgt die Registrierung eines Objekts beim Namensdienst mit der `bind()`-Methode. In diesem Beispiel wird `rebind()` verwendet. Im Unterschied zu `bind` löst `rebind()` keine Ausnahme aus, wenn unter dem angegebenen Namen schon ein Objekt registriert ist. Das bisher registrierte Objekt wird durch das neue Objekt ersetzt. Beim Testen einer Anwendung hat das den Vorteil, dass der Namensdienst nicht immer wieder neu gestartet werden muss. Später sollte aber immer `bind()` verwendet werden, damit eine Doppel-Registrierung auffällt, denn sie stellt meist einen Fehler dar.

Bei der Erzeugung und Registrierung können Fehler auftreten. In Ausnahmen dieses einfachen Beispiel ist das aber eher unwahrscheinlich. Statt einer detaillierten Ausnahmebehandlung wird daher nur ein catch-Block für alle Ausnahmen definiert. Die abgefangene Ausnahme wird einfach ausgegeben.

4 Entwicklung einer Client-Anwendung zum Aufruf von Methoden des Server-Objekts

Der letzte noch zu entwickelnde Teil der verteilten Java-Anwendung ist ein Client. Er ruft von einer anderen JVM aus die `printHelloWorld()`-Methode des Servers auf.

```
1 import java.rmi.*;
2
3 public class HelloWorldClient
4 {
5     public static void main( String[] args )
6     {
7         try
8         {
9             //Referenz auf Server-Objekt vom Namensdienst
10            //holen, downcast erforderlich
```

Beispiel 1d:
HelloWorld
Client.java
Java

I 15 Arten der Netzkommunikation

```
11     HelloWorld remoteObjekt = (HelloWorld)
12     Naming.lookup( "//goliath/HelloWorld" );
13
14     //der Aufruf der Methode des entfernten
15     //Objekts unterscheidet sich nicht von
16     //einem lokalen Aufruf
17     remoteObjekt.printHelloWorld();
18 }
19 catch( Exception e )
20 {
21     System.out.println( e );
22 }
23 }
24 }
```

Die Client-Anwendung besteht nur aus der `main()`-Methode. Sie holt sich über den Namensdienst (`java.rmi.Naming`) eine Referenz auf das Server-Objekt und ruft dann `printHelloWorld()` auf.

Hinweis

Wenn Ihr Computer z.B. den Namen `MeinComputer` hat, dann müssen Sie in der Zeile 12 von Beispiel 1d statt `//goliath/HelloWorld` schreiben: `//MeinComputer/HelloWorld`. Wenn Sie Ihren Computernamen nicht mehr wissen, dann finden Sie ihn im Windows-Betriebssystem unter Systemsteuerung -> System -> Computername.

Wenn das Programm `HelloWorldClient` nicht auf demselben Computersystem wie die Server-Klassen liegt, dann muss in der Zeile 12 von Beispiel 1d anstelle von `goliath` die IP-Adresse des Server-Computers stehen, z.B. `//192.168.178.44/HelloWorld`. Wenn Sie die Adresse Ihres Server-Computers nicht kennen, dann geben Sie im Konsolenfenster des Servers folgenden Befehl ein: `ipconfig`. Die IP-Adresse finden Sie unter IPv4-Adresse.

RMI kapselt Remote-Aufruf

An diesem einfachen Beispiel kann man deutlich sehen, dass sich der eigentliche Aufruf in keiner Weise von einem gewöhnlichen, lokalen Aufruf unterscheidet. Man sieht `remoteObjekt.printHelloWorld()` nicht an, dass das referenzierte Objekt in einer anderen JVM liegt. RMI verbirgt sehr gut, dass es sich um ein entferntes Objekt handelt.

Test auf einem Computer- system

Die `lookup()`-Methode des Namensdienstes erhält als Parameter den Namen des Server-*Hosts* und den Namen des Objekts in Form einer URL. Wenn Client- und Server-Anwendung zu Testzwecken auf einem Computersystem laufen sollen, muss der Parameter `//localhost/HelloWorld` lauten.

Expliziter down cast auf `HelloWorld`

Zunächst gibt `lookup()` eine Referenz vom Typ `java.rmi.Remote` zurück. Daher muss ein expliziter *downcast* ausgeführt werden. Beim Auslesen von Objekten aus einer Sammlung (*Collection*) wird genauso verfahren. Der RMI-Namensdienst arbeitet wie eine Map im *Java Collection Framework*.

Die bisher entwickelte Anwendung umfasst eine Schnittstelle und zwei Klassen (Abb. 15.2-3). Zwischenbilanz

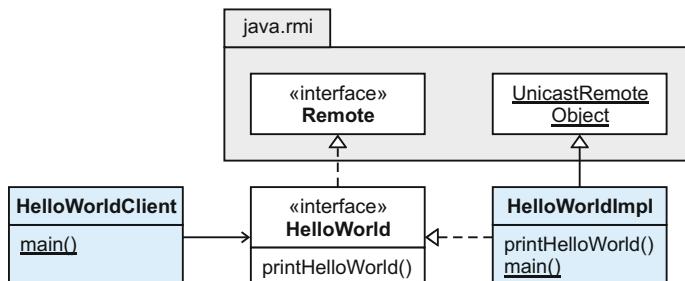


Abb. 15.2-3: Die Bestandteile einer Hello-World-Anwendung mit RMI.

5 Anwendung übersetzen und starten

Zum Übersetzen und Starten der Anwendung müssen eine Reihe von Aktionen nacheinander ausgeführt werden:

■ Übersetzen der Java-Quellen

Die Anwendung besteht aus folgenden Quellcode-Dateien:

- `HelloWorld.java`
- `HelloWorldImpl.java`
- `HelloWorldClient.java`

Liegen alle Quellcode-Dateien auf Ihrem Entwicklungscomputer in einem Verzeichnis, dann reicht zum Übersetzen der Aufruf

`javac *.java.`

Der Java-Compiler erzeugt die zugehörigen .class-Dateien.

■ Erzeugen von Stummel-Objekten

RMI benötigt für den Aufruf von Methoden über die Grenzen einer JVM hinweg sogenannte Stummel-Objekte (*stubs*, siehe »Stummel-Objekte (stubs)«, S. 231). Zunächst reicht es zu wissen, dass sie mit folgendem Aufruf erzeugt werden können:

`rmic -v1.2 HelloWorldImpl.`

Bei `rmic` handelt es sich um den **RMI-Compiler**, der die Datei `HelloWorldImpl_Stub.class` erzeugt.

Die Anwendung besteht aus den Dateien in Abb. 15.2-4.

Auf dem Client-Computer müssen folgende Programme installiert werden:

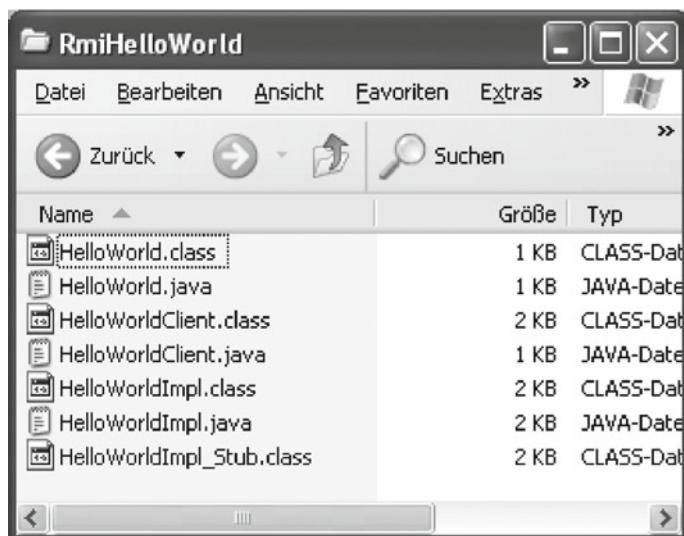
- `HelloWorldClient.class`
- `HelloWorld.class` (Schnittstelle)
- `HelloWorldImpl_Stub.class` (Stummel)

Auf dem Server-Computer müssen folgende Programme installiert werden:

- `HelloWorldImpl.class`
- `HelloWorld.class` (Schnittstelle)
- `HelloWorldImpl_Stub.class` (Stummel)

I 15 Arten der Netzkommunikation

Abb. 15.2-4: Die Dateien von Hello World.



Die Schnittstelle `HelloWorld.class` und die Stummel-Klasse `HelloWorldImpl_Stub.class` müssen also sowohl auf dem Client als auch auf dem Server vorhanden sein.

Test auf 2 Computer-systemen
Um die Anwendung zu testen, ist auf dem **Server** Folgendes zu tun:

■ Starten des Namensdienstes

Der Namensdienst wird von dem Programm `rmiregistry` zur Verfügung gestellt. Durch die Eingabe von `rmiregistry` in einem Konsolenfenster wird der Namensdienst gestartet.

■ Starten des Servers

In einem zweiten Konsolenfenster wird die Server-Anwendung mit `java HelloWorldImpl` gestartet. Sie wartet darauf, aufgerufen zu werden. Außer der Bereit-Meldung »Objekt erzeugt, erwarte Aufrufe« tut sich daher zunächst nichts.

Auf dem **Client** ist Folgendes zu tun:

■ Starten des Clients

Zum Schluss wird die Client-Anwendung in einem Konsolenfenster mit `java HelloWorldClient` gestartet. Der Client setzt einen Aufruf an den Server ab und wird danach wieder beendet.

Im Konsolenfenster des Servers wird »Hello World« ausgegeben. Die Client-Anwendung hat via RMI einen Methodenaufruf in eine andere JVM vorgenommen. Die Server-Anwendung hat auf den Aufruf reagiert, indem sie die entsprechende Ausgabe gemacht hat. Wiederholte Aufrufe des Client erzeugen jeweils eine weitere Zeile auf der Konsole des Servers. Server und Namensdienst laufen weiter. Diese einfache Beispiel-Anwendung sieht keinen Mechanismus zum Beenden vor. Dies kann aber leicht durch Drücken von Strg-C (unter Windows) im jeweiligen Konsolenfenster erfolgen.

Um die Anwendung auf einem Computer (auf goliath) zu testen, werden drei Konsolenfenster Eingabeaufforderung, *Shell* benötigt.

Test auf einem Computer-
system

■ Starten des Namensdienstes

Der Namensdienst wird von dem Programm `rmiregistry` zur Verfügung gestellt. Durch die Eingabe von `rmiregistry` in der ersten Konsole wird der Namensdienst gestartet.

■ Starten des Servers

In der zweiten Konsole wird die Server-Anwendung mit `java HelloWorldImpl` gestartet.

■ Starten des Clients

Zum Schluss wird die Client-Anwendung in der dritten Konsole mit `java HelloWorldClient` gestartet. Der Client setzt einen Aufruf an den Server ab und wird danach wieder beendet.

Im Konsolen-Fenster des Servers wird »Hello World« ausgegeben (Abb. 15.2-5). Im linken Fenster läuft der Client. Rechts oben wurde der Namensdienst (`rmiregistry.exe`) gestartet. Der Server läuft im rechten unteren Fenster.

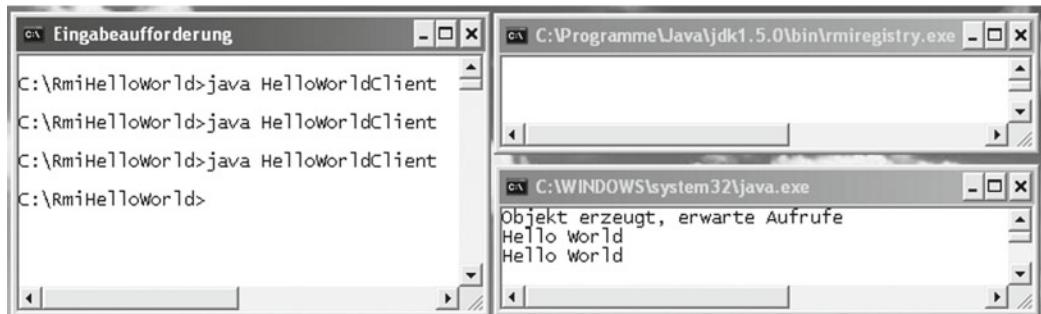


Abb. 15.2-5: Die »Hello World«-Anwendung kann in drei Fenstern auf demselben Computer laufen.

15.2.2 RMI-Begriffe

In der RMI-Welt tauchen immer wieder Begriffe auf, die zunächst definiert werden.¹

Client und Server

Ein Java-Programm besteht aus miteinander kommunizierenden Objekten. Ein Objekt ruft in seinen Methoden die Methoden anderer Objekte auf.

Kommuni-
zierende Objekte

Neu ist die Sichtweise, das aufgerufene Objekt als Server (Dienstanbieter) zu sehen und das aufrufende Objekt als Client (Kunde, Dienstnehmer). Ein Client-Objekt bedient sich der Dienste eines anderen Objekts, um seine eigene Aufgabe erfüllen zu können. Das

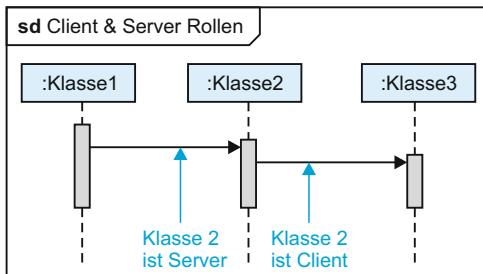
Client =
Dienstnehmer,
Server =
Dienstanbieter

¹ Teile dieses Textes und der Grafiken wurden mit freundlicher Genehmigung des W3L-Verlags aus [ZiAr10, S. 273 f.] übernommen.

I 15 Arten der Netzkommunikation

aufgerufene Objekt, der Server, kann selbst wieder Methoden anderer Objekte aufrufen und agiert diesen gegenüber als Client. Die Begriffe Client und Server sind also Rollen, die immer im Kontext eines Aufrufs vergeben werden müssen (Abb. 15.2-6).

Abb. 15.2-6: Ob eine Klasse oder ein Computer-System Client oder Server ist, kann nicht generell entschieden werden.



Client- & Server-Computer

Nicht nur Objekte können in einer Client-Server-Beziehung zueinander stehen. Auch ganze Computersysteme (*hosts*) können als Client und Server zusammenarbeiten. Für RMI bedeutet dies, dass ein Objekt auf einem Client-Computer eine Methode eines Objekts auf einem Server-Computer aufruft. Client und Server sind wieder Rollen und nur für einen konkreten Aufruf kann entschieden werden, welches Computersystem der Client und welches der Server ist. In der Praxis ist oft von Servern die Rede, ohne dass ein konkreter Methodenaufruf vorliegt. Gemeint sind damit Computersysteme, deren vorwiegende Aufgabe die Bereitstellung von Diensten für andere Computersysteme (Clients) ist. Statt immer explizit von Client-Objekt oder Server-Computer zu sprechen, ist meist nur von Client und Server die Rede. Aus dem Kontext wird immer deutlich, ob ein Objekt oder ein Computersystem gemeint ist.

local und remote

Ein Java-Programm wird von einer virtuellen Maschine ausgeführt. Die JVM ist praktisch ein Computer in Software. Sie stellt Java-Programmen eine Ausführungsumgebung zur Verfügung.

lokal = dieselbe
JVM

Geht man von einem konkreten Objekt aus, sind alle anderen Objekte, die in derselben JVM liegen, lokale Objekte (*local*), relativ zum betrachteten Objekt. Alle Objekte, die in anderen JVMs liegen, sind entfernte Objekte (*remote*) (Abb. 15.2-7). Dabei ist es unerheblich, ob es JVMs auf demselben physischen Computer oder JVMs auf Computersystemen am anderen Ende der Welt sind.

Aufrufe in andere
JVMs

RMI dient als verteilte Objektarchitektur dazu, Methoden entfernter Objekte aufzurufen, wobei es das Ziel ist, entfernte Aufrufe möglichst wie lokale Aufrufe behandeln zu können.

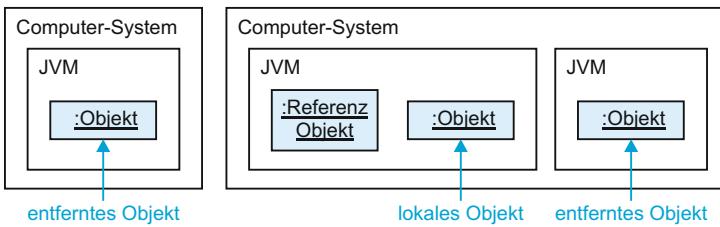


Abb. 15.2-7:
Relativ zu einem Referenz-Objekt sind alle Objekte in derselben JVM lokale Objekte. Alle Objekte in anderen JVMs sind entfernte Objekte.

15.2.3 Stummel-Objekte (stubs)

Wenn eine Java-Anwendung lokal in einer JVM läuft, gelten folgende Aussagen:¹

- Ein Objekt steht an einer bestimmten Stelle im Arbeitsspeicher. Eine Referenz auf ein Objekt kann mehr oder weniger direkt auf eine Speicheradresse abgebildet werden.
 - Bei einem Methodenaufruf legt die aufrufende Methode die Parameter in einem speziellen Speicherbereich (Stapel, *stack*) ab, wo die aufgerufene Methode sie finden und auswerten kann. Mit dem Rückgabewert wird analog verfahren. Das ganze geht sehr schnell vonstatten.
- Wenn eine Anwendung verteilt laufen soll, d. h. Objekte auf verschiedenen JVMs liegen können, sind diese Aussagen *nicht* mehr gültig.
- Eine Referenz auf ein Objekt kann nicht mehr einfach eine Speicheradresse sein. Die JVM mit der Referenz auf ein Objekt und die JVM mit dem referenzierten Objekt haben keinen gemeinsamen Speicher. Eine Adresse in der einen JVM bedeutet in der anderen nichts.
 - Ohne den gemeinsamen Speicher ist die Übergabe von Parametern und Rückgabewerten bei Methodenaufrufen auch nicht mehr über den Stapel möglich, da beide JVMs einen eigenen Stapel haben.

Zur Realisierung entfernter Referenzen und Methodenaufrufe setzt RMI Stummel-Objekte ein.

Lokale Anwendung

Lokale Referenz

Lokaler Aufruf

Verteilte Anwendung

Entfernte Referenz

Entfernter Aufruf

Stummel-Objekte allgemein

Ein **Stummel-Objekt** (*stub*), auch *proxy* genannt, ist ein lokaler Stellvertreter für ein entferntes Objekt (siehe »Das Proxy-Muster (*proxy pattern*)«, S. 83).

In einem lokalen System hat ein Client eine einfache Referenz auf einen Server. In einem verteilten System, wo dies aus den oben genannten Gründen nicht möglich ist, hat er stattdessen eine Referenz auf ein Stummel-Objekt (Abb. 15.2-8).

Client referenziert Stummel-Objekt

¹ Teile dieses Textes und der Grafiken wurden mit freundlicher Genehmigung des W3L-Verlags aus [ZiAr10, S. 274 ff.] übernommen.

I 15 Arten der Netzkommunikation

In Java spielt es dabei keine Rolle, ob ein Server auf einem anderen *host* oder in anderen JVMs auf demselben *host* liegt. Statt des eigentlichen Servers referenziert er ein Stummel-Objekt.

Für den Client sieht das Stummel-Objekt genauso aus, wie das Server-Objekt, d. h. es hat die gleiche Schnittstelle (Abb. 15.2-9).

Abb. 15.2-8:
Stummel-Objekte
sind lokale
Stellvertreter für
entfernte Server-
Objekte.

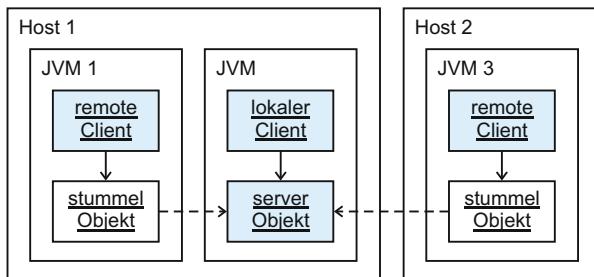
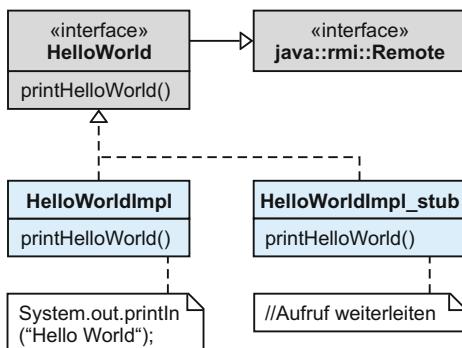


Abb. 15.2-9: Eine
Stummel-Klasse
implementiert
dieselben
Schnittstellen wie
die Server-Klasse.



Methoden-
aufruf

Wenn der Client eine Methode aufruft, so tut er dies auf dem Stummel-Objekt. Das Stummel-Objekt, das ja nur ein Stellvertreter ist, führt die Methode nicht wirklich aus. Es hat aber genug Informationen über den Server, um den Aufruf an diesen weiterleiten zu können. Diese Informationen umfassen

- den *host* (URL oder IP-Adresse)
- die JVM (es können auf einem *host* mehrere JVMs laufen) und
- Informationen zum Auffinden des Server-Objekts innerhalb der JVM.

Das Stummel-Objekt baut eine Netzwerkverbindung zur richtigen JVM auf und teilt ihr mit, auf welchem Objekt welche Methode mit welchen Parametern aufgerufen werden soll. Diese Informationen werden als Datenpakete über das Netz geschickt. Die entfernte JVM nimmt diesen Aufruf entgegen und setzt ihn in einen lokalen Aufruf des Server-Objekts um. Der Rückgabewert wandert auf dem gleichen Weg, nur in entgegengesetzter Richtung, wieder an das Stummel-Objekt zurück, das ihn beim Client ab liefert.

Die Abb. 15.2-10 zeigt den Ablauf als UML-Sequenzdiagramm. Die Darstellung der Netzverbindung ist nicht UML-konform, verdeutlicht aber gut die Serialisierung des Aufrufs.

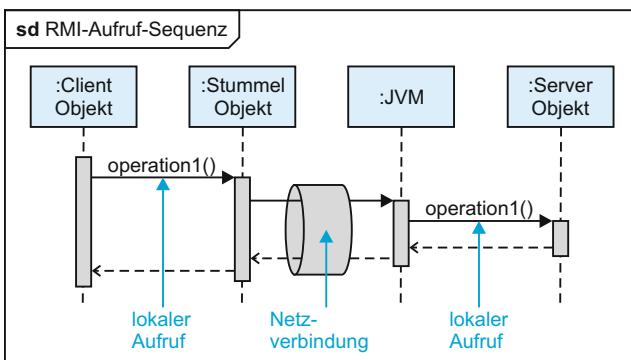


Abb. 15.2-10:
Operationsaufrufe
mit RMI.

All das läuft hinter den Kulissen ab. Client- und Server-Objekte bemerken davon nichts:

- Der Client weiß nicht, dass er statt mit dem Server nur mit seinem Stellvertreter arbeitet. Er führt den Aufruf genauso durch, wie er es in einem lokalen System tun würde.
- Der Server weiß nicht, dass ein Aufruf nicht lokal ist, sondern u. U. vom anderen Ende der Welt gekommen ist.
- Client und Server brauchen *nicht* speziell für verteilte Systeme programmiert zu werden, die Infrastruktur verbirgt die Verteilung.

Der letzte Punkt gilt nur begrenzt. An einigen Stellen einer Anwendung kommt die Verteilung zum Vorschein, z.B. beim Aufruf des Namensdienstes. Auch muss beim Entwurf der Anwendung berücksichtigt werden, dass ein langsames Netzwerk zur Kommunikation verwendet werden muss. Trotzdem machen Stummel-Objekte das Leben leichter. Große Teile der Anwendung können unabhängig von den Besonderheiten einer verteilten Anwendung entwickelt werden.

Stummel-Objekte in RMI

Die Klassen der Stummel-Objekte werden mit dem RMI-Compiler generiert.

Angenommen, eine Anwendung besteht aus folgenden Dateien:

- `HelloWorld.java` (eine Remote-Schnittstelle)
- `HelloWorldImpl.java` implementiert die Remote-Schnittstelle.

Zunächst müssen die Java-Klassen mit dem Java-Compiler übersetzt werden. Anschließend kann der RMI-Compiler gestartet werden:

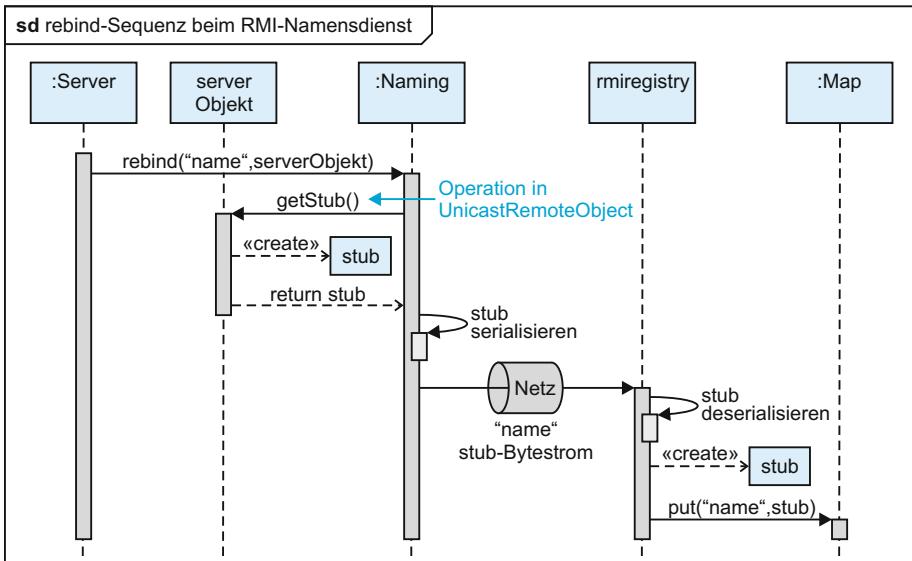
```
rmic -v1.2 HelloWorldImpl.
```

Der RMI-Compiler erzeugt die Datei `HelloWorldImpl_Stub.class`, die Klasse der Stummel-Objekte für `HelloWorld`.

I 15 Arten der Netzkommunikation

| | |
|---|---|
| Ab Java 2 Version 1.2 | Die Option <code>-v1.2</code> weist den RMI-Compiler an, eine Stummel-Klasse für Java ab der Version 1.2 zu erzeugen. Wird diese Option weg gelassen, erzeugt der RMI-Compiler Code, der auch mit früheren Java-Versionen funktioniert. |
| Ab Java 5 | Stummel-Objekte können ab der Java-Version 5 dynamisch generiert werden. Der Aufruf des RMI-Compilers entfällt. Dies funktioniert allerdings nur dann, wenn alle JVMs, auf denen die verteilte Anwendung läuft, in der Version 5 vorliegen. |
| Tipp | <p>Wenn Sie genau wissen möchten, wie eine Stummel-Klasse aussieht, rufen Sie den RMI-Compiler mit <code>rmic -v1.2 -keep HelloWorldImpl</code> auf. Es wird dann auch die Datei <code>HelloWorldImpl_Stub.java</code> erzeugt, die den Quellcode der Stub-Klasse enthält. Es ist für das prinzipielle Verständnis aber nicht erforderlich.</p> |
| Serialisierung | Die Stummel-Klasse ist serialisierbar, d. h. Objekte dieser Klasse können in einen Bytestrom geschrieben und aus einem solchen wieder hergestellt werden. Damit lassen sich Stummel-Objekte z. B. in einer Datei speichern oder über eine Netz-Verbindung zwischen zwei JVMs übertragen. |
| Objekt registrieren | Die beiden folgenden Codezeilen erzeugen ein Objekt von <code>HelloWorldImpl</code> und machen es für entfernte Clients verfügbar: |
| | <pre>HelloWorld serverObjekt = new HelloWorldImpl(); Naming.bind("//localhost/HelloWorld", serverObjekt);</pre> |
| | <p>Wenn das Objekt beim Namensdienst registriert werden soll, holt die <code>bind()</code>-Methode sich ein Objekt der Stummel-Klasse zu <code>HelloWorldImpl</code>. Die Funktionalität zur Erzeugung eines Stummel-Objekts und zur Verknüpfung mit dem Server-Objekt, ist in <code>UnicastRemoteObject</code> realisiert. Die <code>bind()</code>-Methode serialisiert das Stummel-Objekt anschließend in einen Bytestrom. Dieser wird zusammen mit dem Namen des Objekts zum Namensdienst (<code>rmiregistry</code>) übertragen. Ein Eintrag im Namensdienst besteht aus</p> <ul style="list-style-type: none">■ einem String mit dem Namen des Objekts und■ dem serialisierten Stummel-Objekt. |
| | <p>Die Abb. 15.2-11 zeigt den Ablauf der Registrierung.</p> |
| | <p>Ein Client fordert mit der folgenden Zeile eine Referenz auf das Objekt an:</p> |
| | <pre>HelloWorld server = (HelloWorld) Naming.lookup("//goliath/HelloWorld");</pre> |
| Übertragung von serialisierten Stummel-Objekten | Die <code>lookup()</code> -Methode auf dem Client-Computer stellt eine Anfrage an den Namensdienst auf dem Server-Computer, im obigen Beispiel auf <code>goliath</code> . Der Name des gewünschten Objekts wird mit der Anfrage übergeben. Als Ergebnis sendet der Namensdienst des Server-Computers den Bytestrom mit dem serialisierten Stummel-Objekt zurück. |

jekt zurück. Die `lookup()`-Methode deserialisiert das Stummel-Objekt lokal und gibt als Rückgabewert eine lokale Referenz auf das Stummel-Objekt zurück (Abb. 15.2-12).



Immer, wenn jetzt eine Methode des Stummel-Objekts aufgerufen wird, leitet diese den Aufruf an das Server-Objekt weiter und gibt dem Client so die Illusion, lokal zu arbeiten.

Abb. 15.2-11:
Registrierung
beim
Namensdienst.

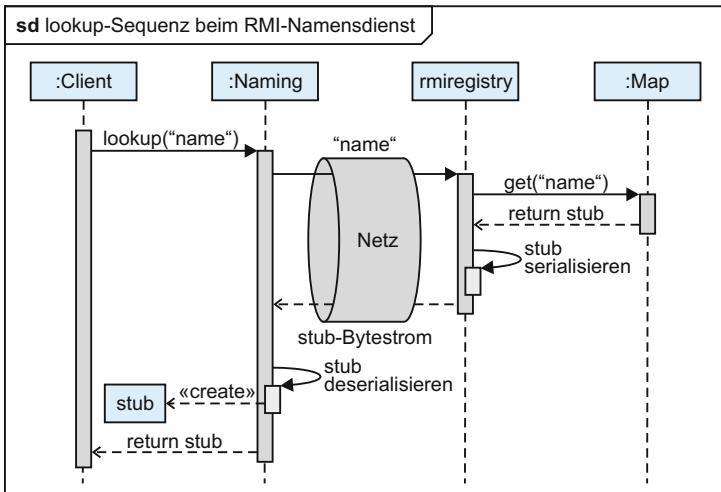


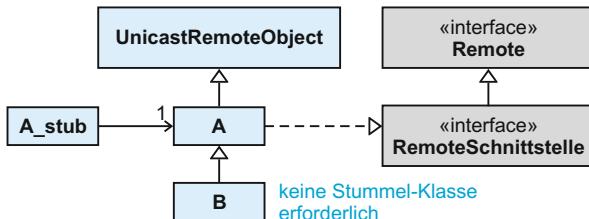
Abb. 15.2-12:
Nutzung des RMI-
Namensdienstes.

Stummel-Klassen müssen von den Klassen generiert werden, die direkt `UnicastRemoteObject` implementieren. Wenn diese Klassen Unterklassen haben, brauchen für diese *keine* Stummel-Klassen gene-

I 15 Arten der Netzkommunikation

riert zu werden (Abb. 15.2-13). Versucht man dies trotzdem, liefert rmid eine entsprechende Fehlermeldung. Wenn eine entfernte Referenz auf ein Objekt einer Unterklasse erzeugt werden muss, verwendet RMI automatisch die zur Oberklasse gehörende Stummel-Klasse.

Abb. 15.2-13: Nur für die Klasse, die direkt von Unicast RemoteObject erbt, müssen Stummel-Klassen generiert werden.



15.2.4 Fallstudie: KV – RMI

Erfolgt bei der Fallstudie »Kundenverwaltung« (KV-Mini) die Netzkommunikation über RMI, dann müssen von den Klassen auf dem Server die Methoden, auf die der Client zugreifen muss, in Form von Schnittstellen zur Verfügung gestellt werden. Diese Schnittstellen müssen von der Schnittstelle **Remote** erben. Wie die Abb. 15.2-14 zeigt (2. Spalte, OOD-Feinentwurf), sind dies die Schnittstellen **KundeI** und **KundenContainerObjektI**.

Das gesamte OOD-Modell zeigt die Abb. 15.2-15. Die Klassen **ClientStart**, **Kundenfenster** und **ObjektSpeicher** sowie die Schnittstelle **KundenContainerEinfachI** bleiben unverändert. Die Klasse **KundenContainerObjekt** wird als **UnicastRemoteObject** implementiert, d.h., dass das Objekt der Klasse **KundenContainerObjekt** direkt an das RMI-System anzubinden ist, sobald es erzeugt wird. Einem angebundenen Remote-Objekt – oder anders gesagt, einem an das RMI-System exportiertem Objekt – kann ein eindeutiger Name im URI-Format zugewiesen werden. Hierfür wird der Java-Namensdienst verwendet.

```
//Erstellen des Container-Objekts. Dabei wird das erstellte  
//Objekt automatisch an das RMI-System angebunden.  
KundenContainerObjektI service = new KundenContainerObjekt();  
  
//Festlegen des Namens für das Remote-Objekt.  
java.rmi.Naming.rebind  
("//localhost:1099/kundenverwaltung", service);
```

Auf dem Client kann man das Remote-Objekt ansprechen, indem zuvor eine Remote-Referenz des Objekts über den Java-Namensdienst abgefragt wird:

```
KundenContainerObjektI datenService = (KundenContainerObjektI)  
java.rmi.Naming.lookup("//localhost:1099/kundenverwaltung");
```

Somit bekommt man ein Stummel-Objekt **datenService**, dessen referenziertes Objekt sich auf dem Server befindet.

15.2 RMI I

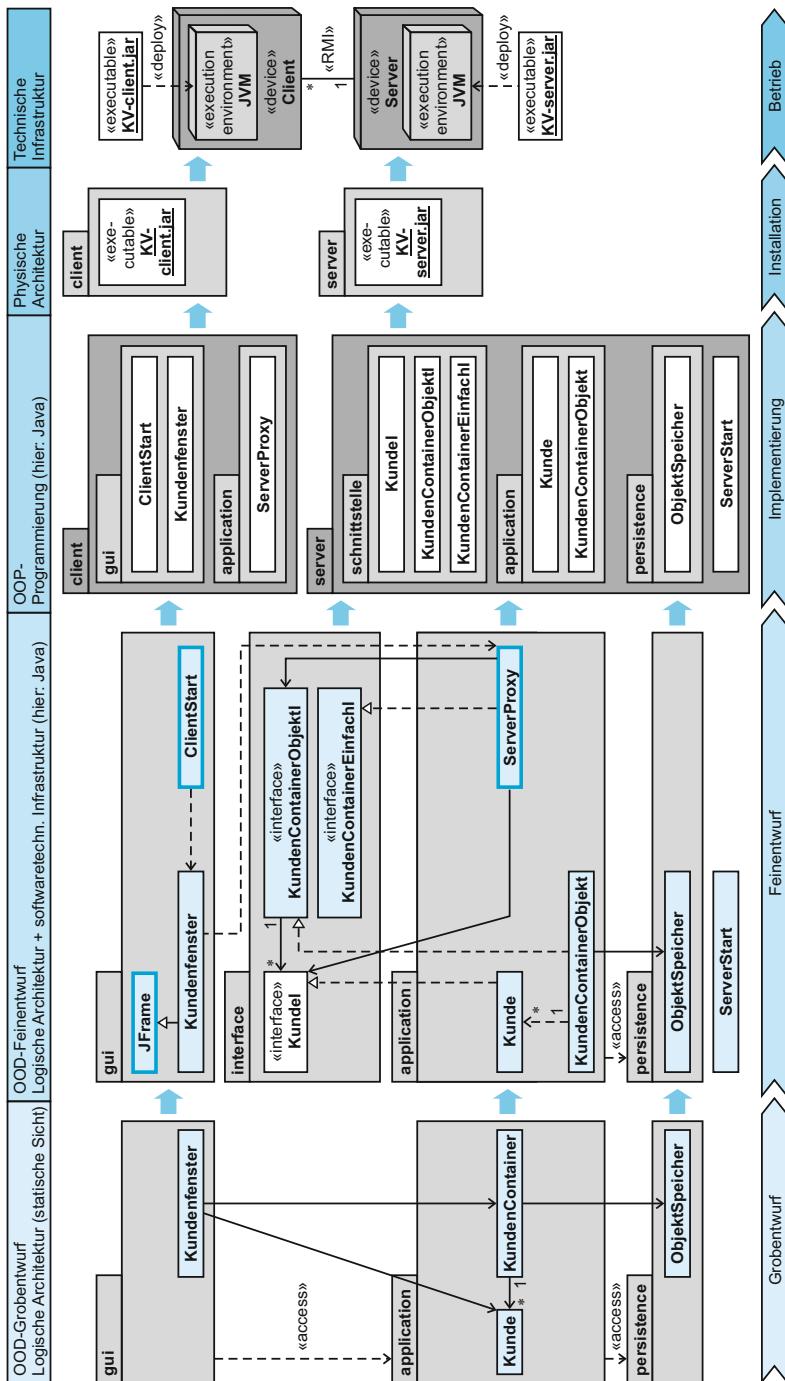


Abb. 15.2-14: Entwicklungsprozess der Fallstudie »KV – RMI«.

I 15 Arten der Netzkommunikation

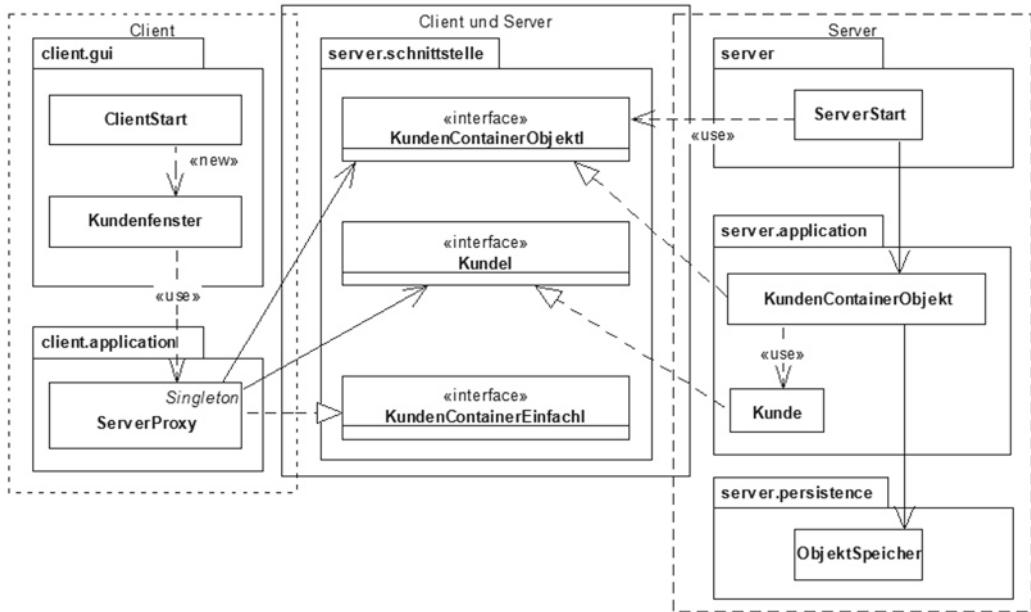


Abb. 15.2-15: OOD-Klassendiagramm der Fallstudie

Die Abb. 15.2-16 zeigt ein Szenario, in dem ein neues Kundenobjekt erstellt wird. Der Ablauf sieht wie folgt aus:

1 Starten des Servers

○ Zur Anwendung hinzugefügte Methoden: RML-Parameter erweitert

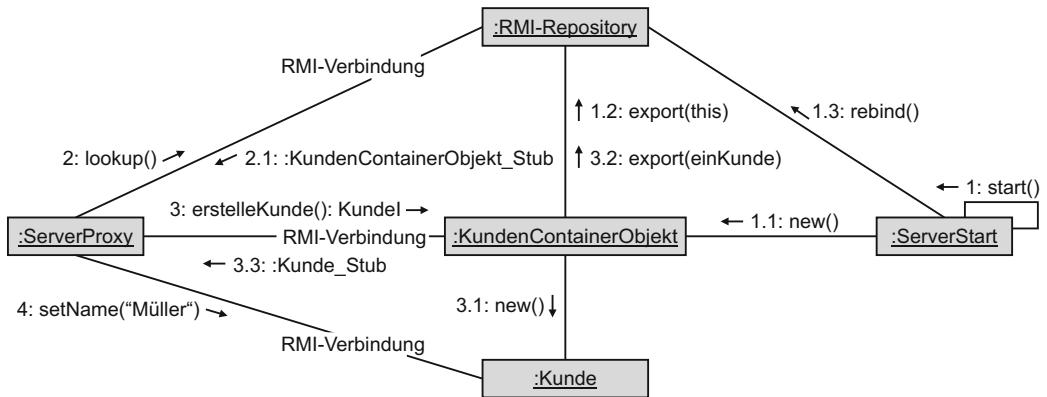
- Zuerst wird in der `main()`-Methode das RMI-Repository gestartet.
 - Anschließend wird die Methode `start()` aufgerufen, die ein Objekt `service` der Klasse `KundenContainerObjekt` erstellt. Dadurch wird automatisch dieses Objekt an den RMI-Repository exportiert (freigegeben). Dies geschieht implizit, d. h. es muss nicht im Quellcode eingetragen sein.
 - Zusätzlich werden alle bereits vorhandenen Kunden in der Klasse `ObjektSpeicher` ausgelesen und ebenfalls an den RMI-Repository exportiert.
 - Dem Objekt `service` der Klasse `KundenContainerObjekt` wird ein Name zugeordnet, damit das Objekt vom Client über den Java-Namensdienst angesprochen werden kann.

2 Starten des Clients

- Beim Starten des Clients wird zunächst eine Verbindung mit dem RMI-Server hergestellt.
 - Der Client fordert ein Stummel-Objekt der Klasse KundenContainer Objekt über das RMI-Laufzeitsystem an, um später die Kundenobjekte zu verwalten.

3 Ein Kundenobjekt erstellen

- Da keine Implementierung der Fachklasse `Kunde` auf dem Client vorhanden ist, muss das Erzeugen eines Kundenobjekts auf dem Server erfolgen.



- Der **KundenContainerObjekt** erstellt ein neues Kundenobjekt in der Methode `erstelleKunde` und exportiert dieses an das RMI-Repository, damit es vom Client angesprochen werden kann.
- Ein Stummel-Objekt des neuen Kunden wird an den Client gesendet.

4 Zugreifen auf das Kundenobjekt

- Nun kann der Client über das Stummel-Objekt auf die Eigenschaften des Kundenobjekts zugreifen.

Dieses Beispiel zeigt einen vollständigen Kommunikationsprozess, in dem der Name eines Kunden anhand seiner Kundennummer über Java-RMI abgefragt wird (Abb. 15.2-17).

Abb. 15.2-16:
Kommunikationsdiagramm für die Erstellung eines neuen Kundenobjekts der Fallstudie »KV – RMI«.

Beispiel

- Der Client Kundenfenster schickt eine Anfrage eines Kundennamen über die Kundennummer »2».
- Die Anfrage wird an **ServerProxy** weitergeleitet, der sie wiederum als Methodenauftrag an das Proxy-Objekt (**KundenContainerObjekt_Stub**) delegiert.
- Dort wird der Aufruf serialisiert (*Marshalling*) und die sogenannte *Marshalled Invocation* wird über RMI an den Server weitergeleitet.
- Auf der Serverseite wird der Methodenauftrag vom RMI-System empfangen und an ein sogenanntes Skeleton-Objekt weitergegeben, das die Aufgabe hat, die Parameter zum Methodenauftrag zu deserialisieren (*Unmarshalling*).
- Durch den Java-Aufruf-Mechanismus (*invocation*) wird die Methode `getNameZuNr()` auf dem Remote-Objekt (*Real-Object*) ausgeführt.
- Das Rückgabeblob wird über Java-RMI (*Marshalling* → RMI-Verbindung → *Unmarshalling* → Weiterleitung) an **ServerProxy** gesendet, der lediglich den Namen an Kundenfenster weiterleitet.

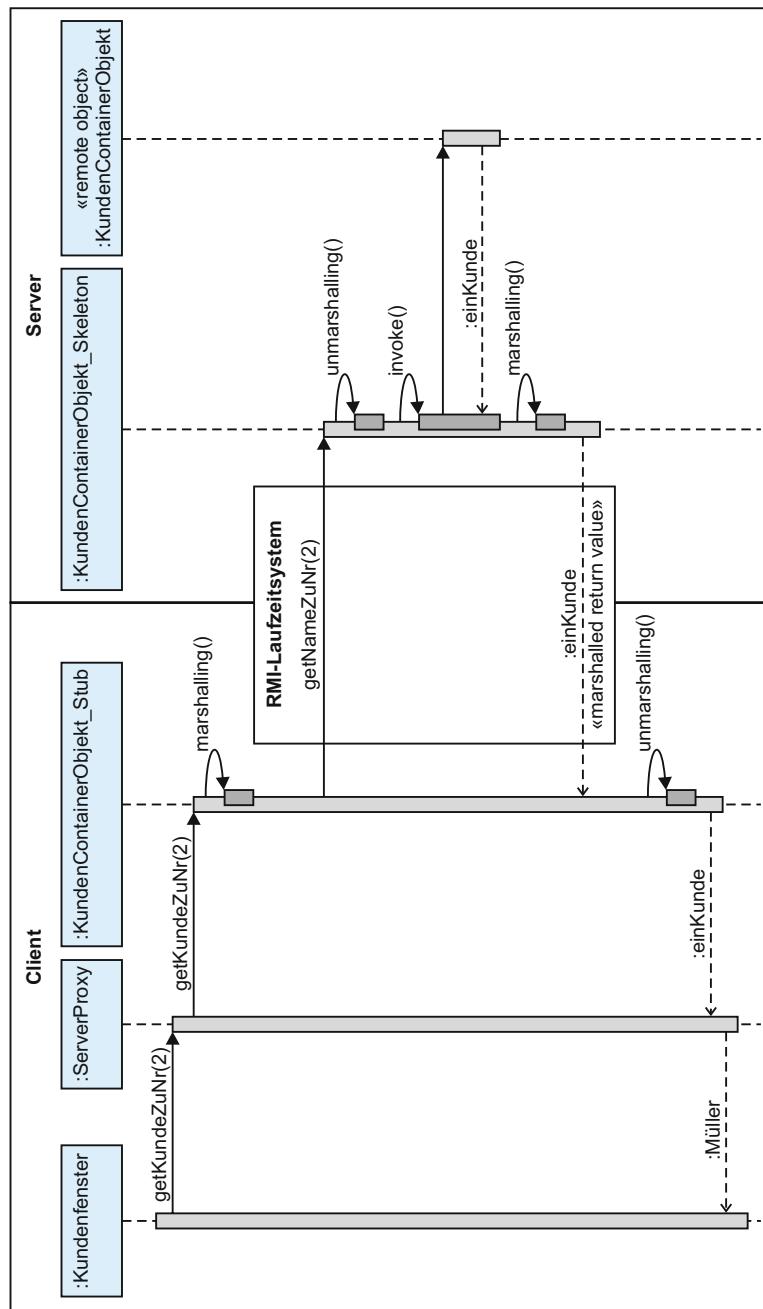
Die Abb. 15.2-17 zeigt das dazu passende Sequenzdiagramm.

Die Programme zu dieser Fallstudie können Sie im E-Learning-Kurs herunterladen.

Laden Sie das Programm auf Ihr Computersystem und führen Sie es aus. Sehen Sie sich die einzelnen Programme im Detail an.

I 15 Arten der Netzkommunikation

Abb. 15.2-17:
Abfrage des
Kundennamens in
der Fallstudie
»KV – RMI«.



15.3 CORBA

CORBA (*Common Object Request Broker Architecture*) ist eine plattformübergreifende **Spezifikation** der **OMG** (*Object Management Group*), die das Erstellen verteilter Anwendungen in heterogenen Umgebungen vereinfacht. Es gibt eine Reihe von kommerziellen und frei verfügbaren softwaretechnischen Plattformen, die der CORBA-Spezifikation entsprechen. Für alle wichtigen Betriebssysteme ist eine CORBA-Plattform verfügbar.

Die Basis für die Standardisierung bildet die OMA (*Object Management Architecture*) (Abb. 15.3-1).

OMA

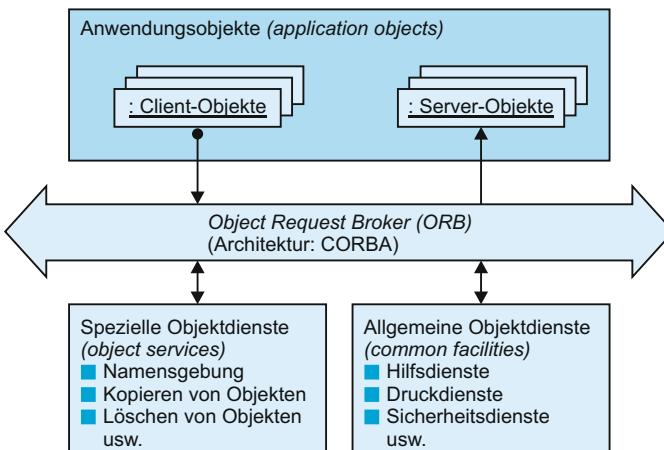


Abb. 15.3-1: OMA
(Object Management Architecture).

Die Hauptbestandteile dieser Architektur sind

- Anwendungsobjekte (*application objects*), die sowohl Client- als auch Server-Objekte sein können,
- ein ORB (*Object Request Broker*), der die Kommunikation zwischen den Objekten vermittelt,
- spezielle Objektdienste (*object services*), die der ORB zur Erfüllung seiner Aufgaben benötigt, z.B. Dienste für die Namensgebung, zum Kopieren und Löschen von Objekten und
- allgemeine Objektdienste (*common facilities*) wie z.B. Hilfe-, Druck- und Sicherheitsdienste.

Die Schlüsselkomponente der Architektur ist der **ORB** (*Object Request Broker*), der als Kommunikationszentrale im Mittelpunkt der Architektur steht. Zu seinen zentralen Aufgaben gehört die Übermittlung von Methodenaufrufen und ihren Ergebnissen zwischen Anwender und Anbieter. Der ORB ist vergleichbar mit einer Telefonzentrale. Da von dieser Architekturkomponente alle anderen Komponenten abhängen, wurde sie unter dem Namen CORBA (*Common Object Request Broker Architecture*) als erstes standardisiert [OMG96].

I 15 Arten der Netzkomunikation

Der ORB benutzt die Schnittstellen von Client- und Server-Objekten, um Anforderungen (*requests*) von Client-Objekten an Server-Objekte weiterzuleiten und die Ergebnisse zurückzuliefern.

IDL Die Schnittstellen werden mit einer Schnittstellen-DefinitionsSprache (CORBA-IDL, *interface definition language*) beschrieben. Die **IDL** basiert auf dem OMG-Objektmodell.

Auf CORBA-Objekte kann von außen nur über ihre Schnittstelle zugegriffen werden.

Quality of Services (QoS) Die CORBA-Spezifikation behandelt auch den Einsatz in speziellen Umgebungen. Standard-CORBA geht von einer gewöhnlichen Hardware- und Betriebssystem-Umgebung aus. Für bestimmte Einsatzgebiete stellen sich jedoch besondere Anforderungen. Unter der Bezeichnung *Quality of Services* (QoS) sind für folgende Bereiche nichtfunktionale Anforderungen in der Spezifikation enthalten:

■ Minimum CORBA

Dabei handelt es sich um eine Variante, die für Systeme mit begrenzten Hardware-Ressourcen (Rechenleistung, Hauptspeicher) entwickelt wurde. Zielplattformen sind z.B. Handys oder PDAs (*personal digital assistant*).

■ Echtzeit-CORBA

Damit auch technische Systeme (z.B. in der Steuerungstechnik) auf CORBA aufgebaut werden können, definiert Realtime-CORBA, wie Echtzeitanforderungen zu handhaben sind.

■ Fehlertolerantes CORBA

Hier werden Aussagen zu fehlertoleranten Systemen gemacht.

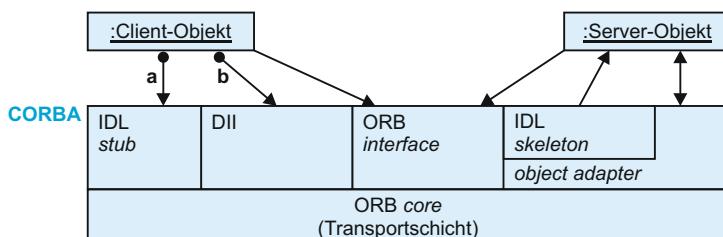
i Auf die wichtigsten Konzepte von CORBA wird detaillierter eingegangen:

- »Die Architektur von CORBA«, S. 242
- »Die Schnittstellendefinitionssprache IDL«, S. 244
- »Standardisierte CORBA-Dienste«, S. 245
- »Fallstudie: KV – CORBA in Java«, S. 246

15.3.1 Die Architektur von CORBA

Die ORB-Architektur entsprechend dem CORBA-Standard zeigt die Abb. 15.3-2.

Abb. 15.3-2:
CORBA.



CORBA besteht aus einem ORB-Kern (ORB *Core*), der als **Transportschicht** angesehen werden kann. Er stellt die Basiskommunikation für die anderen Komponenten zur Verfügung.

ORB-Kern

Ein Server, der Dienstleistungen zur Verfügung stellen will, muss die bereitgestellten Methoden in einer Schnittstelle (*interface*) beschreiben. Die Schnittstelle wird mit der **IDL** (*interface definition language*) (siehe »Die Schnittstellendefinitionssprache IDL«, S. 244) spezifiziert.

Aus einer IDL-Schnittstellenbeschreibung werden generativ folgende Informationen abgeleitet (Abb. 15.3-3): *interface*

- **IDL-Stummel (stub):** Stellt Routinen, ähnlich wie Bibliotheks Routinen, zur Verfügung, die der Client benutzt, um Dienstleistungen des Servers anzufordern.
- **Interface Repository:** Speichert Informationen über die Schnittstelle. Diese Informationen können zur Laufzeit von Clients abgefragt werden, um dynamische Anforderungen aufzubauen. Dazu benutzen sie die DII (*dynamic invocation interface*).
- **IDL-Skelett (skeleton):** Rahmen, der vom Programmierer mit Code gefüllt werden muss. Dieser Code wird ausgeführt, wenn eine Anforderung eintrifft.
- **Implementation Repository:** Verwaltet Informationen, die der ORB benötigt, um Server zu lokalisieren und zu starten.

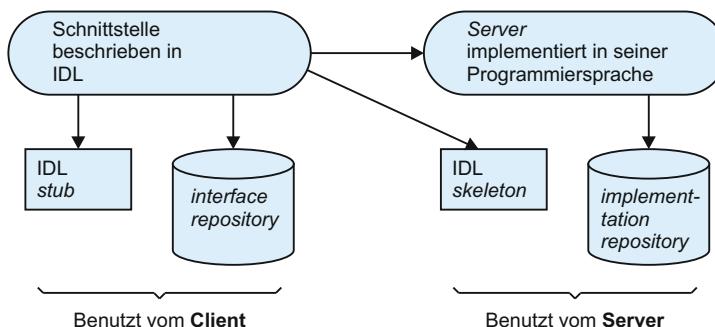


Abb. 15.3-3: Aus der IDL-Schnittstelle abgeleitete Informationen.

Clients interagieren mit dem ORB über IDL-Stummel und DII jeweils in der eigenen Programmiersprache. Ein Client kann auf zwei Arten Dienstleistungen anfordern (*request*):

Clients

- Ist die Schnittstelle in IDL definiert und kennt der Client die Typ- bzw. Klassendefinition des Servers, dann kann eine statische Anforderung erfolgen. Sie wird unter Verwendung des entsprechenden IDL-Stummels durchgeführt (Abb. 15.3-2, a).
- Ist die Schnittstelle des Servers *nicht* bekannt, dann kann die DII verwendet werden (Abb. 15.3-2, b). Die Anforderung wird zur Laufzeit erstellt und verwendet keine IDL-Stummel. Die Anforderung wird stattdessen von der DII-Komponente durchgeführt, die dazu Informationen aus dem *interface repository* verwendet.

I 15 Arten der Netzkommunikation

- Server Wird eine Anforderung vom ORB beim Server abgeliefert, dann weiß der Server nicht, ob die Anforderung statisch oder dynamisch erstellt wurde.
- Objekt-Adapter Anforderungen passieren – vom ORB kommend – den IDL-Stummel, um zum Server zu gelangen. Ein IDL-Skelett hängt in der Regel vom Objekt-Adapter ab (Abb. 15.3-2). Es kann mehrere Skelette für dieselbe Schnittstelle und dieselbe Programmiersprache mit unterschiedlichen Objekt-Adaptoren geben. Ein Objekt-Adapter stellt eine Schnittstelle zu den ORB-Dienstleistungen zur Verfügung, die die Server nutzen können.
- ORB-Interface Andere Dienstleistungen, die von den Servern benutzt werden, werden direkt von der Komponente ORB-Interface zur Verfügung gestellt (Abb. 15.3-2). Diese Schnittstelle ist für alle ORB-Implementierungen identisch und hängt nicht vom Objekt-Adapter ab. Sie wird auch von Clients genutzt.
- repositories* Das *interface repository* (IR) verwaltet die IDL-Definitionen. Diese sind zur Laufzeit verfügbar. Die DII benutzt diese Informationen, um Clients Anforderungen an Server zu erlauben, deren Schnittstelle zur Übersetzungszeit des Clients noch unbekannt war. Die IR-Dienstleistung wird auch vom ORB selbst benutzt, um Anforderungen weiterzuleiten.
- Das *implementation repository* verwaltet Informationen, die der ORB benötigt, um die Server zu lokalisieren und zu starten. Es ist spezifisch für eine Betriebssystem-Umgebung.
- IIOP Damit ORBs verschiedener Hersteller über das Internet miteinander kommunizieren können, wurde das *Internet Inter-ORB Protocol* (IIOP) entwickelt. Es handelt sich dabei um ein in CORBA definiertes Protokoll, um Methodenaufrufe von Objekten auf anderen Computersystemen durchzuführen. IIOP spezialisiert das abstrakte *General Inter-ORB Protocol* (GIOP) für die Kommunikation über TCP/IP.

15.3.2 Die Schnittstellendefinitionssprache IDL

Die Schnittstellendefinitionssprache IDL (*interface definition language*) dient dazu, die Schnittstellen von Klassen zu beschreiben, die als Server Dienstleistungen anbieten. In der IDL werden *keine* Programme geschrieben, sondern nur Spezifikationen. Diese Spezifikationen verwendet der Anwendungsentwickler, um vom Client aus Methoden einer Server-Klasse aufzurufen.

Die Aufrufe der Methoden erfolgen in der Programmiersprache, die für den Client verwendet wird. Die Implementierung der Methoden auf der Server-Seite wird in der Programmiersprache durchgeführt, die für den Server verwendet wird. Diese Programmiersprachen müssen *nicht* übereinstimmen. IDL-Precompiler transformieren die IDL-Spezifikationen in die jeweiligen Zielsprachen.

Die Syntax von IDL ist an C++ angelehnt. Die Grammatik ist eine Teilmenge der C+-Grammatik mit Erweiterungen für verteilte Aufrufe.

Zu CORBA-IDL existieren Abbildungen für die wichtigsten Programmiersprachen (*language mappings*). In diesen Spezifikationen wird genau angegeben, wie die IDL-Konstrukte auf entsprechende Sprachkonstrukte abzubilden sind.

Eine Besonderheit stellt Java dar. Es gibt eine Spezifikation für die Abbildung von IDL auf Java. In Java werden direkt Java-Schnittstellen (*interface*) genutzt. Die CORBA-Spezifikation sieht auch eine Umkehrabbildung (*reverse mapping*) vor. Damit lässt sich aus Java-Schnittstellen IDL-Code generieren.

Java

15.3.3 Standardisierte CORBA-Dienste

Die CORBA-Spezifikation definiert eine Reihe von Diensten. Diese unterteilen sich in zwei große Kategorien:

- Objektdienste (*object services*)
- Allgemeine Dienste (*common facilities*)

Die Objektdienste können als Gegenstück zum API (*application programming interface*) eines Betriebssystem angesehen werden. Sie spezifizieren unabhängig von einem konkreten Anwendungsbereich Dienstleistungen, die viele Klassen häufig benötigen. Alle Objektdienste sind in IDL spezifiziert und werden vom Plattform-Entwickler implementiert. Zu den Objektdiensten gehören z. B.:

Vergleich:
Betriebs-
systemdienste

- Namensdienst (*name service*): Kreierung von Namensräumen und die Abbildung von benutzerdefinierten Objektnamen auf CORBA-Objektreferenzen.
- Lebenszyklusdienst (*lifecycle service*): Verwaltung von Objekten (Erzeugen, Kopieren, Löschen, Abfragen auf Äquivalenz).
- Ereignismeldedienst (*event notification*): Erkennen von Ereignissen und Benachrichtigung von Objekten über Ereignisse.
- Persistenzdienst (*persistence service*): Dauerhaftes Speichern und Laden von Objekten auf/von (externen) Speichern.
- Nebenläufiger Dienst (*concurrency service*): Synchronisierung konkurrierender Zugriffe auf Objekte.
- Auslagerungsdienst (*externalization service*): Datenexport von Objekten in Dateien außerhalb des Objektsystems.
- Beziehungsdiensst (*relationship service*): Erzeugen, Löschen und Verwalten von Beziehungen zwischen Objekten, Navigation über Beziehungen.
- Transaktionsdienst (*transaction service*): Techniken zur Durchführung transaktionsgesteuerter Abläufe mit einem zweistufigen *commit*.

I 15 Arten der Netzkommunikation

- Zeitdienst (*time service*): Synchronisierung der Uhren von Komponentensystemen in einer verteilten Umgebung.
- Sicherheitsdienst (*security service*): Autorisierungs- und Überwachungsfunktionen auf Objektebene.
- Lizenzdienst (*licensing service*): Verwaltung von Objektlizenzen, Abrechnung von Gebrauchsgebühren für Objekte.
- Eigenschaftsdienst (*properties service*): Dynamisches Erzeugen, Löschen und Verwenden von Attributen.
- Anfragedienst (*query service*): SQL-artige Anfrageoperationen für Objekte.

Allgemeine Objektdienste Allgemeine Objektdienste (*common facilities*) sind höherwertige Dienste, die für viele Anwendungsbereiche relevant sind. Vergleichbar sind diese Dienste mit grundlegenden Klassenbibliotheken, die allgemein benötigte Funktionalitäten modular zur Verfügung stellen.

Die allgemeinen Objektdienste lassen sich zwei Kategorien zuordnen:

- Vertikale Marktdienste (*vertical market facilities*): Definieren eine Basisfunktionalität für unterschiedliche Marktsegmente wie Buchhaltungsanwendungen oder CAD.
- Horizontale allgemeine Dienste (*horizontal common facilities*): Dienste, die sich über mehrere Anwendungsbereiche erstrecken. Sie lassen sich in vier Bereiche gliedern:
 - *user interface* definiert den Zugriff auf ein grafisch orientiertes Informationssystem.
 - *information management* legt Modellierung, Definition, Speicherung, Wiedergewinnung, Verwaltung und Austausch von Informationen fest.
 - *systems management* unterstützt die Verwaltung komplexer Informationssysteme.
 - *task management* erlaubt die Automatisierung von Aufgaben, d. h. von Benutzer- und Systemprozessen.

15.3.4 Fallstudie: KV – CORBA in Java

Die Entwicklung einer einfachen CORBA-Klasse umfasst mehrere Schritte:

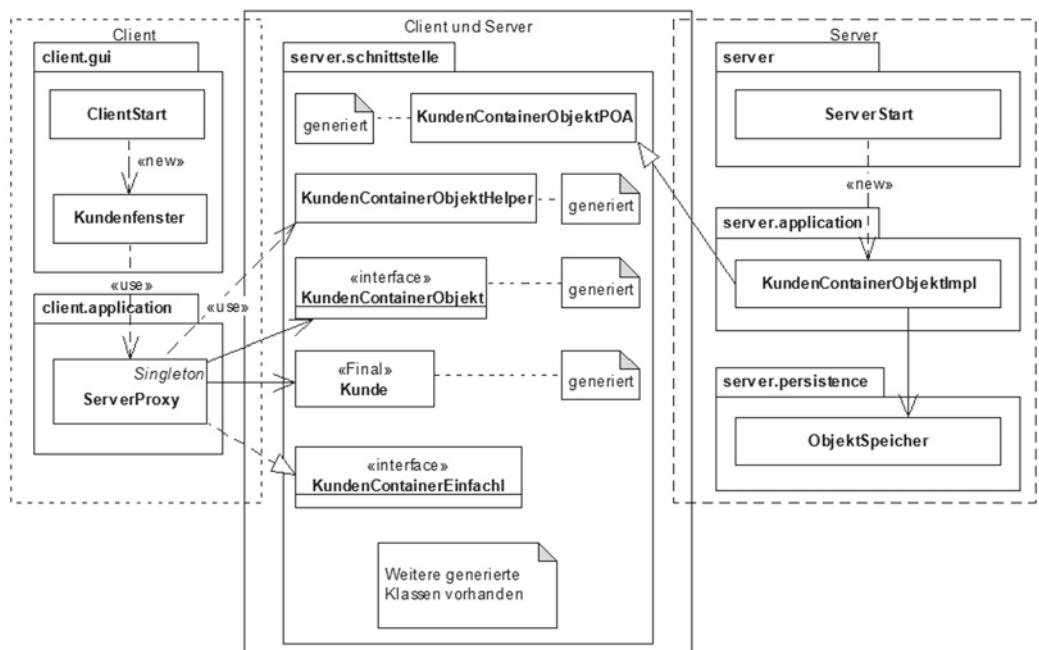
- 1 Spezifikation der Schnittstelle in CORBA-IDL.
- 2 Übersetzung der IDL-Spezifikation mit einem IDL-Compiler. Der Compiler erzeugt automatisch Stummel- und Skelett-Klassen.
- 3 Implementierung der Methoden der Schnittstelle.
- 4 Entwicklung einer Rahmen-Anwendung, in die die entwickelte CORBA-Klasse eingebettet ist. Die Rahmen-Anwendung erzeugt ein Objekt der Klasse und macht es für Clients auf anderen Computersystemen zugreifbar.

5 Entwicklung der Clients. In dem folgenden einfachen Beispiel ist der Client eine einfache Anwendung. In realen Projekten sind Clients zumeist selbst wieder CORBA-Klassen, die eigene Dienstleistungen anbieten, sodass ein Netz von miteinander kommunizierenden Objekten entsteht.

Für die Fallstudie wird der frei verfügbare JacORB als CORBA-Plattform genutzt. Im kostenlosen E-Learning-Kurs zu diesem Buch wird die Installation von JacORB beschrieben.

Installation

Das gesamte OOD-Modell für die »Kundenverwaltung-Mini CORBA« zeigt die Abb. 15.3-4. Die Klassen ClientStart, Kundenfenster und ObjektSpeicher sowie die Schnittstelle KundenContainerEinfachI bleiben unverändert.



```
//kundencontainer.idl
module server
{
    module schnittstelle
    {
        struct Kunde
        {
            string name;
            long nummer;
        };

        interface KundenContainerObjekt
        {
    
```

Abb. 15.3-4: OOD-Klassendiagramm der Fallstudie »KV - CORBA«.

1 Spezifikation der IDL

I 15 Arten der Netzkommunikation

```
void einfuegeKunde(in Kunde einKunde);
Kunde getKundeZuNr(in long nummer);
Kunde erstelleKunde();
long getNaechsteKundenNr();
void endeAnwendung();
};

};

};
```

2 Übersetzung der Spezifikation mit dem IDL-Compiler

Die Datei kundencontainer.idl kann nun mit dem IDL-Compiler des JacORB übersetzt werden. Der Compiler erzeugt eine Reihe von Dateien. Welche Dateien dies sind und wie sie aussehen, wird weitgehend durch die im CORBA-Standard enthaltene Sprachanbindung für Java festgelegt. Der Aufruf des IDL-Compilers erfolgt über die Kommandozeile:

```
idl kundencontainer.idl
```

Der IDL-Compiler erzeugt ein Java-Paket namens schnittstelle und generiert dort einige Klassen:

- Kunde: Die aus der CORBA-IDL erzeugte Struktur für den Datentyp Kunde.
- KundeHelper: Diese Klasse ist dafür zuständig, dass der Client das von ihm gewünschte Objekt vom Server erhält.
- KundeHolder: Diese Klasse ist zuständig für die Übergabe von Parametern.
- _KundenContainerObjektStub: Die Stummel-Klasse.
- KundenContainerObjekt: Die aus der CORBA-IDL erzeugte Java-Schnittstelle.
- KundenContainerObjektHelper: Diese Klasse ist dafür zuständig, dass der Client das von ihm gewünschte Objekt vom Server erhält.
- KundenContainerObjektHolder: Diese Klasse ist zuständig für die Übergabe von Parametern.
- KundenContainerObjektOperations: Eine Hilfsklasse.
- KundenContainerObjektPOA: Die Skelett-Klasse (POA = *programmable object adapter*). Diese Klasse bildet eine Hülle für die eigentliche Implementierung und implementiert die Schnittstelle KundenContainerObjekt.
- KundenContainerObjektPOATie: Eine Hilfsklasse.

Hinweis

In der generierten Java-Klasse Kunde sind die Attribute name sowie nummer als öffentlich gekennzeichnet, was einen direkten Zugriff auf diese Attribute zur Folge hat. Dies entspricht jedoch nicht dem Geheimnisprinzip. Soll das Geheimnisprinzip gewahrt werden, muss die Klasse Kunde im Paket server.schnittstelle entsprechend geändert werden, damit der Zugriff auf die Attribute

explizit über *Getter* bzw. *Setter* durchgeführt werden muss. In dieser Fallstudie wurden *get()*- und *set()*-Methoden hinzugefügt, da diese für die Speicherung notwendig sind.

Die Implementierung erfolgt in der Klasse `KundenContainerObjektImpl`. Diese muss von der generierten Klasse `KundenContainerObjektPOA` erben. Die Implementierung sieht folgendermaßen aus:

```
public class KundenContainerObjektImpl
    extends KundenContainerObjektPOA
{
    private ObjektSpeicher einObjektSpeicher;
    private ArrayList<Kunde> meineKunden;

    public KundenContainerObjektImpl()
    {
        super();
        init();
    }

    private void init()
    {
        einObjektSpeicher = new ObjektSpeicher();
        try
        {
            meineKunden =
                (ArrayList<Kunde>) einObjektSpeicher.leseObjekt();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        if (meineKunden == null)
        {
            meineKunden = new ArrayList<Kunde>();
            System.out.println(
                "Es wurde eine neue Datenbasis angelegt");
        }
    }

    @Override
    public Kunde erstelleKunde()
    {
        Kunde einKunde = new Kunde();
        return einKunde;
    }

    @Override
    public void einfuegeKunde(Kunde einKunde)
    {
        if (getKundeZuNr(einKunde.nummer) == null)
        {
            //Kunde noch nicht vorhanden
        }
    }
}
```

3 Implementierung der Methoden der Schnittstelle

I 15 Arten der Netzkommunikation

```
        meineKunden.add(einKunde);
    }
else
{
    Iterator<Kunde> iter = meineKunden.iterator();
    while (iter.hasNext())
    {
        Kunde kunde = iter.next();
        if (kunde.nummer == einKunde.nummer)
            kunde.name = einKunde.name;
    }
}

@Override
public int getNaechsteKundenNr()
{
    int max = 0;
    Iterator<Kunde> iter = meineKunden.iterator();
    while (iter.hasNext())
    {
        Kunde kunde = iter.next();
        max = Math.max(max, kunde.nummer);
    }
    return max + 1;
}

@Override
public Kunde getKundeZuNr(int nummer)
{
    for(int i=0;i<meineKunden.size();i++)
        if (meineKunden.get(i).nummer==nummer)
            return meineKunden.get(i);
    return null;
}

public void endeAnwendung()
{
    System.out.println("jetzt speichern:");
    for (int i=0;i<meineKunden.size();i++)
        System.out.println("####"+meineKunden.get(i).name);

    einObjektSpeicher.speichereObjekt(meineKunden);
    System.out.println("Datenbasis wurde gespeichert");
}
}
```

4 Entwicklung
einer Rahmen-
Anwendung

Die Rahmenanwendung wird in der Klasse ServerStart realisiert:

```
public class ServerStart
{
    public ServerStart(String[] args)
    {
        org.omg.CORBA.ORB orb =
            org.omg.CORBA.ORB.init(args, null);
```

```

try
{
    org.omg.PortableServer.POA poa =
        org.omg.PortableServer.POAGHelper.narrow(
            orb.resolve_initial_references("RootPOA"));

    poa.the_POAManager().activate();

    org.omg.CORBA.Object tempObjektContainer =
        poa.servant_to_reference(
            new KundeContainerObjektImpl());

    NamingContextExt nc = NamingContextExtHelper.narrow(
        orb.resolve_initial_references("NameService"));
    nc.bind(nc.to_name("einKundeContainer"),
        tempObjektContainer);
}
catch (Exception e)
{
    System.out.println(e.getStackTrace());
}
orb.run();
}

public static void main(String[] args)
{
    new ServerStart(args);
}
}

```

Der folgende Ausschnitt zeigt den Verbindungsauflaufbau des Clients in der Klasse ServerProxy: 5 Entwicklung eines Client

```

public class ServerProxy implements KundenContainerEinfachI
{
    //Singleton-Muster
    private static ServerProxy serverProxy = null;

    private static KundenContainerObjekt datenService;
    private static Kunde einKunde = null;

    public ServerProxy()
    {
        try
        {
            System.out.println("serverproxy");
            //wird bei der Initialisierung benoetigt
            String args[] = null;

            //den Object-Request-Broker initialisieren
            org.omg.CORBA.ORB orb =
                org.omg.CORBA.ORB.init(args, null);

            NamingContextExt nc =
                NamingContextExtHelper.narrow(
                    orb.resolve_initial_references("NameService")));

```

I 15 Arten der Netzkomunikation

```
    datenService=KundenContainerObjektHelper.narrow(      nc.resolve(nc.to_name("einKundeContainer")));  }  catch (Exception e)  {    System.out.println(e.getStackTrace());  }}

public static ServerProxy getObjektreferenz()
{
  if (serverProxy == null)
    serverProxy = new ServerProxy();
  return serverProxy;
}

{ ... }
```

Anschließend können alle .java-Dateien mit einem Java-Compiler übersetzt werden. Dann kann das System gestartet werden:

- Zunächst wird der Namensdienst mit ns C:\ns_ref gestartet. Der Parameter gibt eine Datei an, in die der Namensdienst Ausgaben schreiben kann. Er dient zum initialen Aufbau einer Kommunikationsverbindung zwischen Namensdiensten auf verschiedenen (virtuellen) Maschinen.
- Mit jaco server.ServerStart wird die Rahmen-Anwendung gestartet und das ObjektContainer-Objekt verfügbar gemacht.
- Die Client-Anwendung wird mit jaco client.gui.Kundenfenster gestartet. Um die verteilte Kommunikation beobachten zu können, sollte sie in einer anderen Java-VM laufen (unter Windows lässt sich dies z.B. durch den Start von zwei DOS-Boxen erreichen).

Das vollständige Programm und eine Installationsbeschreibung können Sie im kostenlosen E-Learning-Kurs herunterladen.

 Führen Sie das Programm auf Ihrem Computersystem aus und gehen Sie die Programme Schritt für Schritt durch.

Die Abb. 15.3-5 zeigt den allgemeinen Ablauf einer verteilten Softwareentwicklung in CORBA.

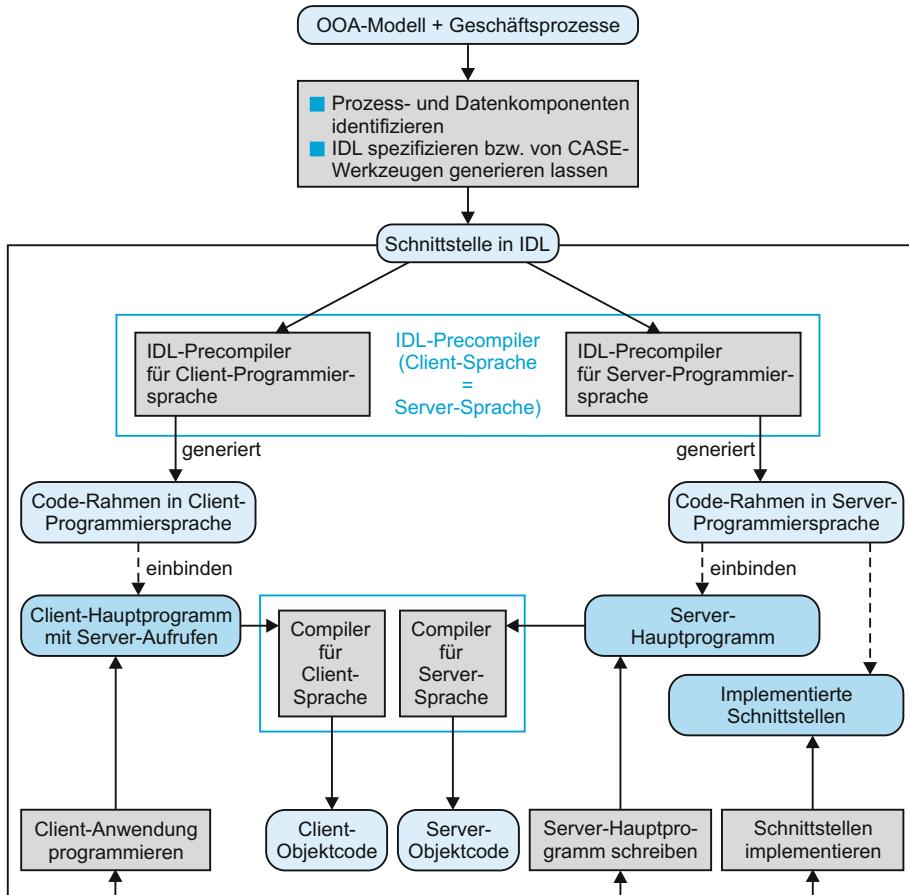


Abb. 15.3-5:
Allgemeiner
Ablauf einer
verteilten Soft-
wareentwicklun
in CORBA.

15.4 XML-RPC

Die Technik **XML-RPC** (*remote procedure calls*) ermöglicht es, Methoden bzw. Prozeduren, die sich auf anderen, miteinander verbundenen Computersystemen befinden, aufzurufen. Die zu übertragenden Daten werden dabei in **XML** (*Extensible Markup Language*) dargestellt. Der Transport der Daten erfolgt über **HTTP** (*Hypertext Transfer Protocol*). Durch die Wahl dieser beiden Standards ist es möglich, mit XML-RPC unterschiedliche Programmiersprachen und Plattformen miteinander kommunizieren zu lassen. Für gängige Programmiersprachen gibt es Bibliotheken, die die Basisfunktionalität für den Methodenaufruf und die Methodenbehandlung bereitstellen. Folgende Aufgaben übernehmen diese Bibliotheken:

- Repräsentation der Datentypen.
- Generierung der Aufrufpakete und Analyse der Antwortpakete.

I 15 Arten der Netzkommunikation

- Übertragung und Empfang der Pakete.
- Erzeugung von Skelett- und Stummel-Rahmen.

Datentypen

Die Tab. 15.4-1 zeigt, welche Datentypen XML-RPC unterstützt.

| Typname in XML-RPC | Beschreibung | Beispiel |
|--------------------|--|---|
| boolean | Boolesche Variable | <boolean>0</boolean> |
| int, i4 | 32-bit-Integer mit Vorzeichen | <int>987</int> |
| double | Gleitkommazahl mit doppelter Präzision | <double>7.6</double> |
| string | Zeichenkette | <string>XML-RPC</string> |
| dateTime.iso8601 | Datum & Uhrzeit im ISO-Format | <dateTime.iso8601>2011-07-16T19:20+01:00</dateTime.iso8601> |
| base64 | Base64-codierte Bytefolge, eignet sich zum Übertragen serialisierter Objekte | <base64>WE1MLVJQQw==</base64> |

Tab. 15.4-1: Datentypen in XML-RPC. Listen von Datenelementen können zu einem `<array>` zusammengefasst und übertragen werden. Eine Liste kann eine beliebige Folge von `<value>`-Elementen aufnehmen, die auch von unterschiedlichem Typ sein können.

`<struct>` Komplexe Datenstrukturen werden mithilfe von `<struct>` aufgebaut. Ein `<struct>`-Element besteht aus Schlüssel-Wert-Paaren. Auf jeden Wert kann durch einen eindeutigen Schlüsselwert zugegriffen werden.

Java Dem Datentyp `<array>` entspricht in Java die Liste (`java.util.List`) oder ein Feld (`array`). `<struct>` kann in Java durch eine *Map* (`java.util.Map`), z. B. eine *HashTable*, realisiert werden.

HTTP-POST

Der Methodenaufruf wird über HTTP-Post an den Empfänger übermittelt. Das zu übertragende XML-Dokument wird im *Request Body* angeordnet und hat folgenden Aufbau:

```
<methodCall>
  <methodName>
    <! --Name der aufgerufenen Methode -->
  </methodName>
  <params>
    <param> <! -- 1. Parameter -->
      <value><Datentyp><! --Wert1--></Datentyp>
    </value>
    <value><Datentyp><! --Wert2 usw.--></Datentyp>
    </value>
  </params>
</methodCall>
```

```

</param>
<param> <!-- 2. Parameter usw. --> </param>
</params>
</methodCall>

```

Die Parameter erhalten keine Namen. Die Parameterposition bestimmt die Semantik. Die Reihenfolge und Bedeutung der Parameter muss zwischen Client und Server abgestimmt werden.

Die Übergabe der Ergebnisse erfolgt analog wie bei der Anfrage:

```

<methodResponse>
  <methodName>
    <! --Name der aufgerufenen Methode -->
  </methodName>
  <params>
    <param> <!-- 1. Parameter --> </param>
    <param> <!-- 2. Parameter usw. --> </param>
  </params>
</methodResponse>

```

Null-Werte (`nil`, `null` oder `nul`) können nicht dargestellt werden. Null-Werte Bei den Parametern muss es sich also immer um konkrete Werte handeln.

Führt ein Methodenaufruf zu einer Ausnahme, dann kann das Antwortdokument statt der Rückgabeparameter auch das Element `<fault>` enthalten, das ebenfalls einen Wert enthalten kann. Fehlermeldungen

Vor- und Nachteile

- + Das Frage-Antwort-Schema gibt das Wesen eines Methodenaufrufs Vorteile gut wieder.
- + Durch die Verwendung des HTTP-Protokolls werden vorhandene Schutzvorrichtungen wie Firewalls und *Proxies* getunnelt.
- + Gute Performanz. Die *Roundtrip*-Zeiten für entfernte Aufrufe liegen in der Regel bei wenigen Millisekunden.
- + In vielen Programmiersprachen relativ einfach realisierbar.
- + Es gibt Implementierungen für Unix, Linux, Windows und Mac.
- + Vielfach eingesetzt im *Online-Publishing*, bei *Weblogs* und für *Microcontent*.
- + Auch für den Einsatz auf mobilen Geräten geeignet.
- Bei den Datentypen gibt es Einschränkungen. Zeichenketten können nur im 7-Bit-ASCII-Format übertragen werden. Das Datumsformat legt keine Zeitzone fest. Nachteile
- Durch die Übertragung im base64-Format, werden die Dokumente um rund 30% vergrößert.
- Da es keine Meta-Beschreibung der Methoden-Signaturen gibt, muss der Entwickler genau wissen, wie die aufzurufende Methode heißt und welche Parameter sie erwartet.

I 15 Arten der Netzkomunikation

- | Java ermöglicht es, die Technik XML-RPC einzusetzen. Die Fallstudie »Kundenverwaltung-Mini« kann daher ebenfalls mit XML-RPC realisiert werden:
- »Fallstudie: KV – XML-RPC«, S. 256

15.4.1 Fallstudie: KV – XML-RPC

In Java gibt es kein standardisiertes API für XML-RPC. Jede Implementierung benutzt eigene Schnittstellen und Klassen, z. B. Apache XML-RPC.

Java In Java erfolgt auf der Serverseite die Implementierung in der Regel durch Servlets, die die Anfrage parsen und eine entsprechende Antwort erzeugen. Für XML-RPC gibt es in Java *kein* standardisiertes API. Jede Implementierung verwendet ihre eigenen Schnittstellen und Klassen.

Apache Eine kompakte Implementierung für Java stellt Apache XML-RPC zur Verfügung. Zusätzlich gibt es Erweiterungen für Servlets und SSL. Jakarta Tomcat kann als Server dienen. Um Apache XML-RPC nutzen zu können, müssen folgende Bibliotheken verfügbar sein:

- Von der Website Apache-Prosite (http://apache.prosite.de/ws_xmlrpc/) muss die Datei apache-xmlrpc-current-bin.zip heruntergeladen und entpackt werden. Die folgenden Bibliotheken müssen im lib-Verzeichnis in das Projekt eingebunden werden (Versionsnummern können abweichen):

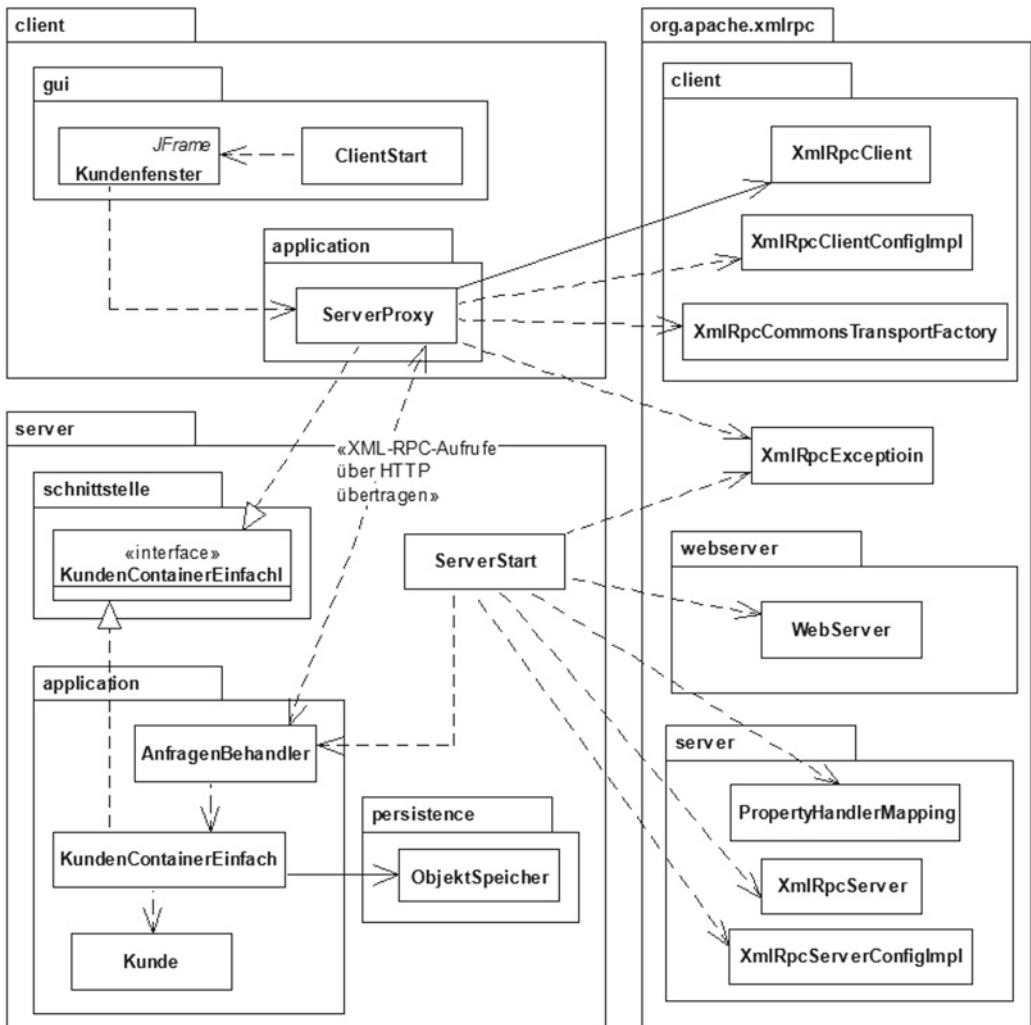
```
commons-logging-1.1.jar  
ws-commons-util-1.0.2.jar  
xmlrpc-client-3.1.3.jar  
xmlrpc-common-3.1.3.jar  
xmlrpc-server-3.1.3.jar
```

- Von der Website Commons-Apache (http://commons.apache.org/codec/download_codec.cgi) muss die Datei commons-codec-1.4-bin.zip heruntergeladen und entpackt werden. Die folgende Bibliothek muss in das Projekt eingebunden werden (Versionsnr. kann abweichen): commons-codec-1.4.jar
- Von der Website Hc-Apache (<http://hc.apache.org/downloads.cgi>) muss die Datei commons-httpclient-3.1.zip (unter Commons HttpClient 3.1, Binary, 3.1.zip) heruntergeladen und entpackt werden. Die folgende Bibliothek muss in das Projekt eingebunden werden (Versionsnr. kann abweichen): commons-httpclient-3.1.jar

Für die Fallstudie »Kundenverwaltung-Mini« kann für die Client-Server-Kommunikation die Technik XML-RPC eingesetzt werden. Für die Implementierung wird Apache XML-RPC verwendet. Das gesamte OOD-Modell zeigt die Abb. 15.4-1.

15.4 XML-RPC I

Die Klassen ClientStart, Kundenfenster, Kunde, KundenContainer Struktur Einfach und ObjektSpeicher sowie die Schnittstelle KundenContainer EinfachI bleiben unverändert.



Die Abb. 15.4-1 zeigt deutlich die Abhangigkeiten vom Paket org.apache.xmlrpc. ServerProxy im client.gui-Paket sowie aus dem server-Paket die Klasse ServerStart greifen jeweils auf org.apache.xmlrpc-Klassen zu. Kundenfenster greift auf ServerProxy zu und dieser auf die Klasse AnfragenBehandler. Die Kommunikation zwischen ServerProxy und AnfragenBehandler lauft uber XML-RPC-Aufrufe, die uber HTTP ubertragen werden. Der ServerProxy ist somit der Stellvertreter fur den AnfragenBehandler auf dem Client.

Abb. 15.4-1: OOD-Klassendiagramm der Fallstudie »KV – XML-RPC«.

I 15 Arten der Netzkommunikation

Die Klasse AnfragenBehandler definiert im Fachkonzept eine Art »Fassade« zu der Klasse KundeContainer Einfach (siehe auch »Das Fassaden-Muster (*facade pattern*)«, S. 69). Sie definiert alle Methoden, die für den Client zugänglich sind.

Dabei akzeptieren alle ihre Methoden nur Parameter, die auch in XML-RPC zulässig sind (String, Integer, Boolean usw.). Im Folgenden ist die Klasse AnfragenBehandler aufgeführt:

Java

```
public class AnfragenBehandler
{
    public boolean einfuegeKunde(String name, int nummer)
    {
        KundenContainerEinfach.getObjektreferenz().
            einfuegeKunde(name, nummer);
        return true;
    }

    public int getNaechsteKundenNr()
    {
        return KundenContainerEinfach.
            getObjektreferenz().getNaechsteKundenNr();
    }

    public String getKundeZuNr(int nummer)
    {
        return KundenContainerEinfach.
            getObjektreferenz().getKundeZuNr(nummer);
    }

    //Methoden muessen etwas zurückgeben,
    //RPC-Methode darf nicht void sein
    public boolean endeAnwendung()
    {
        KundenContainerEinfach.getObjektreferenz().endeAnwendung();
        return true;
    }
}
```

Für veröffentlichte RPC-Methoden (wie `getNaechsteKundenNr()`) ist Folgendes zu beachten:

- Sie dürfen keine Klassenmethoden (nicht `static`) sein.
- Sie müssen einen Rückgabewert liefern, d. h. sie dürfen nicht `void` als Rückgabewert besitzen.
- Wenn die Apache-spezifischen Erweiterungen ausgeschaltet sind (`setEnabledForExtensions()`, siehe unten), dann darf der Wert `null` nicht zurückgeliefert werden.

**Server
Java**

Als nächstes muss ein XML-RPC-Server erstellt werden, über den die Klasse AnfragenBehandler angesprochen werden kann:

```

class ServerStart
{
    private static final int port = 18080;
    public static void main(String[] args)
        throws XmlRpcException, IOException
    {
        WebServer webServer = new WebServer(port);

        XmlRpcServer xmlRpcServer = webServer.getXmlRpcServer();

        PropertyHandlerMapping phm = new PropertyHandlerMapping();
        phm.addHandler("AnfragenBehandler",
            server.application.AnfragenBehandler.class);
        xmlRpcServer.setHandlerMapping(phm);

        XmlRpcServerConfigImpl serverConfig =
            (XmlRpcServerConfigImpl) xmlRpcServer.getConfig();
        serverConfig.setEnabledForExtensions(false);
        serverConfig.setContentLengthOptional(false);

        webServer.start();

        System.out.println("Der Kundenverwaltungs-Server(rpc) läuft," +
            " zum Stoppen bitte Enter drücken.");
        System.in.read();
        webServer.shutdown();
    }
}

```

Es wird ein WebServer erzeugt, der auf dem angegebenen Port »horchen« soll. Dieser liefert durch `getXmlRpcServer()` den dazu gehörigen `XmlRpcServer`. Durch ein `PropertyHandlerMapping`-Objekt lässt sich eine *Handler*-Zuordnung definieren: Zu einem Namen (Schlüssel) wird die entsprechende *Handler*-Klasse (Klasse, die Anfragen bearbeitet) angegeben. Dies erfolgt durch die Methode `addHandler()` und anschließend durch Setzen des *Handler Mappings* für den `XmlRpcServer`. Den *Handler*-Namen (hier `AnfragenBehandler`) benutzt der Client bei seiner Anfrage (siehe unten). Für den Server wird eine Konfiguration eingestellt. Hierbei werden die Apache-spezifischen Erweiterungen zu XML-RPC ausgeschaltet (`setEnabledForExtensions()`) und die Angabe gemacht, dass im *Header* der Anfrage die Größe des Inhalts mit angegeben werden muss, wie es auch in der XML-RPC-Spezifikation gefordert wird. Dann wird der Webserver gestartet, und der Server ist bereit für Anfragen.

Die Klasse `ServerProxy`, die auf Clientseite den Aufbau zum Server durchführt, sieht wie folgt aus:

```

public class ServerProxy implements KundenContainerEinfachI
{
    //Singleton-Muster
    private static ServerProxy serverProxy = null;

```

I 15 Arten der Netzkommunikation

```
private XmlRpcClient xmlRpcClient;
public ServerProxy()
{
    XmlRpcClientConfigImpl config =
        new XmlRpcClientConfigImpl();
    try
    {
        config.setServerURL(new URL
            ("http://127.0.0.1:18080/xmlrpc"));
    }
    catch (MalformedURLException e)
    {
        e.printStackTrace();
    }
    config.setEnabledForExtensions(false);
    config.setConnectionTimeout(60 * 1000);
    config.setReplyTimeout(60 * 1000);

    xmlRpcClient = new XmlRpcClient();

    xmlRpcClient.setTransportFactory(
        new XmlRpcCommonsTransportFactory(xmlRpcClient));
    xmlRpcClient.setConfig(config);
}

public static ServerProxy getObjektreferenz()
{
    if (serverProxy == null)
        serverProxy = new ServerProxy();
    return serverProxy;
}

public void einfuegeKunde(String name, int nummer)
{
    int nr = Integer.valueOf(nummer);
    Object[] params = new Object[]{ name, nr };
    try
    {
        xmlRpcClient.execute
            ("AnfragenBehandler.einfuegeKunde", params);
    }
    catch (XmlRpcException e)
    {
    }
    try
    {
        throw new Exception();
    }
    catch (Exception e1)
    {
        e1.printStackTrace();
    }
}
}

{...}
```

Die Angabe der Server-URL inklusive des Ports 18080 (der im Server oben definiert ist), die Angabe, ob Erweiterungen des Apache XML-RPC ermöglicht sind (um konform zur XML-RPC-Spezifikation zu sein, ist dieser Parameter auf false zu setzen), und die Anfrage- und Antwort-*Timeouts* (Wartezeit, nach der die Anfrage mit einem Fehler abgebrochen wird) werden zuerst festgelegt. Auch andere Einstellungen sind möglich, hierzu können Sie sich die API ansehen: Apache XML-RPC-API (<http://ws.apache.org/xmlrpc/apidocs/index.html>). Dann wird das `XmlRpcClient`-Objekt erzeugt und die `XmlRpcTransportFactory` gesetzt. Die `XmlRpcTransportFactory` ist dafür zuständig, ein `XmlRpcTransport`-Objekt zu erzeugen. Das `XmlRpcTransport`-Objekt wird benötigt, die XML-RPC-*Requests* (Anfragen) zu versenden. Im Beispiel wird `XmlRpcCommonsTransportFactory` gesetzt, die eine HTTP-Transportfabrik ist, d.h. die Anfragen werden über HTTP gesendet. Mit der `execute()`-Methode wird die Anfrage gesendet. Der erste Parameter kennzeichnet die Methode, die aufgerufen werden soll (`<Handler-Name>.<Methodenname>`). Der zweite Parameter `params` enthält die Übergabe-Parameter.

Das Sequenzdiagramm der Abb. 15.4-2 zeigt den Programmablauf beim Speichern der Daten eines Kunden. Das Szenario ist vereinfacht dargestellt. Der `XmlRpcServer` baut den *Handler* eigentlich nicht selbst, sondern eine Klasse, die die Schnittstelle `RequestProcessorFactory` implementiert.

Dynamik

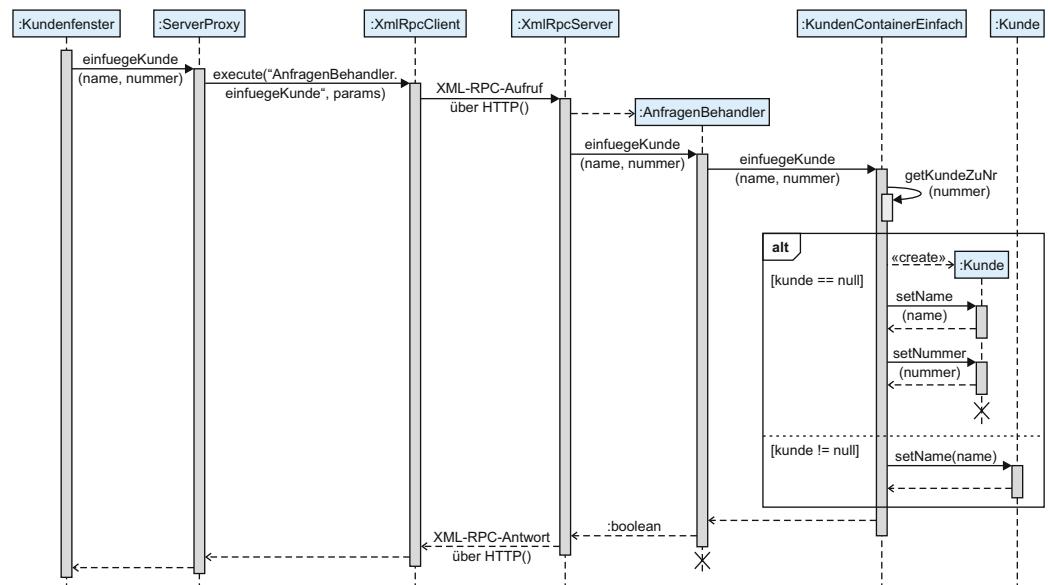


Abb. 15.4-2:
Sequenzdiagramm
zur Fallstudie
»KV – XML-RPC«.

I 15 Arten der Netzkommunikation

Die GUI-Klasse Kundenfenster ruft die `einfuegeKunde()`-Methode von `ServerProxy` mit den Parametern `name` und `nummer` des Kunden auf. `ServerProxy`, der einen `XmlRpcClient` besitzt, ruft auf diesem die `execute()`-Methode auf. Mit `execute()` wird eine XML-RPC-Anfrage durchgeführt. Der erste Parameter gibt die Methode auf einem *Handler* an, die aufgerufen werden soll (`AnfragenBehandler.einfuegeKunde()`), im zweiten Parameter sind die Parameter für diese Methode verpackt.

`XmlRpcClient` sendet diese Anfrage als XML-RPC-Aufruf über HTTP. `XmlRpcServer` auf der anderen Seite empfängt diese Anfrage, erzeugt ein Objekt vom angegebenen *Handler Type* (`AnfragenBehandler`) und ruft auf diesem die angegebene Methode auf (`einfuegeKunde()`). Für jede Anfrage wird das entsprechende *Handler*-Objekt also neu erzeugt.

Nach der Abarbeitung liefert die `einfuegeKunde()`-Methode der Klasse `AnfragenBehandler` einen boolschen Wert zurück, der angibt, ob das Speichern erfolgreich war. Diesen Wert liefert der `XmlRpcServer` als XML-RPC-Antwort an den Client zurück. Tritt während der Bearbeitung ein Fehler auf, dann wird eine `XmlRcpException` geworfen, die in der Klasse `ServerProxy` abgefangen wird. Diese wiederum wirft eine Exception, die in der Klasse `Kundenfenster` abgefangen wird.

OOP Die Programme zu dieser Fallstudie können Sie im E-Learning-Kurs herunterladen.

 Gehen Sie das Programm Schritt für Schritt durch, um die Kommunikation zwischen Client und Server zu verstehen. Laden Sie das Programm herunter und lassen Sie es auf Ihrem Computersystem ablaufen.

15.5 SOAP

SOAP ist ein vom W3C (<http://www.W3C.com>) standardisiertes Netzwerkprotokoll. Ursprünglich stand SOAP für *Simple Object Access Protocol* – heute wird SOAP jedoch als eigenständiger Begriff verwendet. SOAP ist der Nachfolger von XML-RPC (siehe »XML-RPC«, S. 253).

SOAP unterstützt die Übertragung von Daten und erlaubt *Remote Procedure Calls* (RPC). Als Nachrichtenformat wird **XML** benutzt.

SOAP kann mit verschiedenen Transportprotokollen verwendet werden. In der Regel wird HTTP oder HTTPS gewählt. Es kann aber auch JMS (*Java Message Service*), FTP oder SMTP/POP3 verwendet werden.

Arbeitsweise SOAP als Protokoll erlaubt es, beliebige anwendungsspezifische Informationen über ein Netzwerk zu übertragen. In SOAP wird festgelegt, wie die Informationen als Nachricht codiert werden müssen

und wie sie auszuwerten sind. Für entfernte Prozeduraufrufe gibt es eine Spezifikation. Zur Semantik anwendungsspezifischer Daten werden keine Aussagen gemacht. Wird das HTTP(S)-Protokoll verwendet, dann werden die XML-Informationen im Rumpf eines HTTP-Post-Requests an eine gegebene URL geschickt.

Ein zu übertragendes XML-Dokument wird beim Sender zunächst aufgebaut und anschließend evaluiert.

Sollen externe Systeme nicht direkt auf ein Anwendungssystem zugreifen, z. B. aus Sicherheitsgründen oder wegen Kompatibilitätsproblemen, dann kann über die SOAP-Schnittstelle die Anzahl der ausführbaren Methoden eingeschränkt und spezifiziert werden.

SOAP-Nachrichten haben einen festgelegten Aufbau. SOAP wird in der Regel zusammen mit Webservices eingesetzt, wobei die vom Webservice angebotenen Methoden mit WSDL (*Web Service Description Language*) beschrieben werden:

- »SOAP-Nachrichten, Webservices und WSDL«, S. 263

Webservices können in Java einfach implementiert werden:

- »Webservices bereitstellen und nutzen mit JAX-WS«, S. 267

Damit Webservices gefunden werden können, gibt es einen Namensdienst, bei dem Anbieter Webservices registrieren und Clients Webservices finden können:

- »UDDI«, S. 280

Für die Fallstudie »Kundenverwaltung-Mini« kann auch SOAP eingesetzt werden:

- »Fallstudie: KV – SOAP«, S. 283

Anwendungsbereiche



Vor- und Nachteile

- | | |
|---|---------------------------|
| <ul style="list-style-type: none"> + SOAP ermöglicht eine schwache Kopplung von Systemen. + SOAP ist ein flexibles Konzept. - Zum Aufbau und zur Validierung des XML-Dokuments ist Rechenaufwand nötig. - Notwendige mit zu übertragende Metadaten erhöhen das Übertragungsvolumen. | Vorteile Nachteile |
|---|---------------------------|

15.5.1 SOAP-Nachrichten, Webservices und WSDL

Aufbau von SOAP-Nachrichten

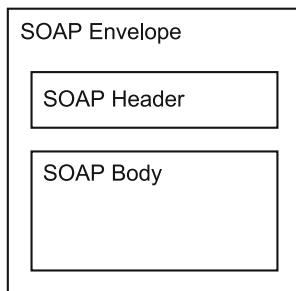
Vereinfacht betrachtet besteht eine SOAP-Nachricht aus den folgenden drei Elementen (Abb. 15.5-1):¹

- 1 Umschlag (SOAP Envelope)**
- 2 Kopf (SOAP Header)**
- 3 Rumpf (SOAP Body)**

¹Teile dieses Textes wurden mit freundlicher Genehmigung des W3L-Verlags aus [Weße06, S. 14 ff.] übernommen.

I 15 Arten der Netzkommunikation

Abb. 15.5-1:
Aufbau einer
SOAP-Nachricht.



Der Umschlag (*Envelope*) gilt als Rahmen für eine SOAP-Nachricht. Innerhalb des Umschlags sind der optionale Kopf (*Header*) und der Rumpf (*Body*) als Elemente enthalten. Der Kopf kann beispielsweise Informationen für die Authentifizierung enthalten. Dadurch wird die Abhängigkeit von HTTP als Transportprotokoll eliminiert (siehe [HeZe03]). Die Nutzlast einer SOAP-Nachricht befindet sich im Rumpf, wobei diese eine der nachfolgenden Ausprägungen hat:

- Anfrage (*Request*)
- Antwort (*Response*)
- Fehlermeldung (*Fault*)

Das Grundgerüst einer SOAP-Nachricht sieht wie folgt aus:

SOAP-
Grundgerüst

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv=
    "http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    Information des Headers
  </soapenv:Header>
  <soapenv:Body>
    Inhalt der SOAP-Nachricht
  </soapenv:Body>
</soapenv:Envelope>
```

Mit SOAP können aber nicht nur textuelle Informationen, sondern auch Binärdaten übertragen werden. Diese Binärdaten werden als SOAP-Attachments bezeichnet.

Webservice

SOAP wird in der Regel in Verbindung mit Webservices eingesetzt.

Definition

»A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner

prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.» [W3C04].

Ein Webservice stellt also eine Dienstleistung auf einem Server zur Verfügung. Die Schnittstelle dieser Dienstleistung ist in einem maschinenlesbaren Format beschrieben (WSDL, siehe unten). Der Nachrichtenaustausch erfolgt über SOAP-Nachrichten.

WSDL

Damit ein Client weiß, *wie* er einen im Internet verfügbaren Webservice via SOAP aufrufen kann, wurde die XML-Beschreibungssprache **WSDL** (*Web Service Description Language*) entwickelt. Sie stellt eine standardisierte Methode zur Definition der von einem Service angebotenen Methoden und deren Parameter dar. WSDL-Dokumente können maschinell ausgewertet werden.

WSDL kapselt alle Informationen (Metadaten), die zum Aufruf eines entfernten Programms notwendig sind. Ein WSDL-Dokument kann mit einer IDL-Schnittstellenbeschreibung von CORBA verglichen werden (siehe »CORBA«, S. 241). Eine IDL-Schnittstellenbeschreibung wird allerdings *nicht* in XML erstellt. Zusätzlich beinhaltet eine WSDL-Beschreibung Informationen, *wo* das entfernte Programm im Netzwerk aufgerufen werden kann. Mit WSDL ist es dadurch möglich, eine automatische Integration von verschiedenen Anwendungen zu realisieren. Fordert ein Client ein WSDL-Dokument an, so kann er auf Basis dieser WSDL-Datei dynamisch Klassen (*Proxies*) erzeugen, mit denen er den Webservice aufruft. Ein geeignetes Szenario für Anwendungen dieser Art stellt die *Enterprise Application Integration* (EAI) dar, bei der verschiedene Anwendungen Informationen austauschen müssen.

Ein WSDL-Dokument wird mit XML beschrieben und beinhaltet folgende fünf Elemente, die in das Wurzelement `<definitions/>` eingebettet werden:

- `<types/>`
- `<message/>`
- `<portType/>`
- `<binding/>`
- `<service/>`

Eine WSDL-Datei wird logisch in einen abstrakten und einen konkreten Teil unterteilt [W3Ce04]. Zu dem abstrakten Teil gehören die XML-Elemente `<types/>`, `<message/>` und `<portType/>`. Im konkreten Teil des Dokuments befinden sich die beiden Elemente `<binding/>` und `<service/>`. Der abstrakte Teil des WSDL-Dokuments beschreibt die Methoden eines Webservice und deren benötigte Datentypen, un-

I 15 Arten der Netzkommunikation

abhängig von einem Transportmechanismus und einer Implementierung. Der konkrete Teil bestimmt, über welches Protokoll und unter welcher Adresse der Dienst erreichbar ist. Das Grundgerüst eines WSDL-Dokuments sieht folgendermaßen aus:

WSDL-
Grundgerüst

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <wsdl:types>
        XML-Schema für die Datentypdefinition
    </wsdl:types>
    <wsdl:message>
        Nachrichten des Webservice
    </wsdl:message>
    <wsdl:portType>
        Methoden des Webservice
    </wsdl:portType>
    <wsdl:binding>
        Protokolle des Webservice
    </wsdl:binding>
    <wsdl:service>
        Adresse des Webservice
    </wsdl:service>
</wsdl:definitions>
```

Zusätzlich kann mit dem Element `<documentation/>` die XML-Datei kommentiert werden. Eine Modularisierung der Datei kann durch das XML-Element `<import/>` bewerkstelligt werden.

Datentyp-
definitionen

Innerhalb des optionalen `<types/>`-Elements werden mit XML-Schemata die von dem Webservice verwendeten Datentypen beschrieben. Allerdings werden lediglich komplexe Datentypen beschrieben, da einfache Datentypen, wie beispielsweise `string` oder `integer`, bereits durch XML-Schemata selbst beschrieben sind. Komplexe Datentypen können mit dem aus der Programmierung bekannten Konzept der strukturierten Datentypen verglichen werden. Die Struktur der komplexen Datentypen wird vollständig in XML beschrieben. Werden lediglich einfache Datentypen verwendet, so wird das `<types/>`-Element in der WSDL-Datei *nicht* benutzt.

Nachrichten &
Methoden

Jeder Webservice bietet mindestens eine Methode an, die ein Client nutzen kann. Zu einer Methode gehören Nachrichten, die zwischen einem Client und einem Server beim Aufruf einer Methode ausgetauscht werden. Jede Nachricht wird innerhalb eines XML-Elements `<message/>` beschrieben. Bei jeder Nachricht wird ebenfalls angegeben, welchen Rückgabewert bzw. welche Parameter sie hat, je nach dem, ob es sich dabei um eine Anfrage (*Request*) oder eine Antwort (*Response*) handelt. Im XML-Element `<portType/>` wird eine Methode eines Webservices beschrieben. Hier werden die einzelnen Nachrichten (`<message/>`) einer Methode zugeordnet. Für jede Methode können maximal zwei Nachrichten ausgetauscht werden.

Ein Beispiel für zwei Nachrichten, die einer Methode zugeordnet Beispiel sind, zeigt der folgende Ausschnitt:

```
<!-- Nachricht für die Anfrage -->
<wsdl:message name="lieferFlugdatenRequest">
  <wsdl:part name="von" type="xsd:string"/>
  <wsdl:part name="nach" type="xsd:string"/>
</wsdl:message>
<!-- Nachricht für das Ergebnis -->
<wsdl:message name="lieferFlugdatenResponse">
  <wsdl:part name="ergebnis" type="xsd:string"/>
</wsdl:message>
<!--Die Methode, bestehend aus zwei Nachrichten-->
<wsdl:portType name="FlugdatenService">
  <wsdl:operation name="lieferFlugdaten"
    parameterOrder="von,nach">
    <wsdl:input
      message="lieferFlugdatenRequest"/>
    <wsdl:output
      message="lieferFlugdatenResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

Innerhalb des `<binding/>`-Elements wird angegeben, an welche Protokolle Protokolle der Webservice gebunden wird. In der Regel ist das die Kombination SOAP/HTTP.

An welcher Stelle ein Client die Implementierung der als Webser- Adresse vices angebotenen Funktionalität im Internet auffinden kann, wird innerhalb des XML-Elements `<service/>` definiert.

15.5.2 Webservices bereitstellen und nutzen mit JAX-WS

Eine Java-API zur Erstellung von Webservices ist **JAX-WS** (*Java API for XML – Web Services*). Sie benutzt Annotationen, um die Entwicklung und das *Deployment* von Webservice-Clients und Service-Endpunkten (*service endpoint*) zu vereinfachen. Ein Service-Endpunkt gibt den Endpunkt für einen Service wieder, der es Clients ermöglicht, den Dienst zu finden und mit ihm zu kommunizieren. Client und Endpunkt kommunizieren dabei über SOAP-Nachrichten. Um SOAP-Nachrichten zu erzeugen, wird SAAJ (*SOAP with Attachments API for Java*) verwendet. Die Umwandlung von XML-Datentypen in Java-konforme Datentypen wird von JAX-WS an JAXB (*Java Architecture for XML Binding*) delegiert.

Server

Soll in Java eine Klasse einen Webservice zur Verfügung stellen, dann muss die Klasse mit folgender Annotation gekennzeichnet werden:

```
@WebService(name="Name des Webservice", serviceName =
"Servicename des Webservice")
```

I 15 Arten der Netzkommunikation

Dadurch wird diese Klasse als ein Webservice-Endpunkt definiert.

Die Klasse muss folgende Eigenschaften besitzen:

- Sie darf nicht `abstract` sein und nicht `final` deklariert sein.
- Sie muss einen öffentlichen `default`-Konstruktur besitzen.

`@SOAPBinding` Die folgende Annotation muss ebenfalls vor der Klasse angegeben werden:

`@SOAPBinding(style= Angabe, wie ein WSDL-Binding in eine SOAP-Nachricht übersetzt werden soll)`

Wenn `style = SOAPBinding.Style.RPC` angegeben ist, dann sollen die Konventionen für *Remote Procedure Calls* verwendet werden. Die Aufrufe enthalten Parameter und es werden Werte zurückgegeben.

Wenn `style = SOAPBinding.Style.DOCUMENT` angegeben ist, dann soll der Client XML-Schemata verwenden.

`@WebMethod` Vor jede Methode, die öffentlich angeboten werden soll, ist folgende Annotation zu schreiben:

`@WebMethod(operationName="Name der Methode, wenn abweichend vom gewählten Methodennamen")`

Jede öffentliche angebotene Methode muss folgende Eigenschaften besitzen:

- Sie muss `public` sein und darf *nicht* als `static` oder `final` deklariert sein.

Beispiel 1a Ein Server soll Staumeldungen auf Autobahnen als Webservice zur Verfügung stellen. Clients können diese Staumeldungen abfragen. Auf der Serverseite wird die hier »vereinfachte« Klasse `Staumelder` programmiert:

Java package stau.server;

Staumelder import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService(name="StaumelderDienst")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class Staumelder
{
 @WebMethod
 public boolean istStau(String autobahn)
 {
 if (autobahn.equals("A40") || autobahn.equals("A42"))
 return true;
 return false;
 }
}

Der Webservice wird unter dem Namen `StaumelderDienst` veröffentlicht. Das `SOAPBinding` wird auf `RPC` gesetzt, d.h. dass die Kommunikation prozedurorientiert stattfindet. Die zu veröffentlichte Methode `istStau()` wird mit der Annotation `@WebMethod` gekennzeichnet.

15.5 SOAP I

Um einen Webservice zu veröffentlichen, wird die Klassenmethode `publish()` der Klasse `Endpoint` verwendet:

```
public static Endpoint publish(java.lang.String address,  
    java.lang.Object implementor)
```

wobei `address` durch eine URI die Adresse und das Transportprotokoll angibt, das benutzt werden soll. `implementor` gibt an, für welches Objekt der Endpunkt erzeugt werden soll.

Für die Klasse `Staumelder` wird ein Endpunkt erzeugt:

Beispiel 1b

```
package stau.server;
```

Java

```
import javax.xml.ws.Endpoint;  
  
public class StauInfoServer  
{  
    public static void main(String[] args) throws Exception  
    {  
        Endpoint.publish("http://localhost:18080/staumelder",  
            new Staumelder());  
    }  
}
```

Der erste Parameter ist die Adresse, unter der dieser Webservice veröffentlicht wird. Der zweite Parameter gibt das Objekt an, das diesen Webservice implementiert.

Nach dem Start von `StauInfoServer` kann im Browser die Adresse `http://localhost:18080/staumelder?wsdl` aufgerufen werden. Dies ist die Adresse, unter der der Dienst veröffentlicht ist, ergänzt um den Zusatz `?wsdl`. Im Browser können Sie sich nun die Beschreibung des Webservices als XML-Dokument anzeigen lassen:

```
<?xml version="1.0" encoding="UTF-8" ?>  
-  
- <definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  
  xmlns:tns="http://server.stau/"  
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
  xmlns="http://schemas.xmlsoap.org/wsdl/"  
  targetNamespace="http://server.stau/" name="StaumelderService">  
<types />  
<message name="istStau">  
  <part name="arg0" type="xsd:string" />  
</message>  
<message name="istStauResponse">  
  <part name="return" type="xsd:boolean" />  
</message>  
<portType name="StaumelderDienst">  
  <operation name="istStau" parameterOrder="arg0">  
    <input message="tns:istStau" />  
    <output message="tns:istStauResponse" />  
  </operation>  
</portType>  
<binding name="StaumelderDienstPortBinding"
```

I 15 Arten der Netzkommunikation

```
type="tns:StaumelderDienst">
<soap:binding transport=
    "http://schemas.xmlsoap.org/soap/http" style="rpc" />
<operation name="istStau">
    <soap:operation soapAction="" />
    <input>
        <soap:body use="literal" namespace="http://server.stau/" />
    </input>
    <output>
        <soap:body use="literal" namespace="http://server.stau/" />
    </output>
</operation>
</binding>
<service name="StaumelderService">
    <port name="StaumelderDienstPort"
        binding="tns:StaumelderDienstPortBinding">
        <soap:address location="http://localhost:18080/staumelder" />
    </port>
</service>
</definitions>
```

Für die nachfolgenden Schritte der Programmierung sollte der Staumelder-InfoServer weiterhin in Betrieb sein.

Client

Damit ein Client die Dienste des Webservers bequem in Anspruch nehmen kann, ist es am einfachsten, wenn der Client Aufrufe an einen lokalen Stummel (*stub*) senden kann. Mithilfe von Generatoren können aus einer gegebenen WSDL-Datei Hilfsklassen für den bequemen Zugriff generiert werden. Mit jedem Java-JDK wird das Werkzeug `wsimport` ausgeliefert.

`wsimport` Über die Konsole wird `wsimport` gestartet. Es wird in das `src`-Verzeichnis des Projekts gewechselt und folgendes Kommando eingegeben:

```
wsimport -keep AdresseDesWebservice?wsdl
```

Mit dem Schalter `-keep` wird angegeben, dass die Quellen generiert werden sollen. Es wird dann noch die Adresse angegeben, von der die Beschreibung des Webservice bezogen werden soll.

Beispiel 1c Im nächsten Schritt werden aus der Webservice-Beschreibung die Service-Klassen für den Client generiert:

```
wsimport -keep http://localhost:18080/staumelder?wsdl
```

Wird gemeldet, dass das Kommando `wsimport` nicht gefunden werden kann, dann muss der komplette Pfad zum `wsimport`-Kommando angegeben werden:

```
C:\java\jdk1.6\bin\wsimport -keep http://localhost:18080/
staumelder?wsdl
```

Hierbei wird C:\java\jdk1.6 als Java-Installationsverzeichnis und Windows als Betriebssystem angenommen.

Es werden für den Client die Klasse StaumelderService und die Schnittstelle StaumelderDienst erzeugt. Der Name StaumelderDienst entspricht dem Webservice-Namen der Klasse Staumelder (siehe oben). Diese Schnittstelle bietet alle Methoden an, die in Staumelder mit @WebMethod gekennzeichnet sind.

```
package stau.server;                                         Java
import javax.jws.WebMethod;                                Staumelder
import javax.jws.WebParam;                                 Dienst
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

/*
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.1.6 in JDK 6
 * Generated source version: 2.1
 */
@WebService(name = "StaumelderDienst",
    targetNamespace = "http://server.stau/")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface StaumelderDienst
{
    @WebMethod
    @WebResult(partName = "return")
    public boolean istStau(
        @WebParam(name = "arg0", partName = "arg0")
        String arg0);
}
```

StaumelderService ist eine Hilfsklasse, die von der Klasse Service erbt. Sie generiert eine Stummel-Klasse mithilfe der von Service geerbten Methoden:

```
package stau.server;                                         Java
import java.net.MalformedURLException;                         Staumelder
import java.net.URL;                                         Service
import java.util.logging.Logger;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceFeature;

/*
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.1.6 in JDK 6
 * Generated source version: 2.1
 *
 */
```

I 15 Arten der Netzkommunikation

```
@WebServiceClient(name = "StaumelderService",
    targetNamespace = "http://server.stau/",
    wsdlLocation = "http://localhost:18080/staumelder?wsdl")
public class StaumelderService
    extends Service
{
    private final static URL STAUMELDERSERVICE_WSDL_LOCATION;
    private final static Logger logger =
        Logger.getLogger(
            staumelderService.class.getName());

    static
    {
        URL url = null;
        try
        {
            URL baseUrl;
            baseUrl =
                staumelderService.class.getResource(".");
            url = new URL(baseUrl,
                "http://localhost:18080/staumelder?wsdl");
        }
        catch (MalformedURLException e)
        {
            logger.warning(
                "Failed to create URL for the wsdl Location:"+
                "'http://localhost:18080/staumelder?wsdl','"+
                "retrying as a local file");
            logger.warning(e.getMessage());
        }
        STAUMELDERSERVICE_WSDL_LOCATION = url;
    }

    public StaumelderService(URL wsdlLocation, QName serviceName)
    {
        super(wsdlLocation, serviceName);
    }

    public StaumelderService()
    {
        super(STAUMELDERSERVICE_WSDL_LOCATION,
            new QName("http://server.stau/", "StaumelderService"));
    }

    /**
     *
     * @return
     *     returns StaumelderDienst
     */
    @WebEndpoint(name = "StaumelderDienstPort")
    public StaumelderDienst getStaumelderDienstPort()
    {
        return super.getPort(new QName("http://server.stau/",
            "StaumelderDienstPort"), StaumelderDienst.class);
    }
```

```
/**
 *
 * @param features
 *      A list of {@link javax.xml.ws.WebServiceFeature}
 *      to configure on the proxy. Supported features not in
 *      the <code>features</code> parameter will have
 *      their default values.
 * @return
 *      returns StaumelderDienst
 */
@WebEndpoint(name = "StaumelderDienstPort")
public StaumelderDienst getStaumelderDienstPort
    (WebServiceFeature... features)
{
    return super.getPort(
        new QName("http://server.stau/", "StaumelderDienstPort"),
        StaumelderDienst.class, features);
}
```

Die Klasse `StauInfoClient` importiert die Schnittstelle `Java StaumelderDienst` und die Klasse `StaumelderService`:
`StauInfoClient`

```
package stau.client;

import stau.server.StaumelderDienst;
import stau.server.StaumelderService;

public class StauInfoClient
{
    public static void main(String[] args) throws Exception
    {
        StaumelderDienst stauService =
            new StaumelderService();

        String autobahn = "A42";
        boolean result = stauService.istStau(autobahn);
        System.out.println("Auf der Autobahn " + autobahn
            + " ist " +
            (result == false ? "kein " : "leider ") + "Stau!");
    }
}
```

Wie das Beispiel zeigt, konstruiert der Client ein Service-Objekt mit dem `new`-Operator und ruft die `getServicePort()`-Methode auf, um einen Stummel (*stub*) zu bekommen. Die Methodenaufrufe werden auf dem Stummel-Objekt ausgeführt, als wäre der Webservice lokal verfügbar.

Die JAX-WS-Laufzeitumgebung schickt die Aufrufe von dem Stummel-Objekt an den Webservice weiter, der dann die Methoden durchführt und das Ergebnis an den Client zurückschickt.

I 15 Arten der Netzkommunikation

Installation Die mithilfe von wsimport erzeugten Dateien müssen zusammen mit der Client-Klasse in eine jar-Datei gepackt und auf dem Client installiert werden. Der Server benötigt diese Dateien nicht.

Beispiel 1d Die Ausführung des Clients ergibt folgendes Ergebnis:
Auf der Autobahn A42 ist leider Stau!
Die SOAP-Nachrichten, die in diesem Beispiel zwischen Client und Server ausgetauscht werden, sehen wie folgt aus:

Der Client sendet folgende Anfrage:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:istStau xmlns:ns2="http://server.stau/">
      <arg0>A42</arg0> </ns2:istStau>
    </S:Body>
  </S:Envelope>
```

Der Server antwortet:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:istStauResponse xmlns:ns2="http://server.stau/">
      <return>true</return>
    </ns2:istStauResponse>
  </S:Body>
</S:Envelope>
```

Im kostenlosen E-Learning-Kurs zu diesem Buch können Sie das Programm herunterladen.

 Installieren Sie die benötigte Software auf Ihrem Computersystem und führen Sie das obige Programm aus.

15.5.3 Webservices nutzen

Webservices kann man heute mithilfe von Suchmaschinen, dem Stichwort Webservice sowie einem Themengebiet, zu dem man Webservices sucht, relativ leicht finden.

Beispiel 1a: Mit den Stichworten `webservice` und `stockquote` findet man z.B. Börsenkurse folgende Website, um Börsenkurse abzufragen: `webservicex.net` (`http://www.webservicex.net/WCF/ServiceDetails.aspx?SID=19`). Die Abb. 15.5-2 zeigt einen Ausschnitt aus der Website.

Eclipse Mithilfe der Entwicklungsumgebung Eclipse (Java EE-Version für Java-EE-Entwickler) ist es einfach möglich, zu einem angebotenen Webservice einen Client zu erstellen, der es erlaubt, diesen Webservice abzufragen.

Anhand eines Beispiels wird gezeigt, welche Schritte durchzuführen sind.

15.5 SOAP I

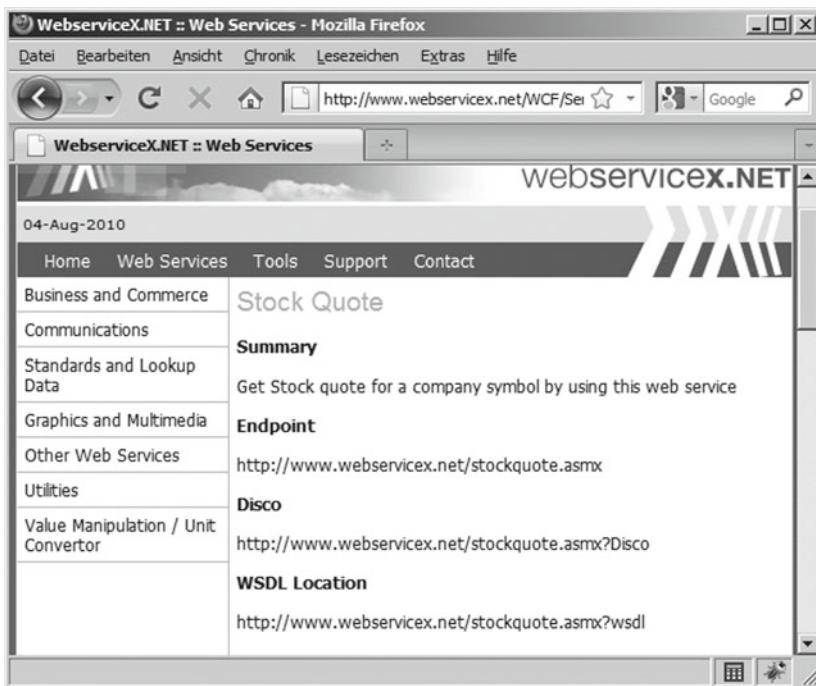


Abb. 15.5-2:
Website
webservicex.net.

Schritt 1: Legen Sie zunächst ein Java-Projekt an. Als Projektnamen können Sie beispielsweise ClientBoerse wählen. In diesem Projekt befindet sich dann der Webservice-Client.

Beispiel 1b

Schritt 2: Klicken Sie wieder auf Control-N. Wählen Sie nun im Ordner Web Services den Eintrag Web Service Client aus, wie es in Abb. 15.5-3 dargestellt ist.

Schritt 3: Klicken Sie auf Next>. In dieses Fenster (Abb. 15.5-4) geben Sie für Webservice-Definition die URL des angebotenen Service ein (WSDL Location), hier: <http://www.webservicex.net/stockquote.asmx?wsdl>

Stellen Sie den Schalter links auf die erste Stufe (Develop Client) ein, wie es die Abb. 15.5-4 zeigt. Außerdem sollte die *Web service runtime* auf *Apache Axis* eingestellt sein.

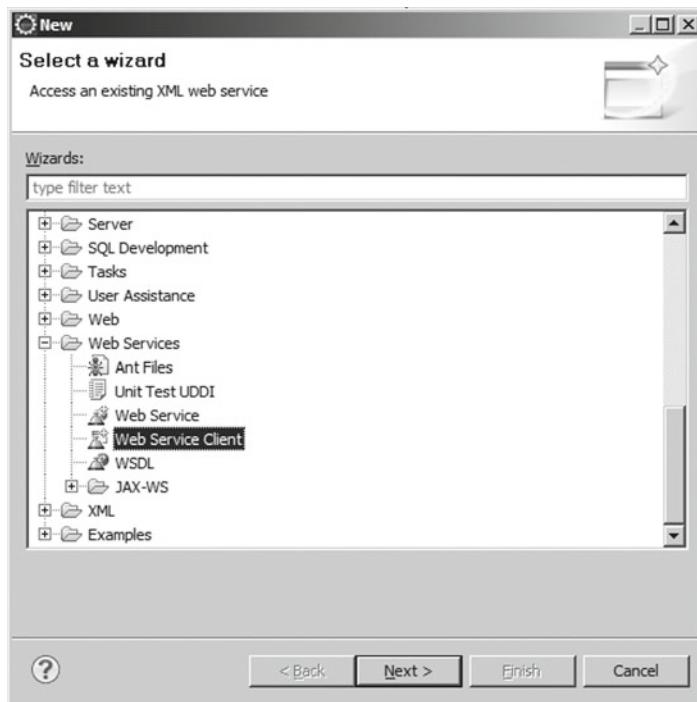
Schritt 4: Klicken Sie rechts auf Client Project:. Wählen Sie als Client Project Ihr gerade angelegtes Projekt aus (Abb. 15.5-5).

Schritt 5: Klicken Sie auf OK, dann auf Finish. Wenn Sie Ihr Projekt öffnen, sehen Sie, dass die Proxy-Klassen erzeugt worden sind (Abb. 15.5-6).

Schritt 6: Fügen Sie dem Paket eine Klasse hinzu, die diese Proxies nutzt. Nennen Sie diese Klasse beispielsweise `MeinWebserviceClient`. Die beiden Methoden in der Klasse sehen folgendermaßen aus (die Erklärung folgt dahinter):

I 15 Arten der Netzkomunikation

Abb. 15.5-3:
Eclipse-Webservice-
Client: Dialog-
Schritt 1.



```
Java package NET.webserviceX.www;

import java.io.ByteArrayInputStream;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;

public class MeinWebserviceClient
{
    public static void main(String[] args)
    {
        try
        {
            String symbol = "^GDAXI";

            //Server fragen
            StockQuoteLocator sql = new StockQuoteLocator();
            StockQuoteSoap sqs = sql.getStockQuoteSoap();
            String antwort = sqs.getQuote(symbol);

            //Parseen der Antwort (Antwort ist im XML-Format)
        }
    }
}
```

15.5 SOAP I

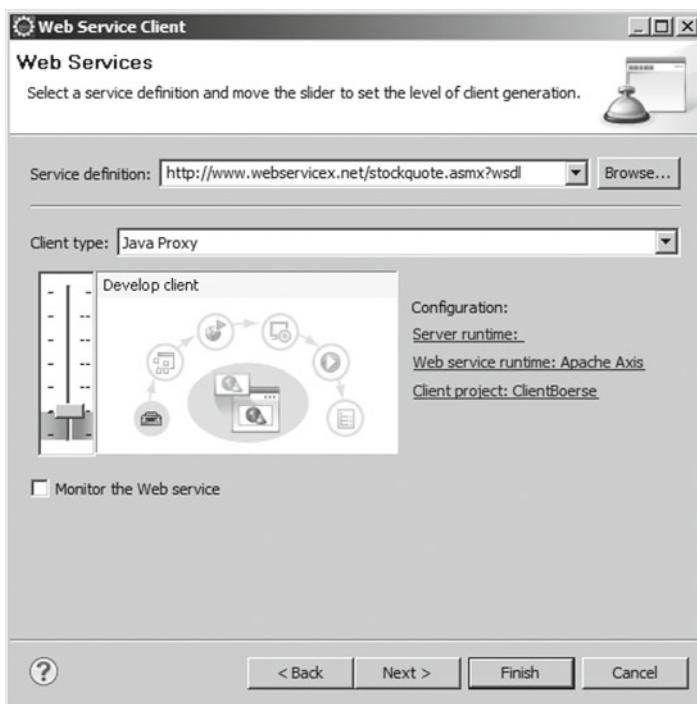


Abb. 15.5-4:
Eclipse-Webservice-
Client: Dialog-
Schritt 2.

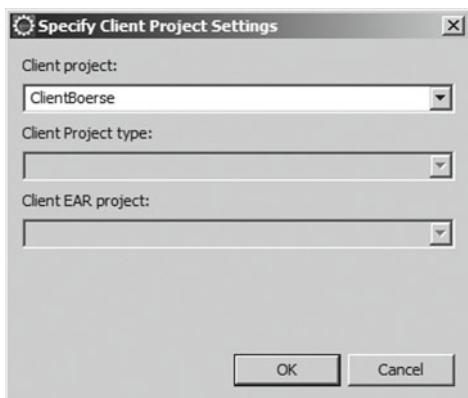
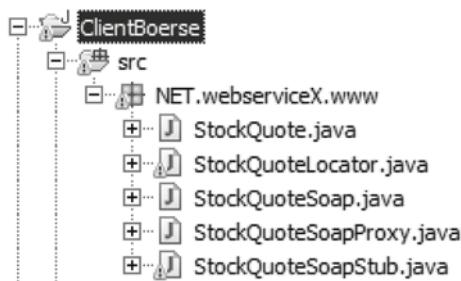


Abb. 15.5-5:
Eclipse-Webservice-
Client: Dialog-
Schritt 3.

```
DocumentBuilderFactory dbf =
    DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc =
    db.parse(new ByteArrayInputStream(antwort.getBytes()));
doc.getDocumentElement().normalize();
NodeList nodeLst = doc.getElementsByTagName( "Stock");
Element stockNode = (Element) nodeLst.item(0);
String letzterKurs = getNodeValue(stockNode, "Last");
String name = getNodeValue(stockNode, "Name");
//Es können hier noch weitere Attribute abgefragt werden
```

I 15 Arten der Netzkomunikation

Abb. 15.5-6:
Auszug aus dem
»Eclipse Project
Explorer«.



```
//Ausgabe
System.out.print("Aktueller Kurs von " + name
    + " ist: " + letzterKurs);
}
catch (Exception e)
{
    e.printStackTrace();
}

static String getNodeValue(Element node, String nodeName)
{
    NodeList fstNmElmntLst =
        node.getElementsByTagName(nodeName);
    Element fstNmElmnt = (Element) fstNmElmntLst.item(0);
    NodeList fstNm = fstNmElmnt.getChildNodes();
    return ((Node) fstNm.item(0)).getNodeValue();
}
```

Da dieser Webservice die Antwort nicht als einfachen Wert, sondern im XML-Format zurückliefert, muss die Antwort noch geparsst werden. Die Antwort des Servers sieht wie folgt aus:

```
<StockQuotes>
<Stock>
    <Symbol>^GDAXI</Symbol>
    <Last>6322.33</Last>
    <Date>8/4/2010</Date>
    <Time>9:24am</Time>
    <Change>+14.42</Change>
    <Open>6287.65</Open>
    <High>6331.84</High>
    <Low>6270.09</Low>
    <Volume>0</Volume>
    <MktCap>N/A</MktCap>
    <PreviousClose>6307.91</PreviousClose>
    <PercentageChange>+0.23%</PercentageChange>
    <AnnRange>5158.60 - 6341.52</AnnRange>
    <Earns>N/A</Earns>
    <P-E>N/A</P-E>
    <Name>DAX</Name>
```

```
</Stock>
</StockQuotes>
```

Es wird der DOM-Parser aus der Standardbibliothek benutzt, um diesen Text zu parsen. Hierfür sind folgende Import-Angaben notwendig:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
```

Der Parser wird so aufgebaut:

```
DocumentBuilderFactory dbf =
    DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
```

Für gewöhnlich werden Dateien geparsert, aber in diesem Fall gibt es nur ein String. Daher wird aus dem String ein `ByteArrayInputStream` gebildet, um ein `InputStream` zu erhalten. Dieser kann dann dem Parser zum Parsen übergeben werden:

```
Document doc =
    db.parse(new ByteArrayInputStream(antwort.getBytes()));
```

Das erhaltene DOM-Dokument kann dann inspiziert werden. Es wird nach dem Stock-Element gesucht und die nachfolgenden Abfragen beziehen sich nur noch auf dieses Element:

```
NodeList nodeLst = doc.getElementsByTagName( "Stock");
Element stockNode = (Element) nodeLst.item(0);
String letzterKurs = getNodeValue(stockNode, "Last");
...
```

`getNodeValue` ist eine Hilfsfunktion, die zu einem gegebenen Element (in diesem Fall ist es das Stock-Element) den Wert innerhalb eines Unterknotens angibt. Mit dieser Funktion können also Attribute der Aktie abgefragt werden. Welche Attribute außer `Name` und `Last` noch verfügbar sind, wird aus der XML-Antwort des Servers (siehe oben) ersichtlich.

Weitere Webservices finden Sie hier:

- Website WSDL (http://www.wsdl11.com/)
- Website Web Service Share (http://www.webserviceshare.com/)
- Website Membrane (http://www.service-repository.com/) (Bei diesen Webservices wird auch die Verfügbarkeit (in %) angegeben).

Tipp

Programmieren Sie selbst einen von Ihnen gewählten Webservice.



I 15 Arten der Netzkommunikation

15.5.4 UDDI

Will ein Serviceanbieter seine Dienstleistungen in Form von Webservices öffentlich anbieten, dann müssen potenzielle Servicenutzer die Möglichkeit erhalten, die angebotenen Webservices zu finden. Aus diesem Grund gibt es *Web Service Registries*, die Listen von Unternehmen und den von ihnen angebotenen Services enthalten.¹

UDDI Viele dieser *Registries* nutzen den Verzeichnisdienst **UDDI** (*Universal Description, Discovery and Integration*). Dieser Verzeichnisdienst bietet standardisierte Methoden, die es Unternehmen und Organisationen ermöglichen, sich und ihre Dienstleistungen zu registrieren. Potenzielle Nutzer können gezielt nach Services für eine bestimmte Aufgabenstellung zu suchen.

Zitat »Eine effiziente Suche nach bestimmten Informationen im Internet ist derzeit nur schwer möglich. Im Internet stehen zu den meisten Themen Informationen bereit, die über Hyperlinks systemübergreifend miteinander verbunden werden können. Jedoch haben die Erfinder des *World Wide Web* nicht daran gedacht, einen effizienten Mechanismus zum Auffinden von Informationen bereitzustellen. Als Folge werden heute Suchmaschinen benutzt, die verschiedenste Strategien anwenden, um Inhalte zu indizieren und somit durchaus unterschiedliche Ergebnisse liefern, mit denen wir in aller Regel nicht zufrieden sind. UDDI löst das Problem des Auffindens von Web Services« [HeZe03].

UDDI ist für die Entwicklung eines Webservice nicht zwingend notwendig. Ein *erstellter* Webservice kann bei einer UDDI-*Registry* angemeldet werden, damit Clients diesen Webservice finden können. Das Suchen und Auffinden von angebotenen Diensten ist der Schwerpunkt von UDDI.

Die Informationen, die UDDI liefert, können in folgende drei Kategorien eingeteilt werden:

- *White Pages*: Zu jedem Unternehmen, das Webservices in einer UDDI-*Registry* veröffentlicht hat, können verschiedene Kontaktinformationen ermittelt werden. Zu diesen Daten zählen beispielsweise Namen, Adressen und Steuernummern.
- *Yellow Pages*: Die veröffentlichten Dienste werden in verschiedene Kategorien eingeteilt, so wie es in einem Branchenbuch üblich ist.
- *Green Pages*: Technische Informationen eines Webservice, wie z. B. WSDL-Dokumente.

Datenmodell In der UDDI-Spezifikation werden fünf Datenstrukturen definiert, die die interne Struktur einer UDDI-*Registry* widerspiegeln. In der Abb. 15.5-7 – siehe auch UDDI.org (<http://www.uddi.org>) – sind die Datenstrukturen dargestellt.

¹Teile dieses Textes und Grafiken wurden mit freundlicher Genehmigung des W3L-Verlags aus [Weße06, S. 20 ff.] übernommen.

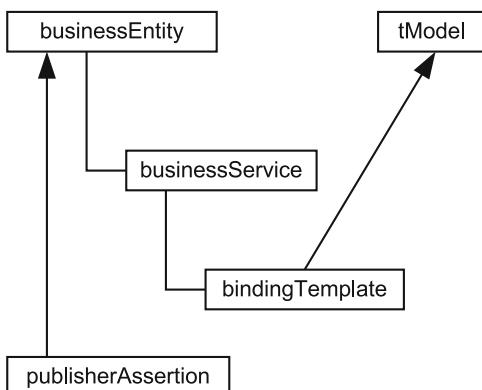


Abb. 15.5-7: Die interne Datenstruktur einer UDDI-Registry.

- **businessEntity:** Repräsentiert ein Unternehmen oder eine Organisation, die einen oder mehrere Dienste anbietet. Innerhalb dieser Datenstruktur sind Kontaktinformationen wie Telefonnummer oder Ansprechpartner gespeichert.
- **businessService:** Jeder Dienst, der von einer **businessEntity** angeboten wird, wird als **businessService** bezeichnet. Diese Dienste können neben *Web Services* auch reale Dienste, wie beispielsweise Schulungen, sein.
- **bindingTemplate:** Innerhalb dieser Datenstruktur werden technische Details eines Dienstes beschrieben. Dazu gehört unter anderem die Adresse für den Zugriff auf einen Dienst.
- **publisherAssertion:** Die Verbindungen zwischen den Firmen und Organisationen der **businessEntity**-Datenstruktur werden innerhalb der **publisherAssertion**-Struktur beschrieben. Eine mögliche Verbindung zweier Unternehmen wäre eine Kunden-Lieferanten-Beziehung.
- **tModel:** Diese Datenstruktur dient dazu, einen Dienst eindeutig zu identifizieren. Neben der eindeutigen ID besitzt ein **tModel** weitere Eigenschaften wie Name oder Beschreibung. Eine **tModel**-Datenstruktur wird von anderen Datenstrukturen der UDDI-*Registry* referenziert – sie hat jedoch keine eigenen Referenzen auf andere Datenstrukturen.

Für die Benutzung der oben beschriebenen Datenstrukturen werden durch das UDDI-API verschiedene Nachrichten definiert. Die Nachrichten werden bei der Kommunikation mit der UDDI-*Registry* verwendet.

Sie werden durch SOAP beschrieben. So existieren Nachrichten für das Ändern oder Löschen eines Dienstes sowie für das An- und Abmelden an einer UDDI-*Registry*.

Details sind auf der Website der OASIS-Groupe nachzulesen, die die UDDI-Spezifikationen erstellt, siehe UDDI.org (<http://www.uddi.org>).

I 15 Arten der Netzkomunikation

Zusammenspiel von SOAP, WSDL und UDDI

Die Abb. 15.5-8 zeigt das Zusammenspiel von SOAP, WSDL und UDDI.²

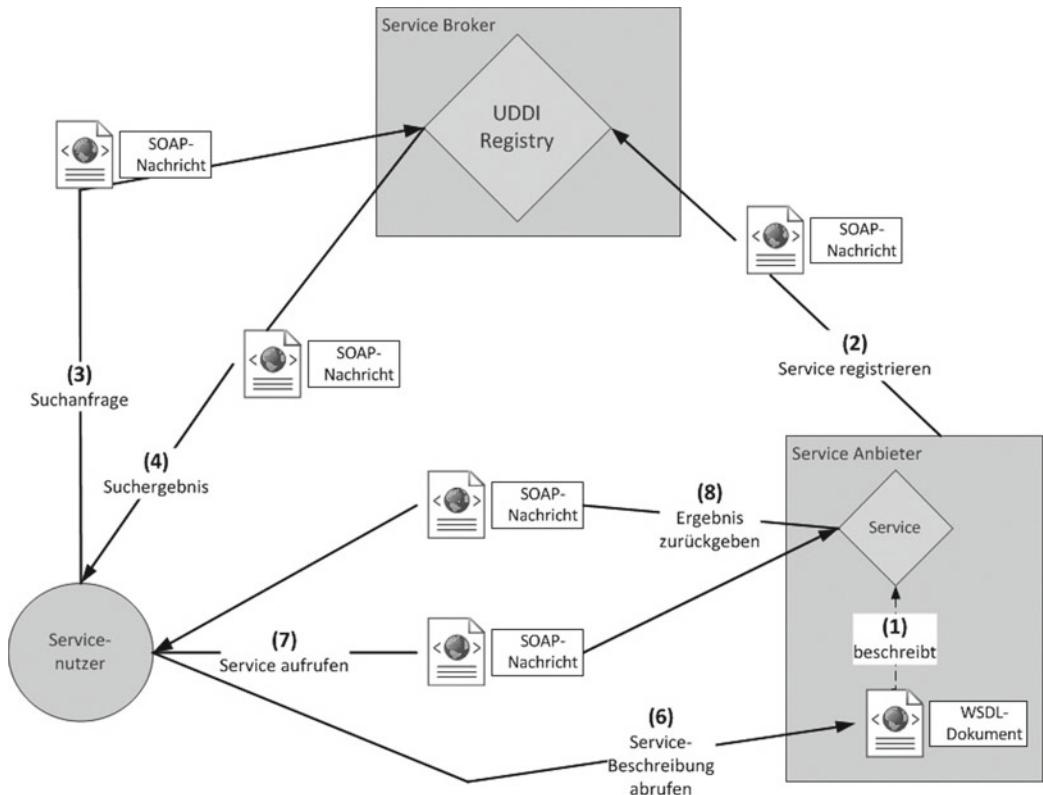


Abb. 15.5-8: SOAP, WSDL und UDDI im Zusammenspiel.

- Der Serviceanbieter erstellt einen Webservice. Die Beschreibung der Schnittstelle erfolgt in Form eines WSDL-Dokumentes.
- Der Serviceanbieter registriert seinen Webservice bei einer UDDI-Registry. Der Datenaustausch erfolgt dabei über SOAP.
- Ein Client sucht einen Webservice. Er stellt bei einer UDDI-Registry eine Suchanfrage. Die Suchanfrage selbst erfolgt mit SOAP.
- Die UDDI-Registry ermittelt in ihrer Datenbank die zur Suchanfrage passenden Webservices und sendet die Liste der Suchergebnisse mit SOAP an den Client zurück.
- Der Client wählt aus den Suchergebnissen einen Webservice aus, den er nutzen will. Die Suchergebnisse enthalten dabei neben einer textuellen Beschreibung der Webservices auch die Adressen der dazugehörigen WSDL-Dokumente.
- Der Client ruft das WSDL-Dokument des Webservice ab, den er aufrufen möchte.

- 7 Mit Hilfe des WSDL-Dokumentes erstellt der Client eine SOAP-Nachricht, um die von ihm gewünschte Methode aufzurufen und sendet diese an den Serviceanbieter.
- 8 Der Webservice des Serviceanbieters führt die gewünschte Methode aus und sendet die Ergebnisse der Methode an den Client.

15.5.5 Fallstudie: KV – SOAP

Für die Fallstudie »Kundenverwaltung-Mini« kann für die Client-Server-Kommunikation die Technik SOAP eingesetzt werden. Das gesamte OOD-Modell zeigt die Abb. 15.5-9. Die Klassen Client-Start, Kundenfenster, Kunde und ObjektSpeicher sowie die Schnittstelle KundenContainerEinfachI bleiben unverändert.

Auf der Serverseite wird eine Klasse KundenanfragenManager als Server WebService deklariert. Sie wird unter dem Namen Kundenverwaltung veröffentlicht. Ihr SOAPBinding wird auf RPC gesetzt. Die zu veröffentlichten Methoden sind als @WebMethod gekennzeichnet:

```
import javax.jws.*;                                     Java
import javax.jws.soap.SOAPBinding;

@WebService(name="Kundenverwaltung")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class KundenanfragenManager
{
    @WebMethod
    public boolean einfuegeKunde(String kndName, int kndNr)
    {
        KundenContainerObjekt objContainer =
            KundenContainerObjekt.getObjektreferenz();
        Kunde k = objContainer.getKundeZuNr(kndNr);
        if (k == null)
        {
            k = new Kunde();
            k.setNummer(kndNr);
            objContainer.einfuegeKunde(k);
        }
        k.setName(kndName);
        return true;
    }

    @WebMethod
    public int getNaechsteKundenNr()
    {
        int nextKdnNr =
            KundenContainerObjekt.getObjektreferenz().
                getNaechsteKundenNr();
        return nextKdnNr;
    }

    @WebMethod
    public String getKundeZuNr(int kndNr)
```

I 15 Arten der Netzkommunikation

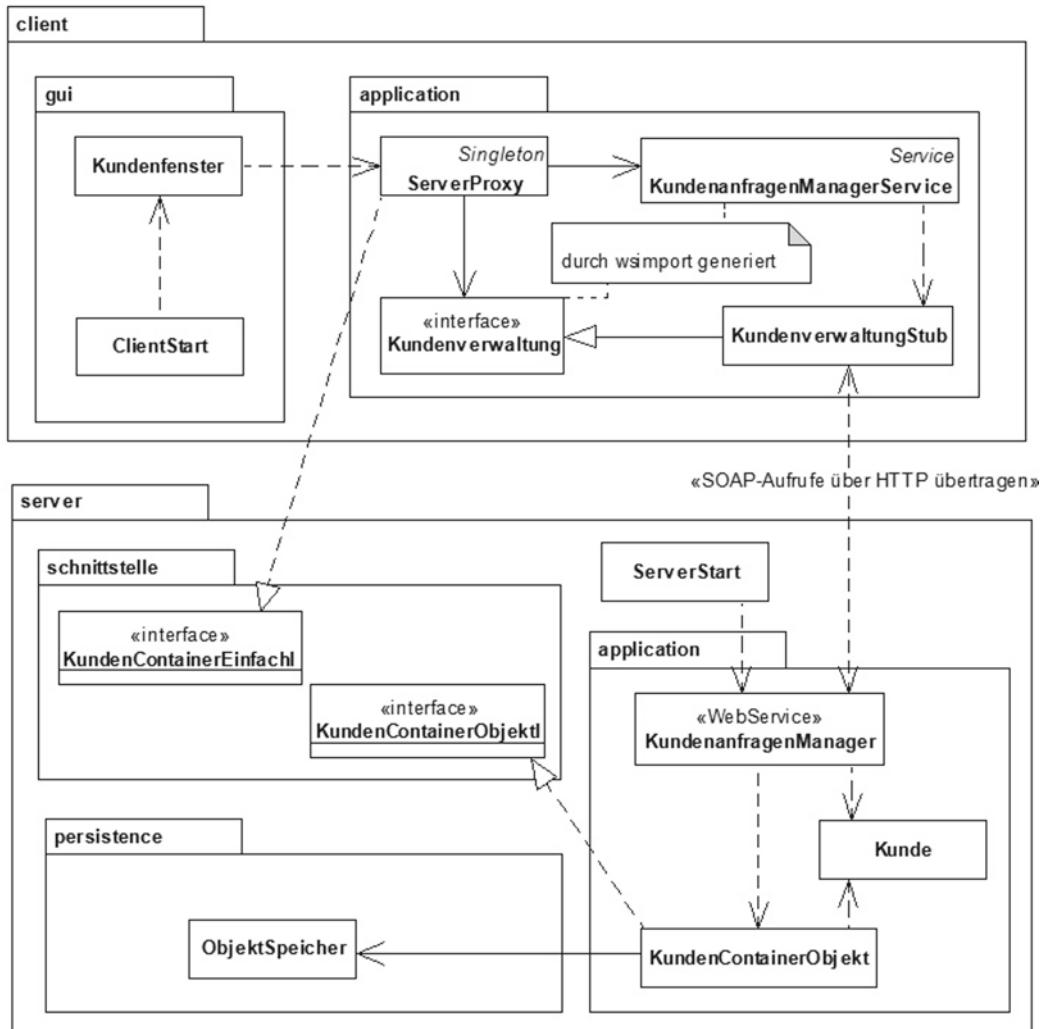


Abb. 15.5-9: OOD-Klassendiagramm der Fallstudie »KV – SOAP«.

```
{
    Kunde k =
        KundenContainerObjekt.getObjektreferenz() .
            getKundeZuNr(kndNr);
    return (k != null) ? k.getName() : "";
}
```

```
@WebMethod
public boolean endeAnwendung()
{
    KundenContainerObjekt.getObjektreferenz() .
        endeAnwendung();
    return true;
}
```

Zu beachten ist, dass eine `public`-Methode auch dann veröffentlicht wird, wenn sie *nicht* mit `@WebMethod` gekennzeichnet ist. Will man eine `public`-Methode explizit von der Veröffentlichung ausschließen, dann muss sie mit `@WebMethod(exclude=true)` annotiert werden. In dieser Klasse sind alle Methoden aufgeführt, die den Clients zur Verfügung stehen sollen.

In der Klasse `ServerStart` wird dieser WebService nach außen veröffentlicht:

```
package server; Java

import java.io.IOException;
import javax.xml.ws.Endpoint;
import server.fachkonzept.KundenanfragenManager;

class ServerStart
{
    public static void main(String[] args) throws IOException
    {
        Endpoint e = Endpoint.publish(
            "http://localhost:18080/kundenverwaltung",
            new KundenanfragenManager() );
        System.in.read();
        e.stop();
    }
}
```

Die Klassen, die die Clients für die Kommunikation mit dem Server benötigen, sollten nicht in das gleiche Paket wie die Webservice-Klasse (`KundenanfragenManager`) generiert werden, sondern in das Paket `client.application`. Hierfür wird die Option `?p` genutzt. Damit kann der Name des Zielpakets angegeben werden:

```
wsimport -p client.application -keep
http://localhost:18080/kundenverwaltung?wsdl
```

Es werden folgende Klassen generiert:

- `KundenanfragenManagerService`
- `Kundenverwaltung`

`Kundenverwaltung` ist eine Schnittstelle, die alle veröffentlichten Methoden von `KundenanfragenManager` anbietet. `KundenanfragenManagerService` ist eine Hilfsklasse, die die Klasse `Service` implementiert. Diese Implementierung ist dann eine Stummel-Klasse. Sie wandelt die Methodenaufrufe in SOAP-Nachrichten um und reicht sie per HTTP an den Server weiter.

In `ServerProxy` wird dies im Konstruktor durchgeführt:

```
package client.application; Java

import server.schnittstelle.KundenContainerEinfachI;

public class ServerProxy implements KundenContainerEinfachI
{
```

I 15 Arten der Netzkommunikation

```
//Singleton-Muster
private static ServerProxy serverProxy = null;

private KundenanfragenManagerService kms;
private Kundenverwaltung kv;

public ServerProxy()
{
    kms = new KundenanfragenManagerService();
    kv = kms.getKundenverwaltungPort();
}

public void einfuegeKunde(String name, int nummer)
{
    try
    {
        kv.einfuegeKunde(name, nummer);
    }
    catch (Exception e)
    {
        try
        {
            throw new Exception();
        }
        catch (Exception e1)
        {
            e1.printStackTrace();
        }
    }
}

{ ... }
```

ServerProxy realisiert damit den Zugriff auf das Fachkonzept, beispielhaft anzusehen an der Methode einfuegeKunde() der Klasse ServerProxy.

OOP Die Programme zu dieser Fallstudie können Sie im E-Learning-Kurs herunterladen.

 Installieren Sie das Programm auf Ihrem Computersystem und führen es aus. Gehen Sie das Programm Schritt für Schritt durch, um die Kommunikation zwischen Client und Server zu verstehen.

15.6 REST

Die Technik **REST** (*REpresentational State Transfer*) ermöglicht es, Webservices zu realisieren. Roy Thomas Fielding beschreibt die Technik in seiner Dissertation [Fiel00]. Bei einer RESTkonformen-Anwendung werden Daten über HTTP übertragen, ohne eine zusätzliche Transportschicht, wie SOAP, oder eine Sitzungsverwaltung über Cookies, einzusetzen.

Jedes einzelne Objekt einer Anwendung, z. B. Kunde oder Anzeige, stellt in REST eine Ressource dar, die von außen über eine **URI** erreichbar ist.

Zunächst wird das Konzept von den REST näher erläutert:

- »Die Konzepte von REST«, S. 287

Anschließend wird gezeigt, wie mit JAX-RS in Java eine RESTkonforme-Anwendung realisiert werden kann:

- »Webservices mit JAX-RS «, S. 289

Die Fallstudie »Kundenverwaltung-Mini« wird ebenfalls mit REST implementiert:

- »Fallstudie: KV – REST«, S. 297



15.6.1 Die Konzepte von REST

Clients initiieren Anfragen an Server. Server bearbeiten die Anfragen und geben eine entsprechende Antwort zurück. Anfragen und Antworten werden in REST als Transfer von Repräsentationen von Ressourcen angesehen. Eine Anwendung kann mit einer Ressource interagieren, wenn sie zwei Dinge kennt: den Identifikator der Ressource und die Aktion, die ausgeführt werden soll. Außerdem muss die Anwendung wissen, wie das Format der zurückgegebenen Information (Repräsentation) zu verstehen ist.

Im Web stellen Seiten, Bilder usw. **Ressourcen** da, die über **URLs** – oder allgemeiner über **URIs** – adressiert werden können.

Ressourcen
Beispiele

Bucheintrag bei Amazon:

http://www.amazon.de/Lehrbuch-Softwaretechnik-Softwaremanagement-Helmut-Balzert/dp/3827411610/ref=sr_1_2? s=books&ie=UTF8&qid=1281602689&sr=1-2

E-Learning-Kurs bei W3L:

<http://www.w3l.de/w3l/jsp/shop/produktdetails.jsp?produktoid=62>

In einer Ressource kann durch **Links** auf weitere Ressourcen verwiesen werden. Ein Client kann anhand von Links von einer Ressource zu einer anderen Ressource navigieren, ohne dass dafür Registrierungsdatenbanken oder ähnliche Infrastrukturen erforderlich sind. Die Verknüpfung von Ressourcen innerhalb einer REST-Architektur bezeichnet man auch als Verbindungshaftigkeit.

Mit HTTP können Nachrichten an die Ressourcen gesendet werden, z. B. durch den Aufruf einer Seite im Webbrower. In REST stellen die HTTP-Methoden GET, PUT, POST und DELETE die generischen Methoden dar, die auf *alle* Ressourcen angewandt werden können, analog wie in SQL die generischen Befehle SELECT, INSERT, UPDATE und DELETE zur Manipulation von Daten verwendet werden. REST verwendet die HTTP-Methoden in folgender Bedeutung:

Methoden

I 15 Arten der Netzkommunikation

- GET: Fragt die Repräsentation einer Ressource ab. GET-Anfragen können beliebig oft gestellt werden. HTTP schreibt vor, dass GET »sicher «(safe) sein muss. Das bedeutet, dass diese Methode nur Informationen beschafft und sonst keine Effekte verursacht.
- POST: Einer Ressource kann etwas hinzugefügt werden. POST-Anfragen können Seiteneffekte haben.
- PUT: Mit PUT können neue Ressourcen erzeugt oder der Inhalt bestehender Ressourcen ersetzt werden.
- DELETE: Ressourcen können gelöscht werden.

Die HTTP-Spezifikation legt fest, dass die Methoden GET, PUT und DELETE **idempotent** sein müssen, d.h. dass das mehrfache Absenden der gleichen Anforderung sich nicht anders auswirkt als ein einzelner Aufruf.

Nachrichten

In REST-Anwendungen können beliebige Dokumente übertragen werden, z.B. HTML-, GIF- und PDF-Dateien. Für strukturierte Daten kann XML verwendet werden. XLink kann für Verweise benutzt werden. Nachrichten müssen in REST selbstbeschreibend sein, d.h. in einer Nachricht muss alles enthalten sein, um die Nachricht interpretieren zu können, einschließlich aller benötigten Metadaten. Dadurch ist es möglich, dass ein Client mithilfe der Repräsentation einer Ressource in der Lage ist, eine Resource auf dem Server zu modifizieren oder zu löschen, vorausgesetzt der Client hat die Erlaubnis dazu. Jede Nachricht enthält also alle Informationen darüber, wie sie zu verarbeiten ist. Beispielsweise wird angegeben, welcher Parser aufzurufen ist (spezifiziert durch den Medientyp, früher Mime-Typ genannt). Antworten geben explizit an, ob sie zwischengespeichert (*caching*) werden dürfen.

Zustandslosigkeit

Es wird zwischen dem Ressourcenzustand und dem Anwendungszustand unterschieden:

- Jede Ressource besitzt einen **Ressourcenzustand**. Die Verwaltung des Zustands ist Aufgabe des Servers. Der Client erhält Informationen über den Ressourcenzustand nur über die ihm zugestellten Repräsentationen einer Ressource.
- Der **Anwendungszustand** gibt an, wo sich der Client innerhalb der Anwendung befindet. Der Anwendungszustand muss vom Client verwaltet werden. Der Webservice besitzt (auf dem Server) keine Informationen über den Anwendungszustand. Daher ist auf der Seite des Servers auch *keine* Sitzungsverwaltung erforderlich. REST verwendet das zustandslose HTTP-Protokoll. Jede HTTP-Botschaft besitzt alle Informationen, die notwendig sind, um die Nachricht zu verstehen. Jede Anfrage eines Clients an den Server ist in sich geschlossen, d.h. sie beinhaltet alle Informationen über den Anwendungszustand, die vom Server für die Verarbeitung der Anfrage benötigt werden.

In REST wird *keine* Methodeninformation in den URI angegeben – Charakteristika im Gegensatz zu anderen RPC-Architekturen.

- + Die Schnittstelle zwischen Systemen wird in REST auf eine überschaubare und standardisierte Menge von Aktionen abgebildet. Vorteile
- + Clients und Server können unabhängig voneinander ausgetauscht und entwickelt werden, solange die Schnittstelle nicht geändert wird.
- + GET-Aufrufe können zwischengespeichert (*caching*) werden, dadurch kann die Leistung verbessert werden. Im Gegensatz dazu verwendet SOAP nur die POST-Methode, die nicht zwischengespeichert werden kann.

Die Zustandslosigkeit von REST unterstützt die Skalierbarkeit (siehe »Nichtfunktionale Anforderungen«, S. 109), da eingehende Anfragen bei einer Lastverteilung auf beliebige Computersysteme verteilt werden können. Zusammenhänge

15.6.2 Webservices mit JAX-RS

JAX-RS(*Java API for RESTful Web Services*) spezifiziert eine API, um in Java RESTkonforme-Anwendungen zu realisieren. Für die Entwicklung und Installation von Webservice-Clients und Service-Endpunkten werden Annotationen verwendet.

Die Referenzimplementierung von JAX-RS wurde im Projekt Jersey erstellt, siehe Website Jersey (<https://jersey.dev.java.net/>). In der Java EE-Plattform ist Jersey standardmäßig enthalten (siehe »Die Java EE-Plattform«, S. 321), in Java SE nicht. Um Jersey in Java SE einzusetzen zu können, müssen drei Java-Archive von der Website Jersey (<https://jersey.dev.java.net/>) heruntergeladen werden:

- jersey-bundle.jar (<http://download.java.net/maven/2/com/sun/jersey/jersey-bundle/1.3/jersey-bundle-1.3.jar>) .jar-Dateien
- jsr311-api.jar (<http://download.java.net/maven/2/javax/ws/rs/jsr311-api/1.1/jsr311-api-1.1.jar>)
- asm.jar (<http://repo1.maven.org/maven2/asm/asm/3.1/asm-3.1.jar>)

Alle drei Java-Archive müssen in den Klassenpfad (*Classpath*) aufgenommen werden.

Mit Jersey lässt sich ein Servlet-Endpunkt definieren, sodass der RESTful-Webservice in einem Servlet-Container – ähnlich zu Apache Tomcat – läuft. Alternativ kann der Webservice den eingebauten Mini-HTTP-Server von Java SE nutzen. Im Folgenden wird der eingebaute Server verwendet. Mini-HTTP-Server

Ein Server soll Staumeldungen auf Autobahnen als Webservice zur Verfügung stellen. Web-Clients können diese Staumeldungen abfragen und Staumeldungen eintragen bzw. ändern. Einen Überblick über die Paket- und Klassenstruktur gibt die Abb. 15.6-1. Beispiel

I 15 Arten der Netzkommunikation

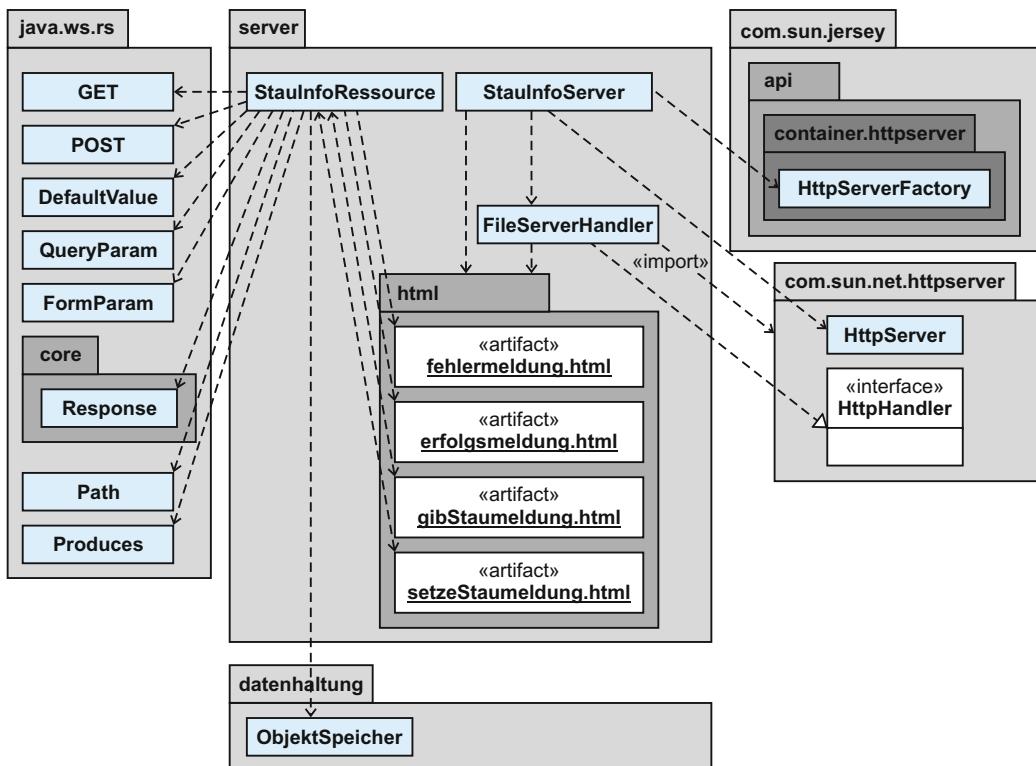


Abb. 15.6-1: **Server**

Paketdiagramm
zum Programm
Staumelder-REST.

Die Klasse StauInfoRessource ist die Ressourcenklasse:

```

package server;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.ArrayList;

import javax.ws.rs.DefaultValue;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;

import datenhaltung.ObjektSpeicher;

@Path("staumelder")
public class StauInfoRessource
{
    public static final String HTML_HEADER =
        "<!doctype html><html><head>"

```

```

<meta http-equiv='Content-Type' content='text/html;
charset=UTF-8'>" +
"<title>Stau-Abfrage</title></head><body>";
public static final String HTML_FOOTER =
"</body></html>";
public static final String IST_STAU =
"Auf der angegebenen Autobahn ist Stau.";
public static final String KEIN_STAU =
"Kein Stau-Eintrag für die angegebene Autobahn.";
public static final String ZURUECK_HTML =
"<br/><FORM><INPUT TYPE=\"button\" VALUE=\"Zurueck\""
onClick=\"history.go(-1);return true;\"/></FORM>";

private static final String stauinfoXmlPfad = "stauinfo.xml";

@GET
@Produces("text/html")
public String istStau(
{
    ArrayList<String> stauAutobahnen = leseStaudateiEin();
    Boolean istStau = stauAutobahnen.contains(autobahn);
    String ausgabe = HTML_HEADER +
        (istStau ? IST_STAU : KEIN_STAU) +
        ZURUECK_HTML + HTML_FOOTER;
    return ausgabe;
}

@POST
public Response speichereStauInfo(
    @DefaultValue("") @FormParam("autobahn") String autobahn,
    @DefaultValue("") @FormParam("iststau") String iststau)
    throws URISyntaxException
{
    if (autobahn.isEmpty() || iststau.isEmpty())
        return Response.seeOther(
            new URI("fehlermeldung.html")).build();
    setzeStau(autobahn, new Boolean (iststau));
    return Response.seeOther(
        new URI("erfolgsmeldung.html")).build();
}

private void setzeStau(String autobahn, boolean iststau)
{
    ArrayList<String> stauAutobahnen = leseStaudateiEin();

    if (stauAutobahnen.contains(autobahn))
    {
        boolean contained = stauAutobahnen.remove(autobahn);
        if (contained)
            schreibeList(stauAutobahnen);
    }
    else
    {
        if (iststau)

```

I 15 Arten der Netzkommunikation

```
        {
            stauAutobahnen.add(autobahn);
            schreibeList(stauAutobahnen);
        }
    }

private static synchronized void schreibeList(
    ArrayList<String> stauAutobahnen)
{
    ObjektSpeicher os = new ObjektSpeicher(stauinfoXmlPfad);
    os.speichereObjekt(stauAutobahnen);
}

private static synchronized ArrayList<String>
leseStaudateiEin()
{
    //fuer BlueJ
    Thread.currentThread().setContextClassLoader(
        StauInfoRessource.class.getClassLoader());

    //Gespeicherte Daten einlesen
    //Falls noch keine Daten gespeichert wurden, kann keine
    //Datei gelesen werden, es gibt dann eine Ausnahme
    ArrayList<String> stauAutobahnen = null;
    ObjektSpeicher os = new ObjektSpeicher(stauinfoXmlPfad);
    try
    {
        stauAutobahnen = (ArrayList)os.leseObjekt();
        if (stauAutobahnen == null)
            stauAutobahnen = new ArrayList<String>();
    }
    catch (Exception e)
    {
        //wenn keine Daten gelesen werden konnten, muss eine
        //neue Datenbasis angelegt werden
        stauAutobahnen = new ArrayList<String>();
    }
    return stauAutobahnen;
}
}
```

GET-Anfrage:
 @Path
 @Path
 ("stauinfo")
 istStau()
Mit der Annotation @Path("stauinfo") wird angegeben, dass die Klasse unter dem Pfad stauinfo erreichbar ist.
Die Methode istStau(@QueryParam("autobahn") String autobahn) bedient die Anfragen des Client und beantwortet, ob es auf einer angegebenen Autobahn einen Stau gibt. Der Eingabeparameter autobahn ist mit QueryParam annotiert, d. h. er wird als HTTP-Anfrageparameter übertragen. Mit @Produces wird angegeben, welchen Medientyp die Methode produziert, in diesem Fall HTML-Text. Im Rumpf der Methode wird als erstes die Staudatei in eine Liste eingelesen. Wenn die angegebene Autobahn in der Liste enthalten ist, bedeutet das, dass es dort einen Stau gibt. Die Ausgabe wird dementsprechend

für den Benutzer lesbar vorbereitet. Die Texte sind in `IST_STAU` und `KEIN_STAU` definiert. Zusätzlich werden die Inhalte in die Bereiche `HTML_HEADER` und `HTML_FOOTER` verpackt. In `ZURUECK_HTML` steht HTML-Code, der zur Anzeige eines Zurück-Druckknopfs dient. Es kann also HTML-Code zurückgegeben werden, weil dies in `@Produces` als Typ angegeben wurde. Die Konstanten sind in der Klasse definiert.

Die Post-Anfrage soll dazu dienen, eine Stau-Information an den Server zu senden. Um die Post-Anfrage zu bearbeiten, ist auf dem Server die Methode `speichereStauInfo()` definiert. An den beiden Eingabeveriablen steht die Annotation `@FormParam`. Für Post-Anfragen muss diese Annotation anstelle von `@QueryParam` verwendet werden, weil bei Post-Anfragen die Parameter im Rumpf des HTTP-Requests übertragen werden und daran bindet `@FormParam` die Variable. Wenn ein übergebener Parameter leer ist, dann wird auf eine Seite mit Fehlermeldung umgeleitet. Ansonsten fährt die Methode fort. Mit `setzeStau` wird die Stau-Information abgespeichert. Am Ende wird umgeleitet auf eine Seite mit Erfolgsmeldung.

Post-Anfrage:
speichere
StauInfo()

```
package server;

import java.io.IOException;
import com.sun.jersey.api.container.httpserver.HttpServerFactory;
import com.sun.net.httpserver.HttpServer;
public class StauInfoServer
{
    public static void main(String argv[])
        throws IllegalArgumentException, IOException
    {
        System.out.println(new java.io.File(".").
getAbsolutePath());
        final String baseUri = "http://localhost:9998/";
        HttpServer server = HttpServerFactory.create(baseUri);
        FileServerHandler h =
            new FileServerHandler ("server/html");
        server.createContext("/gibStaumeldung.html", h);
        server.createContext("/setzeStaumeldung.html", h);
        server.createContext("/erfolgsmeldung.html", h);
        server.createContext("/fehlermeldung.html", h);

        server.start();

        System.out.println(String
            .format("StauInfoServer (Rest-Version) " +
            "ist gestartet unter der URL %s.\n " +
            "Zum Beenden bitte Enter drücken!",baseUri));
        System.in.read();
        server.stop(0);
        System.exit(0);
    }
}
```

Java
StauInfo
Server

I 15 Arten der Netzkommunikation

Den `HttpServer` erhält man von der `HttpServerFactory` über die `create()`-Methode. Der Parameter ist der Basis-Link, über den die Funktionalität aufgerufen werden kann. `StauInfoRessource` wird dann automatisch beim Start gefunden. Nur die zusätzlichen HTML-Dateien (für Abfragen/Eintragen einer Staubmeldung usw.) müssen explizit beim Server eingetragen werden. Dies geht über die `createContext()`-Methode des `HTTPServers`. Dabei wird der Pfad zur HTML-Datei angegeben und als zweites der zuständige `HttpHandler`. Der `HttpHandler` wird aufgerufen, um die HTTP-Anfrage zu bearbeiten. In diesem Fall wird der `FileServerHandler` auf der Basis der Implementierung von Sun bzw. Oracle verwendet. Die Datei liegt im Projekt als Quellcode mit anbei. `FileServerHandler` bearbeitet die HTTP-Anfrage so, dass er im Basispfad, den er zugewiesen bekommt, nach der in der URL angeforderten Datei sucht, und den Inhalt dieser Datei in der Antwort zurückgibt. Der Code der Klasse `FileServerHandler` ist so angepasst, dass das Laden der HTML-Seiten auch in einer jar-Datei funktioniert.

Staubfrage

Client

Die HTML-Datei für eine Stau-Anfrage hat folgenden Inhalt:

```
<!doctype html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
    charset=UTF-8">
<title>Stau-Abfrage</title>
</head>
<body>
<form action="stauinfo" method="get">
Staubfrage für Autobahn
<input type="text" name="autobahn" />
<br />
<input type="submit" value="Anfragen" />
</form>
</body>
</html>
```

Es wird ein Formular definiert mit einem Eingabefeld und einem Absende-Druckknopf. Dabei ist auf folgende Punkte zu achten, damit die Zusammenarbeit mit dem Server funktioniert:

- Der Name des Eingabefeldes muss mit dem Parameter der Annotation `@QueryParam` übereinstimmen (beide haben in diesem Fall den Wert `autobahn`).
 - Das `method`-Attribut von `form` muss auf `get` gesetzt sein.
 - Das `action`-Attribut von `form` muss auf `stauinfo` gesetzt sein.
- Die Abb. 15.6-2 zeigt die Staubfrage im Browser.

15.6 REST I

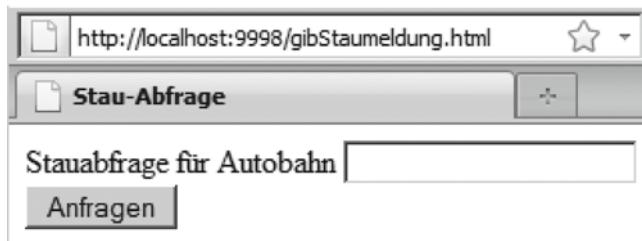


Abb. 15.6-2:
Stauabfrage für
das Programm
Staumelder-REST.

Nach einer Anfrage bekommt man die Meldung der Abb. 15.6-3 angezeigt.

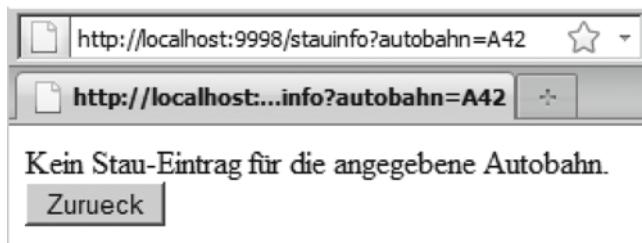


Abb. 15.6-3:
Antwort zur
Stauabfrage
(Programm
Staumelder-REST).

Die HTML-Datei zum Senden einer Stau-Meldung sieht wie folgt aus:

```
<!doctype html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
  charset=UTF-8">
<title>Setzen einer Staumeldung</title>
</head>
<body>
<form action="stauinfo" method="post">
  Staumeldung setzen für Autobahn
  <input type="text" name="autobahn" />
  <br />
  Ist Stau auf dieser Autobahn
  <p>
    <input type="radio" name="iststau" value="true">Ja<br/>
    <input type="radio" name="iststau" value="false">Nein<br/>
  </p>
  <input type="submit" value="Senden" />
</form>
</body>
</html>
```

Stau melden

Ein Formular wird definiert mit zwei Eingabe-Elementen (Text- und Optionsfeld), dazu der Absenden-Druckknopf. Für das Zusammenspiel mit dem Server ist folgendes zu beachten:

- Das `method`-Attribut von `form` muss auf `post` gesetzt sein.
- Das `action`-Attribut von `form` muss auf `stauinfo` gesetzt sein.

I 15 Arten der Netzkommunikation

- Die Namen der Eingabe-Elemente müssen mit den @FormParam-Annotationsparametern übereinstimmen (in diesem Fall `autobahn` und `iststau`).
- Die Werte im Optionsfeld sind boolean-Werte.
Die Abb. 15.6-4 zeigt die Darstellung im Webbrowser.

Abb. 15.6-4:
Staumeldung
setzen (Programm
Staumelder-REST).

The screenshot shows a web browser window with the URL `http://localhost:9998/setzeStaumeldung.html`. The title bar says "Setzen einer Staumeldung". The main content area has a label "Staumeldung setzen für Autobahn" followed by an empty input field. Below it is the question "Ist Stau auf dieser Autobahn" with two radio button options: "Ja" and "Nein". At the bottom is a "Senden" button.

Nach dem Ausfüllen des Formulars und Absenden bekommt der Benutzer die Ausgabe der Abb. 15.6-5.

Abb. 15.6-5:
Antwort auf
Staumeldung
setzen (Programm
Staumelder-REST).

The screenshot shows a web browser window with the URL `http://localhost:9998/erfolgsmeldung.html`. The title bar says "Antwort". The main content area displays the message "Ihre Eingaben wurden erfolgreich übernommen." and a "Zurück" button.

Im kostenlosen E-Learning-Kurs zu diesem Buch können Sie das vollständige Programm herunterladen.

 Installieren Sie das Programm auf Ihrem Computersystem und führen Sie es aus. Gehen Sie das Programm Schritt für Schritt durch, um die Kommunikation zwischen Client und Server zu verstehen.

15.6.3 Fallstudie: KV – REST

Voraussetzung für die Implementierung der Fallstudie »Kundenverwaltung-Mini« mit dem Konzept REST mit der Java SE ist, dass die notwendigen Jersey-Bibliotheken vorhanden sind (siehe »Webservices mit JAX-RS«, S. 289)

Das gesamte OOD-Modell zeigt die Abb. 15.6-6. Die Klassen ClientStart, Kundenfenster, Kunde und ObjektSpeicher sowie die Schnittstelle KundenContainerEinfachI bleiben unverändert.

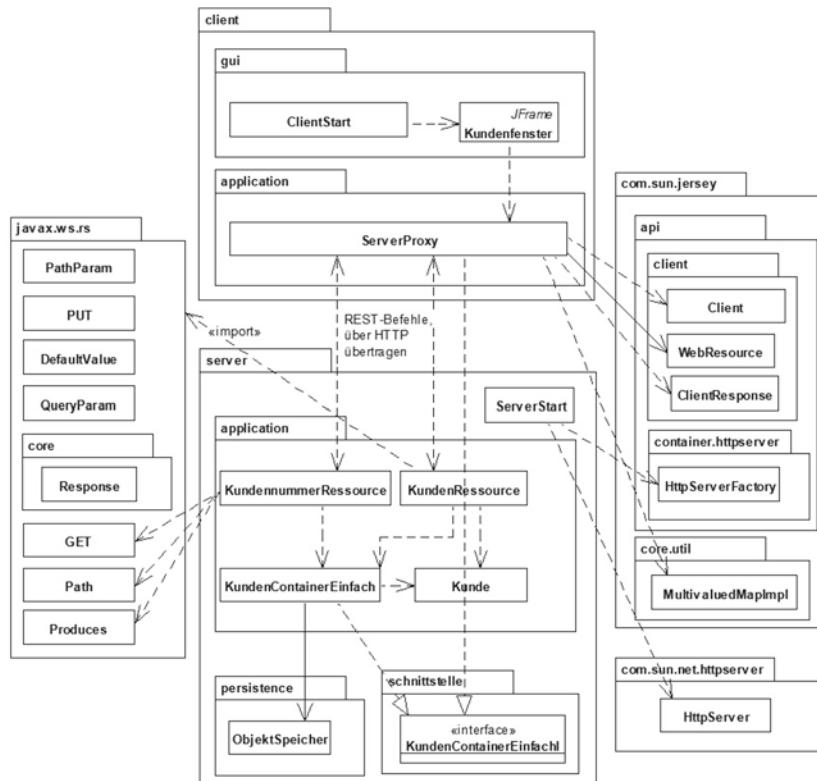


Abb. 15.6-6: OOD-Klassendiagramm der Fallstudie »KV – REST«.

In der REST-Variante der Fallstudie gibt es zwei Ressourcen:

- KundennummerRessource
 - KundenRessource

Ressourcen

Auf diese beiden Ressourcen hat die Klasse ServerProxy auf der Clientseite Zugriff. Das Paketdiagramm der Abb. 15.6-6 zeigt, dass die Ressourcen-Klassen Zugriff auf das Paket javax.ws.rs benötigen.

Die Klasse ServerStart hat eine Abhangigkeit zur Klasse Mini-HTTP-Server `HttpServer`. Diese Klasse reprsentiert den in Java eingebauten Mi-
ni-HTTP-Server. Darin integriert sich Jersey. Beim Start des HTTP-

I 15 Arten der Netzkomunikation

Servers (in diesem Fall auf der Adresse `http://localhost:9998`) wird der Klassenpfad nach allen Klassen durchsucht, die JAX-RS-Annotationen tragen. Die Klasse ServerStart sieht wie folgt aus:

```
Java class ServerStart
{
    public static void main(String[] args) throws IOException
    {
        final String baseUri = "http://localhost:9998/";
        HttpServer server = HttpServerFactory.create(baseUri);
        server.start();
        System.out.println(String.format(
            "Kundenverwaltung (Rest-Version) ist gestartet unter der
            URL %s.\n Zum Beenden bitte Enter drücken!",baseUri));
        System.in.read();
        server.stop(0);
        System.exit(0);
    }
}
```

Der Server gibt nach dem Start folgende Ausgaben aus:

```
22.03.2011 14:49:43 com.sun.jersey.api.core.
    ClasspathResourceConfig init
INFO: Scanning for root resource and provider classes
      in the paths: KV-server.jar
22.03.2011 14:49:43 com.sun.jersey.api.core.
    ScanningResourceConfig logClasses
INFO: Root resource classes found:
      class server.fachkonzept.KundennummerRessource
      class server.fachkonzept.KundenRessource
22.03.2011 14:49:43 com.sun.jersey.api.core.
    ScanningResourceConfig init
INFO: No provider classes found.
22.03.2011 14:49:43 com.sun.jersey.server.impl.application.
    WebApplicationImpl _initiate
INFO: Initiating Jersey application, version
      'Jersey: 1.3 06/17/2010 05:04 PM'
Kundenverwaltung (Rest-Version) ist gestartet unter der
      URL http://localhost:9998/.
Zum Beenden bitte Enter drücken!
```

Die beiden Ressourcen-Klassen werden gefunden, weil sie JAX-RS-Annotationen enthalten, z. B. die Klasse KundenRessource:

```
@Path("/kunde/{kundennummer}")
public class KundenRessource
{
    ...
}
```

Die Path-Annotation gibt die Pfad-Angabe für die Ressource an. Hier wird diese auf Klassenebene angegeben. Eine Angabe auf Methodenebene ist ebenfalls möglich.

Der Pfad `/kunde/{kundennummer}` gibt an, dass die Ressource im Unterpfad `kunde` liegt, und dazu den `template`-Parameter `kundennummer`. Dies ist eine Variable, die zur Laufzeit durch den angegebenen Wert ersetzt wird.

Die GET-Methode der Klasse `KundenRessource` beginnt folgendermaßen:

```
@GET
@Produces("text/plain")
public String getKundeZuNr(
    @PathParam("kundennummer") String kndNr)
```

`@Get` gibt an, dass diese Methode für GET-Anfragen auf diese Ressource zuständig ist. `@Produces` zeigt an, welchen Medientyp diese Methode zurückgibt. In diesem Fall ist es Text. Der Eingabeparameter `kndNr` der Methode ist annotiert mit `@PathParam("kundennummer")`.

Das bedeutet, dass der Wert dieser Variable aus dem Pfad zu dieser Ressource genommen wird. Genauer: Er wird an den mit `{kundennummer}` gekennzeichneten Bereich des Pfades gebunden. Somit kann `getKundeZuNr()` mit folgender URL auch im Browser aufgerufen werden (vorausgesetzt, der Benutzer hat vorher `ServerStart` gestartet):

`http://localhost:9998/kunde/9`

Diese GET-Anfrage hat zur Folge, dass auf dem Server die Ressourcen-Klasse aufgerufen wird, die auf den Pfad `/kunde` registriert ist. Das ist `KundenRessource`. Dabei wird `kundennummer` auf den Wert 9 gesetzt. Die GET-Anfragen werden in dieser Klasse von der Methode `getKundeZuNr()` bearbeitet. Somit entspricht der Aufruf dieser URL folgendem Methodenaufruf auf dem Server:

```
new KundenRessource().getKundeZuNr(9);
```

Da es sich bei der »Kundenverwaltung-Mini« um *keine* Web-Anwendung handelt, kann dieser Ansatz über den Browser *nicht* verwendet werden. Die Aufrufe müssen im Java-Programm erfolgen und zwar in der Klasse `ServerProxy`, die für die GUI der Zugang zum Fachkonzept ist. Diese beinhaltet zwei `WebResource`-Objekte `kndRessource` und `kndNrRessource`. Jedes dieser Objekte steht für eine Ressource. Die `WebResource` kann einfach erhalten werden, wenn die URI bekannt ist:

```
//Konstruktor der Klasse ServerProxy
public ServerProxy()
{
    Client client = Client.create();
    final String baseUri = "http://localhost:9998/";
    kndRessource = client.resource(baseUri + "kunde");
    kndNrRessource = client.resource(baseUri + "kundennummer");
}
```

I 15 Arten der Netzkomunikation

Die GET-Anfrage im ServerProxy, um einen Kundennamen zu erhalten, sieht dann wie folgt aus:

```
String s = kndRessource.path(kndNrStr).accept  
    (MediaType.TEXT_PLAIN).get(String.class);
```

Der Pfad der WebResource wird noch durch die Kundennummer ergänzt, und dann die Methode get() aufgerufen. Als Eingabeparameter wird der Typ des Rückgabewertes angegeben. Als Antwort wird der Name des Kunden zurückgegeben (Leer, wenn der Kunde mit der angegebenen Nummer nicht vorhanden ist).

PUT-Methode Die Methode in KundenRessource, die PUT-Anfragen bearbeitet, ist einfuegeKunde. Ihre Signatur ist:

```
@PUT  
public Response einfuegeKunde(  
    @DefaultValue("") @QueryParam("kundenname") String kndName,  
    @DefaultValue("") @PathParam("kundennummer") String kndNr)
```

Der erste Eingabeparameter wird an einen HTTP-Anfragenparameter gebunden mit dem Namen kundenname, der zweite Parameter an einen Pfad (@PathParam). Wie dieser durch den Client gesetzt wird, wird noch erklärt. Die Rückgabe ist ein Response-Objekt: Entweder wird ein neues Kunde-Objekt angelegt, in diesem Fall wird das Response-Objekt wie folgt erzeugt:

```
response = Response.created(uri).build();
```

uri gibt hier die Adresse der erzeugten Ressource an (uri kann auch den Wert null haben, weil die Adresse dem Client ohnehin bekannt ist). Oder es wird ein bestehendes Kunde-Objekt modifiziert, dann wird das Response-Objekt so erstellt:

```
response = Response.noContent().build();
```

Das heißt, ein leeres Response-Objekt wird zurückgegeben. Der Code zum Aufrufen durch den Client sieht wie folgt aus (Klasse ServerProxy):

```
Java public void einfuegeKunde(String name, int nummer)  
{  
    try  
    {  
        String kndNrStr = Integer.toString(nummer);  
        MultivaluedMap<String, String> queryParams =  
            new MultivaluedMapImpl();  
        queryParams.add("kundenname", name);  
        kndRessource.path(kndNrStr).  
            queryParams(queryParams).  
            accept(MediaType.TEXT_PLAIN).  
            put(ClientResponse.class);  
    }  
    catch (Exception e)  
    {
```

```
try
{
    throw new Exception();
}
catch (Exception e1)
{
    e1.printStackTrace();
}
}
```

Der HTTP-Anfrageparameter mit dem Namen kundenname wird in eine MultivaluedMap eingefügt. In einer MultivaluedMap können Schlüssel-Werte-Paare abgelegt werden. Der Schlüssel ist in diesem Fall kundenname, und dieser wird auch so auf der Seite des Servers abgefragt. Die MultivaluedMap wird der WebResource über die Methode queryParams hinzugefügt. Die Methode für den PUT-Aufruf heißt put.

Die Fallstudie »Kundenverwaltung-Mini – REST« unterscheidet sich von anderen Client-Server-Versionen wie folgt:

Es gibt für den Client keine Möglichkeit endeAnwendung() aufzurufen, wie das z. B. in der SOAP-Variante möglich ist (siehe »Fallstudie: KV – SOAP«, S. 283). Die Methode endeAnwendung() entfällt, weil sie keiner Ressource zuzuordnen ist. Die Methode veranlasste den Server dazu, die Daten persistent zu speichern. Die persistente Speicherung erfolgt nun in der REST-Variante so, dass bei jeder PUT-Anfrage (einfuegeKunde()) die Daten persistent gespeichert werden.

Die Programme zu dieser Fallstudie können Sie im E-Learning- OOP Kurs herunterladen.

Installieren Sie das Programm auf Ihrem Computersystem und führen es aus. Gehen Sie das Programm Schritt für Schritt durch, um die Kommunikation zwischen Client und Server zu verstehen.



15.7 Netzkommunikation in .NET

.NET stellt verschiedene Kommunikationsbibliotheken zur Realisierung verteilter Anwendungen zur Verfügung. Auf den unteren Abstraktionsebenen sind dies

- TCP/UDP Sockets,
- *Named* und *Unnamed Pipes*,
- *Memory Mapped Files* sowie
- Anwendungsprotokolle wie HTTP, FTP, SMTP.

In allen diesen Kommunikationsbibliotheken ist es die Aufgabe des Entwicklers, *selbst* für die *Marshalling/Unmarshalling* von Objekten und Methodenaufrufen zu sorgen.

I 15 Arten der Netzkommunikation

Auf abstrakterer Ebene stellt .NET auch vier Kommunikationsbibliotheken bereit, bei denen die Verteilung durch Proxy-Klassen so gekapselt ist, dass ein Fernaufruf sowohl aus der Sicht des Entwicklers eines Servers als auch des Entwicklers eines Client in weiten Teilen transparent ist. Dies sind:

- *Distributed Component Object Model* (DCOM)
- .NET Remoting
- ASP.NET-Webservices
- *Windows Communication Foundation* (WCF)

DCOM Dabei wird DCOM nur noch zur Kommunikation mit alten COM-basierten Diensten aus der Ära vor .NET verwendet. Die Anzahl der existierenden COM-basierten Dienste darf nicht unterschätzt werden, denn sowohl das Windows-Betriebssystem als auch fast alle zentralen Anwendungen von Microsoft sowohl auf dem Server (z. B. Microsoft Exchange) als auch auf dem Client (Word, Excel, Outlook etc.) basieren noch auf dem COM/DCOM.

.NET Remoting und ASP.NET-Webservices sind zwei komplementäre Ansätze aus den Anfängen von .NET. .NET Remoting ist mit Java-RMI (siehe »RMI«, S. 220) vergleichbar und bietet eine objektorientierte Verteilungstransparenz zwischen zwei .NET-Prozessen.

ASP.NET-Webservices (entsprechen JAX-WS in Java, siehe »Webservices bereitstellen und nutzen mit JAX-WS«, S. 267) bieten plattformunabhängige XML-Web-Services.

Windows Communication Foundation (WCF)

Seit .NET 3.0 hat Microsoft frühere Ansätze in WCF neu- und zusammengefasst. WCF besitzt fünf wesentliche Eigenschaften:

- Die verschiedenen Aspekte verteilter Kommunikation (Schnittstelle, Implementierung, Kommunikationsprotokoll, Infrastrukturdienste, Hosting) lassen sich gut logisch trennen.
- WCF schreibt keine Protokolle und Formate fest vor, sondern erlaubt es dem Nutzer, einen Kommunikationskanal aus verschiedenen Protokollen und Formaten zusammenzustellen.
- Es gibt sowohl Unterstützung für plattformübergreifende Protokolle und Formate (W3C/WS-*) als auch proprietäre Protokolle und Formate der .NET-Welt.
- Ohne Neukompilierung eines Dienstes kann die Kommunikationsform vom Betreiber der Anwendung geändert werden.
- Ohne Neukompilierung können weitere Kommunikationsformen vom Betreiber der Anwendung ergänzt werden.

15.7 Netzkommunikation in .NET I

Die Abb. 15.7-1 veranschaulicht die von WCF unterstützten Protokolle, Formate und Funktionen. In gewissen Grenzen (einige Kombinationen sind nicht möglich) kann der Entwickler der Anwendung oder Betreiber der Anwendung diese zu einem spezifischen Kommunikationskanal miteinander kombinieren.

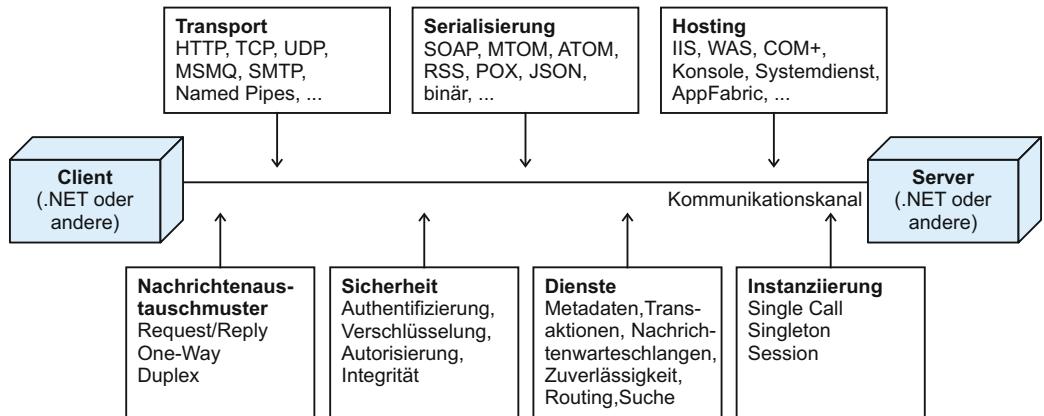


Abb. 15.7-1: WCF ist wie ein Warenkorb, den man aus verschiedenen Regalen bestücken kann.

Architektur einer verteilten Anwendung mit WCF

Die Abb. 15.7-2 stellt die typische Architektur einer verteilten Anwendung mit WCF dar. Rechts sieht man den Server, der aus einer Logikschicht, einer Datenzugriffsschicht und einer Geschäftsobjektbibliothek besteht. Innerhalb der Geschäftsobjektbibliothek kann der Entwickler mit den Annotationen [DataContract] und [DataMember] Einfluss auf die Serialisierung der Geschäftsobjekte nehmen.

Vor die Logik wird üblicherweise eine sogenannte Servicefassade (siehe »Das Fassaden-Muster (*facade pattern*)«, S. 69) gesetzt, die diejenigen Methoden der Geschäftslogik, die von einem Client aus zugänglich sein sollen, als WCF-Dienst bereitstellt.

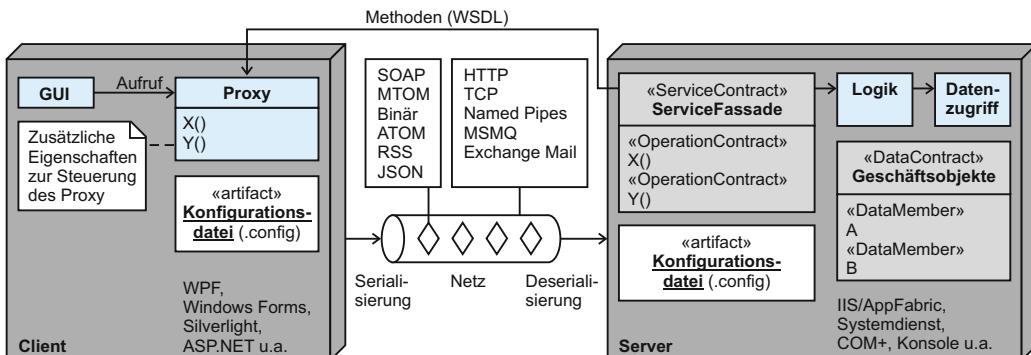
Der Dienst ist eine Klasse mit der Annotation [ServiceContract] und jede einzelne Dienstoperation muss mit [OperationContract] ausgezeichnet werden. Die Eigenschaften des Kommunikationskanals werden meist in der XML-Konfigurationsdatei abgelegt, weil sie dort ohne Neukompilierung vom Betreiber geändert werden können.

Aus den annotierten Klassen und den Konfigurationsdaten kann WCF automatisch die für einen Client notwendigen Metadaten in Form der *Webservice Description Language* (WSDL) erzeugen (siehe »SOAP-Nachrichten, Webservices und WSDL«, S. 263).

Der Client kann die WSDL dazu verwenden, eine Proxyklasse für den Zugriff auf den WCF-Dienst automatisch zu generieren. Der Client kann dann mit den Proxyklassen arbeiten und dessen Methoden aufrufen.

I 15 Arten der Netzkommunikation

Die an die Proxymethoden übergebenen Geschäftsobjekte werden auf dem Aufrufer serialisiert gemäß der Vorgaben des DataContracts und dem vom Kommunikationskanal vorgesehenen Serialisierungsformat (z. B. SOAP, MTOM, RSS, JSON). Auf der Empfängerseite sorgt WCF für die Deserialisierung und Übergabe der Geschäftsobjekte an die Logikschicht.



*Abb. 15.7-2:
Architektur einer
verteilten
Anwendung mit
WCF.*

Das folgende Codefragment zeigt die Definition der Servicefassade in Form einer Schnittstelle für eine Flugbuchung (Definition des WCF-Dienstes):

```

[ServiceContract]
public interface IBuchungsFassade
{
    [OperationContract]
    string NewBuchung(int FNummer, int PNummer);

    [OperationContract]
    Flug GetFlug(int FNummer);

    [OperationContract]
    Passagier GetPassagier(int PNummer);
}

```

Die Implementierung dieser Schnittstelle besteht nur aus Weiterleitungen der Aufrufe an die Logik (Implementierung des WCF-Dienstes):

```

public class BuchungsFassade : IBuchungsFassade
{
    public string NewBuchung(int FNummer, int PNummer)
    {
        return new WWWings_GL.BuchungsBLManager().
            NewBuchung(FNummer, PNummer);
    }
    public WWWings_G0.Passagier GetPassagier(int PNummer)
    {
        Passagier p = new WWWings_GL.PassagierBLManager().
            GetPassagier(PNummer);
    }
}

```

```

        return p;
    }
    public WWWings_G0.Flug GetFlug(int FNummer)
    {
        return new WWWings_GL.FlugBLManager().GetFlug(FNummer);
    }
}

```

Das dritte Codefragment zeigt die Konfiguration des Kommunikationskanals. basicHttpBinding ist die WCF-interne Bezeichnung für die Serialisierung mit SOAP 1.1 gemäß den dem Basic Profile 1.1 der *Web Services Interoperability Organisation* (WS-I) und den ungesicherten Transport über HTTP. Zudem ist ein Metadatenendpunkt definiert, an dem ein Client die WSDL-Daten beziehen kann (Konfiguration des WCF-Dienstes):

```

<system.serviceModel>
    <behaviors><br/>
        <serviceBehaviors>
            <behavior name="BuchungsFassadeBehavior">
                <serviceMetadata httpGetEnabled="true"/>
                <serviceDebug includeExceptionDetailInFaults="true"/>
            </behavior>
        </serviceBehaviors>
    </behaviors>
    <services>
        <service behaviorConfiguration="BuchungsFassadeBehavior"
            name="WWings_Dienste.BuchungsFassade">
            <endpoint address="" binding="basicHttpBinding"
                contract="WWings_Dienste.IBuchungsFassade">
            </endpoint>
            <endpoint address="mex" binding="mexHttpBinding"
                contract="IMetadataExchange"/>
        </service>
    </services>
</system.serviceModel>

```

Nach der Generierung einer Proxyklasse kann ein Client wie folgt auf den WCF-Dienst zugreifen:

```

BuchungsFassadeClient ws = new BuchungsFassadeClient();
string Buchungscode =
    ws.NewBuchung(Flug.FlugNr, Passagier.PersonID);
ws.Close();

```

Die Ausführung von `Close()` ist an dieser Stelle sinnvoll, auch wenn der tatsächliche Transport aktuell über das zustandslose HTTP erfolgt. Zu einem späteren Zeitpunkt könnte das Transportprotokoll aber auf ein zustandsbehaftetes Protokoll wie TCP geändert werden und dann würde das fehlende `Close()` dazu führen, dass der Kommunikationskanal länger als notwendig geöffnet bleibt.

I 15 Arten der Netzkommunikation

Application Server

Ein *Application Server* dient der Bereitstellung von Diensten (Services) für den entfernten Aufruf durch Clients. Seit 2010 gibt es von Microsoft einen eigenständigen *Application Server* mit Namen »Windows Server AppFabric«. AppFabric umfasst IIS, WAS und zusätzliche Komponenten für Verbreitung, Verwaltung und Überwachung sowie das Cachen von Daten. AppFabric ist vergleichbar mit den in der Java-Welt seit langem bekannten Application Servern.

Windows Server AppFabric unterstützt die Bereitstellung von ASP.NET-Anwendungen und WCF-Diensten sowie Workflows.

15.8 Entwurfskonzepte für verteilte Anwendungen

Die einzelnen Teile einer verteilten Anwendung müssen über eine langsame Netz-Verbindung miteinander kommunizieren. Ziel beim Entwurf muss es daher sein, die Anzahl der Kommunikations-Schritte über das Netz (*Roundtrip* genannt) möglichst gering zu halten. Entwurfsmuster können dabei helfen, eine verteilte Anwendung effizient und überschaubar zu halten.

Kein gemeinsamer Speicher

Eine verteilte Anwendung zeichnet sich dadurch aus, dass die einzelnen Subsysteme über *keinen* gemeinsamen Speicher verfügen.¹ Jegliche Kommunikation findet in Form von Nachrichten statt, die über ein Netz übertragen werden müssen. Methodenaufrufe müssen in Nachrichten umgewandelt werden. Dazu steht in der Regel ein entsprechendes Transport-System, z. B. RMI (siehe »RMI«, S. 220), zur Verfügung. Es kapselt die Netzübertragung und ermöglicht es dem Anwendungs-Entwickler, mit gewöhnlichen Methodenaufrufen zu arbeiten.

Keine Referenzen

Im Gegensatz zu lokalen Anwendungen können als Parameter *keine* Referenzen ausgetauscht werden. Ein Methodenaufruf muss alle notwendigen Daten *by value* mitliefern, damit die Methode ausgeführt werden kann. Referenzen müssen als sogenannte Stummel-Objekte übertragen werden, Zugriffe auf die »Referenzen« bedeuten immer Netzverkehr.

Rundreisen

Im Zusammenhang mit Methodenaufrufen über ein Netz wird häufig der Begriff **Rundreise** (*round trip*) benutzt. Damit ist der folgende Zyklus gemeint:

- Serialisierung des Aufrufs durch den Client
- Übertragung über das Netz
- Deserialisierung durch den Server
- Ausführung der Methode

¹Teile des folgenden Textes wurden mit freundlicher Genehmigung des W3L-Verlags aus [ZiAr10, S. 307 ff.] übernommen.

15.8 Entwurfskonzepte für verteilte Anwendungen I

- Serialisierung des Rückgabewertes durch den Server
- Übertragung über das Netz
- Deserialisierung durch den Client

Je nachdem, was die auf dem Server aufgerufene Methode tut, dauert die Vor- und Nachbereitung eines Aufrufs und die Übertragung über das Netz viel länger als die Ausführung der Methode selbst. Rundreisen über das Netzwerk sind teuer.

Beim Entwurf einer verteilten Anwendung muss dies berücksichtigt werden. Der von einem Server zu erbringende Dienst muss so konzipiert werden, dass ein Client ihn mit möglichst wenigen Methoden aufrufen und damit Rundreisen nutzen kann. Dies bedeutet eine Abkehr vom klassischen, nicht verteilten objektorientierten Entwurf. Statt vieler, relativ kleiner Methoden und vieler Objekte arbeitet man besser mit größeren Objekten und Methoden, die viel Funktionalität beinhalten.

Für die Entwicklung verteilter objektorientierter Anwendungen stehen eine Reihe von Entwurfsmustern zur Verfügung. Die Ausrichtung der Architektur an diesen Mustern führt zu performanten und trotzdem leicht verständlichen Anwendungen. In [Fow02] werden viele speziell auf Client-Server-Anwendungen zugeschnittene Entwurfsmuster vorgestellt.

Ein Client muss auf eine Vielzahl von Server-Objekten zugreifen können. Die Beschaffung von Referenzen auf Server-Objekte kann mit Hilfe eines Fabrik-Dienstes effizient und komfortabel erfolgen:

- »Fabrik-Dienst«, S. 307

Das Auslesen von Attributen über das Netz kann mit Wert-Objekten beschleunigt werden. Sie lösen die klassischen `get()`- und `set()`-Methoden ab:

- »Wert-Objekte«, S. 309

Eine Fassade bietet einem Client eine komfortable, performante, speziell auf seine Bedürfnisse zugeschnittene Schnittstelle für ein Server-Objekt:

- »Fassaden«, S. 312

Abkehr von der
OO

Entwurfsmuster

i
Fabrik

Wert-Objekte

Fassaden

15.8.1 Fabrik-Dienst

Eine reale Anwendung verwaltet normalerweise eine große Menge von Objekten. Ein großes Kreditinstitut führt Zehntausende von Konten, ein Online-Handel vertreibt eine ähnlich große Zahl von Produkten.¹

In einer verteilten Anwendung müssen Clients auf jedes einzelne verwaltete Objekt zugreifen können.

Suche über
Schlüssel

¹ Teile des folgenden Textes wurden mit freundlicher Genehmigung des W3L-Verlags aus [ZiAr10, S. 308 ff.] übernommen.

I 15 Arten der Netzkommunikation

Zur Abfrage eines Kontostandes muss ein Client auf das gewünschte Konto-Objekt zugreifen und die entsprechende Methode aufrufen können. Die Anwendung muss daher so konzipiert sein, dass Clients entsprechende Suchfunktionen zur Verfügung stehen, die das gewünschte Objekt zurückliefern. In den meisten Fällen verfügen die Objekte über eindeutige Merkmale, die eine Suche ermöglichen. Ein Konto hat eine Kontonummer, die es von allen anderen Konten unterscheidet. Ein Versandhandel wird für seine Produkte eindeutige Artikelnummern verwalten.

Namensdienst Für die Suche nach Objekten auf anderen Computersystemen steht z.B. in Java-RMI der sogenannte RMI-Namensdienst zur Verfügung (siehe »RMI«, S. 220). Er liefert zu einem Namen ein Objekt. Daher könnte der Namensdienst als Suchdienst für die Objekte einer Anwendung dienen. Jedes Konto-Objekt wird mit seiner Kontonummer als Name beim Namensdienst registriert. Der Zugriff auf ein bestimmtes Konto ist für den Client einfach.

Nachteile Allerdings ergeben sich daraus folgende Nachteile:

- Alle Objekte müssen im Arbeitsspeicher liegen. Eine Bank, um beim Beispiel zu bleiben, müsste für alle Konten Objekte anlegen. Der Server bräuchte einen riesigen Arbeitsspeicher.
- Der Namensdienst erlaubt nur ein einziges Suchkriterium für Objekte. Nach einem Produkt sollte über die Artikelnummer, die Bezeichnung oder über andere Eigenschaften gesucht werden können.
- Zusätzliche Funktionen, wie das Anlegen neuer Objekte oder das Löschen nicht mehr benötigter, bietet der Namensdienst nicht. Sie müssen als eigenständiger Dienst realisiert werden.

Fabrik-Dienst Um diese Nachteile zu vermeiden, wird beim Namensdienst lediglich ein **Fabrik-Dienst** registriert. Er bietet Methoden zum Suchen nach Objekten über verschiedene Kriterien an und kann Objekte erzeugen oder löschen. Je nach Bedarf kann der Fabrik-Dienst im Hintergrund auf eine Datenbank zugreifen, Objekte dort suchen und nur benötigte Daten in den Arbeitsspeicher laden.

Zugriff durch den Client Clients greifen in einem zweistufigen Verfahren auf Fachkonzept-Objekte zu. Der Client fordert vom Namensdienst zunächst eine Referenz auf ein Fabrik-Objekt an. Dieses ist unter einem systemweit bekannten Namen registriert und braucht auf dem Server nur einmal vorhanden zu sein. Der Client braucht die Referenz nur einmal anzufordern und speichert sie für die gesamte Laufzeit. Im zweiten Schritt sucht der Client über Methoden des Fabrik-Objekts nach den eigentlich benötigten Objekten oder legt neue an.

Anwendungsspezifischer Namensdienst Mit Hilfe eines Fabrik-Objekts wird der systemnahe Namensdienst von der anwendungsnahmen Objektverwaltung entkoppelt. Die Abb. 15.8-1 zeigt den Kontrollfluss in einer verteilten Anwendung mit einem Fabrik-Dienst.

15.8 Entwurfskonzepte für verteilte Anwendungen I

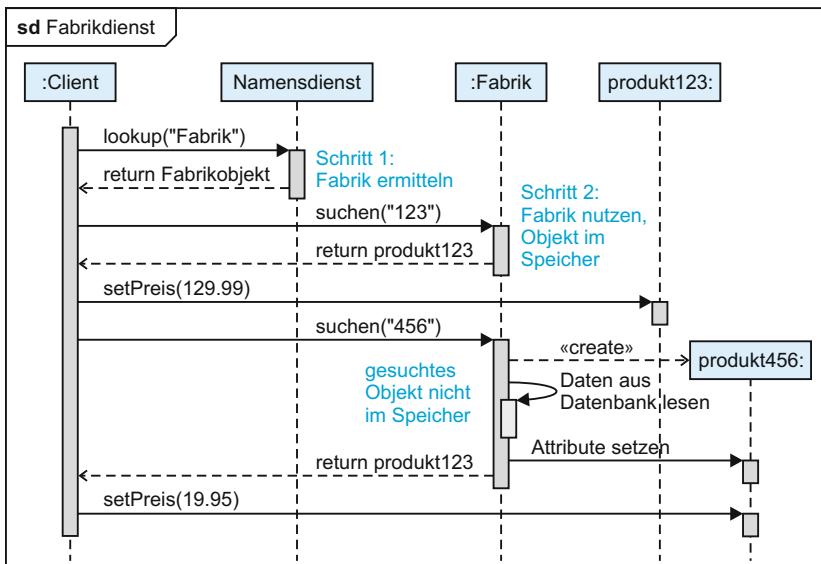


Abb. 15.8-1: Ein Fabrikdienst bietet entfernten Clients die Möglichkeit nach Objekten zu suchen oder neue zu erzeugen. Beim Namensdienst wird nur der Fabrikdienst registriert, nicht jedoch einzelne Objekte.

15.8.2 Wert-Objekte

Rundreisenüber das Netz sind sehr zeitaufwändig. Je mehr Methodenaufrufe zur Nutzung eines Dienstes erforderlich sind, desto langsamer wird die Anwendung.¹ Beim Entwurf einer verteilten Anwendung muss dies berücksichtigt werden. Der von einem Server zu erbringende Dienst muss so konzipiert werden, dass ein Client ihn mit möglichst wenigen Methodenaufrufen und damit Rundreisen nutzen kann.

Eine Anwendung in einer Bank stellt die verwalteten Kunden-Konten durch Objekte der Klasse Konto dar (Abb. 15.8-2). Die Konto-Objekte sollen auf dem Server liegen. Der Client soll über entfernte Aufrufe die Daten des Kontos abfragen können. Bei einer lokal laufenden Anwendung gäbe es für jedes Attribut der Klasse Konto eine `get()`- und eine `set()`-Methode. Um die Daten eines Kontos anzeigen zu können, ruft der Client nacheinander die Attribute ab.

RMI (siehe »RMI«, S. 220) verbirgt z.B. die Verteilung der Anwendung. Die Aufrufe der `get()`- und `set()`-Methoden erscheinen dem Client wie lokale Aufrufe. Sie dauern allerdings Größenordnungen länger als lokale Aufrufe. Wenn der Server Anfragen von vielen Clients erhält, zwingt dieser Entwurf jede noch so leistungsfähige Hardware in die Knie. Die Netzlast wird das Gesamtsystem ausbremsen und zum Stillstand bringen.

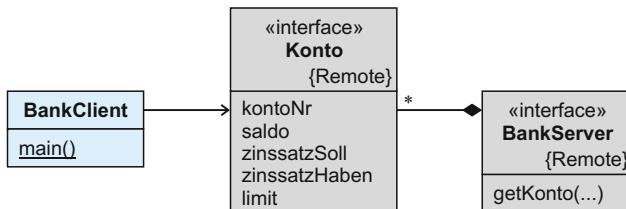
Beispiel 1a:
get() & set()
Methoden

Langsame Aufrufe

¹ Teile des folgenden Textes wurden mit freundlicher Genehmigung des W3L-Verlags aus [ZiAr10, S. 310 ff.] übernommen.

I 15 Arten der Netzkommunikation

Abb. 15.8-2: Die Remote-Schnittstelle Konto ist nicht für Zugriffe durch entfernte Clients optimiert. Zur Abfrage von Attributen sind viele Methodenaufrufe erforderlich.



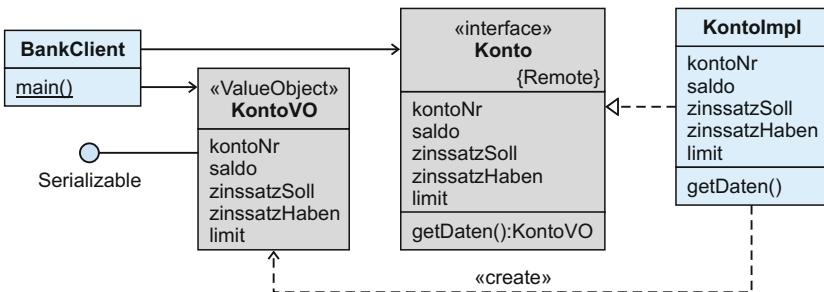
Für eine verteilte Anwendung ist es daher sehr wichtig, dass die Anzahl der entfernten Methodenaufrufe möglichst klein gehalten wird. Wenn weniger Aufrufe zur Erbringung einer Dienstleistung zur Verfügung stehen, muss jeder Aufruf mehr leisten. Methoden müssen daher grobgranularer ausgelegt werden.

Parameter übergeben

Ein wichtiger Aspekt dabei ist die Übergabe von Parametern. Einer entfernten Methode müssen mit einem Aufruf alle benötigten Parameter übergeben werden. Analog muss die Methode alle für den Client wichtigen Daten als Rückgabewert zurückliefern. Für die Realisierung bietet sich z.B. das **Wert-Objekt-Entwurfsmuster** (*value object pattern*) an [Fow02]. Für die Übergabe von Parametern wird eigens eine *zusätzliche Klasse* eingeführt. Sie enthält Attribute für die benötigten Daten. Objekte dieser Klassen sind serialisierbar und können so vom Client zum Server geschickt werden. Für die Rückgabewerte von Methoden werden ebenfalls Wertobjekte eingesetzt. Die Schnittstelle eines Servers enthält statt mehrerer Methoden mit einzelnen Parametern eine einzelne Methode, die in einem Wertobjekt alle Parameter auf einmal übergeben bekommt.

Beispiel 1b Für die Klasse Konto wird eine zusätzliche Klasse KontoVO eingeführt (*VO = value object*). Sie enthält Attribute für die wichtigsten Daten eines Kontos. Die Klasse Konto erhält zwei weitere Operationen *KontoVO getDaten()* und *void setDaten(KontoVO daten)* (Abb. 15.8-3). Ein entfernter Client kann nun mit einem einzigen Aufruf alle Daten des Kontos abfragen. Je mehr Attribute abgefragt werden müssen, desto größer ist die Zeitsparnis.

Abb. 15.8-3: Die Remote-Schnittstelle Konto bietet eine Methode, mit der alle Daten des Objekts durch einen einzigen Aufruf abgefragt werden können. Sie werden als Wertobjekt zum Client geschickt.



15.8 Entwurfskonzepte für verteilte Anwendungen I

Die Größe der bei einem Aufruf zu übertragenden Daten steigt durch die Verwendung von Wertobjekten natürlich an. Wenn es sich dabei allerdings nicht um viele Kilobytes handelt, hat das kaum Auswirkungen auf die Übertragungs-Geschwindigkeit. Ein Datenpaket, in dem der serialisierte Aufruf über das Netz transportiert wird, hat eine feste Größe von etwa 8 Kilobyte. Umfasst ein Aufruf weniger Daten, muss das Paket mit Platzhaltern aufgefüllt werden, der Netzverkehr wird dadurch also nicht geringer. Die meisten in der Praxis vorkommenden Wertobjekte passen in ein Datenpaket. Der Netzverkehr erhöht sich gegenüber einzelnen Parametern nicht.

Geschwindigkeitsvergleich

Der Geschwindigkeitsvorteil, den ein Wertobjekt bietet, lässt sich leicht messen. Abb. 15.8-4 zeigt die Ausgabe einer Client-Anwendung, die speziell für diesen Zweck geschrieben wurde. Sie arbeitet in drei Schritten:

- Im ersten Schritt werden zehntausend Mal alle Attribute eines Ware-Objekts über RMI ausgelesen.
- Im zweiten Schritt wird nur ein einzelnes Attribut zehntausend Mal ausgelesen.
- Im dritten Schritt wird zehntausend Mal vom Ware-Objekt ein Wertobjekt angefordert.

Client- und Server-Anwendung liefen während des Tests auf demselben Computer in unterschiedlichen JVMs.

Testumgebung

```
M:\ValueObjectPerformance>java LagerClient
10000x alle Attribute einzeln lesen: 4697 ms
10000x die Artikelnr lesen: 851 ms
10000x das Value Object lesen: 1572 ms
M:\ValueObjectPerformance>
```

Abb. 15.8-4: Das Auslesen mehrerer Attribute über das Netz ist bei Verwendung eines Value-Objects wesentlich schneller als das einzelne Auslesen.

Wie zu erwarten war, geht das Lesen eines einzelnen Attributs (Schritt 2) am schnellsten. Interessanterweise dauert es weniger als doppelt so lange, wenn alle Attribute in Form eines Wertobjekts übertragen werden (Schritt 3). Das einzelne Auslesen mehrerer Attribute (Schritt 1) ist die mit Abstand langsamste Variante.

Der Einsatz eines Wertobjekts lohnt sich schon, wenn zwei Attribute auszulesen sind. In einem anders aufgebauten Szenario, insbesondere wenn Client und Server auf weit voneinander entfernten Computern laufen, kann der Test andere Zahlen liefern. Latenzzeiten können die Ausführungszeiten extrem erhöhen. Die Geschwindigkeit der drei Varianten relativ zueinander wird aber in etwa gleich bleiben.

I 15 Arten der Netzkommunikation

15.8.3 Fassaden

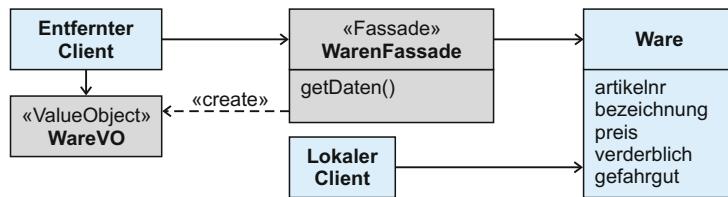
Wenn ein Objekt sowohl von entfernten als auch von lokalen Clients verwendet wird, sollten beide eine für sie passende Schnittstelle erhalten (Abb. 15.8-5):¹

- Der entfernte Client arbeitet mit Wertobjekten und kommt mit wenigen Aufrufen aus (siehe »Wert-Objekte«, S. 309).
- Der lokale Client benutzt eine klassische Schnittstelle mit `get()`- und `set()`-Methoden. Er kommt *ohne* Wertobjekte aus.

Mehrere Sichten

Das **Fassaden-Muster** (siehe »Das Fassaden-Muster (*facade pattern*)«, S. 69) hilft bei der Realisierung mehrerer Sichten auf ein Objekt. Eine Klasse wird zunächst nach der klassischen OO-Lehre mit feingranularen Methoden entwickelt. Für entfernte Clients wird zusätzlich eine Klasse implementiert, die die gleichen Dienste bietet, aber mit Hilfe von grobgranularen Methoden und Wertobjekten. Diese **Fassaden-Klasse** setzt die Aufrufe des entfernten Clients in mehrere lokale Aufrufe der eigentlichen Fachkonzept-Klasse um. Die einzelnen Rückgabewerte von Methoden der Fachkonzept-Klasse werden gesammelt und am Ende in einem weiteren Wertobjekt an den entfernten Client zurückgeschickt.

Abb. 15.8-5: Eine Fassade verkleidet ein Objekt mit einer für entfernte Clients optimierten Schnittstelle.



Integrierte und getrennte Fassaden am Beispiel RMI

Einfache Fassade

Jede RMI-Klasse (siehe »RMI«, S. 220) besitzt bereits eine einfache Fassade: die Remote-Schnittstelle. Beim Entwurf können in der Remote-Schnittstelle grobgranulare Methoden für die entfernten Clients untergebracht werden. Die Implementierungs-Klasse implementiert diese, enthält aber zätzliche, feingranulare Methoden für lokale Clients. Dadurch wird mit relativ wenig Aufwand eine Fassade für entfernte Clients realisiert (Abb. 15.8-6). Lokale Clients haben die Wahl, ob sie die grobgranularen oder die feingranularen Methoden nutzen möchten.

Vermischung von Diensten

Der Nachteil dieser integrierten Variante ist die Vermischung von zwei technisch unterschiedlichen Diensten in einer Klasse. Die Implementierungsklasse muss beide Arten von Methoden implementieren. Sie wird größer und unübersichtlicher.

¹Teile des folgenden Textes wurden mit freundlicher Genehmigung des W3L-Verlags aus [ZiAr10, S. 314 ff.] übernommen.

15.8 Entwurfskonzepte für verteilte Anwendungen I

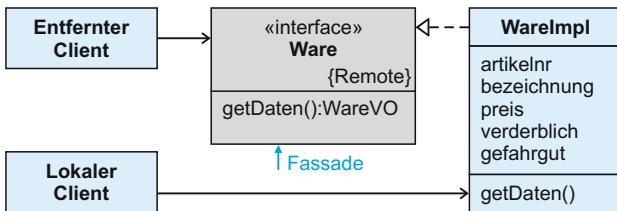


Abb. 15.8-6: Eine einfache Fassade ist eine Schnittstelle, über die entfernte Clients effizient auf ein Objekt zugreifen können.

Eine andere Variante zur Realisierung einer Fassade ist die Auslagerung der Dienste für entfernte Clients in eine eigene Klasse (Abb. 15.8-7). Die eigentliche Fachkonzept-Klasse kann dadurch ohne Rücksicht auf entfernte Clients entwickelt werden. Sie bleibt überschaubar. Die Fassaden-Klasse mit ihrer Remote-Schnittstelle arbeitet als Hüllklasse für die Fachkonzept-Klasse. Sie implementiert nur Dienste für entfernte Clients. Entfernte und lokale Clients können mit dieser Variante unabhängig voneinander betrachtet werden.

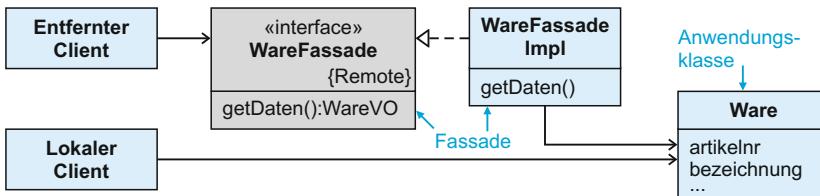


Abb. 15.8-7: Eine Fassade kann von einem separaten Fassaden-Objekt realisiert werden. Dieses greift auf Operationen des verkleideten Objekts zurück. Dadurch wird letzteres nicht durch Fassaden-Code unnötig aufgebläht. Außerdem lassen sich Objekte verkleiden, deren Code nicht verändert werden kann.

Funktionale Fassaden

Die bisher vorgestellten Möglichkeiten zur Realisierung basierten auf einer 1:1-Beziehung zwischen Fassaden-Objekt und Fachkonzept-Objekt. Bei der einfachen Fassade ist dies offensichtlich, da es sich um dasselbe Objekt handelt. Bei der getrennten Fassade gibt es zu jedem Fachkonzept-Objekt ein Fassaden-Objekt. Der Nachteil dieser 1:1-Beziehung ist, dass ein zusätzlicher Dienst realisiert werden muss, der dem entfernten Client eine Referenz auf das Fassaden-Objekt liefert. Dies könnte z. B. ein beim RMI-Namensdienst registrierter Suchdienst sein, der bei einer Bankanwendung eine Methode `getKontoFassade()` zur Verfügung stellt. Als Suchkriterium bietet sich die Kontonummer an. Zum Auslesen der Daten eines Konto-Objekts sind drei Rundreisen über das Netz erforderlich:

- Vom Namensdienst eine Referenz auf den Suchdienst anfordern.
- Vom Suchdienst eine Referenz auf das Fassaden-Objekt anfordern.
- Vom Fassaden-Objekt die Daten anfordern.

I 15 Arten der Netzkommunikation

| | |
|-----------------------|--|
| Schlechte Performance | Wenn von mehreren Konto-Objekten nacheinander Daten abgerufen werden müssen, z.B. um eine Liste darzustellen, fällt der erste Schritt nicht so stark ins Gewicht, denn er braucht nur einmal durchgeführt zu werden. Die Schritte zwei und drei müssen aber für jedes anzugebende Objekt aufgeführt werden. Der Geschwindigkeitsvorteil des Wertobjekts wird dadurch zum Teil aufgehoben. |
| Klassenmethoden | Als Alternative kann der Suchdienst die Methode <code>getDaten()</code> direkt anbieten. Sie bekommt als Parameter die Kontonummer des gewünschten Kontos übergeben. Ein entfernter Client fordert dann nicht erst eine Referenz auf ein Fassaden-Objekt an, sondern direkt die benötigten Daten in Form eines Wertobjekts. Die Methoden des Suchdienstes sind konzeptionell Klassenmethoden, d.h. sie könnten in einem lokalen System als solche realisiert werden. |
| Mengen-Methoden | Für die erwähnte Listen-Darstellung, bei der nacheinander von vielen Konto-Objekten die Daten ausgelesen werden müssen, kann der Suchdienst zusätzlich eine Variante von <code>getDaten()</code> zur Verfügung stellen. Sie erhält als Parameter ein <i>Array</i> oder eine Liste von Kontonummern und liefert entsprechend ein <i>Array</i> oder eine Liste von Wertobjekten zurück. Auf diese Weise können mit einer einzigen Rundreise über das Netz alle benötigten Daten zum Client übertragen werden. Die Belastung des Netzes sinkt dadurch und der Zugriff wird beschleunigt. |
| Funktionale Fassade | Der Suchdienst kann neben den Methoden zum Abrufen von Daten noch weitere Methoden zur Verfügung stellen. Er stellt eine funktionale Fassade zur Abfrage und Manipulation von Konten dar. Ein entfernter Client braucht dann in vielen Fällen keine einzelnen Fassaden-Objekte anzufordern, sondern kann über den Suchdienst einen großen Teil seiner Arbeit verrichten. Die Bezeichnung »Suchdienst« für die Klasse ist dann allerdings <i>nicht</i> mehr zutreffend. |
| Java EE | Ein Dienst, der eine funktionale Schnittstelle zur Verfügung stellt, hinter der sich viele Objekte auf dem Server verbergen können, wird im Kontext der Java EE-Architektur als <i>Session Bean</i> bezeichnet (siehe »Die Java EE-Plattform«, S. 321). |
| Auswahl & Kombination | Welche der vorgestellten Fassaden-Konzepte in der Praxis verwendet werden, hängt von der konkreten Anwendung ab. Eine Bewertung muss immer berücksichtigen, wie ein entfernter Client auf den Server zugreifen muss, d.h. welche Dienstleistungen er benötigt. Die Wahl einer Fassaden-Architektur hängt also von den <i>Use Cases</i> ab, die realisiert werden sollen. Innerhalb einer Anwendung können mehrere Architekturen nebeneinander existieren und schließlich kann auch eine Kombination sinnvoll sein. Einem entfernten Client könnten häufig benötigte Dienste über eine funktionale Fassade zur Verfügung gestellt werden. Für besondere Aufgaben gibt es zusätzlich die Option, ein separates Fassaden-Objekt für ein konkretes Fachkonzept-Objekt anzufordern. |

Grundsätzlich gilt, je komfortabler und flexibler der Zugriff für den Client ist, desto größer ist der Aufwand für die Realisierung auf dem Server. Als Entwickler müssen Sie hier von Fall zu Fall abwägen und dabei immer die Geschwindigkeit der Kommunikation über das Netz im Auge behalten.

Komfort bedeutet
Aufwand

15.9 Vergleich der Konzepte

Im Folgenden werden die wichtigsten Kommunikationsarten anhand von Charakteristika beschrieben und anschließend miteinander verglichen.

Wichtigste Charakteristika der Kommunikationsarten

Sockets

- Kleinster gemeinsamer Nenner des Internets.
- Übermittlung von Byte-Strömen (keine Unterscheidung von Datentypen).
- Es muss ein eigenes Protokoll festgelegt werden.
- Effizient.
- Anwendungsgebiete:
 - TCP: Updates, Backups, Nachladen von Modulen, Kleinstaufgaben.
 - UDP: Kleine, häufig zu übermittelnde Daten.

RMI

- Höheres Abstraktionsniveau verglichen mit Sockets.
- Übermittlung von Daten unterschiedlicher Datentypen.
- Für die Programmiersprache Java, d. h. für homogene Systemlandschaften.
- Objektorientiert und ortstransparent.

CORBA

- Vergleichbar mit RMI, aber programmiersprachenunabhängig.
- Das CORBA-Protokoll IIOP wird von den meisten Firewalls geblockt.

XML-RPC

- Durch die Verwendung von HTTP werden Firewalls und *Proxies* getunnelt.
- Einfachste Form, um Webservices zur Verfügung zu stellen.
- Vorgänger zu SOAP.

SOAP

- SOAP definiert ein Protokoll.

I 15 Arten der Netzkommunikation

- SOAP ist ein Protokollbaukasten, mit dem jeder Entwickler seine eigenen Anwendungsprotokolle entwickelt. Das Protokoll beschreibt den genauen Aufbau einer Anfrage und einer Antwort. Will ein Server weitere Informationen liefern, dann ist ein neuer Webservice notwendig, um mit den bereits installierten Clients kompatibel zu bleiben.
- Hauptanwendungsgebiet: Datenorientierte und lang laufende prozessorientierte Services, synchron und asynchron.

REST

- REST ist eine Architektur, für die es keinen offiziellen Standard gibt.
- Generische Methoden GET, POST, PUT und DELETE.
- Mit URIs globaler Adressraum.
- Eine REST-Anwendung macht Objekte über URIs nach außen sichtbar. Geschäftsobjekte müssen über eine URL erreichbar sein und mit einem Dokument, vorzugsweise in XML, repräsentiert werden können.
- REST ist für Anwendungen mit unbekannter Anzahl von Anwendern und Objekten geeignet.
- REST verwendet für den Transport HTTP und transportiert ganze Repräsentationen von Objekten zum Client. Zwischen spezialisierten lokalen Servern, wie z. B. Applikation-Server und Datenbank, benötigt man effizientere Protokolle wie CORBA, RMI oder DCOM.
- Über eine Firewall kann der Zugriff auf eine REST-Anwendung auf einen Lesezugriff beschränkt werden, indem für externe nur GET-Anfragen erlaubt werden.
- Hauptanwendungsgebiet: Datenorientierte, synchrone und kurz laufende Services.

Vergleich der Kommunikationsarten

Zwei wichtige Kriterien sind bei den verschiedenen Arten der Netzkommunikation zu beachten:

- Architekturstil
- Sicherheit

Architekturstil

Im Zusammenhang mit der Netzkommunikation lassen sich im Wesentlichen drei verschiedene Architekturstile unterscheiden:

- **Schnittstellenorientiert:** Die Netzkommunikation wird so gestaltet, dass auf der Clientseite Methodenaufrufe transparent erfolgen können, d. h. der Client merkt weitgehend nicht, dass die Methoden auf dem Server ausgeführt werden. Als Parameter werden typgesicherte Parameterobjekte (*Remote Procedure Call*) über-

15.9 Vergleich der Konzepte I

geben. Jeder Service hat seine eigene Schnittstelle. Die Kommunikation erfolgt in der Regel synchron, meistens über statisch gebundene Services und über eine direkte physische Verbindung.

- **Nachrichtenorientiert:** Es werden Nachrichten/Dokumente übertragen. Die Kommunikation erfolgt in der Regel asynchron, z. B. indirekt über vermittelnde MOM-Systeme (*Message-oriented Middleware*). Dadurch wird eine lose Kopplung erreicht.
- **Ressourcenorientiert:** Der Schwerpunkt liegt auf dem HTTP-Protokoll und auf per URI identifizierbaren Ressourcen. Ressourcen können in beliebiger Form Informationen und Dienste zur Verfügung stellen.

Sicherheit

Da die Netze von Unternehmen und Organisationen durch Firewalls geschützt werden, spielt bei der Sicherheit der **Schutz bzw. die Durchlässigkeit von Firewalls** eine wesentliche Rolle. Inwieweit kann die Netzkomunikation von Firewalls geblockt bzw. erlaubt werden? Protokolle wie JRMP und IIOP können und werden in der Regel von Firewalls geblockt. Eine Kommunikation ist nur dann möglich, wenn diese Protokolle explizit vom Administrator frei gegeben werden. Verwendet eine Netzkomunikation HTTP als Transportprotokoll, dann ist eine Blockierung durch Firewalls in der Regel unwahrscheinlicher. XML-RPC und SOAP verpacken Anfragen und Antworten in HTTP. Durch dieses *Tunneling* ist ein Objektkommunikation trotz Firewalls möglich. Für Firewalls und Administratoren sehen alle XML-RPC- und SOAP-Nachrichten gleich aus. Um die Bedeutung einer Nachricht zu verstehen, muss z. B. der SOAP-Rumpf betrachtet werden. Bei REST-basierten Anwendungen können Firewalls und Administratoren nach Methoden und URLs filtern. Durch die Beschränkung auf GET-Anfragen kann eine Firewall den Zugriff für Externe auf Lesezugriffe beschränken.

In der Tab. 15.9-1 sind die wichtigsten Kriterien der verschiedenen Kommunikationsarten verglichen.

Alle Kommunikationsarten sind plattform- bzw. betriebssystemunabhängig. Bis auf Java-RMI sind alle Kommunikationsarten unabhängig von einer Programmiersprache.

I 15 Arten der Netzkommunikation

| Kriterien ↓ | Sockets | RMI | CORBA | XML-RPC | SOAP | REST |
|---------------------------------|------------------------------------|-----------------------|-----------------------|-----------------------------------|-----------------------------------|----------------------------|
| Orts-transparenz | Nein | Ja | Ja | Nein | Nein | Nein |
| OO | Nein | Ja | Ja | Bedingt | Ja | Nein |
| Architekturstil | Nachricht.-orientiert | Schnittst.-orientiert | Schnittst.-orientiert | Nachricht.-/schnittst.-orientiert | Schnittst.-/nachricht.-orientiert | Ressourcen-orientiert |
| Server-schnittstelle auf Client | Protokoll des Programmierers | Stummel-Objekt | IDL-Stummel oder DII | execute()-Methode | Stummel-Objekt | REST-Befehle |
| Serverseitiger Zustand | TCP: mit Zustand, UDP: zustandslos | zustands-behaftet | zustands-behaftet | zustands-behaftet | eher zustandslos | zustandslos |
| Namensdienst | Nein | RMI Registry | Name Service | Nein | UDDI | Nein |
| Transport-Protokoll | TCP/UDP | JRMP/ RMI-IIOP | IIOP | HTTP | HTTP, SMTP, JMS... | HTTP |
| Nachrichten-format | Format des Programmierers | Bytes | Bytes | XML | XML + Attachments | Text, XML, HTML, binär... |
| Nachrichten-Protokoll | Protokoll des Progr. | intern | intern | (XML) | SOAP | REST, HTTP |
| Methoden-Signatur | Nicht vorhanden | RMI-Interface | IDL | Abstimmung zw. Client & Server | WSDL | Nicht festgelegt |
| Asyn. Komm. | bei UDP | Nein | Nein | Nein | Ja (JMS, WSN) | Nicht direkt |
| Transaktion, sichere Zustellung | Ja bei TCP | Ja | Ja | Ja | per WSRM & WSTF | keine Unterstützung |
| Firewall | TCP/UDP | i. Allg. blockiert | i. Allg. blockiert | durchlässig | durchlässig | durchlässig, einschränkbar |
| Übertr.-Geschw. | schnell | mittel | mittel | schnell | langsam | schnell |
| Übertr.-Umfang | gering | mittel | mittel | 30 % mehr | mittel | mittel |
| Kopplung | lose | stark | stark | lose | lose | lose |

Tab. 15.9-1:
Vergleich der Kommunikationsarten.

16 Softwaretechnische Infrastrukturen

Soll ein umfangreiches Anwendungssystem entwickelt werden, das viele nichtfunktionale Anforderungen und Randbedingungen erfüllen soll, dann ist zu prüfen, ob eine softwaretechnische Infrastruktur die Realisierung eines solchen Systems wesentlich erleichtern kann. Softwaretechnische Infrastrukturen werden oft auch als softwaretechnische Plattformen – kurz Plattformen – oder technische Referenzarchitekturen bezeichnet. Sie legen die Basistechniken – oft einschl. des Betriebssystems – und damit auch die Programmiersprachen fest. Man spricht auch von *Middleware*.

Gliert man Anwendungssysteme grob in zwei große Kategorien, dann lassen sich Unternehmenslösungen und softwareintensive Systeme unterscheiden.

Für **Unternehmenslösungen** gibt es eine Reihe charakteristischer Anforderungen:

- »Anforderungen an Unternehmensanwendungen«, S. 319

In Abhängigkeit von diesen Anforderungen können eine oder mehrere softwaretechnische Infrastrukturen eingesetzt werden:

- »Die Java EE-Architektur«, S. 321
- »Die .NET-Plattform«, S. 360
- »Infrastrukturen für serviceorientierte Architekturen«, S. 375

16.1 Anforderungen an Unternehmensanwendungen

Software, die zur Unterstützung der Geschäftsabläufe in einem Unternehmen eingesetzt werden soll, unterscheidet sich wesentlich von Software auf einem Einzelplatz-System. **Unternehmensanwendungen** zeichnen sich dadurch aus, dass sie von vielen Benutzern gleichzeitig genutzt werden, verteilt sind (oft über große Entfernung) und mit unternehmenskritischen Daten arbeiten.

In [DePe02] werden folgende Anforderungen an unternehmensexterne Anwendungen formuliert:

■ Mehrbenutzerfähigkeit

Es wird eine Benutzerverwaltung benötigt. Benutzer müssen authentifiziert werden und es muss sichergestellt sein, dass ein Benutzer nur die Teile der Anwendung nutzt, für die er eine Berechtigung hat (siehe »Authentifizierung und Autorisierung«, S. 153).

I 16 Softwaretechnische Infrastrukturen

■ Skalierbarkeit

Die Anzahl der Benutzer ist im Voraus *nicht* bekannt. Wenn eine Anwendung mit der Zeit immer umfangreicher wird, d.h. immer mehr Geschäftsprozesse unterstützt, wird auch die Zahl der Benutzer und die Menge der zu verwaltenden Daten steigen. Steigende Anforderungen an die Leistungsfähigkeit (*performance*) einer Anwendung müssen erfüllt werden können, ohne die Anwendung selbst zu modifizieren, d.h. ganz oder in Teilen neu zu programmieren (siehe »Nichtfunktionale Anforderungen«, S. 109).

■ Verfügbarkeit

Wenn eine Anwendung zur Unterstützung von Geschäftsprozessen eines Unternehmens eingesetzt wird, bedeutet dies meist auch, dass mit einem Ausfall der Anwendung die Geschäftstätigkeit des Unternehmens sehr stark eingeschränkt wird oder ganz zum Erliegen kommt. Es müssen daher Maßnahmen getroffen werden, die diesen Fall ausschließen bzw. sehr unwahrscheinlich machen. Dies lässt sich zwar nicht ausschließlich durch Software realisieren, jedoch muss eine Anwendung mit dem plötzlichen Ausfall bestimmter Teilsysteme zureckkommen (siehe »Zuverlässigkeit«, S. 124).

■ Verbindung mit der Außenwelt

Immer mehr Unternehmen gehen heute dazu über, Geschäftspartner (z.B. Kunden und Lieferanten) in die eigenen Geschäftsprozesse einzubeziehen. Eine Konsequenz ist, dass Benutzer mit einer Anwendung arbeiten, die keine Mitarbeiter des Unternehmens sind. Diese Art von Interaktion findet häufig über das Extranet oder Internet statt. Eine Anwendung muss auch diese Benutzergruppe berücksichtigen. Oft ist es auch erforderlich, dass firmen-interne Anwendungen mit firmen-externen Anwendungen, die z.B. bei Lieferanten in Betrieb sind, interagieren können (siehe »Einflussfaktoren auf die Architektur«, S. 135).

■ Schrittweise Migration

Die Annahme, die Software-Infrastruktur eines Unternehmens könne »von heute auf morgen« komplett umgestellt werden, ist unrealistisch. Eine neue Anwendung wird zunächst nur in Teilen des Unternehmens eingesetzt werden, während andere Teile mit den bisherigen Altsystemen weiter arbeiten. Eine neue Anwendung muss daher so konzipiert sein, dass sie mit vorhandenen Altsystemen harmoniert und eine schrittweise Umstellung unterstützt (siehe »Portabilität«, S. 132).

Eine wichtige Gemeinsamkeit der hier beschriebenen Anforderungen an Unternehmenslösungen ist die Unabhängigkeit von einem konkreten Unternehmen. Es handelt sich um nichtfunktionale Anforde-

rungen. Sie treffen für alle Unternehmenslösungen mehr oder weniger zu, da sie sich nicht auf Geschäftsprozesse beziehen, sondern eher die technischen Randbedingungen betreffen.

Eine Bank benötigt für ihr Computersystem ebenso eine Benutzer-Beispielverwaltung wie ein Versandhaus oder ein Automobil-Hersteller.

16.2 Die Java EE-Plattform

Java EE (*Java Platform, Enterprise Edition* – früher J2EE) spezifiziert eine Softwarearchitektur für Unternehmensanwendungen. Java EE kann als eine Erweiterung von Java SE (*Java Platform, Standard Edition*) angesehen werden, um die Entwicklung von verteilten, transaktionsbasierten, robusten, leistungsfähigen und hoch verfügbaren Anwendungen zu erleichtern. Java EE ist eine Obermenge der Java SE-Plattform, d. h. Java SE-APIs können in jeder Java EE-Komponente benutzt werden.

Zunächst wird die Architektur von Java EE näher betrachtet:

- »Die Java EE-Architektur«, S. 321

Die eigentliche Anwendung – in der Regel die Geschäftslogik oder die Geschäftsprozesse – wird durch *Enterprise Java Beans* realisiert:

- »EJBs«, S. 326

Damit Java EE-Programme ausgeführt werden können, wird ein Java EE-Server benötigt. Ein bekannter Open-Source-Server ist der »JBoss Application Server«. Im kostenlosen E-Learning-Kurs zu diesem Buch wird die Installation beschrieben. Um anwendungsübergreifend auf *Enterprise Java Beans* zugreifen zu können, müssen sie gefunden werden können. Dafür gibt es das *Java Naming and Directory Interface*:

- »JNDI«, S. 328

Jetzt sind alle Vorbereitungen getroffen, um eine *Session Bean* zu programmieren und auf sie zuzugreifen:

- »Erstellung und Nutzung einer Session Bean«, S. 330

Auch die Fallstudie »Kundenverwaltung« kann unter Verwendung von Java EE programmiert werden – sowohl als Client-Server-Anwendung als auch als Web-Anwendung:

- »Fallstudie: KV mit Java EE als Client-Server-Anwendung«, S. 333
- »Fallstudie: KV mit Java EE als Web-Anwendung«, S. 351

16.2.1 Die Java EE-Architektur

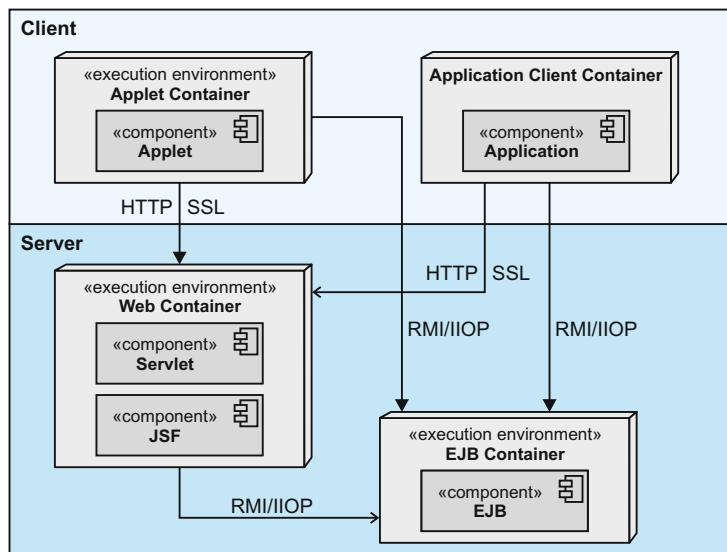
Die Java EE-Architektur spezifiziert verschiedene Container. Container sind Java EE-Laufzeitumgebungen, die verschiedene Services für die Komponenten, die sie beherbergen, zur Verfügung stellen.

I 16 Softwaretechnische Infrastrukturen

Container

Die Java EE-Infrastruktur ist in logische Bereiche gegliedert, die Container genannt werden. Jeder Container hat eine spezielle Rolle, unterstützt eine Menge von APIs und stellt Services für Komponenten zur Verfügung. Container verbergen die technische Komplexität und erhöhen die Portabilität. Die Abb. 16.2-1 zeigt die logischen Beziehungen zwischen den Containern. Die Pfeile repräsentieren die Protokolle, die von den Containern benutzt werden, um gegenseitig aufeinander zugreifen zu können.

Abb. 16.2-1: Die Standard-Java EE-Container.



Komponenten

- **Applet-Container** werden von den meisten Webbrowsern zur Verfügung gestellt, um Applet-Komponenten auszuführen. Der Container stellt eine sichere Ablauf-Umgebung für Applets zur Verfügung (*sandbox*).
- **Web-Container**, auch bekannt als Servlet-Container, stellen Services bereit, um Webanwendungen (bestehend aus Servlets, JSPs, *Servlet Filters*, *Web Event Listeners*, JSP- und JSF-Seiten, Web-Services) zu verwalten, auszuführen und auf HTTP-Anfragen von Web-Clients zu antworten. Servlets unterstützen auch SOAP- und REST-Webservice-Endpunkte.
- **EJB-Container** verwalten die Ausführung von EJBs (*Enterprise Java Beans*), die die Geschäftslogik der Anwendung implementieren. Der Container erzeugt Exemplare von EJBs, verwaltet ihren Lebenszyklus und stellt Services für die Transaktion, Sicherheit,

Nebenläufigkeit, Verteilung, asynchronen Aufruf sowie den Namensservice zur Verfügung. Auf die EJBs kann lokal und entfernt über RMI (oder HTTP für SOAP- und REST-Web-Services) zugegriffen werden.

- **Application Client-Container** (ACC) unterstützen Anwendungen, die auf einem Client ausgeführt werden. Dabei handelt es sich typischerweise um GUI- oder Stapelverarbeitungs-Programme, die Zugriff auf alle Funktionalitäten der Java EE-Plattform haben. ACC unterstützt Java SE-Anwendungen durch Sicherheitsmanagement und Namens-Service.

Services

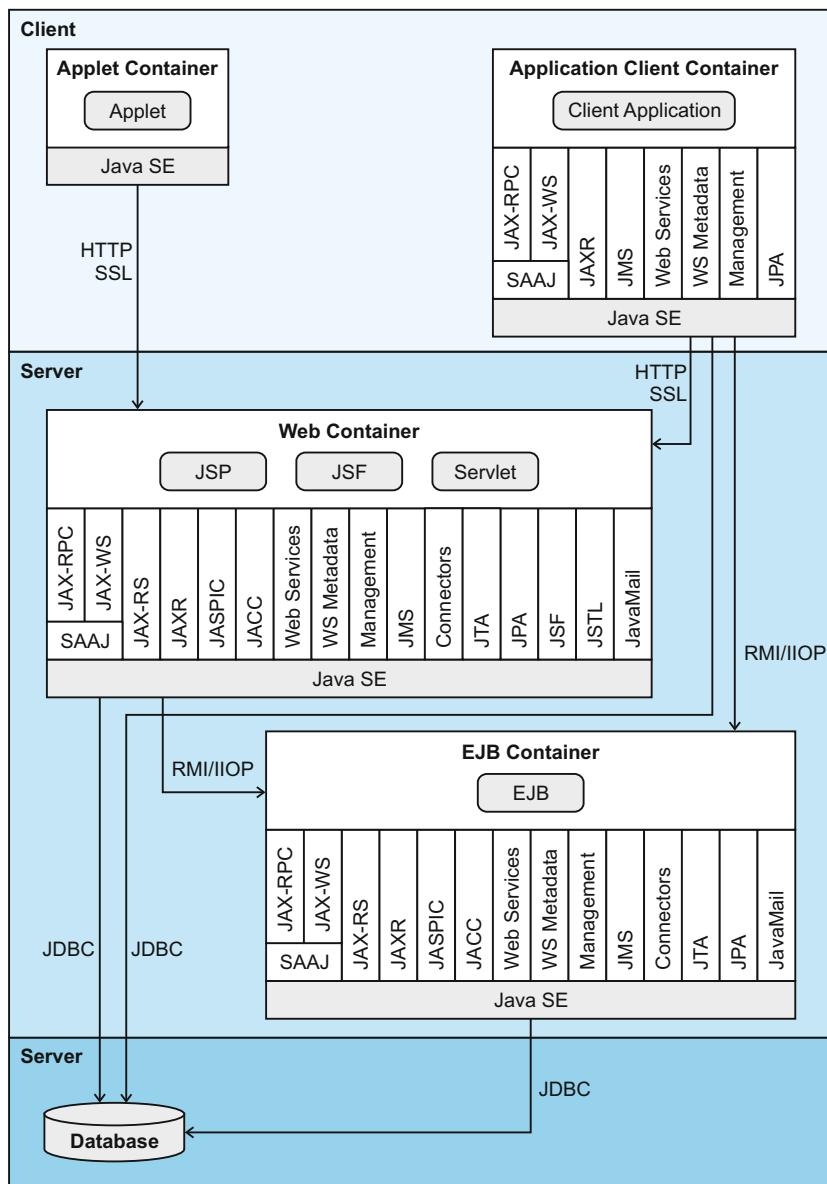
Jeder Container stellt Services für die Komponenten zur Verfügung, die in ihm installiert sind. Die Abb. 16.2-2 zeigt die Services, die von dem jeweiligen Container zur Verfügung gestellt werden.

Java EE stellt folgende Services bereit:

- **JTA (Java Transaction API)**: Erlaubt der Anwendung die Steuerung der Transaktionsverwaltung (siehe »Transaktionen«, S. 177).
- **JPA (Java Persistence API)**: Stellt eine einheitliche und datenbank-unabhängige Schnittstelle für die Abbildung von Objekten auf Relationen bereit (ORM, *object-relational mapping*). Mithilfe der Abfragesprache JPQL (*Java Persistence Query Language*) können Objekte abgefragt werden, die in der zugrunde liegenden Datenbank gespeichert sind (siehe »JPA – Java Persistence API«, S. 431).
- **JMS (Java Message Service)**: Erlaubt es Komponenten asynchron mittels Nachrichten miteinander zu kommunizieren. Es werden zuverlässige Punkt-zu-Punkt-Verbindungen (P2P) als auch das *Publish-Subscribe*-Modell (*pub-sub-model*) unterstützt.
- **JCA (Java EE Connector Architecture)**: Erlaubt es, andere Systeme transparent zu integrieren (*Enterprise Application Integration*). Dabei kann es sich um Datenbanken, Mainframes oder ERP-Systeme handeln.
- **JavaMail**: Erlaubt den Zugriff auf E-Mail-Dienste wie SMTP, POP3, IMAP oder NNTP.
- **JAF (JavaBeans Activation Framework)**: Erlaubt es, die verschiedenen MIME-Typen zu erkennen und zu bearbeiten. JAF ist Bestandteil von **Java SE** und wird von JavaMail benutzt. Teil von Java SE
- **JNDI (Java Naming and Directory Interface)**: Gemeinsame Schnittstelle, mit der alle Java-Klassen auf Namens- und Verzeichnisdienste zugreifen können. JNDI ist Bestandteil von **Java SE**. Über JNDI erfolgt insbesondere der Zugriff auf Java EE-Komponenten.
- **JMX (Java Management Extensions)**: Unterstützt die Verwaltung und Überwachung von Java-Anwendungen. Bestandteil von **Java SE**. Management

I 16 Softwaretechnische Infrastrukturen

Abb. 16.2-2: Java EE-Container und ihre Services (in Anlehnung an [Gonc09, S.7]).



Sicherheits-
Services

- **JAAS (Java Authentication and Authorization Service):** Unterstützt die Authentifizierung und Autorisierung von Benutzern (siehe »Authentifizierung und Autorisierung«, S. 153).
- **JACC (Java Authorization Contract for Containers):** Erlaubt die Definition von Sicherheitsrichtlinien für die verschiedenen Container.

- **JAX-WS** (*Java API for XML Web Services*): Unterstützt die Erstellung von Webservices und der zugehörigen Clients, die über XML kommunizieren, z. B. über SOAP (siehe »Webservices bereitstellen und nutzen mit JAX-WS«, S. 267). JAX-RPC (*Java API for XML-based RPC*) dient dazu, *Remote Procedure Calls* auf XML-Basis auszuführen. Bestandteil von JAX-WS.
- **SAAJ** (*SOAP with Attachments API for Java*): Ermöglicht es, SOAP-Nachrichten mit dem Paket javax.xml.soap zu erzeugen und zu senden.
- **JAX-RS** (*Java API for RESTful Web Services*): Unterstützt die Erstellung von Web-Services und der zugehörigen Clients, die über REST kommunizieren (siehe »Webservices mit JAX-RS«, S. 289).
- **JAXP** (*Java API for XML Processing*): Unterstützt das Parsing von Dokumenten mit SAX- und DOM-APIs sowie XSLT.
- **StAX** (*Streaming API for XML*): Cursorbasierte XML-Verarbeitung zur Ergänzung der DOM- und SAX-Parser.

Webservices

XML-Verarbeitung

Protokolle

Die in den Containern installierten Komponenten können durch verschiedene Protokolle aufgerufen werden. Beispielsweise kann ein Servlet, das sich in einem Web-Container befindet, sowohl über HTTP als auch über einen Webservice mit einem EJB-Endpunkt, installiert in einem EJB-Container, aufgerufen werden. Folgende Protokolle werden unterstützt:

- **HTTP und HTTPS** (HTTP in Kombination mit SSL – *Secure Sockets Layer*): Die clientseitige API ist im Paket java.net in Java SE definiert. Die serverseitige API ist definiert in Servlets, JSPs, JSF-Schnittstellen, SOAP- und REST-Web-Services.
- **RMI-IIOP**: RMI (*Remote Method Invocation*) erlaubt es, entfernte Objekte unabhängig vom zugrunde liegenden Protokoll aufzurufen (siehe »RMI«, S. 220). RMI-IIOP ist eine Erweiterung, um CORBA zu integrieren (siehe »CORBA«, S. 241).

Java EE-Server

Es gibt zahlreiche Implementierungen für Java EE-Server. Bekannte Open-Source-Server sind:

- GlassFish
 - Sun GlassFish Enterprise Server (Referenzimplementierung von Sun auf der Basis von GlassFish)
 - JBoss Application Server (Apache Tomcat optional)
- Verbreitete kommerzielle Server sind:
- IBM WebSphere Application Server (WAS)
 - Oracle Application Server

I 16 Softwaretechnische Infrastrukturen

- BEA WebLogic
- SAP NetWeaver Application Server

Literatur [Gonc09]

16.2.2 EJBs

Im EJB-Container (siehe »Die Java EE-Architektur«, S. 321), der sich auf dem Server befindet, werden EJBs (*Enterprise Java Beans*) verwaltet, die die Geschäftslogik der Anwendung implementieren.

EJB-Typen

Es werden folgende Typen von EJBs unterschieden:

- *Session Beans*: Implementieren die Anwendung – in der Regel die Geschäftslogik – und stellen sie in Form von Services zur Verfügung.
- *Message Driven Beans* (MDBs): Ermöglichen eine asynchrone Kommunikation von EJBs über den *Java Message Service* (JMS). Sie werden verwendet, um externe Systeme durch den Empfang von asynchronen Mitteilungen zu integrieren.

Hinweis

Zusätzlich gab es noch *Entity Beans*, die eine Schnittstelle zur Datenbank herstellen. Heute wird dafür JPA verwendet (siehe »JPA – Java Persistence API«, S. 431).

3 Arten

Session Beans

Von den *Session Beans* gibt es drei Ausprägungen:

- *Stateless Session Beans*: Sind zustandslos und vergessen nach der Durchführung der von einem Client gewünschten Methode den Client wieder. Ruft ein Client eine weitere Methode derselben Klasse auf, dann arbeitet er wahrscheinlich mit einem anderen Objekt der Klasse als beim vorherigen Methodenaufruf. Jedes Objekt kann von jedem Client benutzt werden.

Tipp

Wenn Sie sich – beispielsweise in einer Web-Anwendung – den Sitzungszustand bereits an anderer Stelle merken, z.B. in einem Servlet, dann verwenden Sie immer *Stateless Session Beans*.

- *Stateful Session Beans*: Verfügen über ein sogenanntes »Konversationsgedächtnis«, in dem sie sich den Zustand ihrer Interaktion mit dem Client »merken«. Ein Client arbeitet stets mit dem gleichen Objekt der Klasse. Wegen der Zustandsspeicherung und der korrekten Zuordnung eines Objekts zum dazugehörigen Client ist der Aufwand deutlichressourcenintensiver als bei *Stateless Session Beans*.

Beispiel

In einer Web-Anwendung werden für einen Warenkorb *Stateful Session Beans* benötigt, da der Warenkorb individuell für jeden Benutzer ist.

- *Singleton Session Beans*: Werden nur einmal für die gesamte Anwendung erzeugt. Alle Clients arbeiten also mit demselben Objekt.

EJBs können auch als Webservice-Endpunkte benutzt werden, siehe »SOAP«, S. 262, und »REST«, S. 286.

Eine EJB soll nach Aufruf der Methode `gibGruss()` den Text "Hello World" zurückgeben. In einer Schnittstelle werden die öffentlichen Methoden festgelegt:

```
package interfaces;
//Remote-Schnittstelle der Stateless Session Bean
public interface HelloWorld
{
    //Von der EJB bereitgestellter Service
    public String gibGruss();
}
```

Die Implementierung der Schnittstelle kann wie folgt aussehen:

```
package ejb;

import javax.ejb.Remote;
import javax.ejb.Stateless;
import java.io.Serializable;
import java.rmi.RemoteException;

@Stateless
@Remote(interfaces.HelloWorld.class)
public class HelloWorldBean implements HelloWorld
{
    public String gibGruss() throws RemoteException
    {
        return "Hello EJB World";
    }
}
```

Sollen individualisierte Grußnachrichten zurückgegeben werden, dann muss eine *Stateful Session Bean* verwendet werden:

```
package interfaces;
//Remote-Schnittstelle der Stateful Session Bean
public interface HelloWorld
{
    //Von der EJB bereitgestellte Services
    public speichereGruss(String gruss);
    public String gibGruss();
}

package ejb;

import javax.ejb.Remote;
import javax.ejb.Stateful;
import java.io.Serializable;
import java.rmi.RemoteException;
```

I 16 Softwaretechnische Infrastrukturen

```
@Stateful  
@Remote(interfaces.HelloWorld.class)  
public class HelloWorldBean implements HelloWorld  
{  
    //Merke den Gruss  
    private String gruss = "Sende mir einen Gruss";  
  
    public void speichereGruss(String gruess)  
    {  
        this.gruss = gruss;  
    }  
    public String gibGruss() throws RemoteException  
    {  
        return gruss;  
    }  
}
```

Annotationen Durch die Annotation @Stateless bzw. @Stateful wird eine Java-Klasse zu einer EJB.

Die Annotation @Remote(paketname.Schnittstellenname.class) gibt an, dass diese EJB die angegebene Remote-Schnittstelle realisiert.

Literatur [Sun09, S. 252 ff.]

16.2.3 JNDI

Namensdienst

Ein Namensdienst (*naming service*) stellt einen Service dar, der Namen mit Ressourcen verbindet und diese dann einheitlich und einfach über ihre Namen finden lässt. In der Softwarewelt gibt es aber nicht nur einen einzigen einheitlichen Namensdienst, sondern jede bedeutende Entwicklungs- und Laufzeitumgebung hat ihren eigenen Dienst dazu entwickelt. Beispiele dafür sind die Registrierung von RMI (*Remote Method Invocation*) oder der Verzeichnisdienst LDAP (*Lightweight Directory Access Protocol*).

JNDI **JNDI** (*Java Naming and Directory Interface*) soll nun – für die Ressourcen, die in der Java-Programmierwelt benutzt werden – eine ordnende und vereinheitlichende Instanz für verschiedene Namens- und Verzeichnisdienste bilden, also ein Art Meta-Dienst.

Dies kann dadurch realisiert werden, dass über die Anforderungen einzelner Dienste ein allgemeiner Service gelegt wird, der dann bei der Programmierung ausschließlich angesprochen wird, ohne noch die darunterliegenden besonderen Dienste berücksichtigen zu müssen.

Das Prinzip ist einfach: Ein Anwendungsprogramm greift über eine Programmierschnittstelle (API) auf den *Java Naming Manager* zu, der wiederum über ein *Service Provider Interface* (SPI) zu den verschiedenen Namens- und Verzeichnisdiensten Verbindung aufnimmt. Die Abb. 16.2-3 verdeutlicht diese Vorgehensweise.

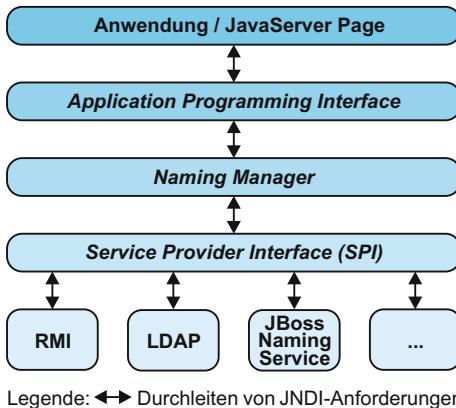


Abb. 16.2-3:
Systemaufbau des
JNDI-Dienstes.

Auch Java EE ist in diesem Sinne ein *Service Provider* (SP) und implementiert die SP-Schnittstelle. Im Falle des »JBoss Application Server« handelt es sich um den *JBoss Naming Service*.

Verzeichnisdienst

Ein Verzeichnisdienst (*Directory Service*) ist eine Erweiterung eines Namensdienstes und ermöglicht es, Objekten nicht nur einen Namen, sondern auch weitere Attribute zuzuordnen, z. B. Lese- und Schreibrechte bei einer Datei. Über die Attribute kann zusätzlich gesucht werden.

Einsatz von JNDI in einer Java EE-Anwendung

In einer Java EE-Anwendung wird JNDI zum Lokalisieren von Ressourcen innerhalb des Anwendungsservers benutzt. Auch beim anwendungsübergreifenden Zugriff auf EJBs wird JNDI genutzt.

Dabei ist es nicht nur möglich, EJBs innerhalb des Anwendungsservers zu lokalisieren, sondern ein externer Client kann mithilfe von JNDI auf EJBs zugreifen, die von einem entfernten Anwendungsserver verwaltet werden.

JNDI-Zugriff

Die Nutzung von JNDI zum Zugriff auf Ressourcen (*JNDI Lookup*) erfolgt immer nach dem folgenden Schema:

- Zunächst muss als Einstiegspunkt in das System, auf das zugegriffen werden soll, ein sogenannter initialer Kontext erzeugt werden.
- Abhängig vom Dienst, auf den zugegriffen werden soll, werden bei der Erzeugung des initialen Kontexts unterschiedliche Konfigurationsdaten übergeben.

I 16 Softwaretechnische Infrastrukturen

- Wurde der Kontext erzeugt, dann kann auf den gewünschten Namens- oder Verzeichnisdienst zugegriffen werden.
- Vor Beendigung des Programmes sollte der Kontext geschlossen werden, um die beanspruchten Ressourcen wieder freizugeben.

Beispiel Der folgende Codeausschnitt zeigt, wie zunächst die Konfiguration für den Zugriff auf die Ressourcen eines JBoss-Anwendungsservers definiert und anschließend der initiale Kontext erzeugt wird:

```
Java try
{
    //Konfiguration
    Hashtable<String, String> konfig =
        new Hashtable<String, String>();
    konfig.put(Context.INITIAL_CONTEXT_FACTORY,
        "org.jnp.interfaces.NamingContextFactory");
    konfig.put(Context.PROVIDER_URL, "localhost:1099");
    //JBoss-Standard-Port
    konfig.put("CONTEXT.URL-PKG_PREFIXES,
        "org.jboss.naming:org.jnp.interfaces");

    //Kontext erzeugen
    Context ctx = new InitialContext(konfig);

    //Zugriff auf eine EJB mit dem Namen MeineEJB
    MeineEJB ejb = (MeineEJB) ctx.lookup("MeineEJB/remote");

    ctx.close();
}
catch(NamingException e)
{
    //Wird erzeugt, wenn der Zugriff auf eine Ressource
    //fehlschlägt
    //Fehlerbehandlung
}
```

16.2.4 Erstellung und Nutzung einer *Session Bean*

Erstellung einer *Session Bean*

Session Beans lassen sich mit wenig Aufwand aus normalen Java-Klassen erzeugen. Die hierfür benötigten Schritte sind davon abhängig, ob die Dienste der *Session Bean* nur lokal oder auch von Remote-Clients genutzt werden sollen. Es lassen sich lokale und Remote-Schnittstellen unterscheiden. Die in einer lokalen Schnittstelle definierten Methoden sind nur innerhalb derselben Anwendung sichtbar, während die Methoden einer Remote-Schnittstelle von anderen Anwendungen innerhalb des Anwendungsservers – aber auch aus entfernten JVMs – heraus aufgerufen werden können.

16.2 Die Java EE-Plattform I

Eine *Session Bean* kann sowohl beide Schnittstellenarten gleichzeitig als auch nur eine Variante implementieren.

Im Folgenden wird die Erstellung einer *Session Bean* für Remote-Clients erläutert, die das Speichern und Laden von Kundenobjekten ermöglicht.

Um aus einer einfachen Java-Klasse eine *Session Bean* für Remote-Clients zu machen, sind folgende Schritte erforderlich:

- 1 Für die Java-Klasse muss eine Schnittstelle erstellt werden, die alle öffentlichen Methoden der Klasse enthält, das sogenannte *Business Interface* oder *Remote Interface*.
- 2 Die Java-Klasse muss mithilfe von Annotationen als *Session Bean* gekennzeichnet werden.

Der folgende Codeausschnitt zeigt das *Business Interface* der *Session Bean* für eine Kundenverwaltung:

```
package interfaces;

public interface Kundenverwaltung
{
    public void speichereKunden(Kunde einKunde);
    public void loescheKunden(Kunde einKunde);
    public Kunde sucheKunden(int kundennummer);
}
```

Die Java-Klasse, die die Schnittstelle implementiert, wird mithilfe der Annotation `@Stateful` als *Stateful Session Bean* gekennzeichnet. Die Annotation `@Remote` definiert die Remote-Schnittstelle der *Session Bean*. Mithilfe der Annotation `@Local` kann eine lokale Schnittstelle angegeben werden:

```
package session;                                         Java

import javax.ejb.Remote;
import javax.ejb.Stateful;

@Stateful
@Remote(interfaces.Kundenverwaltung.class)
public class KundenverwaltungBean
    implements KundenVerwaltung
{
    public void speichereKunden(Kunde einKunde)
    {
        //Implementierung
    }
    public void loescheKunden(Kunde einKunde)
    { //...}
    public Kunde sucheKunden(int kundennummer)
    { //...}
}
```

I 16 Softwaretechnische Infrastrukturen

Hinweis

Ab EJB 3.1 benötigt eine lokale *Session Bean* kein lokales *Business Interface* mehr. *Session Beans* ohne lokales *Business Interface* werden automatisch als *Local Session Beans* angesehen. Es ist jedoch anzuraten in jedem Fall ein *Business Interface* zu erstellen, um den Unterschied zu normalen Java-Klassen zu verdeutlichen.

Nutzung einer *Session Bean*

Session Beans mit einem *Remote Interface* können von allen Klassen innerhalb des Anwendungsservers und auch von Clients, die in anderen JVMs ausgeführt werden, aufgerufen werden.

Der Zugriff erfolgt mithilfe von JNDI (siehe »JNDI«, S. 328), wobei der zum Auffinden der *Session Bean* erforderliche JNDI-Pfad sich – je nach verwendetem Anwendungsserver unterscheiden kann.

Expliziter JNDI-Lookup

Der folgende Codeausschnitt zeigt, wie mithilfe eines JNDI-Lookups eine Instanz der *Session Bean* Kundenverwaltung erzeugt wird.

Beispiel 1b Das erzeugte Objekt ist vom Typ der Remote-Schnittstelle, da der Client die konkrete Implementierung der *Session Bean* nicht kennt:

```
import interfaces.Kundenverwaltung;

public class MeinClient
{
    public void speichereKunden(Kunde einKunde)
    {
        //Konfiguration festlegen
        ...

        //JNDI Lookup der Session Bean
        Context ctx = new InitialContext(konfig);
        Kundenverwaltung kv = (Kundenverwaltung)
            ctx.lookup("KundenverwaltungBean/remote");
        ctx.close();
        //Arbeiten mit der Session Bean
        kv.speichereKunden(einKunde);
    }
}
```

Dependency Injection

Für Clients innerhalb desselben Anwendungsservers existiert in Java EE die Möglichkeit der sogenannten *Dependency Injection*. Dafür wird ein Attribut vom Typ der Local-Schnittstelle mit der Annotation @EJB versehen.

Der EJB-Container führt dann beim ersten Zugriff auf das Attribut automatisch einen JNDI-Lookup durch, sodass dieser nicht explizit vom Entwickler formuliert werden muss.

16.2 Die Java EE-Plattform I

Dieses Beispiel zeigt den Zugriff auf die Session Bean Beispiel 1c Kundenverwaltung mithilfe von *Dependency Injection*.

```
import interfaces.Kundenverwaltung;

public class MeinClient
{
    @EJB
    Kundenverwaltung kv;

    public void speichereKunden(Kunde einKunde)
    {
        kv.speichereKunden(einKunde);
    }
}
```

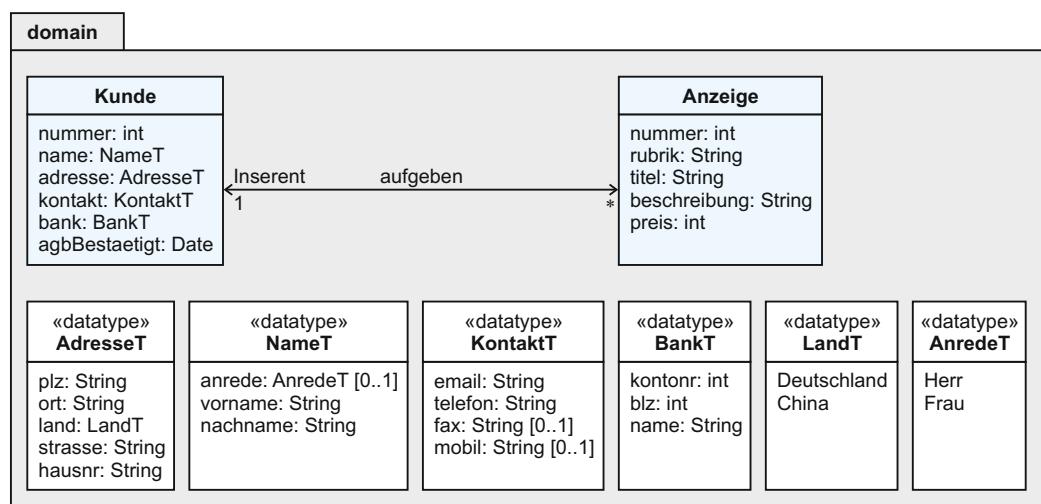
Dependency Injection kann nicht nur in anderen Session Beans, sondern auch in Web Beans genutzt werden.

[Sun09, S. 257 ff.]

Literatur

16.2.5 Fallstudie: KV mit Java EE als Client-Server-Anwendung

Die Fallstudie »Kundenverwaltung« wird in erweiterter Form als »Anzeigenverwaltung« als Client-Server-Anwendung mit Java EE realisiert. Die Abb. 16.2-4 zeigt die Spezifikation in Form eines UML-OOA-Diagramms (siehe auch »Fallstudie: KV – Überblick«, S. 15).



Der Desktop-Client wird mithilfe der Java-Bibliothek Swing realisiert. Als softwaretechnische Infrastruktur auf der Serverseite und für die Kommunikation zwischen Client und Server kommt Java EE zum Einsatz.

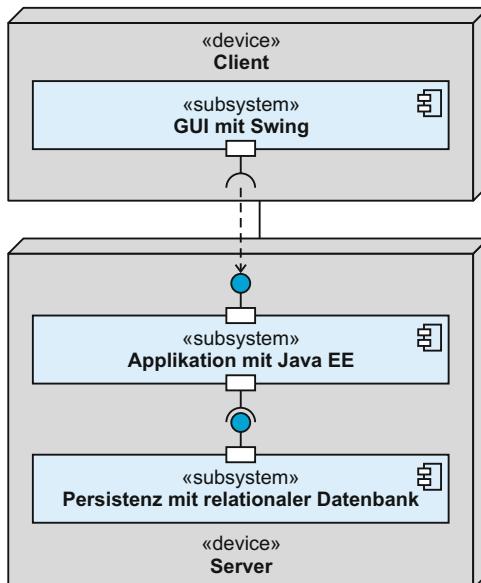
Abb. 16.2-4: OOA-Klassendiagramm für die Fallstudie »Kundenverwaltung«.

I 16 Softwaretechnische Infrastrukturen

Die Kommunikationsart ist abhängig von der eingesetzten softwaretechnischen Infrastruktur. Für die Persistenz wird eine relationale Datenbank verwendet.

Die Abb. 16.2-5 zeigt die Verteilung der Subsysteme.

Abb. 16.2-5:
Verteilung der Anwendungskomponenten bei der Fallstudie KV mit Java EE als CS-Anwendung.



Die grundlegende Architektur zeigt die Abb. 16.2-6.

Das Subsystem Applikation

Der Subsystem Applikation gliedert sich in das eigentliche Fachkonzept und die Geschäftslogik:

- **Fachkonzept:** Das fachliche Modell der Anwendung befindet sich auf der Serverseite im Paket `server.schnittstelle.fachkonzept`. Es wird durch JPA-Entities realisiert (siehe »*JPA – Java Persistence API*«, S. 431). Dabei handelt es sich um normale Java-Klassen, die um Annotationen der JPA (*Java Persistence API*) ergänzt wurden.
- **Geschäftslogik:** Die Anwendungslogik wird von den EJBs im Paket `server.application` realisiert.

Die Fachkonzeptklassen, welche dem Client zur Verfügung gestellt werden, implementieren `Serializable`, damit Objekte dieser Klassen zwischen Client und Server ausgetauscht werden können. Der Client verwaltet lokale Listen der Kunden und Anzeigen zur Verringerung der Remote-Zugriffe (Caching-Funktion des Business Delegate-Musters).

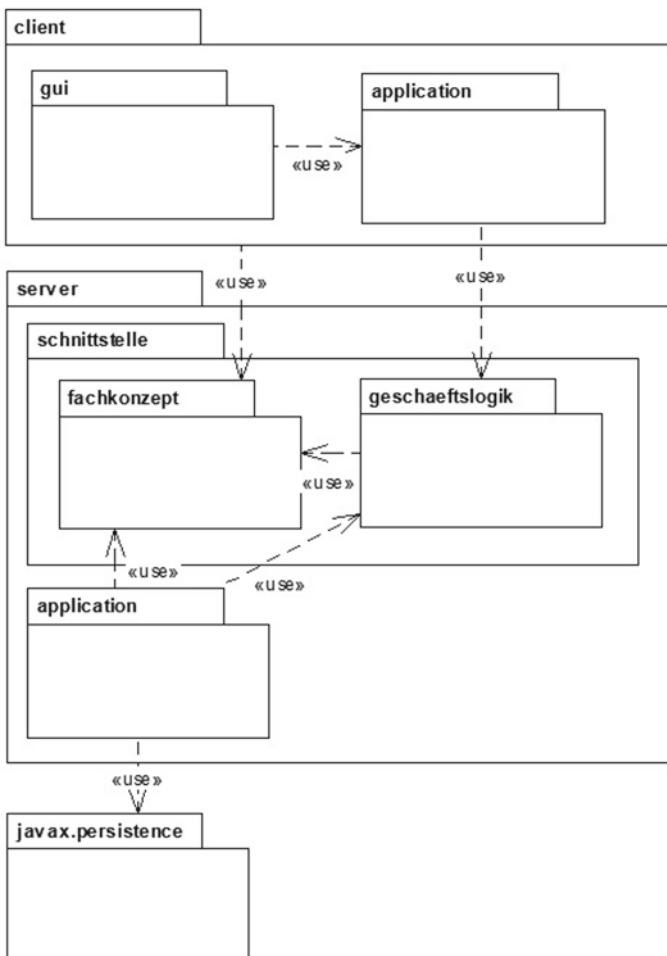


Abb. 16.2-6:
Grundlegende Architektur der Fallstudie KV mit Java EE als CS-Anwendung.

Das Subsystem GUI

Die Klassen der Benutzungsoberfläche befinden sich im Paket `client.gui`. Sie nutzen die Fachkonzeptklassen, welche sich im Paket `server.schnittstelle.fachkonzept` befinden. Auf die Dienste der EJBs greifen die GUI-Klassen *nicht* direkt, sondern nur über eine Klasse aus dem Paket `client.remote` zu.

Der Zugriff auf die EJBs erfolgt mithilfe von JNDI. Die Clientanwendung fragt das *Remote Interface* der *Session Bean Anzeigenverwaltung* ab. Die *Session Bean Anzeigenverwaltung* wiederum fragt in der Serveranwendung mittels *Dependency Injection* die *Local Interfaces* der *Session Beans AnzeigenBean* und *KundenBean* ab.

Das Subsystem Persistenz

Zur Speicherung der Daten in einer relationalen Datenbank werden Dienste der JPA aus dem Paket `javax.persistence` verwendet.

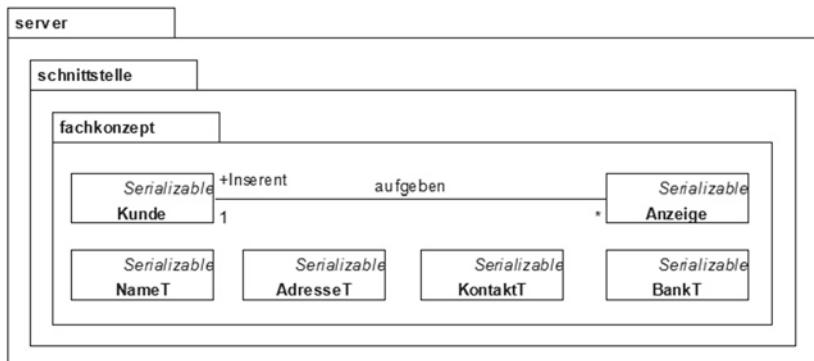
I 16 Softwaretechnische Infrastrukturen

Fachkonzept und objekt-relationale Abbildung

Die in der Abb. 16.2-4 dargestellten Fachkonzeptklassen müssen wie folgt erweitert werden (Abb. 16.2-7):

- Alle Fachkonzeptklassen müssen die Schnittstelle Serializable implementieren, damit Objekte zwischen Client und Server ausgetauscht werden können.
- Alle Klassen, deren Objekte in der relationalen Datenbank gespeichert werden sollen, müssen um Annotationen ergänzt werden, die der JPA mitteilen, wie die Klassen auf ein relationales Datenbankschema abzubilden sind.

Abb. 16.2-7:
Erweiterung der
Fachkonzeptklas-
sen um Remote-
Schnittstellen bei
der Fallstudie KV
mit Java EE als CS-
Anwendung.



Damit die Objekte der Fachkonzeptklassen mithilfe der JPA in einer relationalen Datenbank gespeichert werden können, müssen die Klassen um Annotationen ergänzt werden, die die objekt-relationale Abbildung definieren. Für die Klasse `Kunde` sieht dies wie folgt aus (siehe auch »Fallstudie: KV – JPA«, S. 436):

```
Java @Entity
@Table(name = "kunden")
public class Kunde implements Serializable
{
    @Id
    @Column(name="kundennr")
    private long nummer;

    @Embedded
    @Target(NameT.class)
    private NameI name;

    @Embedded
    @Target(AdresseT.class)
    private AdresseI adresse;

    @Embedded
    @Target(KontaktT.class)
    private KontaktI kontakt;
```

```

@Embedded
@Target(BankT.class)
private BankI bank;
private Date agbBestaetigt;

@OneToMany(mappedBy="kunde", cascade = CascadeType.REMOVE,
    targetEntity=Anzeige.class, fetch=FetchType.EAGER)
private List<AnzeigeI> anzeigeListe;

... // Zugriffsmethoden
}

```

Die Annotation `@Entity` markiert die Klasse als JPA-Entity. Die Objekte der Klasse können dadurch mithilfe der JPA in der Datenbank gespeichert werden. Mit der Annotation `@Table` lassen sich Angaben zur Datenbanktabelle machen, in der die Objekte der Klasse gespeichert werden sollen. In diesem Fall wird festgelegt, dass die Datenbanktabelle `kunden` heißen soll. Fehlt diese Angabe, entspricht der Tabellenname dem Klassennamen.

Jede Entity-Klasse benötigt ein Attribut, das in der Datenbank als Primärschlüssel verwendet werden kann. In diesem Fall ist dies das Attribut `nummer`, das mit der Annotation `@Id` als Primärschlüssel definiert wird. Die Angabe `@Column(name="kundennr")` legt fest, dass das Attribut `nummer` in einer Spalte mit dem Namen `kundennr` gespeichert werden soll. Fehlt diese Angabe, entspricht der Spaltenname dem Attributnamen.

Weil die Klassen `NameT`, `AdresseT`, `KontaktT` und `BankT` Informationen zu genau einem Kunden enthalten und nicht ohne diesen existieren können, sollen sie in der Datenbank *nicht* in eigenen Tabellen gespeichert, sondern in die Kundentabelle integriert werden. Dies wird mithilfe der Annotation `@Embedded` festgelegt.

Weil das Attribut in der Java-Klasse vom Typ der jeweiligen Schnittstelle (z.B. `NameI`) und nicht der eigentlichen Klasse (z.B. `NameT`) ist, muss zusätzlich über die Annotation `@Target(NameT.class)` definiert werden, welche konkrete Klasse die einzubettenden Objekte erzeugt.

Die einzubettenden Klassen selbst werden mit der Annotation `@Embeddable` markiert:

```

@Embeddable
public class NameT implements Serializable
{
    @Enumerated(EnumType.STRING)
    private Anrede anrede = Anrede.Undefiniert;
    private String vorname;
    private String nachname;

    ... // Zugriffsmethoden
}

```

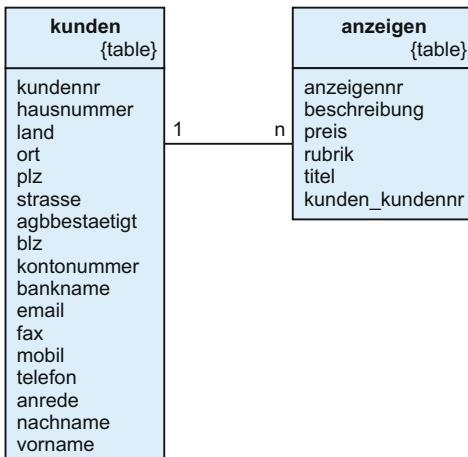
I 16 Softwaretechnische Infrastrukturen

Für unterschiedliche Assoziationsarten stehen in der JPA verschiedene Annotationen zur Verfügung. In diesem Fall handelt es sich um eine 1:*- bzw. 1:n-Assoziation. Die dazugehörige Annotation heißt `@OneToOne`. Sie benötigt mehrere Attribute. Das Attribut `mappedBy` gibt den Namen des Attributs am anderen Ende der Assoziation an. Das Attribut `CascadeType` bestimmt, in Folge welcher Ereignisse die assoziierten Objekte aktualisiert werden sollen. Im Beispiel legt die Angabe `CascadeType.REMOVE` fest, dass beim Löschen eines Kunden auch die dazugehörigen Anzeigen gelöscht werden. Die Angabe `targetEntity` enthält den Namen der referenzierten Klasse. Diese Angabe ist erforderlich, wenn das Attribut vom Typ einer Schnittstelle ist. Das Attribut `fetch` bestimmt schließlich, ob beim Laden eines Kunden die dazugehörigen Anzeigen direkt mit geladen oder erst bei Bedarf abgefragt werden. Zur Auswahl stehen die Modi `FetchType.LAZY` (verknüpfte Objekte nicht direkt initialisieren) und `FetchType.EAGER` (verknüpfte Objekte direkt mit initialisieren). Aus Performance-Gründen sollte möglichst `FetchType.LAZY` gewählt werden. Um die Fallstudie an anderen Stellen zu vereinfachen wird in diesem Fall aber `FetchType.EAGER` verwendet. Die Gegenseite der Assoziation wird in der Klasse `Anzeige` wie folgt definiert:

```
@ManyToOne(targetEntity=Kunde.class)  
private Kunde kunde;
```

Auf Basis der Annotationen wird beim Starten der Anwendung im EJB-Container automatisch ein relationales Datenbankschema für die Anwendung generiert bzw. ein möglicherweise vorhandenes Schema aktualisiert. Die Abb. 16.2-8 zeigt das Datenbankschema für die Fachkonzeptklassen der Abb. 16.2-4.

Abb. 16.2-8:
Datenbankschema
der Fachkonzept-
klassen bei der
Fallstudie KV mit
Java EE als CS-
Anwendung.



Die Attribute der Klassen Kunde, NameT, AdresseT, KontaktT und BankT werden in der Tabelle kunden vereint. Die Tabelle anzeigen enthält als Fremdschlüssel die Kundennummer des verknüpften Kunden (kunde_kundennr).

Geschäftslogik

Die Abb. 16.2-9 zeigt die für die Realisierung der Geschäftslogik relevanten Klassen und Schnittstellen. Es wird das Entwurfsmuster *Session Facade* aus dem J2EE-Musterkatalog [Sun02] verwendet. Die Geschäftslogik wird durch drei EJBs realisiert.

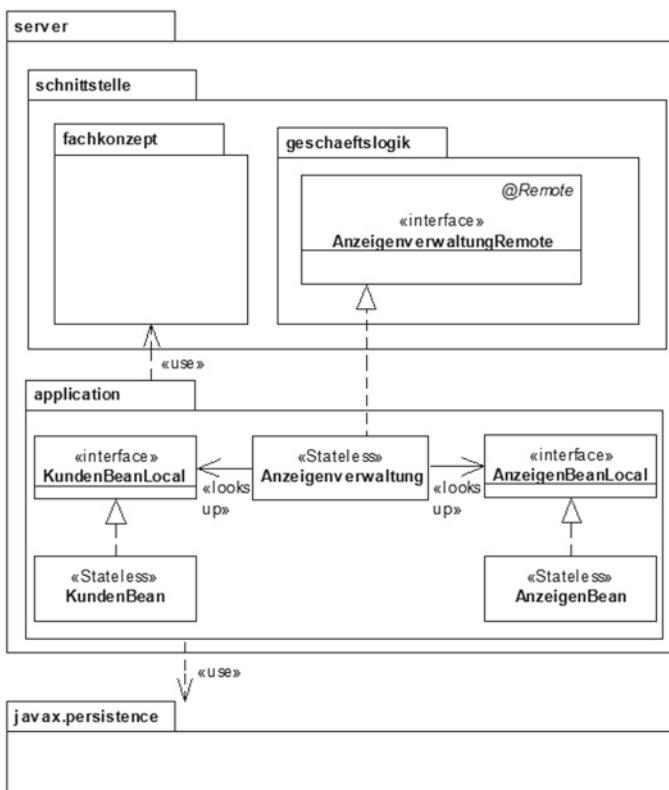


Abb. 16.2-9:
Klassen und
Schnittstellen der
Geschäftslogik bei
der Fallstudie KV
mit Java EE als CS-
Anwendung.

Die Klasse KundenBean

Die SLSB (*Stateless Session Bean*) KundenBean dient zum Speichern von Kunden-Objekten in der Datenbank und Laden von Kunden aus der Datenbank. Dafür nutzt die Klasse KundenBean Methoden der EntityManager-Schnittstelle der JPA. Der folgende Codeausschnitt zeigt zwei Methoden der Klasse KundenBean zum Speichern eines neuen Kunden und zum Laden einer Liste aller Kunden:

I 16 Softwaretechnische Infrastrukturen

```
Java @Stateless  
public class KundenBean implements KundenBeanLocal  
{  
    @PersistenceContext(unitName = "avCS-Persistenz")  
    private EntityManager entityManager;  
  
    // Neuen Kunden in der Datenbank speichern  
    public void speichereKunde(Kunde einKunde)  
    {  
        entityManager.merge(einKunde);  
        entityManager.flush();  
    }  
  
    // Liste aller Kunden aus der Datenbank laden  
    public List<Kunde> getKundenListe()  
    {  
        Query query = entityManager.createQuery(  
            "select k from Kunde k  
            order by k.name.nachname, k.name.vorname");  
        return (List<Kunde>)query.getResultList();  
    }  
  
    // Weitere Methoden...  
}
```

Zunächst wird durch die Annotation `@PersistenceContext` die `PersistenceUnit` für den Zugriff auf die Datenbank geladen. Diese über die Schnittstelle `EntityManager` angesprochene Komponente der JPA ermöglicht aus Java-Klassen heraus den Zugriff auf eine relationale Datenbank. Wie später im Abschnitt "Installation" erklärt wird, lässt sich die `PersistenceUnit` in der Konfigurationsdatei `persistence.xml` definieren.

Die Schnittstelle `EntityManager` stellt verschiedene Methoden bereit, die es erlauben, Objekte in der Datenbank zu speichern und aus der Datenbank zu laden. Der Aufruf `entityManager.merge(einKunde)` speichert das übergebene Kunden-Objekt in der Datenbank. Der Befehl `entityManager.flush()` sorgt dafür, dass die an die Datenbank zu übergebenden Daten sofort und ohne Verzögerung in die Datenbank geschrieben werden.

In der zweiten Methode wird zunächst ein Objekt vom Typ `Query` erzeugt. Dieses Objekt stellt eine Datenbankabfrage im SQL ähnlichen HQL-Format (*Hibernate Query Language*) dar. Im Beispiel werden alle Kunden-Objekte sortiert nach ihren Namen aus der Datenbank geladen. Die Liste der Abfrageergebnisse kann mithilfe der Methode `getResultList()` vom `Query`-Objekt abgerufen werden.

Neben den beiden abgebildeten Methoden enthält die Klasse `KundenBean` weitere Methoden zum Zugriff auf die Datenbank, unter anderem zum Löschen von Kunden-Objekten und zur Suche eines Kunden anhand seiner Kundennummer.

Die Schnittstelle KundenBeanLocal

Die Schnittstelle KundenBeanLocal dient der Klasse KundenBean als *Business Interface*:

```
@Local
public interface KundenBeanLocal {
    public void entferneKunde(Kunde einKunde);
    public Kunde getKundeZuNr(long nummer);
    public List<Kunde> getKundenListe();
    public long getNaechsteKundenNr();
    public void speichereKunde(Kunde einKunde);
}
```

Mit der Annotation `@Local` wird festgelegt, dass lokale Anfragen, also Anfragen aus demselben Anwendungsfenster, auf die *Session Bean* KundenBean über dieses *Business Interface* stattfinden.

Die Klasse AnzeigenBean

Die SLSB AnzeigenBean enthält analog zur Klasse KundenBean Methoden zum Speichern und Laden von Anzeigen-Objekten mithilfe der EntityManager-Schnittstelle der JPA.

Die Schnittstelle AnzeigenBeanLocal

Die Schnittstelle AnzeigenBeanLocal dient der Klasse AnzeigenBean als *Business Interface* und ist analog zur Schnittstelle KundenBeanLocal.

Die Klasse Anzeigenverwaltung

Die SLSB Anzeigenverwaltung ist zuständig für die Erzeugung neuer Objekte und die Weiterleitung von Anfragen, die einen Zugriff auf die Datenbank erfordern, an die *Session Beans* Kundenbean und AnzeigenBean. Sie ist über ihr *Business Interface* AnzeigenverwaltungRemote ansprechbar. Der Zugriff auf diese *Session Beans* erfolgt über *Dependency Injection*. Alternativ wäre auch eine Abfrage über JNDI möglich. Der folgende Codeausschnitt zeigt einen Teil der Klasse Anzeigenverwaltung:

```
@Stateless
public class Anzeigenverwaltung
    implements AnzeigenverwaltungRemote
{
    /*
     * Dependency Injection: Lokales Business Interface
     * der Session Bean KundenBean
     * Alternativ: JNDI-Lookup
     * kundenBean =
     * (KundenBeanLocal) ctx.lookup("KundenBean/local");
    */
    @EJB
    private KundenBeanLocal kundenBean;

    //Analog zu KundenBeanLocal
```

Java

I 16 Softwaretechnische Infrastrukturen

```
@EJB  
private AnzeigenBeanLocal anzeigenBean;  
  
public Anzeigenverwaltung()  
{  
}  
  
//Methode zum Erstellen eines neuen Kunden-Objekts  
public Kunde erstelleKunde()  
{  
    Kunde kunde = new Kunde();  
    kunde.setNummer(kundenBean.getNaechsteKundenNr());  
    return kunde;  
}  
  
//Methode zum Speichern eines Kunden in der Datenbank.  
public void speichereKunde(Kunde einKunde)  
{  
    kundenBean.speichereKunde(einKunde);  
}  
//Weitere Methoden... (Entfernen & Suchen eines Kunden,  
//Rückgabe einer Kundenliste)  
  
//Methode zum Erstellen eines neuen Anzeigen-Objekts  
public Anzeige erstelleAnzeige()  
{  
    Anzeige anzeigen = new Anzeige();  
    anzeigen.setNummer(anzeigenBean.getNaechsteAnzeigenNr());  
    return anzeigen;  
}  
  
//Methode zum Speichern einer Anzeige in der Datenbank.  
public void speichereAnzeige(Anzeige eineAnzeige)  
{  
    anzeigenBean.speichereAnzeige(eineAnzeige);  
}  
//Weitere Methoden... (Entfernen & Suchen einer Anzeige,  
//Rückgabe einer Anzeigenliste)  
}
```

Die *Session Beans* KundenBean sowie AnzeigenBean werden über *Dependency Injection* abgefragt. In der Klasse Anzeigenverwaltung erfolgt lediglich eine Weiterleitung der Clientanfragen an die entsprechende *Session Bean* (Entwurfsmuster *Session Facade*).

Die Schnittstelle AnzeigenverwaltungRemote

Die Schnittstelle AnzeigenverwaltungRemote dient der Klasse Anzeigenverwaltung als *Business Interface*:

```
@Remote  
public interface AnzeigenverwaltungRemote  
{  
    public void speichereAnzeige(Anzeige eineAnzeige);  
    public void speichereKunde(Kunde einKunde);  
    public void entferneAnzeige(Anzeige anzeigen);
```

```

public void entferneKunde(Kunde einKunde);
public Anzeige erstelleAnzeige();
public Kunde erstelleKunde();
public Anzeige getAnzeigeZuNr(long nummer);
public Kunde getKundeZuNr(long nummer);
public List<Kunde> getKunden();
public List<Anzeige> getAnzeigen();
public String endeAnwendung();
}

```

Mit der Annotation `@Remote` wird festgelegt, dass Anfragen, die nicht aus demselben Anwendungsfenster stammen, auf die *Session Bean* Anzeigenverwaltung über dieses *Business Interface* stattfinden.

Die Client-Anwendung

Die Klassen der Client-Anwendung lassen sich den beiden folgenden Aufgabengebieten zuordnen:

- Darstellung des Benutzungsoberflächen
- Kommunikation mit der Server-Anwendung

Die Benutzungsoberfläche

Die Benutzungsoberfläche der Client-Anwendung wird mithilfe der Java-Bibliothek Swing realisiert. Die Abb. 16.2-10 zeigt die für die Benutzungsoberfläche relevanten Klassen aus dem Paket `client.gui`.

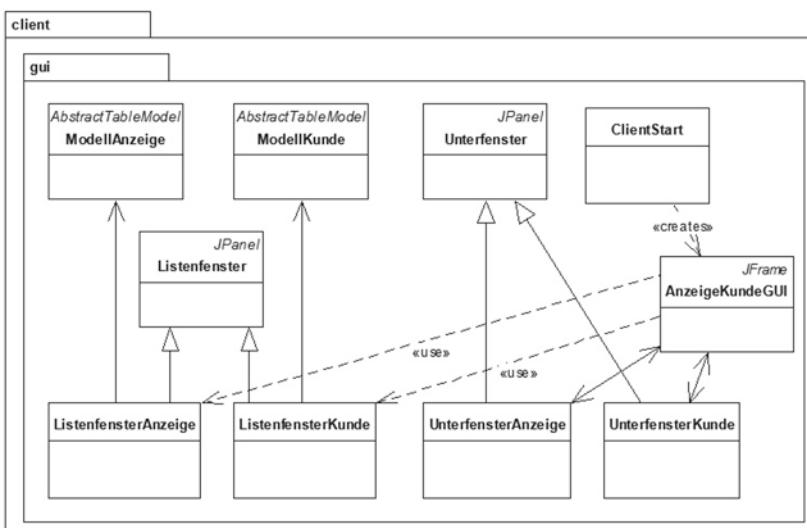


Abb. 16.2-10:
Relevante Klassen
der Benutzungs-
oberfläche bei der
Fallstudie KV mit
Java EE als CS-
Anwendung.

Die Klasse `ClientStart` erzeugt ein neues Objekt der Klasse `AnzeigeKundeGUI`, das das eigentliche Hauptfenster der Benutzungs-oberfläche realisiert. Die beiden Klassen `UnterfensterAnzeige` und

I 16 Softwaretechnische Infrastrukturen

UnterfensterKunden erben von der Klasse Unterfenster und dienen zur Neuerfassung oder Bearbeitung einzelner Anzeigen und Kunden. Die beiden Klassen ListenfensterAnzeige und ListenfensterKunde erben von der Klasse Listenfenster und enthalten Listen aller Anzeigen bzw. Kunden. Als Datenmodell für die beiden Tabellen dienen die Klassen ModellAnzeige und ModellKunde, welche von der Klasse AbstractTableModel erben.

Die Kommunikation mit der Server-Anwendung

Die Kommunikation mit der Server-Anwendung erfolgt über die Klasse AnzeigenverwaltungDelegate (Entwurfsmuster *Business Delegate*). Diese greift über das *Business Interface* AnzeigenverwaltungRemote auf die *Session Bean* Anzeigenverwaltung zu. Damit wird eine Entkopplung der Benutzungsoberfläche von der Kommunikation mit der Server-Anwendung erreicht. Zum Auffinden der *Session Bean* wird JNDI benutzt.

Die Gesamtstruktur im Detail

Die Abb. 16.2-11 zeigt detailliert die zuvor nur grob dargestellte Gesamtstruktur der Client-Server-Anwendung. Aus Gründen der Übersichtlichkeit sind die Beziehungen zwischen einigen Klassen nur auf Paket- und nicht auf Klassenebene abgebildet.

Hinweis

Um das Beispiel einfach zu halten, sind keine Mechanismen vorhanden, die Nebenläufigkeit berücksichtigen. Aus diesem Grund ist es möglich, dass bei gleichzeitigem Zugriff auf denselben Kunden Werte verloren gehen.

Verwendete Java EE-Entwurfsmuster

In dieser Fallstudie werden die Java EE-Entwurfsmuster (früher als J2EE-Muster bezeichnet) *Session Facade* und *Business Delegate* eingesetzt, welche im Folgenden kurz erläutert werden.

Das Entwurfsmuster Sitzungsfassade

Name(n): Das Entwurfsmuster Sitzungsfassade (*Session Facade*) (siehe »Das Fassaden-Muster (*Facade pattern*)«, S. 69) gehört zu den J2EE-Mustern [Sun02].

Grundidee: Die Details der Geschäftslogik werden hinter einer *Session Bean* verborgen.

Anwendungsbereich(e): Es dient dazu, die Abhängigkeiten zwischen Clients und der Realisierung der Geschäftslogik zu reduzieren.

16.2 Die Java EE-Plattform I

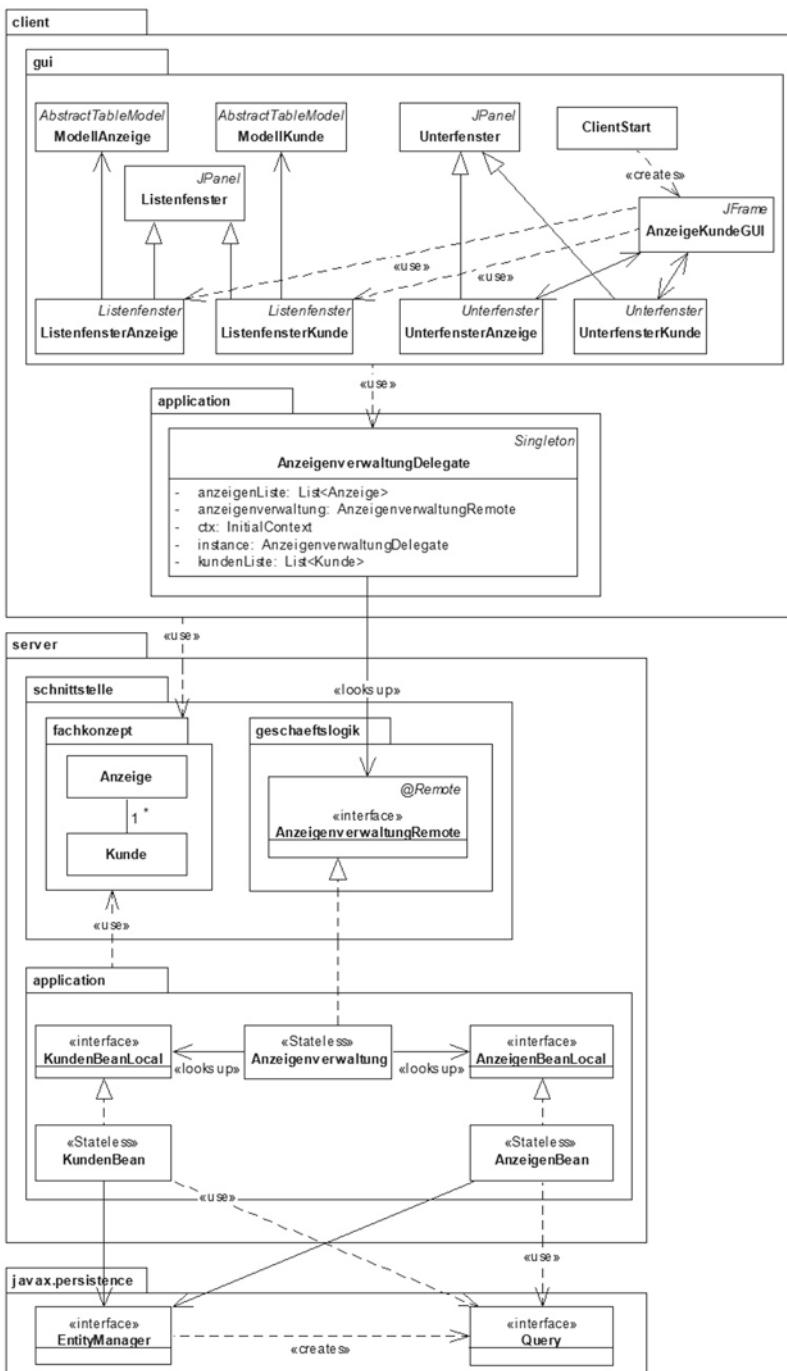


Abb. 16.2-11:
Gesamtstruktur
der Fallstudie KV
mit Java EE als CS-
Anwendung.

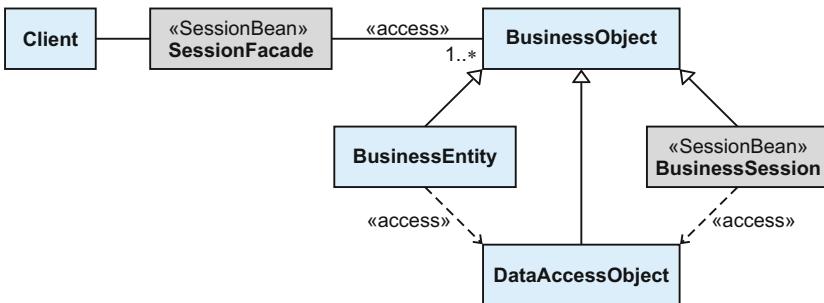
I 16 Softwaretechnische Infrastrukturen

Problem: Kommuniziert ein Client mit zu vielen verschiedenen Komponenten der Geschäftslogik, dann ergibt sich daraus eine große Abhängigkeit des Clients von der Implementierung der Anwendungslogik.

Lösung: Eine einzelne *Session Bean* dient als Fassade, die den Clients eine einheitliche, »grobkörnige« (*coarse-grained*) Schnittstelle zum Zugriff auf die Geschäftslogik bietet und die Komplexität der Implementierung der Geschäftslogik verbirgt.

Struktur
Die Abb. 16.2-12 zeigt die Struktur des Musters (in Anlehnung an [ACM03]).

Abb. 16.2-12:
Allgemeine
Struktur des
»Session Facade-
Musters«.



Realisierung in der Fallstudie: Die *Session Bean* Anzeigenverwaltung dient als Fassade für die Clientanwendung und entkoppelt die Clientanwendung von den *Session Beans* KundenBean und AnzeigenBean. Änderungen an der KundenBean und der AnzeigenBean haben keine Auswirkungen auf die Clientanwendung. Umgekehrt haben Änderungen an der Kommunikationsart mit dem Client *keine* Auswirkungen auf die beiden *Session Beans*. Entscheidet man sich im Laufe der Zeit für eine andere Kommunikationsart oder fügt z. B. einen Web-Client hinzu (siehe »Fallstudie: KV mit Java EE als Web-Anwendung«, S. 351), dann muss nur die Sitzungsfassade angepasst werden.

Das Entwurfsmuster *Business Delegate*

Das *Business Delegate*-Muster gehört zu den J2EE-Mustern [Sun02] und dient der Entkopplung der Präsentationsschicht von der Geschäftslogik.

Problem: Kommuniziert die Präsentationsschicht direkt mit der Geschäftslogik, sind die Komponenten der Präsentationsschicht anfällig für Änderungen an der Geschäftslogik oder dem Kommunikationsmechanismus.

Lösung: Eine als *Business Delegate* bezeichnete Komponente reduziert die Abhängigkeit zwischen der Präsentationsschicht und der Implementierung der Geschäftslogik. Sie dient als eine clientseiti-

ge Abstraktion der Geschäftslogik. Die Präsentationsschicht greift nicht direkt, sondern nur über die *Business Delegate*-Komponente auf die Dienste der Geschäftslogik zu. Außerdem verbirgt die Komponente die Kommunikationsdetails mit der Server-Anwendung, indem sie den JNDI-Zugriff auf die EJBs übernimmt. Schließlich kann das Muster zur Verringerung des Netzwerkverkehrs genutzt werden, indem Daten in der *Business Delegate*-Komponente zwischengespeichert werden, sodass nicht jeder an die Geschäftslogik gerichtete Methodenaufruf der Präsentationsschicht einen Remote-Zugriff zur Folge hat.

Struktur: Die Abb. 16.2-13 zeigt die Struktur des Business Delegate-Musters.

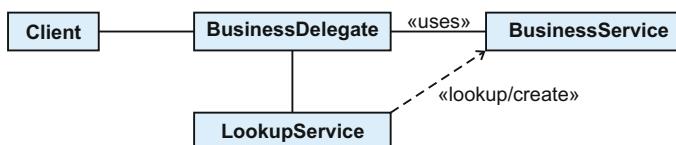


Abb. 16.2-13:
Allgemeine
Struktur des
»Business
Delegate-Musters«.

Die Klasse *BusinessDelegate* fungiert als Vermittler zwischen dem Client und der Geschäftslogik. Sie verwendet einen *LookupService*, um die Komponenten der Geschäftslogik aufzurufen.

Realisierung in der Fallstudie: Die Abb. 16.2-14 zeigt die in der Realisierung des Musters beteiligten Klassen.

Die Klasse *AnzeigenverwaltungDelegate* realisiert das Singleton-Muster, sodass alle Klassen der Kommunikationsschicht mit demselben Exemplar der Klasse arbeiten. Dieses Exemplar entkoppelt die Präsentationsschicht von der Server-Anwendung durch die folgenden Maßnahmen:

- Es verbirgt die Details der Kommunikation mit der Server-Anwendung vor der Präsentationsschicht. Der Zugriff auf die *Session Bean Anzeigenverwaltung* findet über das *Business Interface AnzeigenverwaltungRemote* statt. Außerdem fängt es die möglicherweise bei der Kommunikation auftretenden Ausnahmen ab und ersetzt sie durch benutzerfreundlichere Fehlermeldungen.
- Das Exemplar leitet die Anfragen der Präsentationsschicht an die Server-Anwendung weiter und verbirgt dabei die Implementierungsdetails der Geschäftslogik. Änderungen an der *Session Bean Anzeigenverwaltung* oder ihres *Business Interface AnzeigenverwaltungRemote* haben somit keine direkten Auswirkungen auf die Präsentationsschicht.

I 16 Softwaretechnische Infrastrukturen

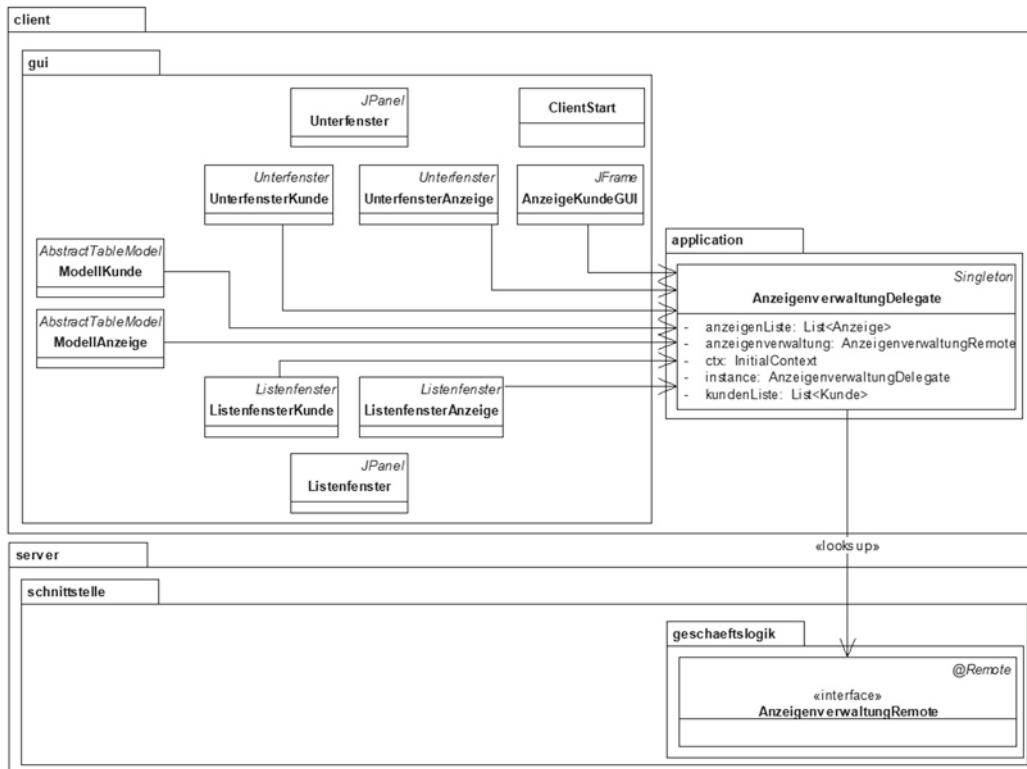


Abb. 16.2-14: Realisierung des Business Delegate-Musters bei der Fallstudie KV mit Java EE als CS-Anwendung.

■ Zur Verringerung der Netzwerkbela stung speichert das Exemplar der Klasse AnzeigenverwaltungDelegate die Kunden- und Anzeigenlisten zwischen. Nur wenn Änderungen an Objekten vorgenommen werden, sendet sie diese Änderungen an die Serveranwendung und fragt die aktualisierten Objekte erneut vom Server ab. Der folgende Codeausschnitt zeigt einen Teil der Klasse AnzeigenverwaltungDelegate:

```

Java
public class AnzeigenverwaltungDelegate
{
    private static AnzeigenverwaltungDelegate instance = null;
    private AnzeigenverwaltungRemote anzeigenverwaltung;
    //Referenz zur Remote-EJB

    //Lokale Listen der Kunden und Anzeigen
    //zur Verringerung der Remote-Zugriffe
    //-> Caching-Funktion des Business Delegate-Musters
    private List<KundeI> kundenListe;
    private List<AnzeigeI> anzeigenListe;

    private AnzeigenverwaltungDelegate()
    {
        try
        {

```

```

ctx = new InitialContext();

//JNDI-Lookup der Session Bean Anzeigenverwaltung
//über ihr Remote-Interface
anzeigenverwaltung = (AnzeigenverwaltungRemote)
    ctx.lookup("Anzeigenverwaltung/remote");
}

catch(NamingException e)
{
    e.printStackTrace();
}

//Methode für den Zugriff auf die Singleton-Instanz
public static AnzeigenverwaltungDelegate getInstance()
{
    if(instance == null)
        instance = new AnzeigenverwaltungDelegate();
    return instance;
}

//Zugriff auf eine Methode der Session Bean Anzeigenverwaltung
public void entferneAnzeige(Anzeige anzeige) throws Exception
{
    try
    {
        anzeigenverwaltung.entferneAnzeige(anzeige);
        aktualisiereListen();
    }
    catch(RemoteException e)
    {
        throw new Exception("Fehler beim Loeschen der Anzeige!");
    }
}
//Weitere Methoden...
}

```

Installation

Die Abb. 16.2-15 zeigt, wie die zuvor dargestellten Pakete in Form von zwei JARs (Java ARchives) auf den Client und den Server verteilt werden.

Die Pakete `client` und `server.schnittstelle` werden samt ihrer Unterpakete im Archiv `avCS-Client.jar` zusammengefasst, an die Clients verteilt und dort installiert. Zusätzlich benötigen die Clients die JBoss-Client-Bibliotheken aus dem Unterordner `client` des JBoss-Installationsverzeichnisses.

In der Fallstudie werden die notwendigen Bibliotheken automatisch beim Erstellen der Clientanwendung in den Ordner, in dem sich auch die Datei `avCS-Client.jar` befindet, kopiert.

I 16 Softwaretechnische Infrastrukturen

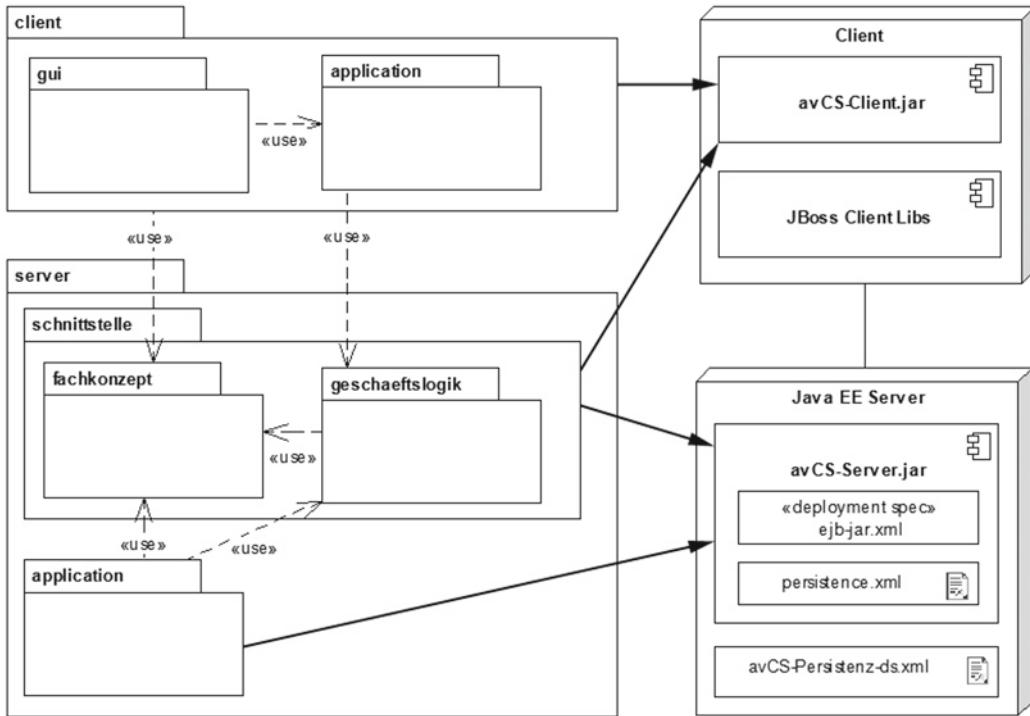


Abb. 16.2-15: Die Pakete `server.schnittstelle` und `server.application` und ihre Unterpakete werden im Archiv `avCS-Server.jar` zusammengefasst.
Deployment der Client-Server-Anwendung.

Zusätzlich enthält das Archiv den *Deployment Descriptor ejb-jar.xml*, der dem Anwendungsserver signalisiert, dass sich EJBs darin befinden, und die Konfigurationsdatei `persistence.xml`.

Letztere enthält die Definition der JPA PersistenceUnit, die für die Kommunikation mit der Datenbank zuständig ist. Über den in der Konfigurationsdatei definierten Namen `avCS-Persistenz` greifen die EJBs `KundenBean` und `AnzeigenBean` auf diese zu.

Im Abschnitt `jta-data-source` wird definiert, welche Datenquelle die Verbindungsparameter für den Zugriff auf die relationale Datenbank enthält:

```

<persistence xmlns="...">
    <persistence-unit name="avCS-Persistenz">
        <provider>
            org.hibernate.ejb.HibernatePersistence
        </provider>
        <!-- Name der Datenquelle, die in avCPersistenz-ds.xml definiert ist. -->
        <jta-data-source>java:/avCS-Datenquelle</jta-data-source>
        <properties>
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.DerbyDialect"/>
            <property name="hibernate.hbm2ddl.auto">
                
```

```
        value="update"/>
    <property name="hibernate.show_sql" value="false"/>
    <property name="hibernate.format_sql" value="false"/>
</properties>
</persistence-unit>
</persistence>
```

Das Archiv wird in einen Java EE-Anwendungsserver kopiert und von diesem ausgeführt. Zusätzlich muss die XML-Datei avCS-Persistenz-ds.xml an den Server übergeben werden, weil sie die Definition der in der Datei persistence.xml referenzierten Datenquelle avCS-Datenquelle enthält:

```
<datasources>
    <local-tx-datasource>
        <jndi-name>avCS-Datenquelle</jndi-name>
        <use-java-context>true</use-java-context>
        <!-- Nutzung von JavaDB -->
        <connection-url>
            jdbc:derby:avCS_db;create=true
        </connection-url>
        <driver-class>
            org.apache.derby.jdbc.EmbeddedDriver
        </driver-class>
    </local-tx-datasource>
</datasources>
```

Hinweis

Alle drei Fallstudien »Client-Server-Anwendung«, »Web-Anwendung« (»Fallstudie: KV mit Java EE als Web-Anwendung«, S. 351) und »Webservice« (»Fallstudie: KV mit Java EE als Webservice«, S. 377) können parallel installiert werden, da kein gegenseitiger Zugriff stattfindet.

Laden Sie die Fallstudie auf ihr Computersystem herunter, lassen Sie das Programm ablaufen und gehen Sie das Programm Schritt für Schritt durch.



16.2.6 Fallstudie: KV mit Java EE als Web-Anwendung

Die Fallstudie »Kundenverwaltung« wird in erweiterter Form als »Anzeigenverwaltung« als Web-Anwendung mit Java EE realisiert. Im Folgenden wird die Architektur so ausgelegt, dass die Anzeigenverwaltung sowohl als Client-Server-Anwendung als auch als Web-Anwendung gleichzeitig benutzt werden kann.

Web-Anwendung vs. Client-Server-Anwendung

Durch Einsatz von Entwurfsmustern ist es möglich, einen Großteil der bereits vorhandenen Klassen der Client-Server-Version zu übernehmen (siehe »Fallstudie: KV mit Java EE als Client-

I 16 Softwaretechnische Infrastrukturen

Server-Anwendung«, S. 333). Dazu gehören die Pakete `client.gui` und `server.schnittstelle.fachkonzept` sowie die Schnittstellen `AnzeigenBeanLocal`, `KundenBeanLocal` und `AnzeigenverwaltungRemote`, die unverändert übernommen werden können.

Die Klassen `AnzeigenverwaltungDelegate` und `Anzeigenverwaltung` unterscheiden sich lediglich bei der Nutzung der *Session Beans (JNDI)* bzw. *Dependency Injection*) und die *Session Beans* `AnzeigenBean` und `KundenBean` besitzen lediglich einen unterschiedlichen Persistenzkontext.

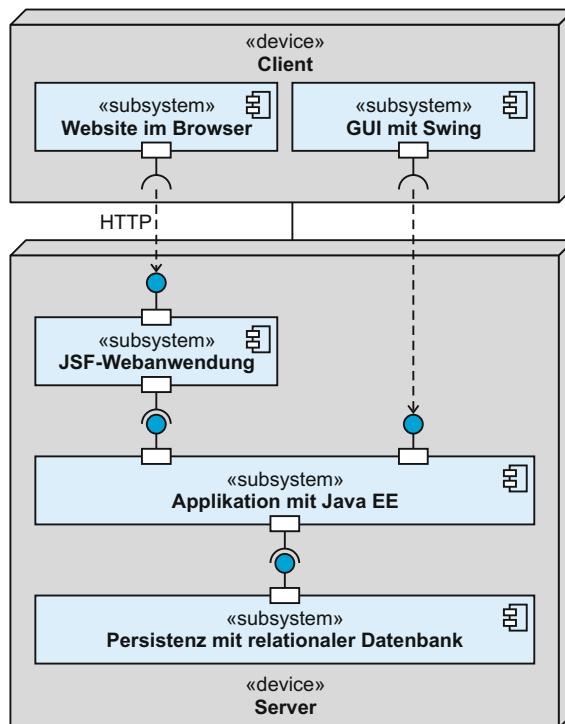
Die JSF-Web-Anwendung

Die **JSF**-Webanwendung greift als ein weiterer Client auf die Dienste der Java EE-Anwendung zu und generiert Webseiten, die im Browser des Benutzers angezeigt werden. Im Gegensatz zum Desktop-Client aus der Fallstudie »KV mit Java EE als Client-Server-Anwendung« wird eine JSF-Webanwendung im Anwendungsserver in derselben JVM wie die Java EE-Anwendungslogik ausgeführt.

Während man die Desktop-Anwendung als Remote-Client bezeichnet, handelt es sich bei der Web-Anwendung um einen lokalen Client.

Die Abb. 16.2-16 zeigt die Verteilung der Subsysteme.

Abb. 16.2-16:
Verteilung der Anwendungskomponenten bei der Fallstudie KV mit Java EE als Web-Anwendung.



Die Benutzungsoberfläche der Web-Anwendung wird durch *Facelets* realisiert. Dies sind XHTML-Dateien, die in ihrem Aufbau stark JSPs ähneln und neben HTML-Tags spezielle JSF-Tags enthalten.

Von einer Startseite mit dem Namen `home.xhtml` aus existieren Verweise zu Listenansichten für Kunden (`kundenListe.xhtml`) und Anzeigen (`anzeigenliste.xhtml`) sowie zu einem Assistenten für das Erstellen einer neuen Anzeige.

Ein Kunde gelangt dabei zunächst auf die Seite `kundennummer.xhtml`, auf welcher er seine Kundennummer eingeben kann, sofern er bereits ein registrierter Kunde ist.

Andernfalls ruft der Kunde direkt die nächste Seite `kunde.xhtml` auf. Dort hat er die Möglichkeit, seine Daten einzugeben. Schließlich gelangt der Kunde auf die Seite `anzeige.xhtml`, die ein Eingabeformular für die Erfassung einer neuen Anzeige enthält.

Die Abb. 16.2-17 zeigt die einzelnen Komponenten der JSF-Webanwendung.

Der folgende Codeausschnitt zeigt beispielhaft die Datei `anzeige.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8"/>
        <title>Anzeigendetails</title>
        <link rel="stylesheet" type="text/css"
              href="layout/layout.css" />
    </head>
    <body>
        <h:form id="anzeigeForm" styleClass="panelGrid">
            <h:messages globalOnly="true"
                        errorClass="meldung_fehler_global"
                        infoClass="meldung_info_global"
                        layout="table" style="width: 99%;" />
        </h:form>
    </body>

```

Bitte füllen Sie alle mit * markierten Felder aus!

```
<h:panelGrid columns="3" styleClass="panelGrid">
    <h:outputText value="Anzeigenummer"/>
    <h:inputText id="in_nummer"
                  value="#{anzeigeWebBean.aktAnzeige.nummer}"
                  disabled="true" />
    <h:message for="in_nummer"
                  errorClass="meldung_fehler_lokal" />

    <h:outputText value="Rubrik *"/>
    <h:inputText id="in_rubrik"
                  value="#{anzeigeWebBean.aktAnzeige.rubrik}"
                  required="true"/>
    <h:message for="in_rubrik"
```

I 16 Softwaretechnische Infrastrukturen

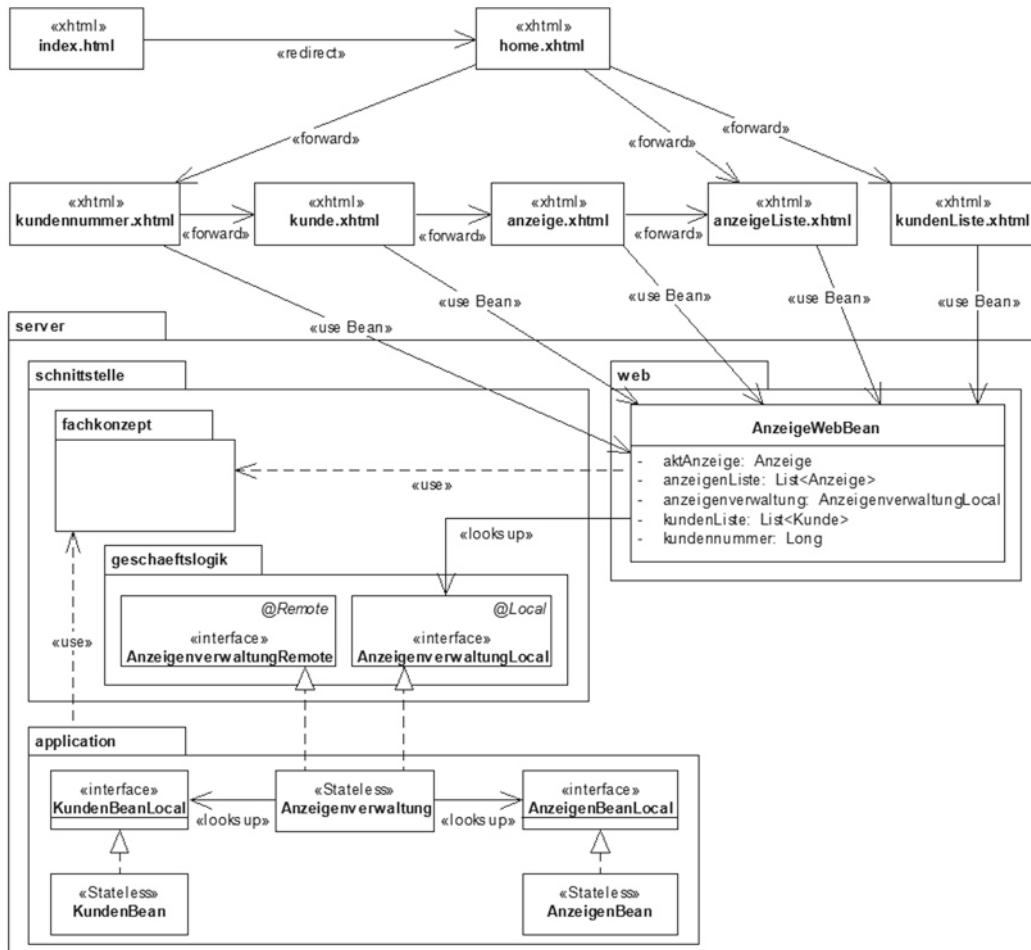


Abb. 16.2-17:
Komponenten der
JSF-
Webanwendung
bei der Fallstudie
KV mit Java EE als
Web-Anwendung.

```

        errorClass="meldung_fehler_lokal" />

<h:outputText value="Titel *"/>
<h:inputText id="in_titel"
    value="#{anzeigeWebBean.aktAnzeige.titel}"
    required="true" />
<h:message for="in_titel"
    errorClass="meldung_fehler_lokal" />

<h:outputText value="Beschreibung"/>
<h:inputTextarea id="in_beschreibung"
    value="#{anzeigeWebBean.aktAnzeige.beschreibung}" />
<h:message for="in_beschreibung"
    errorClass="meldung_fehler_lokal" />

<h:outputText value="Preis *"/>
<h:inputText id="in_preis"
    value="#{anzeigeWebBean.aktAnzeige.preis}" />
  
```

```

    required="true" />
<h:message for="in_preis"
errorClass="meldung_fehler_lokal" />

<h:outputText value="Kunde"/>
<h:inputText id="in_kunde"
value="#{anzeigeWebBean.aktAnzeige.kunde.name.nachname},
#{anzeigeWebBean.aktAnzeige.kunde.name.vorname}"
disabled="true" />
<h:message for="in_kunde"
errorClass="meldung_fehler_lokal" />

<h:commandButton styleClass="button"
action="#{anzeigeWebBean.speichereAnzeige}"
value="Speichern" />
<h:commandButton styleClass="button"
action="#{anzeigeWebBean.beendeAssistenten}"
immediate="true" value="Abbrechen"/>
</h:panelGrid>
</h:form>
</body>
</html>
```

Im HTML-Tag werden alle benötigten JSF-Tag-Bibliotheken geladen. JSF-Tags sind nur innerhalb eines form-Tags erlaubt. Das panelGrid-Tag generiert eine Tabelle für die darauf folgenden Formular-Elemente. Die Eingabefelder werden durch die inputText-Tags realisiert. Dabei ist jedes Feld über das value-Attribut mit einem Attribut einer sogenannten *Managed Bean* mit dem Namen AnzeigeWebBean verknüpft. Auch die Druckknöpfe im unteren Bereich des Formulars rufen Methoden der *Managed Bean* auf. *Managed Beans* sind Java-Klassen, die in einer Konfigurationsdatei mit dem Namen faces-config.xml registriert werden und danach aus den Facelets heraus ansprechbar sind. Als *Managed Bean* für alle oben genannten Seiten fungiert die Klasse AnzeigeWebBean. Im Attribut aktAnzeige verwaltet diese die von einem Benutzer während der Durchführung des Anzeigen-Assistenten gemachten Eingaben. In den Attributen anzeigenListe und kundenListe werden die von der Anwendungslogik abgefragten Objekt-Listen für die Dauer der Sitzung mit einem Benutzer zwischengespeichert. Zum Erzeugen neuer Objekte sowie zum Speichern und Laden von Objekten greift die *Managed Bean* über die Schnittstelle AnzeigenverwaltungLocal auf die Dienste der in der Fallstudie »KV mit Java EE als Client-Server-Anwendung« vorgestellten Anwendungslogik zu.

Der folgende Codeausschnitt zeigt einen Teil der Klasse AnzeigeWebBean:

```

public class AnzeigeWebBean
{
/*
 * Dependency Injection: Lokales Business Interface
```

I 16 Softwaretechnische Infrastrukturen

```
* der Session Bean Anzeigenverwaltung
* Alternativ: JNDI-Lookup
* anzeigenverwaltung = (AnzeigenverwaltungLocal)
* ctx.lookup("avWEB-Server-ear/Anzeigenverwaltung/local");
*/
@EJB
private AnzeigenverwaltungLocal anzeigenverwaltung;

private Long kundennummer;
private Anzeige aktAnzeige;
private List<Anzeige> anzeigenListe;
private List<Kunde> kundenListe;

public AnzeigeWebBean()
{
}

//Getter und Setter...

public String starteAssistenten()
{
    this.aktAnzeige = anzeigenverwaltung.erstelleAnzeige();
    return "START";
}

public String beendeAssistenten()
{
    return "STARTSEITE";
}

public String speichereAnzeige()
{
    if(aktAnzeige.getKunde().getAgbBestaetigt() == null)
        aktAnzeige.getKunde().setAgbBestaetigt(new Date());
    anzeigenverwaltung.speichereKunde(aktAnzeige.getKunde());
    anzeigenverwaltung.speichereAnzeige(aktAnzeige);
    aktualisiereListen();
    return "ANZEIGENLISTE";
}

//Weitere Methoden...
}
```

Da die *Managed Bean* AnzeigeWebBean auf die Dienste der Anwendungslogik zugreift, wird im oberen Bereich *Dependency Injection* für den Zugriff auf das lokale *Business Interface* AnzeigenverwaltungLocal der EJB Anzeigenverwaltung genutzt. Dies ist möglich, da die JSF-Webanwendung im Anwendungsfenster in derselben JVM wie die Java EE-Anwendungslogik aufgeführt wird. Danach erfolgt die Deklaration von Attributen, die aus den *Facelets* heraus ansprechbar sind. Das Betätigen von Druckknöpfen und Links in den *Facelets* löst immer eine Methode einer *Managed Bean* aus. Im Codeausschnitt sind die beiden zu den

Druckknöpfen aus der oben beschriebenen Datei anzeige.xhtml gehörigen Methoden abgebildet. Zum Speichern einer neuen Anzeige greift die *Managed Bean* auf die Dienste der EJB Anzeigenverwaltung zu. Jede Methode gibt einen String zurück, anhand dessen die nächste aufzurufende Seite bestimmt wird.

Die Festlegung der Seitenaufrufe, abhängig von dem übergebenen String, wird in der Konfigurationsdatei faces-config.xml definiert:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config ... >

<!--faces-config.xml ist der Standardname-->

<!--Name der JavaBean, das zur Laufzeit referenziert wird-->
<!--AnzeigeWebBean: Zugriff auf die Anzeigenverwaltung-->
<managed-bean>
    <managed-bean-name>anzeigeWebBean</managed-bean-name>
    <managed-bean-class>
        server.web.AnzeigeWebBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<!--Hier könnte noch ein Converter eingesetzt werden, um in
den XHTML-Dateien die Ausgaben ansprechender zu gestalten,
z. B. <h:outputText value="#{einKunde.nummer}"
converter="kundeConverter"/>-->

<!--Navigationsregeln-->
<!--Regel für die mögliche Navigation von home.xhtml.
Falls als Ergebnis START zurückgegeben wird, dann soll
die Seite kundennummer.xhtml aufgerufen werden-->
<navigation-rule>
    <display-name>Startseite</display-name>
    <from-view-id>/home.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>START</from-outcome>
        <to-view-id>/kundennummer.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>

<!--weitere Navigationsregeln-->
<application>
    <!--Facelets aktivieren-->
    <view-handler>
        com.sun.facelets.FaceletViewHandler
    </view-handler>
    <!--Datei mit Textmeldungen, vorhanden in
    WEB-INF/classes/bundle/messages.properties-->
    <message-bundle>bundle.messages</message-bundle>
</application>
</faces-config>
```

I 16 Softwaretechnische Infrastrukturen

Die Gesamtstruktur im Detail

Die Abb. 16.2-18 zeigt die Gesamtstruktur der Anzeigenverwaltung mit Web- und Desktop-Client.

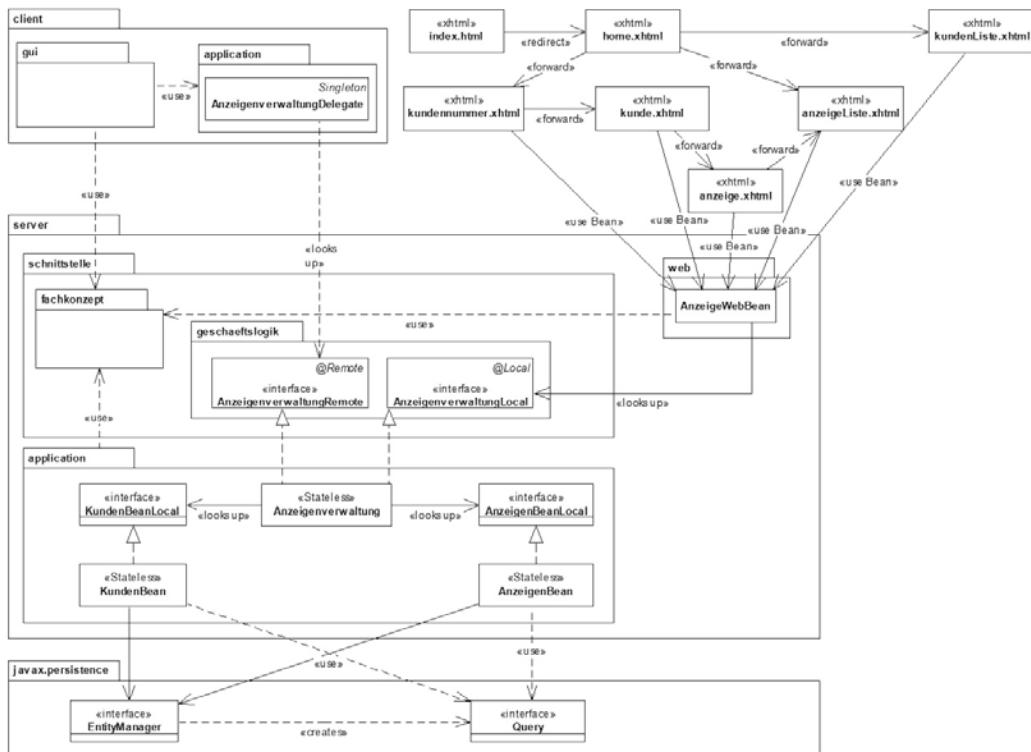


Abb. 16.2-18:
Gesamtstruktur
der Fallstudie KV
mit Java EE als
Web-Anwendung.

Der aus der Fallstudie »KV mit Java EE als Client-Server-Anwendung« bekannte Desktop-Client (siehe »Fallstudie: KV mit Java EE als Client-Server-Anwendung«, S. 333) ist im oberen linken Bereich zu sehen und greift über das von außen zugreifbare *Business Interface* *AnzeigenverwaltungRemote* auf die EJB *Anzeigenverwaltung* zu. Im oberen rechten Bereich ist der aus *Facelets* und einer *Managed Bean* bestehende Web-Client zu sehen. Die *Managed Bean* kann über das lokale *Business Interface* *AnzeigenverwaltungLocal* auf die EJB *Anzeigenverwaltung* zugreifen.

Installation

Eine Java EE-Anwendung, die sowohl eine Web-Anwendung als auch eine auf EJBs basierende Anwendungslogik enthält, wird in Form einer EAR-Datei installiert.

Die Web-Anwendung, bestehend aus den *Facelets*, den Paketen `server.web` und `server.schnittstelle` sowie den Konfigurationsdateien `web.xml` und `faces-config.xml`, wird in ein WAR (*Web Archive*) gepackt. Die im Paket `server.application` realisierte Anwendungslogik und die dazugehörigen Schnittstellen im Paket `server.schnittstelle` werden zusammen mit den Konfigurationsdateien `ejb-jar.xml` und `persistence.xml` in ein JAR (*Java ARchives*) gepackt. Beide Archive werden zusammen mit der Konfigurationsdatei `application.xml` in einem Archiv vom Typ EAR (*Enterprise Archive*) zusammengefasst und an den Anwendungsserver übergeben. In der Datei `application.xml` wird definiert, welche JAR- und WAR-Dateien zu der EAR-Anwendung gehören. Theoretisch kann ein EAR auch mehrere Archive jeder Sorte enthalten.

Die Abb. 16.2-19 zeigt, wie die zuvor dargestellten Pakete in JAR-, WAR- bzw. EAR-Dateien verteilt werden.

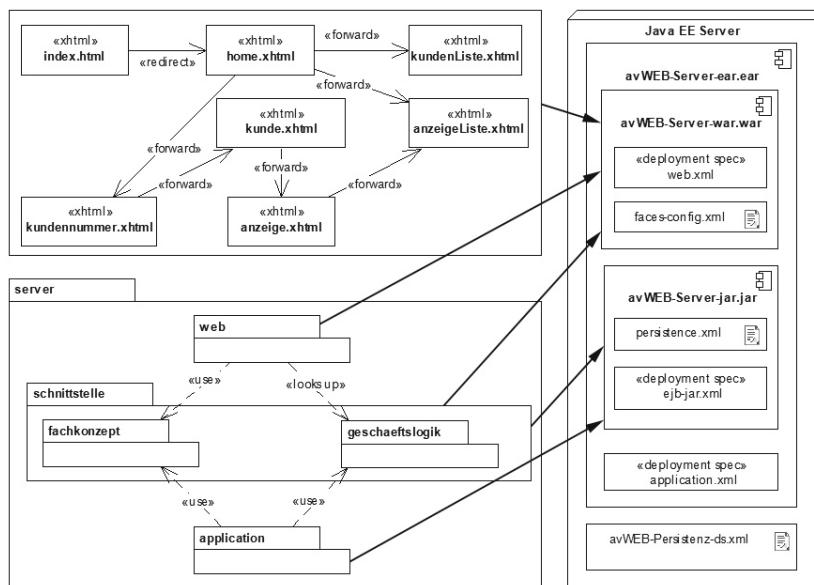


Abb. 16.2-19:
Verteilung der
Pakete in Dateien
bei der Fallstudie
KV mit Java EE als
Web-Anwendung.

Der folgende Codeausschnitt zeigt die Konfigurationsdatei `application.xml` der Anzeigenverwaltung:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<application ... >
    <display-name>anzeigenverwaltung</display-name>
    <module>
        <ejb>avWEB-Server-jar.jar</ejb>
    </module>
    <module>
        <web>
            <web-uri>avWEB-Server-war.war</web-uri>
            <!--http://localhost:8080/anzeigenverwaltung-->
        </web>
    </module>
</application>
```

I 16 Softwaretechnische Infrastrukturen

```
<context-root>anzeigenverwaltung</context-root>
</web>
</module>
</application>
```

Hinweis Alle drei Fallstudien »Client-Server-Anwendung« (»Fallstudie: KV mit Java EE als Client-Server-Anwendung«, S. 333), »Web-Anwendung« und »Webservice« (»Fallstudie: KV mit Java EE als Webservice«, S. 377) können parallel installiert werden, da kein gegenseitiger Zugriff stattfindet.

 Laden Sie die Fallstudie auf ihr Computersystem herunter, lassen Sie das Programm ablaufen und gehen Sie das Programm Schritt für Schritt durch.

16.3 Die .NET-Plattform

Das **.NET Framework** (kurz .NET) ist eine – in direkter Konkurrenz zu Java SE und Java EE stehende – betriebssystem- und hardware-neutrale Softwareentwicklungsplattform. .NET wurde ursprünglich von der Firma Microsoft entwickelt und ist in Teilen bei der ECMA und ISO standardisiert. Microsoft selbst liefert auch heute noch die wichtigsten Implementierungen von .NET:

- Das vollständige Microsoft .NET Framework für die Windows-Betriebssysteme.
- Das Microsoft .NET Compact Framework (.NET CF) für Windows Embedded und Windows Mobile 6.x (außer Windows Phone 7).
- Microsoft Silverlight für Windows-Betriebssysteme (einschließlich Windows Phone 7) und Mac OS.
- XNA für die Computerspielekonsole in der Spielekonsole XBox und Windows.
- Micro Framework (.NET MF) für Kleinstgeräte (seit November 2009 ein Open Source-Projekt).

Durch die Open-Source-Initiative Mono, die inzwischen zur Firma Novell gehört, steht auch eine Implementierung von .NET für Linux und andere Unix-basierte Systeme und Mac OS zu Verfügung. Eine Differenzierung zwischen allgemeinen Anwendungen (Java SE) und Unternehmensanwendungen (Java EE) gibt es nicht. .NET differenziert vielmehr zwischen Funktionen für den Client (.NET Framework Client Profile) und zusätzlichen Funktionen für Server (.NET Framework Extended). .NET hat viele Anleihen bei Java genommen, sowohl bei der Sprachsyntax als auch den Klassenbibliotheken. Aber auch Java hat inzwischen einiges bei .NET abgeschaut. Als Beispiele sind Annotationen, *Autoboxing* und generische Datentypen ab Java 5 zu nennen.

Zunächst werden die wichtigsten Eigenschaften der .NET-Plattform näher betrachtet:

- »Eigenschaften des .NET Framework«, S. 361

Anschließend wird auf die .NET-Architektur eingegangen:

- »Zur .NET-Architektur«, S. 368

Ein kurzer Überblick beschreibt die verfügbaren Werkzeuge:

- »Werkzeuge«, S. 372

16.3.1 Eigenschaften des .NET Framework

Technische Merkmale

Wesentliche Merkmale des .NET Framework sind:

- Plattformunabhängigkeit durch Zwischensprache/Intermediation mit Just-in-Time-Compiler wie bei Java (*Write Once Run Anywhere*-Prinzip).
- Sprachunabhängigkeit (mehr als 70 verschiedene Programmiersprachen) mit sprachübergreifenden Aufrufen und sprachübergreifender Vererbung.
- Durchgängige Objektorientierung: Auch elementare Datentypen wie Zahlen und Zeichenketten sind Objekte.
- Unterstützung für wiederverwendbare Softwarekomponenten.
- Verschiedene Anwendungarten (Fat-Clients, Standard-Webanwendungen, Rich Internet Applications, Systemdienste, Webdienste, Pocket-PC-Anwendungen, SmartPhone-Anwendungen)
- Einheitliche Laufzeitumgebung mit Diensten wie Codeüberprüfung (Sicherheit, Array-Grenzen etc.), *Threading*, Speicherbereinigung und Ausnahmebehandlung.
- Umfangreiche Klassenbibliothek mit mehr als 12.000 Klassen, nutzbar in allen .NET-fähigen Programmiersprachen.
- XML-basierte Konfiguration von Anwendungen (Abkehr von der Windows-Registrierungsdatenbank).
- Parallelbetrieb verschiedener .NET Framework-Versionen: Jede .NET-Anwendung startet automatisch mit der Framework-Version, für die sie entwickelt wurde.
- Bereitstellen von Metadaten über den Programmcode (automatische Metadatengenerierung und manuelle Metadaten).
- Lose Schnittstellenverträge, die es ermöglichen, dass man Klassenmitglieder ergänzt, ohne den Schnittstellenvertrag zu brechen. Der Vertrag wird erst gebrochen, wenn man Mitglieder oder Parameter entfernt bzw. Datentypen ändert.
- XCopy-Deployment: Verteilung von Anwendungen durch einfaches Kopieren der Programmdateien sowie der zugehörigen Bibliotheken und Ressourcendateien.
- Interoperabilität zu älteren Plattformen (COM, Windows-32-API) sowie über XML-Webservices zu anderen Plattformen.

I 16 Softwaretechnische Infrastrukturen

- Schutz vor »gefährlichem« Programmcode durch Sandbox-Konzepte wie in Java.

Anwendungsarten

.NET unterstützt eine breite Palette von Anwendungsarten. Hier sind insbesondere zu nennen:

- Windows-Anwendungen auf Basis von Windows Forms oder *Windows Presentation Foundation* (WPF) oder Microsoft Silverlight in der Out-Of-Browser (OOB)-Variante.
- Webserver-Anwendungen auf Basis von ASP.NET Webforms, ASP.NET Model View Controller und ASP.NET Dynamic Data.
- Web-Client-Anwendungen (alias Browseranwendung) auf Basis von ASP.NET AJAX (HTML/JavaScript ohne Plug-Ins) oder mit Plug-Ins durch Einsatz von WPF im Browser (*Web Browser Application – WBA* alias XBAP) oder Microsoft Silverlight. Ab .NET 4.0 wird *Windows Forms* im Browser mit IEHost.dll nicht mehr unterstützt.
- Erweiterungen für die Microsoft Office-Produkte (Word, Excel, Outlook, PowerPoint etc.) auf Basis der *Visual Studio Tools for Microsoft Office* (VSTO).
- Konsolenanwendungen auf Basis der .NET-Klasse `System.Console`.
- Windows-Dienste auf Basis der Klassen in der Komponente `System.ServiceProcess`.
- XML-Webservices auf Basis von ASP.NET Webservices.
- WMI-Provider auf Basis der Klassen in `System.Management`.
- Prozeduren, Funktionen und Datentypen für den Microsoft SQL Server 2005/2008 (inkl. R2) in Form der QLCLR.

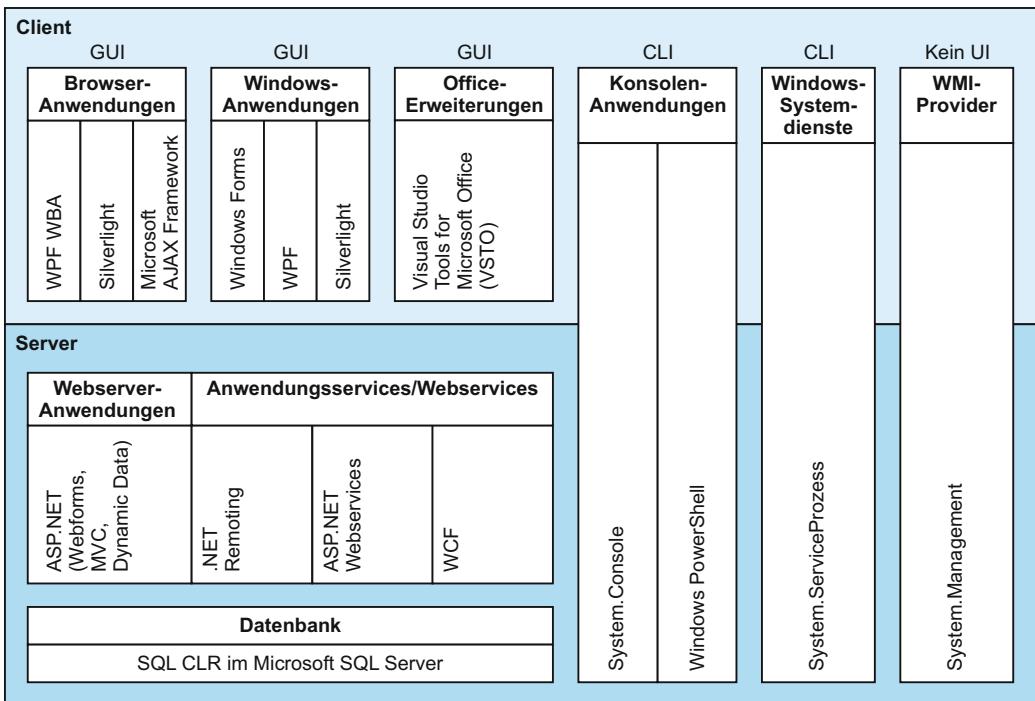
Die Abb. 16.3-1 verdeutlicht die Vielfalt der Anwendungsarten.

Ausführungsumgebungen

.NET-Anwendungen werden in sogenannten .NET Runtime Hosts (das Pendant zu den Containern in Java) ausgeführt. .NET bietet insbesondere folgende *Runtime Hosts*:

- **Shell Host:** Client-Anwendungen, die direkt auf dem Betriebssystem ausgeführt werden, entweder mit grafischer Oberfläche oder als Konsolenanwendungen.
- **PowerShell Host:** Spezieller *Runtime Host* für die Windows PowerShell, eine Kommandozeilenshell von Windows.
- **Microsoft Office Host:** Erweiterungen der Microsoft Office-Produkte z. B. um Funktionsleisten und *Smart Tags*.
- **Browser Host:** Einbettung grafischer Oberflächen in Webseiten im Microsoft Internet Explorer und Firefox (ursprünglich für Windows Forms, inzwischen nur noch für WPF).

16.3 Die .NET-Plattform I



Legende: CLI = *Command Line Interface* (Kommandozeilenschnittstelle)

- **Silverlight Host:** Einbettung grafischer Oberflächen in Webseiten in zahlreiche Browser auf Basis von Silverlight.
- **Web Server/Application Server Host:** Bereitstellung von Webanwendungen mit ASP.NET und Webservices mit der *Windows Communication Foundation* (WCF). Die wichtigste Implementierung eines solchen Hosts ist der »Microsoft Internet Information Server« (IIS) zusammen mit Microsoft AppFabric. Diese Produkte sind Bestandteile des Windows-Betriebssystems (auch in Client-Betriebssystemen). Eine Alternative für den Open Source-Webserver Apache liefert das Mono-Projekt.
- **SQL Server Runtime Host:** Codeausführung innerhalb der Datenbanksystems Microsoft SQL Server, z.B. Trigger, Funktionen, *Stored Procedures* und *User Defined Types*.

Alle *Runtime Hosts* können direkt mit einer Datenbank oder über einen Web Server/Application Server kommunizieren. Beim Silverlight-Host und dem Browser-Host ist der Direktzugriff auf Datenbanken aber eine Randerscheinung.

Abb. 16.3-1: .NET-Anwendungsarten.

I 16 Softwaretechnische Infrastrukturen

Laufzeitumgebung

Alle *Runtime Hosts* greifen auf eine gemeinsame Laufzeitumgebung zu, die *Common Language Runtime* (CLR). Die CLR übernimmt zentrale Aufgaben, insbesondere:

- Übersetzung der von .NET verwendeten plattformneutralen Zwischensprache (genannt *Managed Code*) in die plattformspezifische Maschinensprache (genannt *Native Code*).
- Automatische Speicherverwaltung durch einen *Garbage Collector*.
- Fehlerbehandlung (*Exception Handling*).
- Abschottung des Programmcodes von den Betriebssystemressourcen (*Sandboxing*).
- Abgrenzung von Anwendungen innerhalb eines Prozesses durch *Application Domains*.
- Interoperabilität mit Nicht-.NET-Anwendungen im gleichen Prozess.
- Bereitstellung einer umfangreichen Klassenbibliothek.

Programmiersprachen

Während die Programmiersprache Java in den 1990er Jahren mit dem Leitsatz »Eine Sprache für alle Plattformen« auf dem Markt getreten ist, zielte Microsoft zunächst mehr auf »Eine Plattform für alle Sprachen« ab. Von Anfang an stand die Integration zahlreicher verschiedener Programmiersprachen im Mittelpunkt von .NET. Inzwischen weichen die beiden gegensätzlichen Positionen von Java und .NET auf: Es gibt .NET-Laufzeitumgebungen für andere Plattformen und es gibt Compiler für andere Programmiersprachen, die auch Java-Bytecode erzeugen können.

Insgesamt existieren mittlerweile über 70 verschiedene Programmiersprachen für .NET. Darunter sind nicht nur objektorientierte Sprachen wie C#, Java, C++, Visual Basic und Delphi, sondern auch funktionale Sprachen wie SML, Caml und Haskell sowie »alte Tanten« wie Fortran und Cobol vertreten. Es gibt sowohl kommerzielle als auch kostenlose Sprachen.

Bibliotheken

Die .NET-Laufzeitumgebung umfasst mehrere tausend vordefinierte Klassen (abhängig von der Implementierung und der Version), die alle .NET-Sprachen verwenden können. Während alle Klassen in .NET offiziell eine einzige Klassenbibliothek bilden, die .NET Framework Class Library – FCL, etabliert es sich zunehmend in der Literatur und der Praxis, von der FCL als einer Sammlung verschiedener Bibliotheken zu sprechen.

16.3 Die .NET-Plattform I

Die Klassenbibliothek ist in Namensräume organisiert. Einige Teile der Klassenbibliothek haben neben dem Namensraum noch einen alternativen Namen. Die Tab. 16.3-1 listet die wichtigsten .NET-Bibliotheken auf.

| Namensraum (Alternativer Name) | Beschreibung |
|--|---|
| Microsoft.CSharp | C#-Compiler |
| Microsoft.JScript | JScript .NET-Compiler |
| Microsoft.VisualBasic | Visual Basic-Compiler |
| Microsoft.Win32 | Zugriff auf die Registrierungsdatenbank & Systemereignisse |
| System | Elementare Datentypen, Kommandozeilenfenster, Speicherverwaltung, <i>Application Domain</i> -Verwaltung |
| System.ServiceProcess | Erstellung von & Kontrolle über Windows-Systemdienste |
| System.Activities (Windows Workflow Foundation) | Computergestützte Arbeitsabläufe |
| System.AddIn (Managed Add-In-Framework, MAF) | Erstellen von Add-Ins (komplexeres Add-In-Modell mit Prozessisolierung) |
| System.CodeDom (Code Document Object Model) | Zugriff auf Quellcode, Erstellen & Kompilieren von Code |
| System.Collections | Objektmengen (untypisiert/typisierte, threadsicher/nicht-thread-sicher) |
| System.ComponentModel | Unterstützung für Komponenten, die in einen Komponentencontainer aufgenommen werden können |
| System.ComponentModel.Composition (Managed Extensibility Framework, MEF) | Erstellen von Add-Ins (einfaches Add-In-Modell ohne Prozessisolierung) |
| System.Configuration | Zugriff auf Assembly-Konfigurationsdaten & globale Konfigurationsdaten |
| System.Configuration.Install | Erstellung von Installationsprogrammen |
| System.Data (ADO.NET) | Zugriff auf Datenquellen aller Art, insbesondere DBs |
| System.Deployment (Click-Once-Deployment) | Verteilten von Anwendungen |
| System.Device.Location | Klassen zur Unterstützung ortsbasierter Dienste |
| System.Diagnostics | Debugging, Tracing, Systemprozesse, Ereignisprotokoll, Performance-Counter |

Tab. 16.3-1: Die wichtigsten Bibliotheken im Microsoft .Net Framework Version 4.0. Teil a.

I 16 Softwaretechnische Infrastrukturen

Tab. 16.3-2: Teil b.

| Namensraum (Alternativer Name) | Beschreibung |
|---|---|
| System. Diagnostics.Contracts (Code Contracts) | Vor- & Nachbedingungen sowie Invarianten |
| System. Diagnostics.Eventing | Ereignisprotokolle |
| System. Diagnostics.SymbolStore | Unterstützung von Debug-Symbolen |
| System.DirectoryServices | Zugriff auf Verzeichnisdienste aller Art, insbesondere LDAP und Active Directory, einschl. Directory Services Markup Language (DSML) |
| System.Drawing (Windows Graphics Device Interface, GDI+) | Zeichnen und Drucken |
| System.Dynamic (Dynamic Languages Runtime, CLR) | Dynamische Typisierung / Unterstützung für dynamische Sprachen |
| System.EnterpriseServices | Zugriff auf COM+-Dienste |
| System.Globalization | Ländereinstellungen |
| System.IO | Dateisystem- & Dateizugriff |
| System.IO.Compression | Datenkomprimierung |
| System.IO.IsolatedStorage | Unterstützung isolierter Speicherbereiche im Dateisystem |
| System.IO. MemoryMappedFiles | Inter-Prozess-Kommunikation mit Memory Mapped Files |
| System.IO.Packages | Zusammenfassen von Daten zu einem ZIP-Paket |
| System.IO.Pipes | Benannte & unbenannte Pipes |
| System.IO.Ports | Zugriff auf die seriellen Ports |
| System.Linq (Language Integrated Query, LINQ) | Allgemeine Such- & Abfragesprache |
| System.Management (Windows Management Instrumentation, WMI) | Netz- & Systemmanagement |
| System.Media | Abspielen von Audio-Daten |
| System.Messaging (Microsoft Message Queue Service, MSMQ) | Nachrichtenwarteschlangen |
| System.Net | Zugriff auf Netzwerkprotokolle (TCP, UDP, HTTP, DNS etc.) |
| System.Net.Mail | E-Mail-Versand |
| System.Net. NetworkInformation | Netzwerkverfügbarkeit, statische Daten aus dem TCP/IP-Protokoll-Stack |
| System.Net.PeerToPeer | Peer-To-Peer-Kommunikation |

16.3 Die .NET-Plattform I

Tab. 16.3-3: Teil c.

| Namensraum (Alternativer Name) | Beschreibung |
|--|---|
| System.Net.Sockets | Zugriff auf die Socket-Schnittstelle |
| System.Numerics | Große & komplexe Zahlen |
| System.Reflection | Metadaten von .NET-Komponenten, Codeerzeugung zur Laufzeit |
| System.Resources | Unterstützung kulturabhängiger Ressourcen |
| System.Runtime.Caching | Zwischenspeicherung von Werten (Caching) |
| System.Runtime.CompilerServices | Unterstützung für Compiler-Bau |
| System.Runtime.InteropServices | Interoperabilität zu COM-Komponenten |
| System.Runtime.Remoting (.NET Remoting) | Objektorientierte Inter-Prozess-Kommunikation über HTTP, TCP und IPC |
| System.Runtime.Serialization | Serialisierung von .NET-Objekten (Binär, SOAP) |
| System.Security | Verschlüsselung, Signierung, Zugriffsrechte, Sandboxing |
| System.ServiceModel (WCF) | Serviceorientierte Inter-Prozess-Kommunikation über HTTP, TCP, SMTP, MSMQ u. a. mit zahlreichen Serialisierungsformaten z. B. SOAP, MTOM, RSS, ATOM, JSON sowie einem eigenen Binärformat |
| System.ServiceModel.Routing | Routing von Nachrichten |
| System.ServiceModel.Syndication | Unterstützung für RSS und ATOM |
| System.Text | Zeichenkettenfunktionen, Textcodierung |
| System.Text.RegularExpressions | Reguläre Ausdrücke |
| System.Threading | Unterstützung von Multi-Threading / Multi-Tasking |
| System.Timers | Zeitgesteuerte Ereignisse |
| System.Transactions | Transaktionssteuerung |
| System.Web (ASP.NET) | Serverseitige Webanwendungen, die HTML, CSS & JavaScript für den Browser erzeugen |
| System.Web.ApplicationServices & System.Web.Security | Benutzer- & Rollenverwaltung, Authentifizierung |
| System.Web.Profile | Benutzerprofildatenverwaltung |
| System.Web.Services (ASP.NET Webservices) | SOAP-basierte Webservices (älteres Modell, ersetzt durch System.ServiceModel) |
| System.Windows | Steuerelemente & andere visuelle Elemente für die Gestaltung von Windows-Desktop-Anwendungen mit WPF |

I 16 Softwaretechnische Infrastrukturen

Tab. 16.3-4: Teil d.

| Namensraum (Alternativer Name) | Beschreibung |
|--|--|
| System.Windows.Forms | Steuerelemente für die Gestaltung von Windows-Desktop-Anwendungen mit Windows Forms |
| System.Workflow (Windows Workflow Foundation) | Computergestützte Arbeitsabläufe (älteres Modell, ersetzt durch System.Activities) |
| System.Xaml | Allgemeine Unterstützungsklassen für die Arbeit mit XAML (ein XML-Dialekt, der u. a. in WPF/Silverlight und Workflow verwendet wird) |
| System.Xml | Lesen und Bearbeiten von XML insbesondere Standards wie XML DOM, XLST, XPath, XSD |

16.3.2 Zur .NET-Architektur

Softwarekomponentenkonzept

Die Unterstützung wiederverwendbarer Softwarekomponenten gehört zu den Kernkonzepten von .NET. Im .NET Framework wird als *Assembly* (im Deutschen von einigen Autoren auch »Assemblierung« genannt) eine .NET-Softwarekomponente bezeichnet. Eine *Assembly* liegt in komplizierter Form als eine DLL oder EXE vor. Sie enthält eine oder mehrere .NET-Klassen. Eine EXE-Assembly (alias Managed EXE) ist eine startbare Anwendung. Eine DLL-Assembly (alias Managed DLL) kann nicht einzeln gestartet werden. Ihr Zweck ist die Verwendung im Rahmen einer anderen Anwendung. Sowohl EXE- als auch DLL-Assemblies sind wieder verwendbare Softwarekomponenten. Eine echte Unterscheidung in Anwendungen und Komponenten gibt es nicht mehr.

Assemblies bestehen meist aus nur einer Datei (Single-File-Assembly). Grundsätzlich möglich ist aber die Verteilung auf mehrere Dateien (Multi-File-Assembly). Der Autor einer *Assembly* kann diese mit einer digitalen Signatur (genannt *Strong Name*) versehen. Der Nutzer einer *Assembly* kann damit die Authentizität und Integrität der Softwarekomponente prüfen. Zudem kann der Nutzer das Sicherheitssystem von .NET so konfigurieren, dass nur die *Assembly* bestimmter Autoren zur Ausführung gelangen können.

Neben der *Assembly* als Softwarekomponente gibt es im .NET Framework noch eine alternative Definition des Begriffs Softwarekomponente: »While the term *component* has many meanings, in the .NET Framework a component is a class that implements the *System.ComponentModel.IComponent* interface or one that derives directly or indirectly from a class that implements this interface« [MSDN01]. Nimmt man diese Aussage wörtlich, dann sind Komponenten keine ganzen *Assembly*, sondern nur einzelne Klassen, die

die Schnittstelle `IComponent` besitzen. Diese Definition steht jedoch im Widerspruch zu der Gleichsetzung von kompilierten Dateien (DLL, EXE) und Softwarekomponenten, die sich in vielen anderen MSDN-Dokumenten (z. B. [MSDN02] und [MSDN03]) findet.

Wenn man einen Komponentenbegriff im engeren Sinn – Komponente als binärer Softwarebaustein (siehe »Was ist eine Softwarearchitektur?«, S. 23) – zugrunde legt, so könnte man eine *Assembly* als Softwarekomponente bezeichnen. Vielfach wird dies auch in der Microsoft-.Net-Welt so getan. Nach der UML ist eine *Assembly* jedoch streng genommen ein <<artifact>> und kann Komponenten manifestieren. Eine Klasse, die die Schnittstelle `IComponent` implementiert, ist eine Komponente im Sinne der UML. Die Klasse `System.Windows.Forms.Control` erbt z. B. von `Component`, der Standardimplementierung von `IComponent`. Jedes Windows Forms-Steuerelement ist also eine Komponente im Sinne der UML.

Hinweis

In der Praxis werden Softwarekomponenten auf Basis von *Assemblies* realisiert. Die Nutzung von `IComponent` ist sehr selten.

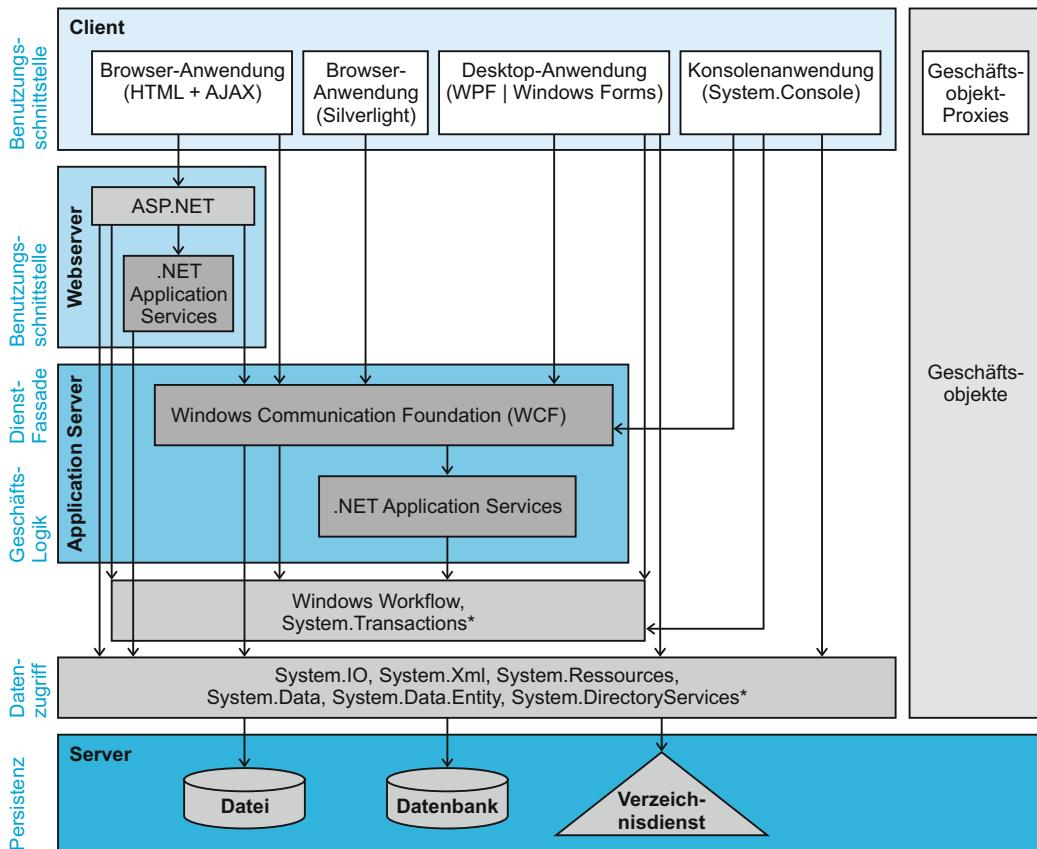
.NET-Anwendungsarchitektur

Die Abb. 16.3-2 zeigt mehrere Varianten typischer .NET-Anwendungsarchitekturen mit verschiedenen Clients und verschiedenen Datenspeichern. Während sich in der .NET-Welt die Trennung der Anwendungsarchitekturen in logische Schichten weitestgehend durchgesetzt hat, werden bei der physikalischen Infrastruktur häufig weiterhin nur zwei Schichten eingesetzt. Gründe sind insbesondere der zunächst durch jede weitere physikalische Schicht eintretende Leistungsverlust und den durch jede weitere physikalische Schicht steigenden Entwicklungsaufwand.

Bei Konsolenanwendungen sowie Desktop-Anwendungen besteht die Wahl, diese mit zwei Schichten zu implementieren mit direktem Zugriff auf die Datenspeicher oder aber einen *Application Server* in Sinne der Drei-Schichten-Architektur zwischenzuschalten. Im Fall einer Zwei-Schichten-Architektur werden Geschäftslogik- und Datenzugriffsbibliotheken (z. B. `System.Data`) im Prozess des Clients ausgeführt.

Bei einer Silverlight-Anwendung besteht der Weg mit zwei Schichten nicht, außer wenn man die Silverlight-Anwendung außerhalb der *Sandbox* ausführt und dann auf ältere COM-Komponenten für den Datenzugriff zugreift (diese Ausnahme ist in der Abb. 16.3-2 nicht eingezzeichnet). Der Regelfall ist der Zugriff auf einen *Application Server*, der die Geschäftslogik- und Datenzugriffsbibliotheken in seinem Prozess ausführt, da Silverlight diese Bibliotheken nicht selbst besitzt.

I 16 Softwaretechnische Infrastrukturen



Legende: *Wahlweise in Client-, Webserver- oder Application Server-Prozess geladen
→ = Zugriff

Abb. 16.3-2: Typische .NET-Anwendungsarchitekturen. Bei HTML- und AJAX-basierten Webanwendungen auf Basis von ASP.NET ist immer ein Webserver involviert. Hier können wahlweise die Geschäftslogik- und Datenzugriffsbibliotheken in den Webserververprozess geladen werden oder aber diese Bibliotheken können wieder über einen Application Server genutzt werden. Ein AJAX-basierter Client kann auch direkt mit einem Application Server kommunizieren.

Seitlich ganz rechts in der Abb. 16.3-2 sind selbstdefinierte Geschäftsobjekte eingezeichnet, die durch alle Schichten durchgereicht werden. Clients, die über einen Application Server kommunizieren, können wahlweise statt der Original-Geschäftsobjekte entsprechende Proxy-Klassen verwenden. Dies ist zum Beispiel beim Einsatz von AJAX notwendig, da JavaScript die in .NET auf dem Application Server definierten Geschäftsobjekte nicht einbinden kann.

Geschäftslogik

Softwarekomponenten, die in der Geschäftslogik einer .NET-Anwendung zum Einsatz kommen, sind grundsätzlich normale .NET-Assemblies, unabhängig davon, ob sie im Anwendungsserver gehostet werden oder Teile des eigentlichen Anwendungsprozesses sind. Mit bestimmten Grenzen lassen sich Geschäftslogik-Assemblies auch zwischen verschiedenen Implementierungen austauschen, z.B. im .NET Framework und Silverlight ohne Neukompilierung gleichzeitig nutzen.

Auf der Ebene der Geschäftslogik bietet Microsoft bisher recht wenig Unterstützung. Seit .NET 2.0 gibt es verschiedene allgemeine Anwendungsdienste (z. B. Benutzerverwaltung, Rollenverwaltung und Profildatenspeicherung) und seit .NET 3.0 die Windows Workflow Foundation (WF).

Anwendungsdienste

.NET stellt seit Version 2.0 einige Anwendungsdienste bereit, die Entwickler in .können. Die Anwendungsdienste bestehen aus vordefinierten Klassen, Datenbankschemata und Konfigurationsmöglichkeiten über XML-Konfigurationsdateien.

Folgende Anwendungsdienste stehen bereit:

- Mitgliedschaftssystem: Benutzerverwaltung mit Authentifizierung, Erzeugung neuer Benutzer, Kennwortvergabe, Freigeben und Sperren von Benutzern.
- Profildatensystem: Speicherung beliebiger Zusatzdaten zu Benutzern im Mitgliedschaftssystem.
- Rollensystem: Zuordnung von Benutzern zu Rollen.

Jedes dieser Anwendungsdienste unterstützt unterschiedliche Datenspeicher, z.B. die Speicherung in Datenbanken oder der Zugriff auf LDAP-basierte Verzeichnisdienste wie dem *Microsoft Active Directory*. Ursprünglich waren die Anwendungsdienste nur in ASP.NET-Webanwendungen verfügbar. In neueren .NET-Versionen können diese aber auch in Webservices bzw. über Webservices in beliebigen anderen Anwendungsarten verwendet werden.

Windows Workflow Foundation

Die *Windows Workflow Foundation* (WF) ist eine Infrastruktur für langlebige, computergestützte Arbeitsabläufe, typischerweise mit Benutzerinteraktionen, während deren der *Workflow* ruhen muss. WF ist weder ein Server noch eine Endbenutzeranwendung, sondern nur eine .NET-basierte Klassenbibliothek (Namensraum `System.Activities`, früher: `System.Workflow`) mit Laufzeitumgebung zur Erstellung selbiger. Die *Workflows* kann ein Entwickler in einem Designer-Werkzeug innerhalb von Visual Studio grafisch gestalten. Dieses Designer-Werkzeug kann auch in eigene .NET-Anwendungen

I 16 Softwaretechnische Infrastrukturen

integriert werden, sodass die Gestaltung von *Workflows* auf Endbenutzer übertragen werden kann. Die Speicherung der *Workflows* erfolgt in XML-Dateien.

Zur Unterstützung der Langlebigkeit kann WF einen *Workflow* persistieren und zu einem späteren Zeitpunkt an der persistierten Stelle wieder aufnehmen. Zudem unterstützt WF eine Ablaufverfolgung (*Tracking*), mit der nachvollzogen werden kann, wann sich ein *Workflow* in welchem Zustand befand.

WF unterstützt verschiedene Arten von Workflow-Sprachen, zum Beispiel Sequenzdiagramme, Flussdiagramme und Zustandsdiagramme. Alle bisher verfügbaren Notationen sind jedoch Microsoft-proprietary.

WF selbst stellt nur primitive Workflow-Notationen wie Sequenz, Variablenzuweisung, Schleife, Bedingungen, Methodenaufruf, Serviceaufruf, Parallelisierung, Persistenz, Verzögerung, Transaktion und Kompensation bereit. Es obliegt dem Entwickler selbst, Geschäftsprozessaktivitäten zu schaffen.

Kritisch bei WF ist die gegenüber normalem Programmcode stark verlangsamte Ausführungszeit.

16.3.3 Werkzeuge

Das Microsoft .NET Framework ist in modernen Windows-Betriebssystemen enthalten (z. B. Windows 7 und Windows Server 2008 R2 enthalten das .NET Framework Version 3.5 Service Pack 1). Später ausgelieferte Versionen des .NET Frameworks (z. B. 4.0) können bei Microsoft.com kostenlos heruntergeladen und zusätzlich installiert werden.

Das .NET Framework selbst enthält bereits Compiler für die Programmiersprachen C#, Visual Basic .NET, JScript.NET sowie die Microsoft Intermediate Language (MSIL). Es wird aber weder eine Entwicklungsumgebung noch ein spezieller Editor mitgeliefert. Grundsätzlich kann jeder beliebige Texteditor zur Erfassung des Quellcodes eingesetzt werden. Dies ist jedoch nicht komfortabel.

Visual Studio

Visual Studio gibt es in verschiedenen Varianten, einige davon sind kostenfrei. Es gibt zahlreiche Erweiterungen (genannt *Add-Ins* oder *Extensions*) für Visual Studio zum Beispiel für weitere Programmiersprachen, Hilfen bei der Codeeingabe und die Integration in Quellcodeverwaltungen. Es gibt sowohl kostenfreie Erweiterungen als auch kommerzielle Erweiterungen, sowohl von Microsoft als auch anderen Anbietern.

Anders als in der Java-Welt findet man darüber hinaus nur noch wenige alternative Entwicklungsumgebungen in der .NET-Welt. Heute gibt es noch zwei Open Source-Entwicklungsumgebungen, SharpDevelop und MonoDevelop als Alternative zu Visual Studio. Es gibt nur wenige Entwicklungsumgebungen für .NET-Sprachen abseits von C# und Visual Basic, mit denen man .NET-Anwendungen erstellen kann, z.B. Embarcadero RAD Studio für Delphi .NET und Eiffel Studio für Eiffel.NET. Kommerzielle Unternehmen konzentrieren sich darauf, funktionale Erweiterungen für Visual Studio zu liefern. Alle Initiativen, .NET in die quelloffene Entwicklungsumgebung Eclipse zu integrieren, sind bisher im Sande verlaufen.

Erstellung einer .NET-Anwendung mit Visual Studio

Das übergeordnete Konzept in Visual Studio sind Projektmappen (*Solutions*). Eine Projektmappe enthält ein oder mehrere Projekte. Ein Projekt basiert auf einer Projektvorlage. Visual Studio stellt zahlreiche Projektvorlagen bereit, z.B. für die verschiedenen Oberflächentechniken, aber auch zur Integration von .NET-Programmcode in andere Anwendungen wie Microsoft Office, Microsoft SQL Server und Microsoft SharePoint sowie Microsoft Cloud-Plattform »Windows Azure«.

In den meisten Fällen entsteht aus einem Projekt durch Kompilieren ein Kompilat in Form einer DLL- oder EXE-Datei. In einigen Ausnahmefällen (z.B. Webanwendungen) entstehen aus einem Projekt auch mehrere Kompilate. Innerhalb einer Projektmappe können Projekte in unterschiedlichen Programmiersprachen gemischt werden. Ein Projekt ist aber im Regelfall in einer Programmiersprache umzusetzen. Auch hier bilden Webprojekte eine Ausnahme, weil man dort für jede einzelne Webseite die Programmiersprachen frei wählen kann. Innerhalb eines Projekts gibt es Projektelemente, z.B. Klasse, Fenster, Steuerelement, XML-Ressourcen, XML-Konfigurationsdatei, Datenbankdatei. Auch für die Projektelemente stellt Visual Studio zahlreiche Vorlagen bereit.

Projekte können innerhalb der Projektmappe in Ordner gegliedert werden. Auch innerhalb eines Projekts können Ordner verwendet werden, um die Projektelemente zu gruppieren. Die Ordnerstruktur hat nur indirekt Einfluss auf das Kompilat. In einigen Programmiersprachen beeinflusst die Ordnerstruktur bei der Erstellung einer Klasse deren Namensraum. Der Namensraum kann aber manuell geändert werden. Auch darf der Name einer Datei von der in der Datei realisierten .NET-Klasse abweichen. Visual Studio stellt neben verschiedenen Quelltexteditoren auch grafische Editoren (»Designer«) bereit, z.B. für Windows Forms, Windows Presentation Foundation (WPF), Webforms/HTML-Dokumente, Bitmaps (.bmp, .cur, .ico), Klas-

I 16 Softwaretechnische Infrastrukturen

sendiagramme, XML-Ressourcendateien (.resx), XML-Schemata (.xsd), Workflows, typisierte *DataSets* und Objektrelationale Abbildungen (ORM). Die Abb. 16.3-3 zeigt ein Beispielprojekt in Visual Studio 2010.

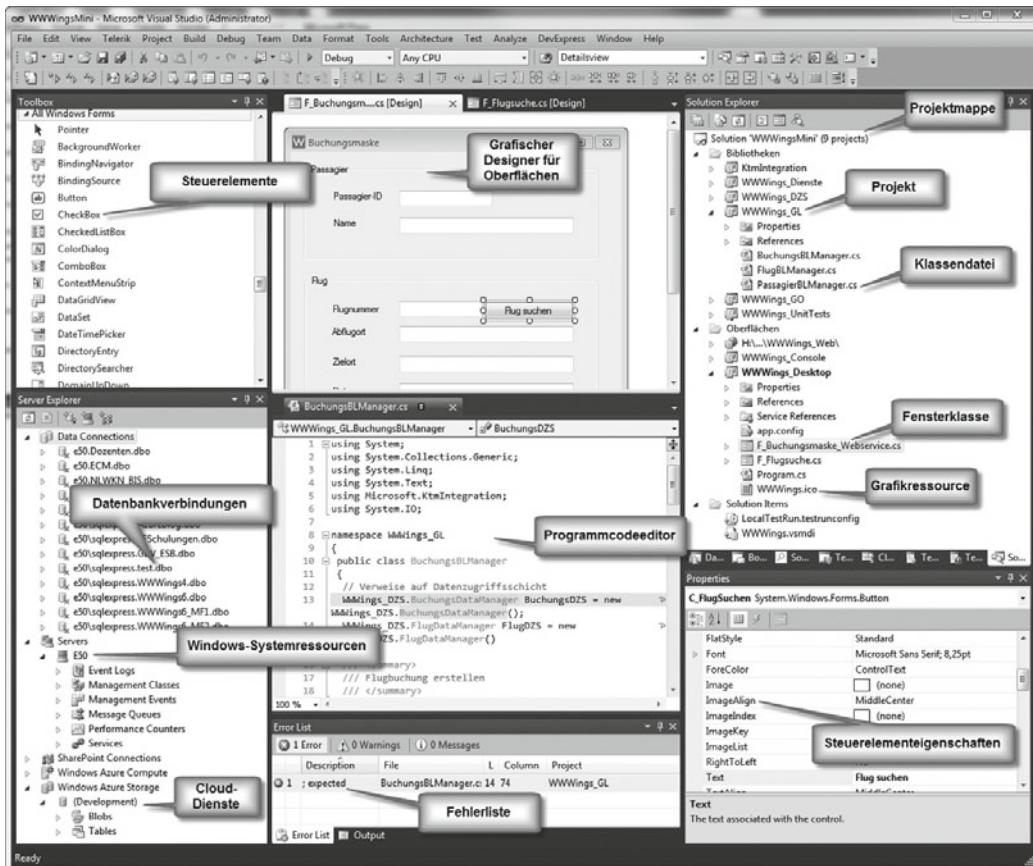


Abb. 16.3-3:
Beispiel für ein
Projekt in
Microsoft Visual
Studio 2010.

Mehrschichtige .NET-Anwendungen vs. RAD

.NET erlaubt sowohl monolithisches Entwickeln (Oberfläche, Logik und Datenzugriff in einer Softwarekomponente) im Sinne des *Rapid Application Development* (RAD) als auch das mehrschichtige Entwickeln (oft auch als *Enterprise Development* bezeichnet).

RAD-Anwendungen entstehen oft durch den Einsatz von Assistenten und Ziehen/Fallenlassen innerhalb einer grafischen Entwicklungsumgebung. Ein Entwickler kann so in wenigen Minuten umfangreiche Anwendungen »zusammenklicken«, für die er mit »richtigem« Programmieren mehrere Stunden oder Tage benötigte hätte. Leider ist

16.4 Infrastrukturen für serviceorientierte Architekturen I

das Ergebnis von RAD in den seltensten Fällen wiederverwendbar, wartbar, skalierbar oder sicher. Ein typisches Merkmal von RAD ist die enge Vermischung von Darstellung, Logik und Datenzugriff.

Microsoft hat in den letzten Jahren sehr viel Wert auf das Mehrschichtenmodell gelegt und dabei zum Teil Entwickler vernachlässigt, die sich an extremem RAD im Sinne von Microsoft Access, Visual Basic 6.0 oder Visual FoxPro begeistern können. Mit der Technik LightSwitch versucht Microsoft, RAD und Mehrschichtigkeit innerhalb des .NET-Konzeptes zu verheiraten.

16.4 Infrastrukturen für serviceorientierte Architekturen

Bei der serviceorientierten Architektur (**SOA**) handelt es sich um ein technikunabhängiges **Konzept**, bei dem Softwaresysteme aus lose gekoppelten Funktionsbausteinen (Services) mit klar umrissenen fachlichen Aufgaben aufgebaut werden.

Eine SOA lässt sich nach [BFB+07] in folgende drei Schichten strukturieren (siehe auch »Das Schichten-Muster (*layers pattern*)«, S. 46):

3 Schichten

- Schicht 1: »Technische Integration« oder »Enterprise Application Integration« (EAI): Widmet sich der technischen Anbindung von bestehenden Anwendungen an die SOA.
- Schicht 2: »Enterprise Service Bus« (ESB): Diese nächsthöhere Integrationsschicht erledigt das *Message Routing* und die Datentransformation, umso eine protokollunabhängige Kommunikationssschicht anzubieten.
- Schicht 3: »Prozessorchestrierung«: Stellt Methoden und Werkzeuge zur Verfügung, um die Verwaltung und die Orchestrierung von verteilten Geschäftsprozessen zu ermöglichen.

Schicht 1: »Technische Integration« oder EAI

Diese Schicht kann technisch auf verschiedene Art und Weise umgesetzt werden:

- Realisierung durch **Webservices**
- Einsatz von CORBA (siehe »CORBA«, S. 241)
- Verwendung von Microsoft DCOM oder Microsoft .NET Remoting (siehe »Netzkommunikation in .NET«, S. 301).

Favorisiert wird heute die Realisierung durch den Einsatz von **Webservices**. Der Unterschied zwischen Webservices und traditionellen Middleware-Ansätzen (CORBA, DCOM, .NET Remoting) besteht im Grad der Kopplung zwischen den Komponenten. Durch die standardisierten Schnittstellen von Webservices entsteht eine lose Koppelung, in der einzelne Komponenten dynamisch austauschbar sind.

I 16 Softwaretechnische Infrastrukturen

- SOAP, WSDL, UDDI Bei der Verwendung von Webservices sind **WSDL**, **UDDI** und **SOAP** die grundlegenden Bausteine einer SOA-Infrastruktur. WSDL wird benutzt, um den Service zu beschreiben. UDDI erlaubt die Registrierung eines Services und die Suche nach Services. SOAP stellt die Transportschicht zur Verfügung, um Mitteilungen zwischen Service-Konsument und Service-Provider auszutauschen. Ein Konsument kann einen Service im UDDI-Verzeichnisdienst suchen, bekommt die WSDL für den Service und kann den Service mithilfe von SOAP aufrufen.
- Java EE Die Java EE-Plattform (siehe »Die Java EE-Plattform«, S. 321) unterstützt Webservices auf verschiedene Art und Weise, sodass es möglich ist, mit Java EE eine serviceorientierte Architektur zu realisieren.
- .NET Die .NET-Plattform bietet eine umfangreiche Unterstützung mit Webservices (einschließlich der sogenannten WS-* -Protokolle wie WS-Security, WS-AtomicTransaction u. a. im Rahmen von ASP.NET sowie in der Windows Communication Foundation (WCF), siehe »Netzkommunikation in .NET«, S. 301). Neben den standardisierten Webservices bietet .NET auch proprietäre Microsoft-Protokolle wie DCOM und .NET Remoting.
- Konnektoren Um bereits bestehende Anwendungen an die SOA anzupassen, müssen Konnektoren entwickelt werden. Beispiele für Konnektorarchitekturen sind OpenEAI und OpenSyncro.

Schicht 2: »Enterprise Service Bus«

- ESB Zur Realisierung des **ESB** (*Enterprise Service Bus*) gibt es mehrere Architektur-Alternativen und Implementierungen:
- JBI (*Java Business Integration*)
 - Beispiel für eine Implementierung: Apache ServiceMix
 - SCA (*Service Component Architecture*)
 - Beispiel für eine Implementierung: Apache Tuscany
 - WCF (*Windows Communication Foundation*)
 - Beispiele für eine Implementierung: Bestandteil des .NET Framework ab Version 3.0, Microsoft BizTalk Server (BizTalk ist das EAI-/ESB-Produkt von Microsoft)
- Literatur [PiRö07]

Schicht 3: »Prozessorchestrierung«

Unternehmensanwendungen müssen die Geschäftsprozesse eines Unternehmens – evtl. einschl. der Kunden und Lieferanten – abwickeln. Zur Beschreibung von Geschäftsprozessen gibt es verschiedene Modellierungssprachen, die durch Werkzeuge ausgeführt und zur Orchestrierung verwendet werden können, z. B.

16.5 Fallstudie: KV mit Java EE als Webservice I

- BPEL (*Business Process Execution Language*)
- Beispiele für Implementierungen: ActiveBPEL, Apache ODE (*Orchestration Director Engine*) und jBPM von JBoss
- XPDL (*XML Process Definition Language*)
- Beispiele für Implementierungen: Enhydra Shark, OBE (*Open Business Engine*), WfMOpen und jBPM von JBoss
- BPMN (*Business Process Modeling Notation*)
- Beispiel für eine Implementierung: Bizagi BPMN Process Modeler
- Windows Workflow Foundation
- Beispiele für eine Implementierung: Bestandteil des .NET Framework ab Version 3.0, Microsoft BizTalk Server, Microsoft SharePoint Server

16.5 Fallstudie: KV mit Java EE als Webservice

Java EE unterstützt Webservices. Die Fallstudie »Anzeigenverwaltung« als Erweiterung der »Kundenverwaltung« soll wie folgt modifiziert werden:

Anstatt die Anzeigen über eine Weboberfläche per Hand einzutippen, möchte der Verlag Firmenkunden die Möglichkeit bieten, Anzeigen über einen Webservice einzustellen. Der Verlag, der die Zeitung »SWT-Zeitung« anbietet, entschließt sich, noch weitere Zeitungen »ET-Zeitung« und »IT-Zeitung« anzubieten. Die Abb. 16.5-1 zeigt die sich daraus ergebene Architektur.

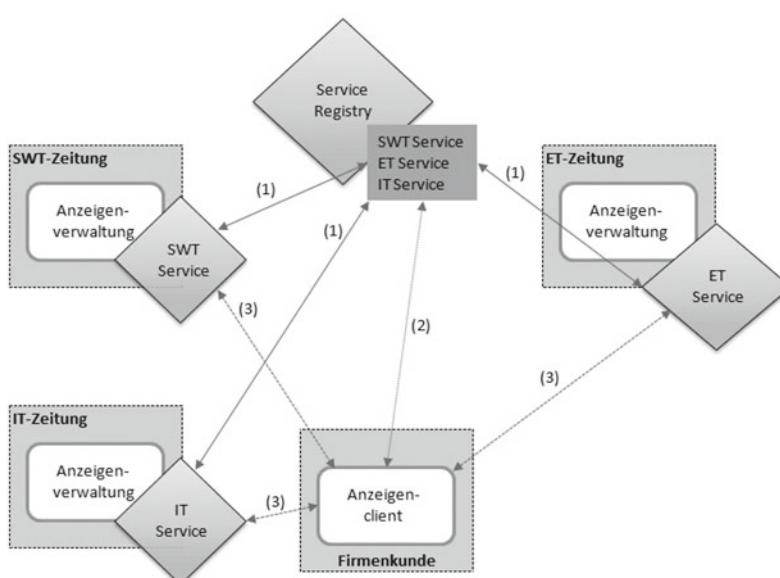


Abb. 16.5-1:
Grundlegende
Architektur aller
Webservices der
Fallstudie KV mit
Java EE als
Webservice.

I 16 Softwaretechnische Infrastrukturen

Damit potenzielle Kunden auf diesen Webservice aufmerksam werden, wird er bei einer unabhängigen UDDI-Registry registriert (1). Ein Kunde kennt zunächst nur dieses *Registry*, nicht aber die SWT-Zeitung. Er durchsucht die *Registry* anhand des allgemeinen Suchbegriffs »Zeitung« nach Webservices (2) und kann anschließend ohne großen Aufwand Anzeigen in mehreren Zeitungen schalten (3).

Zur Realisierung dieses Beispiels sind folgende Schritte notwendig:

- Die Anzeigenverwaltung muss um einen Service ergänzt werden, der ihre Funktionalität über eine standardisierte Schnittstelle zugänglich macht.
- Der Service ist in einem Service-Verzeichnis zu registrieren, damit Kunden seine genaue Adresse abfragen können.
- Auf der Seite des Kunden ist eine Client-Anwendung zu entwickeln. Diese durchsucht das Service-Verzeichnis nach Diensten, welche das Erstellen von Zeitungsanzeigen ermöglichen. Über ein Eingabeformular kann der Benutzer die Daten der zu erstellenden Zeitungsanzeige eintragen. Zusätzlich enthält das Formular eine Liste der im Service-Verzeichnis gefundenen, in Frage kommenden Dienste. Beim Absenden des Formulars wird die neue Zeitungsanzeige an alle vom Benutzer gewählten Anzeigenverwaltungen gesendet.

Webservice

Die Funktionalität der Anzeigenverwaltung soll in Form eines Webservice öffentlich zugänglich gemacht werden (Abb. 16.5-2).

Der Webservice wird im Java EE-Server ausgeführt und greift wie die JSF-Webanwendung als lokaler Client auf die Dienste der Java EE-Anwendung zu.

Die Architektur der Anzeigenverwaltung wird lediglich um die Klasse `AnzeigenService` im Paket `server.webservice` erweitert (Abb. 16.5-3).

In Java EE kann aus einer normalen Java-Klasse ohne großen Aufwand ein Webservice gemacht werden. Folgende Schritte sind durchzuführen:

- Die Annotation `@WebService` kennzeichnet eine Klasse als Webservice. Zusätzlich kann per `@Stateless` oder `@Stateful` angegeben werden, ob der Webservice den Status der Sitzung mit einem Client speichern soll.
- Alle Methoden, die den Clients zugänglich gemacht werden sollen, sind mit `@WebMethod` zu markieren.

16.5 Fallstudie: KV mit Java EE als Webservice I

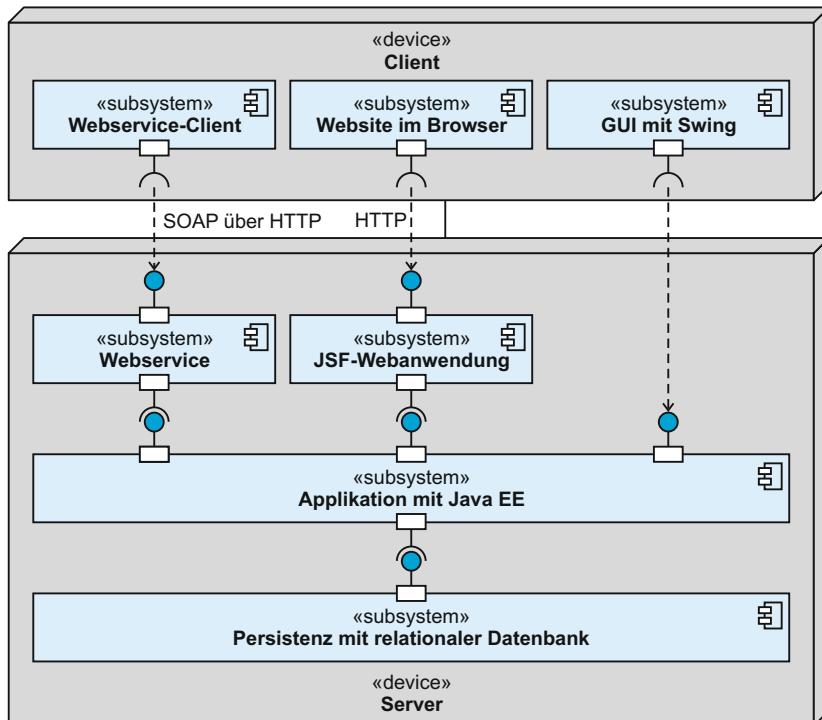


Abb. 16.5-2:
Grundlegende Architektur eines Webservice der Fallstudie KV mit Java EE als Webservice.

- Die Namen und Typen der Parameter einer mit `@WebMethod` markierten Methode können durch die Ergänzung der Annotation `@WebParam` vor jedem Eingabeparameter definiert werden.

Der folgende Ausschnitt zeigt diese Schritte am Beispiel der Klasse `AnzeigenService` der »SWT-Zeitung«:

```

package server.webservice; Java

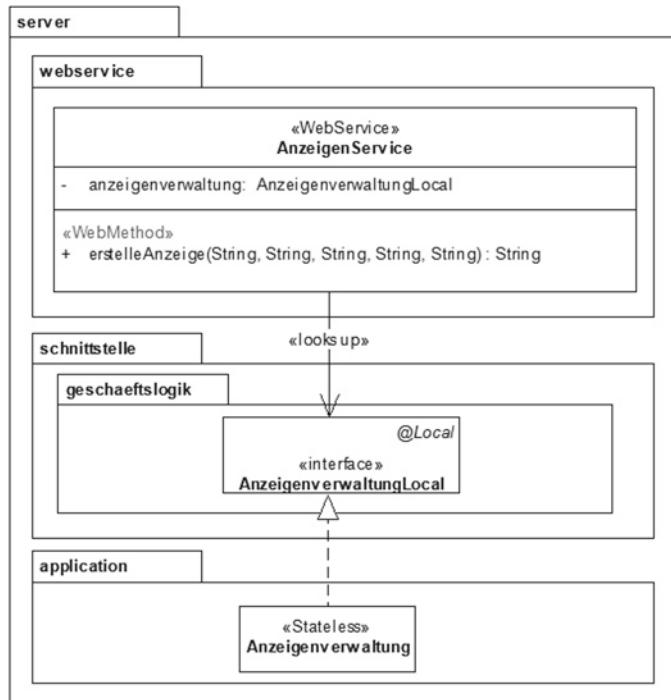
import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import server.schnittstelle.fachkonzept.Anzeige;
import server.schnittstelle.fachkonzept.Kunde;
import server.schnittstelle.geschaeftslogik.
AnzeigenverwaltungLocal;

/**
 * Web Service zum Erstellen einer neuen Anzeige
 */
@Stateless
@WebService
public class AnzeigenService {

```

I 16 Softwaretechnische Infrastrukturen

Abb. 16.5-3:
Erweiterung der Architektur um Webservices bei der Fallstudie KV mit Java EE als Webservice.



```

/*
 * Dependency Injection: Lokales Business Interface
 * der Session Bean Anzeigenverwaltung
 * Alternativ: JNDI-Lookup
 * anzeigenverwaltung = (AnzeigenverwaltungLocal)
 * ctx.lookup("avWEB-Server-earSWT/Anzeigenverwaltung/local");
 */
@EJB
private AnzeigenverwaltungLocal anzeigenverwaltung;

public AnzeigenService()
{
}
@WebMethod
public String erstelleAnzeige(
    @WebParam(name="kundennummer") String kundennummer,
    @WebParam(name="rubrik") String rubrik,
    @WebParam(name="titel") String titel,
    @WebParam(name="beschreibung") String beschreibung,
    @WebParam(name="preis") String preis)
{
    try
    {
        Kunde kunde = anzeigenverwaltung.
            getKundeZuNr(Long.valueOf(kundennummer));
        if (kunde == null)
            return "Fehler: Kunde nicht gefunden!";
    }
}
  
```

16.5 Fallstudie: KV mit Java EE als Webservice I

```
Anzeige anzeige = anzeigenverwaltung.erstelleAnzeige();
anzeige.setRubrik(rubrik);
anzeige.setTitel(title);
anzeige.setBeschreibung(beschreibung);
anzeige.setPreis(Integer.valueOf(preis));
anzeige.setKunde(kunde);
anzeigenverwaltung.speichereAnzeige(anzeige);
return "Anzeige erstellt";
}
catch(Exception e)
{
    e.printStackTrace();
    return "Beim Speichern der Anzeige ist" +
        "ein Fehler aufgetreten!";
}
}
```

Die Klasse AnzeigenService realisiert einen Webservice zum Erstellen einer neuen Anzeige. Dafür enthält sie eine mit @WebMethod gekennzeichnete Methode mit dem Namen erstelleAnzeige. Diese erhält als Eingabeparameter die Nummer des Kunden, der die Anzeige aufgibt, sowie die Rubrik, den Titel, die Beschreibung und den Preis der Anzeige. Durch das Hinzufügen der Annotation @WebParam(name="parameterName") vor jedem Eingabeparameter wird der Name festgelegt, unter welchem sich die Parameter über die Service-Schnittstelle ansprechen lassen. Die in Java vergebenen Namen der Parameter werden ohne die Annotation nicht automatisch in die Service-Definition übernommen. Fehlt diese Angabe, werden die Parameter in der Service-Definition durchnummertiert.

Die Gesamtstruktur im Detail

Die Abb. 16.5-4 zeigt die Gesamtarchitektur der Anzeigenverwaltung inklusive des Webservice, der Web-Anwendung und des Desktop-Clients .

Abgesehen von der Klasse AnzeigenService müssen keine weiteren Klassen im Vergleich zur Fallstudie »KV als Web-Anwendung« (»Fallstudie: KV mit Java EE als Web-Anwendung«, S. 351) hinzugefügt werden.

Automatische Generierung des WSDL-Dokumentes durch den Server

Beim Starten des JBoss-Anwendungsservers wird für Klassen, die mit der Annotation @WebService gekennzeichnet sind, automatisch eine Service-Definition im WSDL-Format generiert. Nach dem Serverstart kann im Browser unter der Adresse <http://localhost:8080/jbossws/services> eine Übersicht über die registrierten Webservices aufgerufen werden.

I 16 Softwaretechnische Infrastrukturen

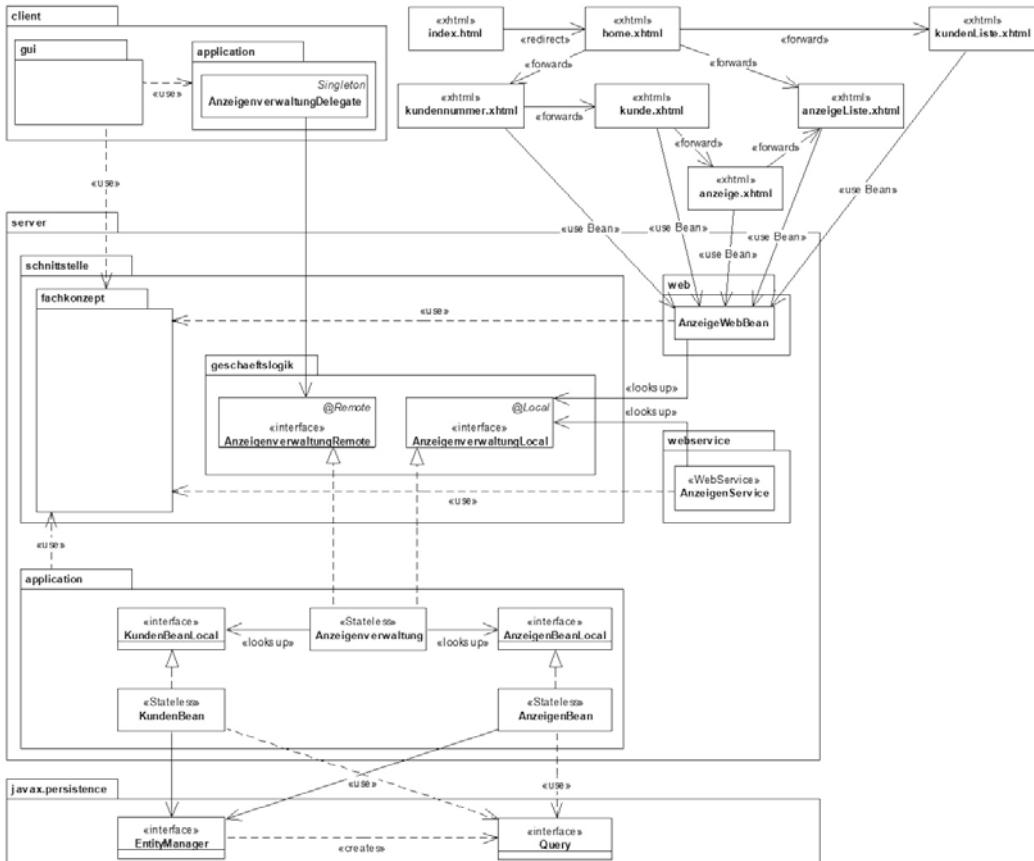


Abb. 16.5-4: Die Übersicht enthält auch Links zu den automatisch generierten WSDL-Dokumenten. Das WSDL-Dokument zur Klasse AnzeigenService der »SWT-Zeitung« ist unter der Adresse <http://localhost:8080/avWEB-Server-jarSWT/AnzeigenService?wsdl> zu finden. Der folgende Codeausschnitt zeigt die vom JBoss Anwendungsserver generierte Service-Definition dieser Klasse:

```
<?xml version='1.0' ... >
<wsdl:definitions name="AnzeigenServiceService"
    targetNamespace="http://webservice.server/">
<wsdl:types>
    <xss:schema elementFormDefault="unqualified"
        targetNamespace="http://webservice.server/"
        version="1.0" xmlns:tns="http://webservice.server/"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xss:element name="erstelleAnzeige"
            type="tns:erstelleAnzeige" />
        <xss:element name="erstelleAnzeigeResponse"
            type="tns:erstelleAnzeigeResponse" />
        <xss:complexType name="erstelleAnzeige">
            <xss:sequence>
```

16.5 Fallstudie: KV mit Java EE als Webservice I

```
<xs:element minOccurs="0"
    name="kundennummer" type="xs:string" />
<xs:element minOccurs="0"
    name="rubrik" type="xs:string" />
<xs:element minOccurs="0"
    name="titel" type="xs:string" />
<xs:element minOccurs="0"
    name="beschreibung" type="xs:string" />
<xs:element minOccurs="0"
    name="preis" type="xs:string" />
</xs:sequence>
</xs:complexType>
<xs:complexType name="erstelleAnzeigeResponse">
    <xs:sequence>
        <xs:element minOccurs="0"
            name="return" type="xs:string" />
    </xs:sequence>
</xs:complexType>
</xs:schema>
</wsdl:types>

<wsdl:message name="erstelleAnzeigeResponse">
    <wsdl:part element="tns:erstelleAnzeigeResponse"
        name="parameters">
    </wsdl:part>
</wsdl:message>

<wsdl:message name="erstelleAnzeige">
    <wsdl:part element="tns:erstelleAnzeige"
        name="parameters">
    </wsdl:part>
</wsdl:message>

<wsdl:portType name="AnzeigenService">
    <wsdl:operation name="erstelleAnzeige">
        <wsdl:input message="tns:erstelleAnzeige"
            name="erstelleAnzeige">
        </wsdl:input>
        <wsdl:output message="tns:erstelleAnzeigeResponse"
            name="erstelleAnzeigeResponse">
        </wsdl:output>
    </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="AnzeigenServiceServiceSoapBinding"
    type="tns:AnzeigenService">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="erstelleAnzeige">
        <soap:operation soapAction="" style="document" />
        <wsdl:input name="erstelleAnzeige">
            <soap:body use="literal" />
        </wsdl:input>
        <wsdl:output name="erstelleAnzeigeResponse">
            <soap:body use="literal" />
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

I 16 Softwaretechnische Infrastrukturen

```
</wsdl:output>
</wsdl:operation>
</wsdl:binding>

<wsdl:service name="AnzeigenServiceService">
    <wsdl:port binding="tns:AnzeigenServiceServiceSoapBinding"
        name="AnzeigenServicePort">
        <soap:address location="http://localhost:8080/
            avWEB-Server-jarSWT/AnzeigenService" />
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Im oberen Bereich des Dokumentes werden der Typ erstelleAnzeige und die dazugehörigen Eingabeparameter mit den in der Java-Klasse per Annotation festgelegten Namen kundennummer, rubrik, titel, beschreibung und preis definiert. Der Typ erstelleAnzeigeResponse repräsentiert den Rückgabewert der Methode erstelleAnzeige. Im mittleren Bereich des Dokumentes werden diese Typen einem Element mit dem Namen operation als input- und output-Werte zugewiesen.

Im unteren Bereich des Dokumentes wird im Abschnitt service festgelegt, dass der Webservice unter der Adresse <http://localhost:8080/avWEB-Server-jarSWT/AnzeigenService> zu erreichen ist.

Service-Verzeichnis

Wie bereits beschrieben, generiert der Anwendungsserver automatisch eine Service-Beschreibung AnzeigenService für die jeweiligen Zeitungen. Diese Service-Beschreibungen müssen jedoch ebenfalls den potenziellen Kunden zur Verfügung gestellt werden. Aus diesem Grund ist es sinnvoll, den Webservice in einem Service-Verzeichnis zu registrieren. Dazu wird eine eigenständige Konsolen-Anwendung entwickelt, mit der die Services aller Zeitungen in einem öffentlichen Service-Verzeichnis registriert werden können.

Registrierung von Services

Als Service-Verzeichnis in der Fallstudie wird jUDDI verwendet. Für den Zugriff auf XML-Verzeichnisse wie jUDDI existiert in Java die Bibliothek javax.xml.registry. Die Abb. 16.5-5 zeigt die Struktur der Konsolen-Anwendung für den Zugriff auf das Service-Verzeichnis.

Die Klasse Registrierung nutzt die aufgeführten Methoden der Klasse Abfrage. Innerhalb der init()-Methode der Klasse Abfrage wird mithilfe der Klassen aus diesem Paket javax.xml.registry zunächst eine Verbindung zum Service-Verzeichnis aufgebaut. Bevor neue Services erstellt werden, werden zunächst alle Services,

16.5 Fallstudie: KV mit Java EE als Webservice I

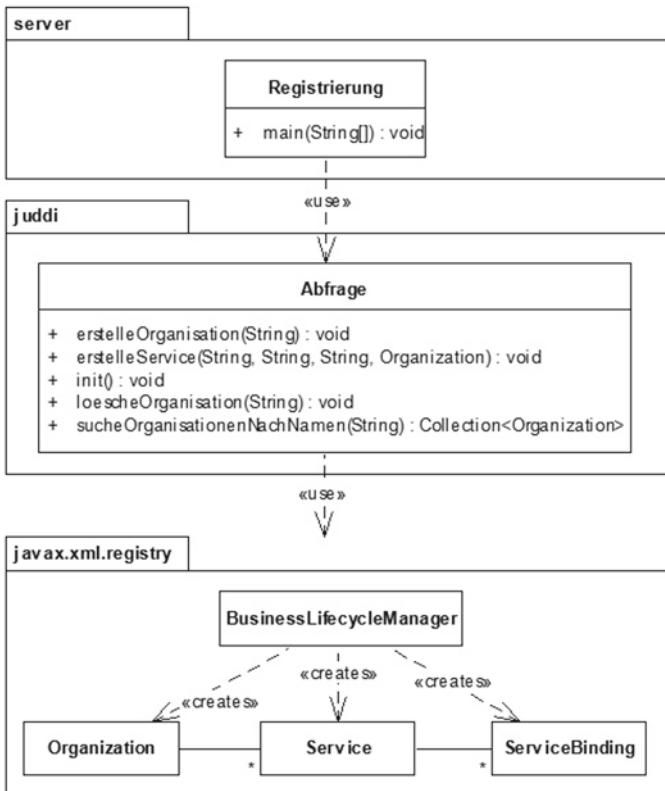


Abb. 16.5-5:
Registrierung von
Webservices bei
der Fallstudie KV
mit Java EE als
Webservice.

bei denen »Zeitung« im Namen vorkommt, gelöscht. Die Klasse **BusinessLifecycleManager** stellt Methoden zum Hinzufügen und Bearbeiten von Einträgen im Service-Verzeichnis bereit. Innerhalb von UDDI-Verzeichnissen sind Services immer Organisationen zugeordnet. Ein und derselbe Service kann unter mehreren Adressen erreichbar sein. Die Adressen dieser konkreten Service-Endpunkte werden in Objekten vom Typ **ServiceBinding** gespeichert. Die Klasse **Abfrage** nutzt den **BusinessLifecycleManager**, um die verschiedenen Zeitungen als Organisationen zu registrieren und ordnet jeder dieser Organisationen eine eigene Instanz des **AnzeigenService** zu. Der folgende Codeausschnitt zeigt die für die Registrierung von Services vorhandenen Inhalte der Klasse **Abfrage**:

```

public class Abfrage
{
    //Verbindungsparameter, Port 8090
    private static final String PUBLICATION_MANAGER_URL =
        "http://localhost:8090/juddi/publish";
    private static final String INQUIRY_MANAGER_URL =
        "http://localhost:8090/juddi/inquiry";

    //ID des von uns über die jUDDI-Webkonsole
  
```

I 16 Softwaretechnische Infrastrukturen

```
//gespeicherten Publishers "juddi"
private static final String JUDDI_PUBLISHER_ID = "juddi";

//QueryManager zum Abfragen von Registry-Inhalten
private BusinessQueryManager mBusinessQueryMgr;

//LifeCycleManager zum Schreiben von Inhalten in das Verzeich.
private BusinessLifeCycleManager mBusinessLifecycleMgr;

private Connection mConnection; //Verbindung zum UDDI Registry

public Abfrage()
{
    try
    {
        init();
    }
    catch (JAXRException e)
    {
        e.printStackTrace();
    }
}

public void init() throws JAXRException
{
    //Instanz der ConnectionFactory ermitteln,
    //um eine Verbindung zum UDDI Registry herzustellen
    ConnectionFactory jaxrConnectionFactory =
        ConnectionFactory.newInstance();

    //Verbindungsparameter wie oben definiert übergeben
    Properties uddiVerbindungsKonfiguration =
        new Properties();
    uddiVerbindungsKonfiguration.setProperty(
        "javax.xml.registry.queryManagerURL",
        INQUIRY_MANAGER_URL);
    uddiVerbindungsKonfiguration.setProperty(
        "javax.xml.registry.lifeCycleManagerURL",
        PUBLICATION_MANAGER_URL);
    uddiVerbindungsKonfiguration.setProperty(
        "javax.xml.registry.security.authenticationMethod",
        "UDDI_GET_AUTHTOKEN");
    jaxrConnectionFactory.
        setProperties(uddiVerbindungsKonfiguration);

    //Verbindung herstellen
    mConnection = jaxrConnectionFactory.createConnection();

    //QueryManager und LifecycleManager laden
    RegistryService registryService =
        mConnection.getRegistryService();
    mBusinessQueryMgr =
        registryService.getBusinessQueryManager();
    mBusinessLifecycleMgr =
        registryService.getBusinessLifeCycleManager();
```

16.5 Fallstudie: KV mit Java EE als Webservice I

```
PasswordEncoder passwordAuthentication =
    new PasswordAuthentication(
        JUDDI_PUBLISHER_ID, "".toCharArray());
Set<PasswordEncoder> credentials =
    new HashSet<PasswordEncoder>();
credentials.add(passwordAuthentication);
mConnection.setCredentials(credentials);
}

public Collection<Organization> sucheOrganisationenNachNamen
    (String suchbegriff) throws JAXRException
{
    Collection<String> findQualifiers =
        new ArrayList<String>();
    findQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);
    Collection<String> suchMuster =
        new ArrayList<String>();
    suchMuster.add("%" + suchbegriff + "%");

    //Suche nach Organisationen, deren Name
    //den Suchbegriff enthält
    BulkResponse response =
        mBusinessQueryMgr.findOrganizations(
            findQualifiers, suchMuster, null, null, null, null);
    Collection orgs = response.getCollection();
    return orgs;
}

public void erstelleOrganisation(String organisationsName)
    throws JAXRException
{
    //Erzeugen einer neuen Organisation
    //mithilfe des LifeCycleManagers
    Organization organisation = mBusinessLifecycleMgr.
        createOrganization(organisationsName);

    //Der LifeCycleManager erwartet eine Collection mit zu
    //speichernden Organisationen als Parameter
    Collection<Organization> zuSpeicherndeOrganisationen =
        new ArrayList<Organization>();
    zuSpeicherndeOrganisationen.add(organisation);

    //Organisationsliste speichern
    BulkResponse response = mBusinessLifecycleMgr.
        saveOrganizations(zuSpeicherndeOrganisationen);

    //Key der neuen Organisation ausgeben,
    //sofern keine Ausnahmen auftreten
    if (response.getExceptions() == null)
    {
        Collection<Key> responseCollection =
            response.getCollection();
        Key key = responseCollection.iterator().next();
        System.out.println("Eine neue Organisation mit dem
            Namen + \\" + key)
```

I 16 Softwaretechnische Infrastrukturen

```
        + organisationsName + "\" wurde gespeichert.");
        System.out.println("Der Organisation wurde der folgende
                           Schlüssel zugewiesen: " + key.getId());
    }
} else
{
    System.out.println("Beim Speichern der Organisation
                       + \"" + organisationsName + "\" ist ein
                       Fehler aufgetreten.");
}
}

public void erstelleService(String name, String beschreibung,
                            String wsdlUrl, Organization organisation)
throws JAXREException
{
    //Die Methoden der Klasse Service erwarten Texte als Objekte
    //der Klasse InternationalString
    InternationalString nameAlsIS = mBusinessLifecycleMgr.
        createInternationalString(name);
    InternationalString beschreibungAlsIS =
    mBusinessLifecycleMgr.
        createInternationalString(beschreibung);

    //Service erstellen
    Service service =
    mBusinessLifecycleMgr.createService(nameAlsIS);
    service.setDescription(beschreibungAlsIS);

    //Service Binding erstellen
    ServiceBinding binding =
    mBusinessLifecycleMgr.createServiceBinding();
    InternationalString serviceBindingBeschreibung =
    mBusinessLifecycleMgr.createInternationalString(
        "Binding fuer Service " + name + " mit der
        Beschreibung " + beschreibung);
    binding.setDescription(serviceBindingBeschreibung);

    //URL des WSDL-Dokumentes speichern
    binding.setValidateURI(false);
    binding.setAccessURI(wsdlUrl);

    //Service Bindings werden einem Service
    //als Liste von Bindings hinzugefügt
    Collection<ServiceBinding> serviceBindings =
        new ArrayList<ServiceBinding>();
    serviceBindings.add(binding);
    service.addServiceBindings(serviceBindings);

    //Services werden einer Organisation als
    //Liste von Services zugeordnet
    Collection<Service> services = new ArrayList<Service>();
    services.add(service);
    organisation.addServices(services);
}
```

16.5 Fallstudie: KV mit Java EE als Webservice I

```
//Der LifecycleManager erwartet eine Liste von
//Organisationen, die er speichert
Collection<Organization> zuSpeichernedeOrganisationen =
    new ArrayList<Organization>();
zuSpeichernedeOrganisationen.add(organisation);

//Speichern der Organisation und damit auch
//des neu verknüpften Service
BulkResponse response = mBusinessLifecycleMgr.
    saveOrganizations(zuSpeichernedeOrganisationen);

//Erfolgsmeldungen ausgeben, falls keine
//Ausnahmen auftreten
if (response.getExceptions() == null)
{
    Collection<Key> responseCollection =
        response.getCollection();
    Key key = responseCollection.iterator().next();
    System.out.println("Der Organisation mit" +
        " dem Schlüssel " + key.getId() + " wurde der " +
        "Service \\" + name + "\\" hinzugefügt.");
}
else
{
    System.out.println("Beim Speichern des Service \\" + name + "\\" ist ein Fehler aufgetreten.");
}

public void loescheOrganisation(String suchbegriff)
    throws JAXRException
{
    for(Organization org:
        sucheOrganisationenNachNamen(suchbegriff))
    {
        System.out.println("Organisation: " +
            org.getName().getValue());
        System.out.println("Key: " + org.getKey().getId());

        Collection keys = new ArrayList();
        keys.add(org.getKey());
        BulkResponse response =
            mBusinessLifecycleMgr.deleteOrganizations(keys);
    }
    System.out.println("");
}
```

Abfrage von Services

Für einen Kunden, der die Webservices der verschiedenen Zeitungen nutzen möchte, um eine Anzeige in mehreren Zeitungen zu schalten, muss eine Client-Anwendung entwickelt werden, welche die folgenden grundsätzlichen Funktionen erfüllt:

I 16 Softwaretechnische Infrastrukturen

- Die Client-Anwendung muss das Service-Verzeichnis nach in Frage kommenden Services zum Erfassen von Zeitungsanzeigen durchsuchen und diese auflisten.
- Die Client-Anwendung muss die Daten einer neuen Anzeige erfassen und an alle aus der Liste der verfügbaren Dienste gewählten Services versenden.

Die Abb. 16.5-6 zeigt die Architektur der Client-Anwendung.

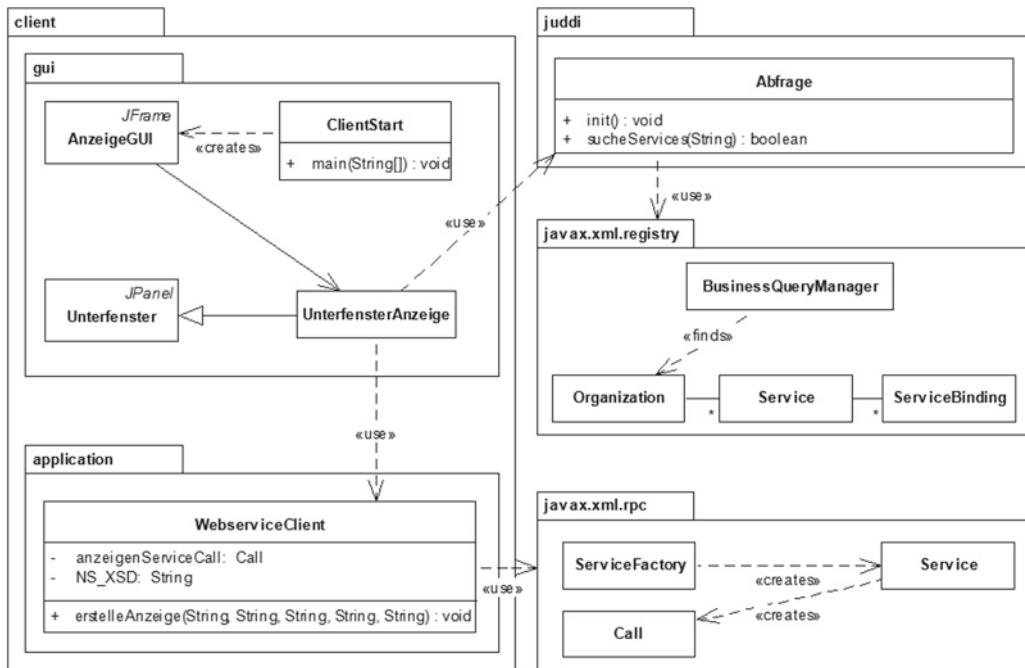


Abb. 16.5-6:
Abfrage von
Webservices bei
der Fallstudie KV
mit Java EE als
Webservice.

Die Benutzungsoberfläche wird durch vier Klassen im Paket `client.gui` realisiert. Die Klasse `ClientStart` ist für die Erzeugung eines Objekts der Klasse `AnzeigeGUI` verantwortlich. Die Klasse `AnzeigeGUI` stellt das Hauptfenster der Anwendung dar. Das Formular zum Erfassen einer neuen Anzeige wird in der Klasse `UnterfensterAnzeige` umgesetzt, welche von `Unterfenster` erbt. In das Formular der Klasse `UnterfensterAnzeige` gibt der Benutzer die Details der Anzeige sowie seine Kundennummer ein. Anschließend muss er aus einer Liste der verfügbaren Services diejenigen wählen, an welche die Anzeigendaten übermittelt werden sollen.

Die Liste der verfügbaren Services wird mithilfe der Klasse `Abfrage` vom Service-Verzeichnis abgefragt. Für den Zugriff auf das Service-Verzeichnis wird wieder die Bibliothek `javax.xml.registry` genutzt. Innerhalb der `init()`-Methode der Klasse `Abfrage` wird zunächst eine Verbindung zum Service-Verzeichnis aufgebaut. Die Suche nach Ser-

16.5 Fallstudie: KV mit Java EE als Webservice I

vices erfolgt mithilfe der Klasse BusinessQueryManager. Der folgende Codeausschnitt zeigt die beiden relevanten Methoden der Klasse Abfrage:

```
public class Abfrage
{
    //Attribute analog zur Registrierung

    public Abfrage()
    {
        //Analog zur Registrierung
    }

    public void init() throws JAXRException
    {
        //Analog zur Registrierung
    }

    public Map<String, String>
        sucheServices(String suchbegriff)
    {
        Map<String, String> serviceInfos =
            new HashMap<String, String>();
        try
        {
            for(Organization org:
                sucheOrganisationenNachNamen(suchbegriff))
            {
                System.out.println("Organisation: " +
                    org.getName().getValue());
                System.out.println("Key: " + org.getKey().getId());
                //Services zur Organisation durchlaufen
                for(Service service:
                    (Collection<Service>)org.getServices())
                {
                    System.out.println("  |____Service: " +
                        service.getName().getValue());
                    System.out.println("    |____Beschreibung: " +
                        service.getDescription().getValue());
                    //Service-Bindings zum Service aufrufen und die
                    //darin gespeicherten Links ausgeben
                    for(ServiceBinding sb:
                        (Collection<ServiceBinding>)service.
                            getServiceBindings())
                    {
                        System.out.println("      |____Link: " +
                            sb.getAccessURI());
                        serviceInfos.put(service.getName().getValue(),
                            sb.getAccessURI());
                    }
                }
                System.out.println();
                System.out.println();
            }
        }
    }
```

I 16 Softwaretechnische Infrastrukturen

```
        catch (JAXRException theException)
        {
            theException.printStackTrace();
        }
        return serviceInfos;
    }
}
```

Der Zugriff auf die aus der Liste gewählten Webservices wird in der Klasse `WebserviceClient` realisiert. Diese nutzt Klassen der Bibliothek `javax.xml.rpc`, um per *Remote Procedure Call* auf einen Webservice zuzugreifen. Die Besonderheit dieser Zugriffsmethode besteht darin, dass keine Klassen oder Schnittstellen der Server-Anwendung benötigt werden. Der folgende Codeausschnitt zeigt die Klasse `WebserviceClient`:

```
public class WebserviceClient
{
    private String NS_XSD;
    private Call anzeigenServiceCall;

    public WebserviceClient(String adresse)
    {
        try
        {
            //XSL-Namespace definieren
            this.NS_XSD = "http://www.w3.org/2001/XMLSchema";

            //Fabrik zum Erstellen der Services
            ServiceFactory factory =
                ServiceFactory.newInstance();

            Service anzeigenService = factory.createService(
                new QName(
                    "http://webservice.server/",
                    "AnzeigenService"));

            //Erzeugen eines Objekts vom Typ Call
            this.anzeigenServiceCall =
                anzeigenService.createCall(new QName(
                    "http://webservice.server/",
                    "AnzeigenServicePort"));

            this.anzeigenServiceCall.setTargetEndpointAddress(
                adresse.substring(0,adresse.length()-5));
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    public String erstelleAnzeige(
        String kundennummer, String rubrik, String titel,
        String beschreibung, String preis)
```

16.5 Fallstudie: KV mit Java EE als Webservice I

```
{  
    String result = "Beim Zugriff auf den Webservice " +  
        "ist ein Fehler aufgetreten!";  
    try  
{  
        //Namen der aufzurufenden Methode festlegen.  
        //Hier: erstelleAnzeige  
        anzeigenServiceCall.setOperationName(  
            new QName(  
                "http://webservice.server/",  
                "erstelleAnzeige"));  
  
        //Rückgabetyp der Methode definieren. Hier: string  
        QName QNAME_TYPE_STRING =  
            new QName(NS_XSD, "string");  
        anzeigenServiceCall.  
            setReturnType(QNAME_TYPE_STRING);  
  
        //Namen, Typen und Richtung (IN = Eingabeparameter)  
        //der Eingabeparameter der Methode definieren  
        anzeigenServiceCall.addParameter(  
            "kundennummer", QNAME_TYPE_STRING,  
            ParameterMode.IN);  
        anzeigenServiceCall.addParameter(  
            "rubrik", QNAME_TYPE_STRING,  
            ParameterMode.IN);  
        anzeigenServiceCall.addParameter(  
            "titel", QNAME_TYPE_STRING,  
            ParameterMode.IN);  
        anzeigenServiceCall.addParameter(  
            "beschreibung", QNAME_TYPE_STRING,  
            ParameterMode.IN);  
        anzeigenServiceCall.addParameter(  
            "preis", QNAME_TYPE_STRING,  
            ParameterMode.IN);  
  
        //Parameterwerte übergeben  
        String[] params = { kundennummer , rubrik,  
            titel, beschreibung, preis };  
  
        //Methode aufrufen und Rückgabe-String speichern  
        result = (String)  
            anzeigenServiceCall.invoke(params);  
    }  
    catch (Exception e)  
{  
        e.printStackTrace();  
    }  
    return result;  
}
```

Im Konstruktor wird zunächst mithilfe einer Fabrik ein Objekt vom Typ Service erstellt. Dabei wird der im WSDL-Dokument unter targetNamespace definierte Namensraum <http://webservice.server>

I 16 Softwaretechnische Infrastrukturen

übergeben. Anschließend wird ein Objekt vom Typ Call initialisiert, welches den Service-Aufruf kapselt. Diesem wird die Adresse des WSDL-Dokumentes übergeben. In der Methode erstelleAnzeige werden dem Call-Objekt zusätzlich der Name der aufzurufenden Operation, der erwartete Rückgabetyp und die Namen und Typen der Eingabeparameter, wie sie im oben abgebildeten WSDL-Dokument zu finden sind, mitgeteilt. Schließlich werden die konkreten Werte der Eingabeparameter in einem Feld zusammengefasst und der Webservice mithilfe des Call-Objektes unter Angabe dieser Parameterwerte aufgerufen.

Hinweis

Bevor in der Fallstudie bei mehreren Zeitungen eine Anzeige hinzugefügt werden kann, muss über die Web-Anwendung (Zugriff über Browser) ein Kunde erstellt werden. Die beim Hinzufügen genutzte Kundennummer muss bei allen Zeitungen vorhanden sein.

Installation von jUDDI

In der Fallstudie wird die kostenlose Implementierung *Java implementation of the Universal Description, Discovery, and Integration (UDDI v3) specification for (Web) Services (jUDDI)* von Apache eingesetzt. Die Installation des Tomcat-Pakets *judditomcat-2.0.1.zip* ist notwendig, welches unter http://www.apache.org/dist/ws/juddi/2_0_1/ heruntergeladen werden kann. Den im Archiv enthaltenen Ordner *apache-tomcat-5.5.27* in ein beliebiges Verzeichnis kopieren und die Umgebungsvariable CATALINA_HOME auf diesen Ordner verweisen lassen.

Damit *jUDDI* parallel zum JBoss-Server gestartet werden kann, muss der HTTP-Port in der Datei *server.xml* im Unterordner *conf* an der Stelle Define a non-SSL HTTP/1.1 Connector on port 8080 angepasst werden:

```
<Connector port="8090" maxHttpHeaderSize="8192" maxThreads="150"
minSpareThreads="25" maxSpareThreads="75" enableLookups="false"
redirectPort="8443" acceptCount="100" connectionTimeout="20000"
disableUploadTimeout="true" />
```

Anschließend kann der Server mithilfe der Stapelverarbeitungsprogramme *bin/startup.bat* gestartet bzw. mit *bin/shutdown.bat* gestoppt werden. Für die Ausführung der Fallstudie muss über die Web-Oberfläche von *jUDDI* ein *Publisher* angelegt werden, welcher die Berechtigung zum Lesen und Schreiben von Daten im Service-Verzeichnis besitzt. Nach dem Start des Servers kann die Web-Oberfläche unter der Adresse <http://localhost:8090/juddi-console/> aufgerufen werden.

16.5 Fallstudie: KV mit Java EE als Webservice I

Alle Aktionen in der Web-Oberfläche erfordern die Authentifizierung mithilfe eines sogenannten *AuthTokens*. Dieses kann unter dem Link `get_authToken` im Abschnitt UDDI Publish API erzeugt werden. Bei der Erstinstallation ist in *jUDDI* der Name `jdoe` mit einem leeren Passwort voreingestellt, daher müssen diese Informationen an der Stelle `userID="jdoe"` und `cred=""` eingetragen werden. Nach einem Klick auf Submit muss der Wert im Bereich `<authInfo>...</authInfo>` in die Zwischenablage kopiert werden.

Neue *Publisher* lassen sich unter dem Verweis `save_publisher` im Abschnitt *jUDDI* API anlegen. Dazu muss der Wert in der Zwischenablage im Bereich `<authInfo>...</authInfo>` eingefügt und die Werte `publisherID` bzw. `publisherName` eingetragen werden. In der Fallstudie wird *juddi* genutzt:

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv=
    "http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Body>
        <save_publisher generic="1.0" xmlns="urn:juddi-org:api_v1">
            <authInfo>
                authToken:CB13ED70-4A1E-11E0-AD70-E7F69218086E
            </authInfo>
            <publisher>
                publisherID="juddi"
                publisherName="juddi"
                admin="true"
                enabled="false"
                emailAddress="*****"/>
            </publisher>
        </save_publisher>
    </soapenv:Body>
</soapenv:Envelope>
```

Nach dem Klicken auf Submit wird der neue *Publisher* gespeichert und *jUDDI* ist erfolgreich für die Ausführung der Fallstudie eingerichtet worden.

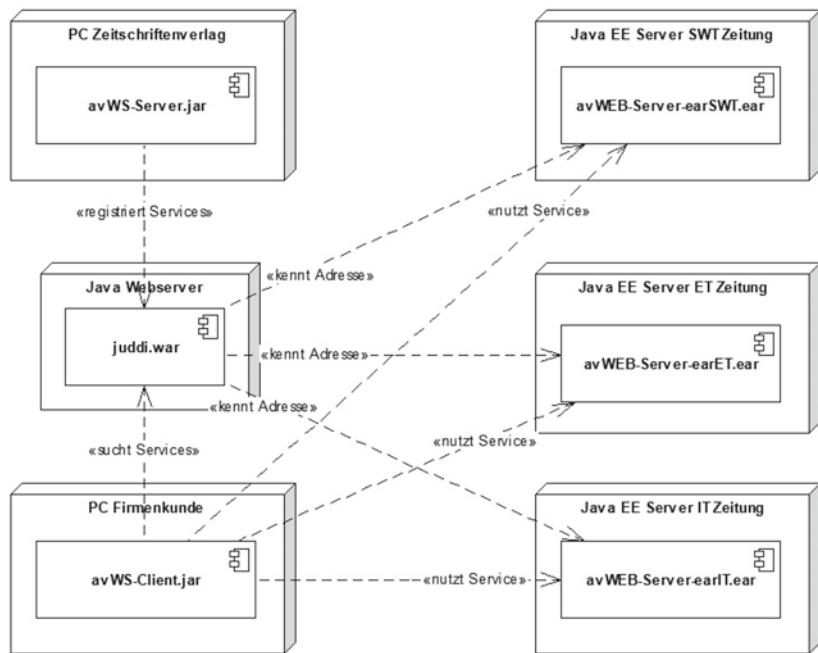
Installation der Anwendung

Die Abb. 16.5-7 zeigt die Verteilung der verschiedenen Anwendungen.

Die den Webservice enthaltende Anzeigenverwaltung wird als EAR-Anwendung auf den Anwendungsservern der verschiedenen Zeitungen ausgeführt. Somit hat jede Zeitung ihre eigene Instanz der Anwendung, welche über den Webservice `AnzeigenService` ansprechbar ist. Das Service-Verzeichnis *jUDDI* wird als Web-Anwendung auf einem Java Webserver (im Fallbeispiel auf einem Tomcat-Webserver) ausgeführt. Die für die Registrierung von Webservices notwendigen Klassen werden in die Datei `avWS-Server.jar` hinzugefügt. Diese registriert die vorhandenen Zeitungen »SWT-Zeitung«, »ET-Zeitung« sowie »IT-Zeitung« beim Service-Verzeichnis. Die für die Abfrage von

I 16 Softwaretechnische Infrastrukturen

Abb. 16.5-7:
Verteilung der
Pakete in Dateien
bei der Fallstudie
KV mit Java EE als
Webservice.



Webservices und Eintragung von Anzeigen bei mehreren Zeitungen notwendigen Klassen sind in der Datei `avWS-Client.jar` zusammengefasst und können an die potenziellen Kunden gegeben werden. Die Anwendung greift zunächst auf das Service-Verzeichnis zu, um die Adressen der Webservices zu erfragen und sendet die zu speichernden Anzeigen dann direkt an die Webservices der verschiedenen Zeitungen.

Hinweis

Alle drei Fallstudien »Client-Server-Anwendung« (»Fallstudie: KV mit Java EE als Client-Server-Anwendung«, S. 333), »Web-Anwendung« (»Fallstudie: KV mit Java EE als Web-Anwendung«, S. 351) und »Webservice« können parallel installiert werden, da kein gegenseitiger Zugriff stattfindet.

Java

Das vollständige Programm für den Client (»PC Zeitschriftenverlag« und »PC Firmenkunde«) sowie die Programme zur Erstellung der Webservices SWT (»Java EE Server SWT Zeitung«), ET (»Java EE Server ET Zeitung«) und IT (»Java EE Server IT Zeitung«) finden Sie im E-Learning-Kurs zu diesem Buch.

17 Architekturen »Eingebetteter Systeme«

Immer dann, wenn Hardware- und Softwarekomponenten in ein umfassenderes Produkt integriert sind, um produktspezifische Funktionsmerkmale zu realisieren, spricht man von **Eingebetteten Systemen** (ES). Das Spektrum von Produkten, welche unter Verwendung von »Eingebetteten Systemen« realisiert werden, umfasst zahlreiche Branchen wie etwa Fahrzeugbau, Automatisierungs- und Produktionstechnik, Luft- und Raumfahrt, Medizintechnik, Umwelt- und Energietechnik, *Consumer Electronics*, Mobilkommunikation, Bahntechnik und Sicherheitstechnik [Nati09].

Die in den »Eingebetteten Systemen« enthaltene Software übernimmt dabei typischerweise eine der folgenden Aufgaben:

- 1** Steuerung, Regelung und Überwachung der Umgebung oder von Teilsystemen.
- 2** Daten- bzw. Signalverarbeitung, wie zum Beispiel das Filtern von Sensordaten, Ver- und Entschlüsselung bei der Datenkommunikation sowie (De-)Codieren von Datenströmen.
- 3** Interaktion mit Benutzern oder anderen Systemen.
- 4** Systemmanagementfunktionen, wie Ressourcenmanagement, Konfiguration und Diagnose.

Die Rechen- und Speicherkapazität der verwendeten Knoten reicht von Kleinstrechnern bis hin zu leistungsfähigen Computern der PC-Klasse.

Im Bereich der eingebetteten Systeme wird der Architektur der Software eine entscheidende Bedeutung beigemessen, da die Architektur die Basis für alle darauf folgenden Aktivitäten bildet (siehe »Was ist eine Softwarearchitektur?«, S. 23). Das bedeutet insbesondere, dass die Architektur des eingebetteten Systems maßgeblich über die Erreichbarkeit der gestellten Qualitätsanforderungen entscheidet.

Zu den zentralen Systemqualitäten in eingebetteten Systemen zählen, aufgrund ihrer typischen Einsatzbereiche, Verlässlichkeit (siehe »Nichtfunktionale Anforderungen«, S. 109), Ressourceneffizienz (siehe »Leistung und Effizienz«, S. 128) und Echtzeitfähigkeit, die einen großen Einfluss auf die Softwarearchitektur haben.

Die Architektur eines eingebetteten Systems muss so entworfen werden, dass sie die Erfüllung der gestellten Anforderungen zum einen zulässt und zum anderen sehr wahrscheinlich macht. Dazu werden bereits in sehr frühen Phasen entsprechende Abschätzungen

I 17 Architekturen »Eingebetteter Systeme«

gemacht, die über die Tragfähigkeit potenzieller Architekturkonzepte eine Aussage ermöglichen. Dies ist essenziell, um die richtigen Entscheidungen zu treffen und falsche Entscheidungen sowie Entwurfsfehler zu frühen Zeitpunkten mit relativ geringen Kosten zu korrigieren. Damit diese Abschätzungen realistisch möglich sind, müssen allerdings entsprechende Modelle angelegt werden, die den Entscheidungsprozess letztendlich objektiv und skalierbar machen.

Im Folgenden werden dazu Architektursichten vorgestellt, die typischerweise beim Entwurf eingebetteter Architekturen zum Einsatz kommen:

- »Anforderungsspezifikation«, S. 398
- »Typische Sichten Eingebetteter Systeme«, S. 400
- »Architekturmuster für Eingebettete Systeme«, S. 406

17.1 Anforderungsspezifikation

Bevor man mit dem Entwurf der Architektur eines eingebetteten Systems beginnt, sollten alle relevanten Anforderungen, die sich auf die Auslegung der Software- und Systemarchitektur auswirken können, zu einem gewissen Grad identifiziert und analysiert sein. Dies ist besonders wichtig, da eine unvollständige Liste von Anforderungen zu kostspieligen Nacharbeiten führen kann (siehe Lehrbuch der Softwaretechnik – Basiskonzepte und Requirements Engineering, Teil IV, [Balz09a]).

Um Anforderungen an die Architektur spezifizieren zu können, müssen in der Regel drei Phasen durchlaufen werden: Erhebung, Modellierung und Analyse. Bei der Erhebung der Anforderungen müssen zunächst alle **Stakeholder** identifiziert werden. Sobald sie identifiziert wurden, werden diese in Interviews befragt. Das Ergebnis der Erhebung liefert die notwendigen Informationen, um mit der Modellierung der Anforderungen zu beginnen. Im Falle von Qualitätsanforderungen stellt sich die Herausforderung, dass diese oft nur implizit definiert oder kommuniziert sind (siehe »Nichtfunktionale Anforderungen«, S. 109).

- Beispiele
- Das System muss schnell auf Eingaben reagieren.
 - Das System muss einfach erweiterbar sein.

Aus diesem Grund werden Qualitätsanforderungen operationalisiert. Das bedeutet, man zerlegt Anforderungen so weit, bis sie quantifizierbar sind.

- Beispiel
- Die Architektur muss so ausgelegt sein, dass die Hardware des Radarsensors im Rahmen der Weiterentwicklung ohne Anpassungen der Software vom Kundendienst ausgetauscht werden kann.

17.1 Anforderungsspezifikation I

Neben den Qualitätsanforderungen gilt es auch sogenannte Einschränkungen oder Vorgaben (*Constraints*) zu identifizieren und zu analysieren. Diese können technischer, organisatorischer, politischer, gesetzlicher, normativer oder physikalischer Natur sein.

Eine normative Vorgabe im Bereich der Entwicklung sicherheitskritischer Systeme könnte die Einhaltung des Standards ISO 26262 [ISO 26262] in der Automobilindustrie sein. Beispiel

Bei komplexen Systemen ist es unerlässlich, diese in einem Werkzeug-gestützten Modell zu erfassen, zu repräsentieren, zu analysieren und zu verwalten.

In verschiedenen Domänen eingebetteter Systeme haben sich unterschiedliche Modellierungsansätze entwickelt, die mehr oder weniger in die industrielle Praxis eingegangen sind.

- EAST-ADL mit Fokus auf Automobilbau, siehe Website ATESST Beispiele (<http://www.atesst.org>).
- SAE-AADL [SAE AS5506] mit Fokus auf Luft- und Raumfahrt.
- SysML als allgemeine Modellierungssprache für Systeme, siehe Website OMG Systems Modelling Language (<http://www.omg.org/>).

Im Folgenden wird die Notation der SysML zur Veranschaulichung herangezogen.

Zur Modellierung von Anforderungen bietet die SysML ein sogenanntes **Anforderungsdiagramm**. Anforderungsdiagramme sind sehr hilfreich bei der Erfassung und Strukturierung großer Mengen von Anforderungen.

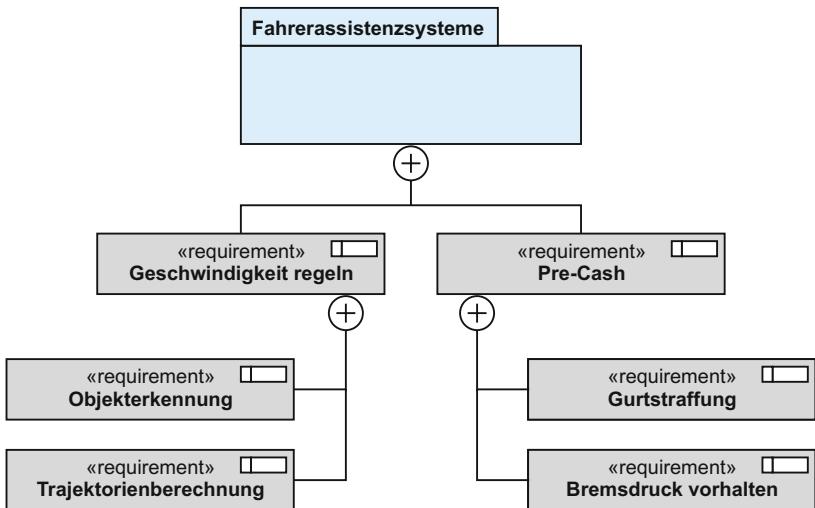
Als ein wichtiges Strukturierungsinstrument dient hierbei vor allem die Modellierung von Abhängigkeitsbeziehungen zwischen Anforderungen, die man bei Weiterentwicklungen oder Änderungen der Systemanforderungen benötigt.

Wie die Abb. 17.1-1 zeigt, können Anforderungen mittels Anforderungsdiagrammen hierarchisiert werden, um somit handhabbarer und nachvollziehbar zu werden.

Neben der reinen Hierarchisierung von Anforderungen können Anforderungen abgeleitet werden, die auf den unteren Ebenen existieren. Dazu können sogenannte Anwendungsfälle modelliert werden, die konkretere Nutzungsszenarien beschreiben, um somit sehr systematisch den Anforderungsraum zu analysieren.

I 17 Architekturen »Eingebetteter Systeme«

Abb. 17.1-1:
SysML-Anforde-
rungsdiagramm.



17.2 Typische Sichten Eingebetteter Systeme

Architekturen komplexer Systeme werden über verschiedene Sichten beschrieben (siehe »Was ist eine Softwarearchitektur?«, S. 23). Auch wenn man im Allgemeinen mit einem Satz von Standardsichten beginnen kann, um Architekturentscheidungen zu dokumentieren, müssen die benutzten Architekturensichten auf die adressierten Anforderungen, die Systemart, sowie die *Stakeholder* ausgerichtet werden. Im Folgenden wird beschrieben, wie die Standardsichten **Kontextsicht**, **Bausteinsicht**, **Dynamische Sicht** und **Verteilungssicht** an die Besonderheiten eingebetteter Systeme angepasst werden können. Es werden folgende Sichten unterschieden:

- Die Daten-zentrische Kontextsicht
- Die Daten-zentrische Bausteinsicht
- Die Verteilungssicht eingebetteter Systeme

Die Daten-zentrische Kontextsicht

Bei eingebetteten Systemen dient die Kontextsicht – ebenso wie im Falle von Informationssystemen – zur Beschreibung des Systems als »Blackbox« in seinem Ausführungskontext. Das heißt, die Kontextsicht zeigt das eingebettete System in seiner Interaktion mit anderen Akteuren. Akteure können dabei externe Systeme sein, Sensoren, Aktuatoren oder aber auch Benutzer. Im Falle von eingebetteten Systemen liegt zudem ein klarer Fokus auf Datenflüssen, die zwischen Systemen und Systembestandteilen ausgetauscht und bearbeitet werden.

17.2 Typische Sichten Eingebetteter Systeme I

Die Daten-zentrische Kontextsicht eingebetteter Softwarearchitekturen verfolgt damit die folgenden Ziele:

- Übersicht über das eingebettete Systems und seine Umgebung. Ziele
- Abgrenzung des Systems gegenüber externen Akteuren.
- Explizite Darstellung der Abhängigkeiten des Systems von bestimmten Kontextfaktoren.

Die Abb. 17.2-1 zeigt die generellen Modellelemente einer Daten-zentrischen Kontextsicht.

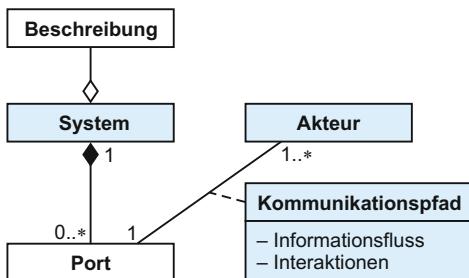


Abb. 17.2-1: Metamodell zur Beschreibung der Daten-zentrischen Kontextsicht.

Das Systemelement bezeichnet dabei das **System**, welches entworfen wird, in einer potenziellen Systemlandschaft. Das System kann darüber hinaus durch zusätzliche Informationen beschrieben sein.

Akteure sind Rollen, welche die Funktionalität und/oder Daten des Systems nutzen oder Funktionalität und Daten für das System bereitstellen. Rollen sind typischerweise externe Systeme, Benutzer, Sensoren oder Aktuatoren.

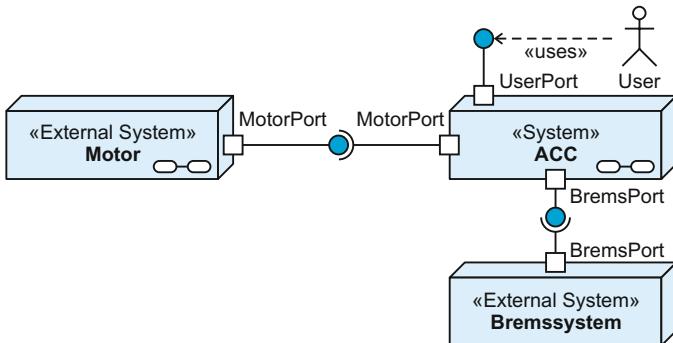
Ports sind Kommunikationsendpunkte zwischen dem System und potenziellen Akteuren. Es wird unterschieden zwischen Objektflussports und Standardports. Objektflussports werden benutzt, um Informationen zu modellieren, die in das System hinein oder aus dem System hinaus transportiert werden. Standardports werden benutzt um Interaktionen mit der Systemfunktionalität zu modellieren.

Ein **Kommunikationspfad** beschreibt dabei konkrete Interaktionen zwischen Akteuren, die über bestimmte Ports mit dem System verbunden sind. Informationsflüsse zeigen explizit den Informationsaustausch zwischen dem betrachteten System und externen Systemen.

In der Abb. 17.2-2 sieht man den Systemkontext eines Systems **Beispiel ACC**, das mit zwei externen Systemen verbunden ist. Neben dem **MotorPort**, der Informationsflüsse hin zum Motor vermittelt, werden ein Port zum Bremssystem sowie eine Benutzungsschnittstelle angeboten.

I 17 Architekturen »Eingebetteter Systeme«

Abb. 17.2-2:
Beispiel für eine
Daten-zentrische
Kontextsicht.



Die Daten-zentrische Bausteinsicht

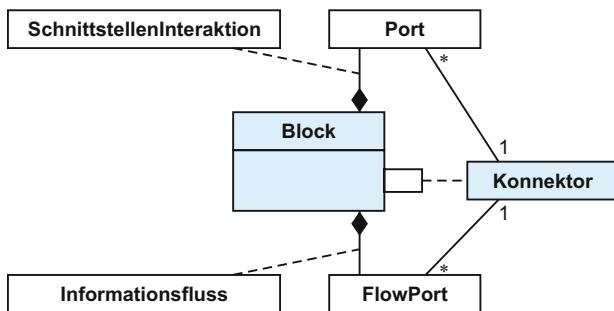
Im Gegensatz zur Kontextsicht, die das System als »Blackbox« darstellt, zeigt die Bausteinsicht die interne Organisation des Systems mit funktionalen Einheiten, den sogenannten Bausteinen. In eingebetteten Systemen arbeiten diese Bausteine häufig auf einem kontinuierlichen Datenstrom und liefern als Ausgabe wiederum Daten, die von anderen Bausteinen genutzt und weiterverarbeitet werden. Die Daten-zentrische Bausteinsicht kann über Anreicherung mit dynamischer Information als Grundlage zur Analyse von Qualitäts-eigenschaften herangezogen werden. Dazu werden die Bausteine selbst oder aber auch die Verbindungen zwischen den Bausteinen mit Hilfe von **Verhaltensdiagrammen** modelliert.

Im Kontext der Modellierung eingebetteter Softwarearchitekturen verfolgt die Daten-zentrische Bausteinsicht die folgenden Ziele:

- Ziele
- Übersicht über den internen funktionalen Aufbau des Systems.
 - Funktionale Hierarchisierung des Systems zur Beherrschung der Komplexität.
 - Bewertung und Vorhersage von Qualitätseigenschaften.

Die Abb. 17.2-3 zeigt die generellen Modellelemente einer Daten-zentrischen Kontextsicht.

Abb. 17.2-3: Meta-Modell zur Block-basierten Modellierung.



17.2 Typische Sichten Eingebetteter Systeme I

Ein **Block** repräsentiert eine funktionale, kohäsive Einheit, die wiederum in weitere Blöcke unterteilt werden kann. Ein **Port** ist – wie bei der Kontextsicht – als Kommunikationsendpunkt zu interpretieren. Ein (Standard)Port wird zur Modellierung von Schnittstellen-Interaktionen verwendet. Ein **FlowPort** wird zur Modellierung von Informationsflüssen eingesetzt. Ein **Konnektor** verbindet entweder zwei oder mehr Blöcke, die über Informationsflüsse oder Schnittstellen kommunizieren. Hier werden Informationsfluss und Schnittstellen-Interaktion unterschieden. Informationsfluss bedeutet, dass Blöcke kontinuierlich Informationen austauschen. Im Falle der Schnittstellen-Interaktion kommunizieren Blöcke über den gegenseitigen Aufruf von Operationen, die der Block zur Verfügung stellt.

Um die entsprechenden Strukturen des Systems in einer solchen Form modellieren zu können, gibt es sogenannte SysML-Blockdiagramme. Es gibt zwei verschiedene Arten von Blockdiagrammen:

- Blockdefinitionsdiagramm und
- internes Blockdiagramm.

Das **Blockdefinitionsdiagramm** wird eingesetzt, um eine Hierarchie von funktionalen Blöcken zu definieren (Abb. 17.2-4).

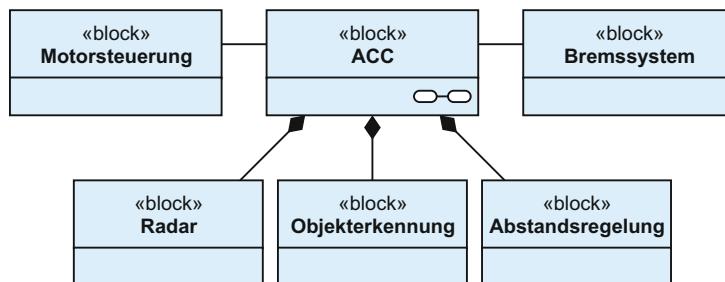


Abb. 17.2-4:
Blockdefinitions-
diagramm.

Der Vorteil einer solchen Darstellung ist eine übersichtliche Strukturierung aller relevanten Hierarchieebenen in Bezug auf eine bestimmte Funktionalität. Die Hierarchie wird durch Dekomposition der jeweiligen Blöcke definiert. Dazu werden die internen Blockdiagramme der SysML verwendet (Abb. 17.2-5).

Wenn Verhalten bezüglich der Kommunikation zwischen Blöcken oder auch Verhalten der Blöcke selbst modelliert werden soll, werden in der SysML **Zustandsdiagramme**, **Sequenzdiagramme** sowie **Aktivitätsdiagramme** eingesetzt. Welches der Diagramme zum Einsatz kommt, hängt letztendlich von der Zielstellung des Modells ab. Sollen sichtbare Zustände gezeigt werden, dann sind Zustandsdiagramme am sinnvollsten. Wenn algorithmische Eigenschaften modelliert werden sollen, dann werden Aktivitätsdiagramme eingesetzt. Sequenzdiagramme sind von Vorteil, wenn die Darstellung der konkreten Abfolge beim Informationsaustausch relevant ist.

I 17 Architekturen »Eingebetteter Systeme«

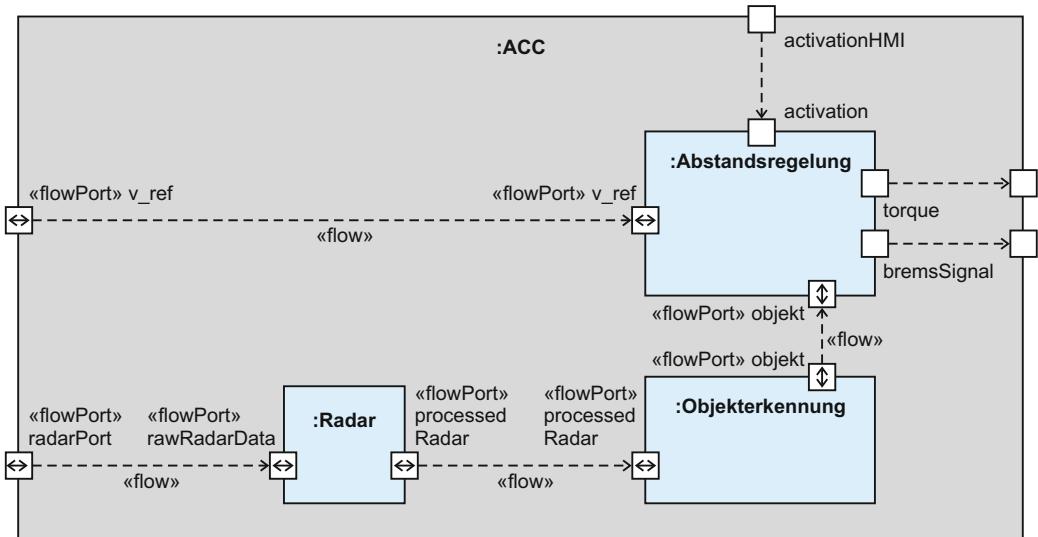
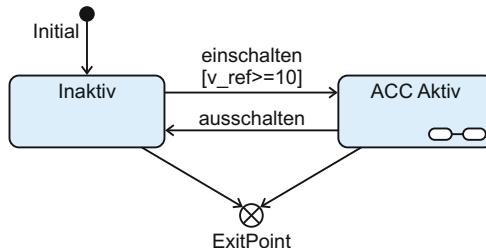


Abb. 17.2-5: In der Abb. 17.2-6 ist ein Zustandsdiagramm dargestellt, welches die extern sichtbaren Zustände des ACC-Systems beschreibt.

Internes Blockdiagramm.

Abb. 17.2-6: Zustandsdiagramm zur Modellierung von Verhalten.



Die Verteilungssicht eingebetteter Systeme

Eine Verteilungssicht zeigt die Relation der Softwareanteile bezüglich der Hardwareumgebung und deren Topologie. Im Kontext der Modellierung eingebetteter Softwarearchitekturen verfolgt die Verteilungssicht daher die folgenden Ziele:

- Ziele
- Übersicht über die Hardware-Topologie.
 - Zuweisung von Software zu Hardwareeinheiten.
 - Übersicht über die Hardwareeigenschaften, wie Prozessor oder Speicher.

Die Abb. 17.2-7 zeigt die generellen Modellelemente einer Verteilungssicht.

Ein **Knoten** repräsentiert ein Hardwaregerät, das Software zur Ausführung bringt. Ein **Artefakt** beschreibt ein Paket von Softwareeinheiten (Blöcken), die zusammen zur Ausführung gebracht werden können. Die Abbildung von funktionalen Blöcken auf Knoten wird

17.2 Typische Sichten Eingebetteter Systeme I

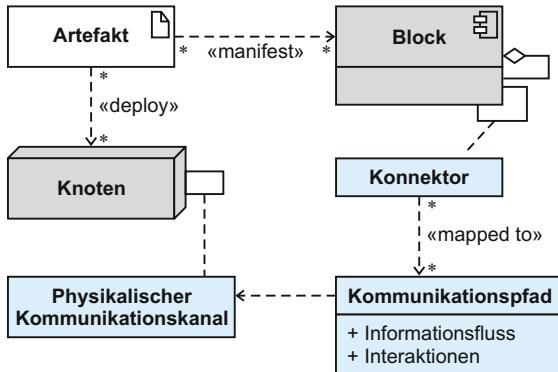


Abb. 17.2-7: Metamodell zur Verteilungssicht.

dabei als *Deployment* bezeichnet. Ein **Kommunikationspfad** beschreibt die physikalische Verbindung von zwei oder mehr Knoten. Ein Kommunikationspfad kann darüber hinaus bereits über technische Protokolle, wie zum Beispiel CAN [Enge02] oder FlexRay (siehe Website FlexRay (<http://www.flexray.com/>)), spezifiziert sein. Durch die Abbildung von Blöcken auf Knoten werden die **Konnektoren** (wie in der Bausteinsicht definiert) implizit auf physikalische Kommunikationskanäle abgebildet. Die sich daraus ergebende physikalische Systemverteilung ist zudem relevant bei der Abschätzung von Qualitätseigenschaften.

In der Abb. 17.2-8 ist eine Verteilungssicht dargestellt.

Beispiel

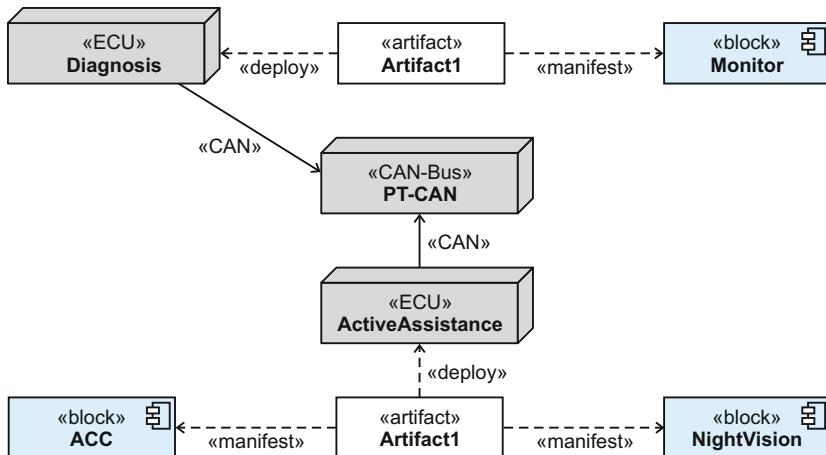


Abb. 17.2-8:
Beispielhafte
Verteilungssicht
eines
eingebetteten
Systems.

I 17 Architekturen »Eingebetteter Systeme«

17.3 Architekturmuster für Eingebettete Systeme

NFRs Für eingebettete Systeme stellt sich während des Entwurfs die Frage, in welcher Weise eine bestimmte Anforderung gelöst werden soll. Laufzeit-Anforderungen betreffen Qualitätseigenschaften der Systeme im Betrieb. Zu den wichtigsten Qualitätseigenschaften bei »Eingebetteten Systemen« zählen oft die Funktionssicherheit (siehe »Betriebssicherheit und Funktionssicherheit«, S. 121), die Verfügbarkeit (siehe »Zuverlässigkeit«, S. 124) und die Echtzeitfähigkeit. Im Grunde stellen alle nicht erfüllten Laufzeit-Anforderungen bei »Eingebetteten Systemen« eine beträchtliche Bedrohung für die Systemumgebung oder die Nutzerakzeptanz dar und sind damit im Architekturentwurf konstruktiv zu adressieren.

Ereignisse & Maßnahmen Dies geschieht dadurch, dass man Ereignisse klassifiziert, auf die das System in angebrachter Weise reagieren können soll. Die Ereignisklassen priorisieren dabei die Ereignisse durch Zuordnung von Auftrittswahrscheinlichkeiten und der dazugehörigen Schwere der Konsequenz. In Kombination kann dadurch eine Risikobewertung durchgeführt werden. Basierend auf der Risikobewertung werden systematisch Maßnahmen abgeleitet, die den Entwurf des eingebetteten Systems dahingehend beeinflussen, dass zum einen das Auftreten der Ereignisse im Betrieb möglich ist, zum anderen innerhalb kürzester Zeit geeignete Maßnahmen ergriffen werden können, um die etwaigen Konsequenzen zu minimieren.

Es liegt daher in der Verantwortung des Architekten, den »Bedrohungen« zur Laufzeit geeignete Gegenmaßnahmen entgegen zu stellen. Die Entscheidung, mit welcher Maßnahme ausgewählte Anforderungen adressiert werden, lässt sich nicht komplett formalisieren und damit auch nicht automatisieren. Vielmehr ist sie stark an die Kompetenz des Architekten – insbesondere an dessen Kreativität – gebunden. Nichtsdestotrotz, lassen sich auch hier typische und bewährte Lösungen in Form von Architekturmustern explizit und übertragbar machen. Die Architekturmuster zur Adressierung von Bedrohungen zur Laufzeit können in die folgenden Kategorien eingeteilt werden:

Architekturmuster **1 Erkennungsmuster:** Hier werden Lösungen beschrieben, die Fehlverhalten so schnell wie möglich identifizieren. Sie bilden die Grundlage für weitere Mechanismen zur Reduktion der Bedrohung. Als Beispiel für die Qualitätseigenschaft Verfügbarkeit kann hier das *Watchdog*-Muster angeführt werden, in dem über periodisch verschickte Signale die Funktionsweise einer Komponente kontinuierlich überprüft werden kann.

2 Wiederherstellungsmuster: Hier werden Lösungen beschrieben, welche die Auswirkungen von Fehlverhalten abmildern und das System nach einem Fehlverhalten schnellstmöglich wieder in einen normalen Zustand überführen. Eine gängige Praxis bei »Eingebetteten Systemen« ist hier, nach einem Fehlverhalten im System, den Fehler in den Fehlerspeicher einzutragen und das System durch Neustart wieder in den Normalzustand zu überführen.

3 Vermeidungsmuster: Hier werden Lösungen beschrieben, die die Wahrscheinlichkeit für den Eintritt des Fehlverhaltens senken. Dies kann zum Beispiel durch die redundante Auslegung von Funktionen erfolgen, die redundante Komponenten zur Kompensation von Ausfällen verwenden.

Offensichtlich stehen das Erkennen und das Wiederherstellen in enger Wechselbeziehung zueinander, da das reine Erkennen ohne das Wiederherstellen in der Regel nicht ausreicht und im Gegenzug die Wiederherstellung nur nach erkanntem Fehlverhalten funktionieren kann.

Aufgrund ihrer zentralen Bedeutung für »Eingebettete Systeme« werden für die Laufzeit-Anforderung **Funktionssicherheit** zwei Muster näher vorgestellt:

- »Das PSC-Muster (*protected single channel pattern*)«, S. 407
- »Das *Homogeneous Redundancy*-Muster«, S. 409

17.3.1 Das PSC-Muster (*protected single channel pattern*)

Name(n)

Das PSC-Muster (*protected single channel pattern*) ist eines der einfachsten Muster zur Erhöhung der Sicherheit einer Steuerung oder Regelung [Powe99].

Grundidee

Der Grundgedanke des PSCP besteht darin, die Funktionssicherheit eines Einkanal-Systems durch die Integration zusätzlicher Prüfungen und Aktionen an geeigneten Stellen innerhalb dieses Kanals zu erhöhen. Dabei werden zusätzliche Maßnahmen getroffen, um die Korrektheit der Daten und des Programms zu überprüfen und zu gewährleisten.

Anwendungsbereich(e)

- Das Muster dient primär zur Fehlererkennung und *nicht* zur Fehlerbehandlung. Verglichen mit anderen Mustern sind die erzielbaren Verbesserungen der Funktionssicherheit gering.
- Das PSCP ist sehr gut geeignet zur Erkennung von transienten Hardwarefehlern. Zur Erkennung von permanenten Hardwarefehlern ist das Muster dagegen nur bedingt einsetzbar. Ebenso für die

I 17 Architekturen »Eingebetteter Systeme«

Erkennung von Software- bzw. Entwurfsfehlern ist die Anwendung anderer Muster, wie zum Beispiel des *Heterogeneous Redundancy*-Musters, sinnvoller.

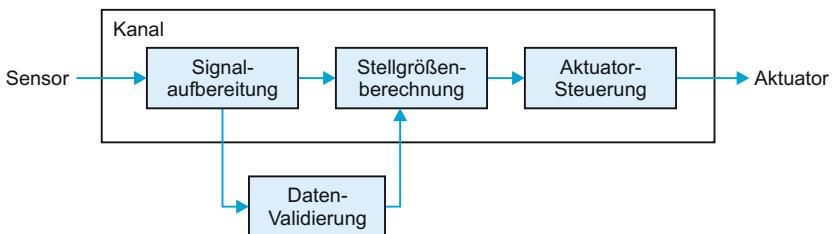
- Dadurch, dass die Umsetzung des PSCP allerdings mit relativ geringem Aufwand möglich ist und in den meisten Fällen keine zusätzliche Hardware erfordert, eignet es sich insbesondere für sehr kostensensitive Systeme.
- Außerdem ist es sehr gut mit anderen Mustern kombinierbar und kann somit idealerweise als Ausgangsbasis für die Anwendung weiterer Muster dienen.

Varianten

Im Wesentlichen existieren zwei Varianten des Musters: Bei der *Open-Loop*-Variante werden lediglich die Eingabewerte des Kanals überwacht, bei der *Closed-Loop*-Variante zusätzlich auch der Aktuator oder die Strecke selbst.

Open-Loop Der prinzipielle Aufbau des Musters in der *Open-Loop*-Variante ist in Abb. 17.3-1 dargestellt.

Abb. 17.3-1:
Protected Single
Channel Muster –
Open Loop.



Im oberen Teil der Abbildung ist der einzige Kanal des Systems dargestellt. Innerhalb dieses Kanals werden zunächst in der Komponente »Signalaufbereitung« die Eingangswerte aus der Sensorik aufbereitet. Anschließend werden die aufbereiteten Werte in der Komponente »Stellgrößenberechnung« zur Berechnung der Stellgrößen für die Aktuatorik verwendet. Bevor die Stellgrößen an die Aktuatorik ausgegeben werden können, müssen sie in der Komponente »Aktuator-Steuerung« in für den Aktuator geeignete Signale transformiert werden.

Bei der Anwendung des PSCP wird das Verhalten dieses Kanals durch die Komponente »Daten-Validierung« überwacht, um eventuelle Abweichungen vom gewünschten Systemverhalten zu erkennen. Dazu werden zusätzliche Prüfungen integriert, die entweder direkt die Hardware des Systems, die vom System zu verarbeitenden Daten oder das auszuführende Programm selbst überwachen. Genaue Vorgaben, wie diese Prüfungen zu realisieren sind, werden vom Muster selbst allerdings nicht vorgegeben. Die konkrete Realisierung muss problemspezifisch durch den Entwickler umgesetzt werden.

17.3 Architekturmuster für Eingebettete Systeme I

In der Abb. 17.3-2 ist die *Closed-Loop*-Variante des Musters dargestellt.

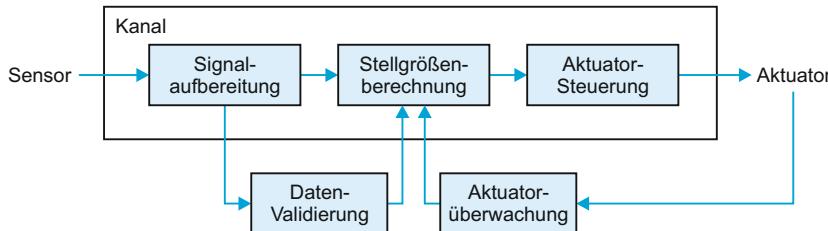


Abb. 17.3-2:
Protected Single
Channel Muster –
Closed Loop.

Darin wird zusätzlich die Komponente »Aktuator-Überwachung« ergänzt, um auch die korrekte Umsetzung des Steuersignals durch den Aktuator überprüfen zu können.

Für die Implementierung des PSCP sind verschiedene Strategien anwendbar, die auch untereinander kombiniert werden können. Die Überprüfung der Hardware kann beispielsweise den Speicher (ROM, RAM, EEPROM), die Berechnungseinheiten (ALU) oder die Peripherie umfassen. Zusätzlich können Überwachungen der Daten und des Kontrollflusses im Programm durchgeführt werden. Beispiele für die Überwachung auf Softwarearchitekturebene sind *Watchdog-Timer* und Plausibilisierungs-Tests.

Hinweise zur
Implemen-
tiering

17.3.2 Das *Homogeneous Redundancy*-Muster

Name(n)

Das *Homogeneous Redundancy*-Muster (auch *switch-to-back-up*-Muster) erhöht die Verfügbarkeit des Systems, indem es einen alternativen Kanal anbietet, auf den im Fehlerfall ausgewichen wird. Im Gegensatz zum *Heterogeneous Redundancy*-Muster ist sowohl die Hardware als auch die Software des alternativen Kanals eine exakte Kopie des primären Kanals.

Anwendungsbereich(e)

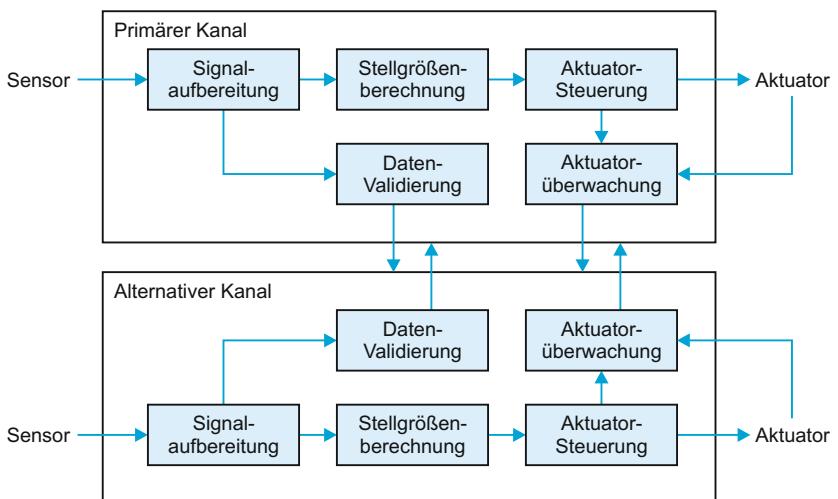
Das Muster schützt vor Hardware-Fehlern und Ausfällen. Software-Fehler können *nicht* erkannt werden, da sich jeder Software-Fehler in einem Kanal auch in den duplizierten Kanälen befindet. Das Muster ist für kostensensitive Systeme ungeeignet, weil sich die Hardware-Kosten verdoppeln. Aufgrund der Verwendung der identischen Implementierung für beide Kanäle sind die zusätzlichen Software-Entwicklungskosten sehr gering. Der zusätzliche Implementierungsaufwand hängt von der Fehlererkennung und von der Realisierung des Wechsels zum aktiven Kanal ab.

I 17 Architekturen »Eingebetteter Systeme«

Struktur des Musters

Das Muster erweitert den ursprünglichen Kanal um eine Fehlererkennung und dupliziert ihn. Die Fehlererkennung erfolgt analog zur *Closed-Loop*-Variante des PSC-Musters (siehe »Das PSC-Muster (*protected single channel pattern*)«, S. 407) in der Daten-Validierung und der Aktuatorüberwachung. Der einzige Unterschied besteht darin, dass die Fehlererkennung bei der Detektion eines Fehlers einen Kontrollwechsel zu dem anderen Kanal veranlasst. Die resultierende Musterstruktur ist in der Abb. 17.3-3 dargestellt.

Abb. 17.3-3:
Homogeneous
Redundancy
Muster.



Es wird immer nur einer der beiden Kanäle ausgeführt. Erkennt der ausführende Kanal einen Fehler, dann übergibt er die Kontrolle an den anderen Kanal. Welcher der beiden Kanäle mit der Ausführung beginnt (primärer Kanal) ist irrelevant, da beide Kanäle identisch sind. Neben der sequenziellen Ausführung der Kanäle ist auch eine parallele Ausführung denkbar. Diese parallele Ausführung hat den Vorteil, dass bei einem Kanalwechsel kein Berechnungsschritt verloren geht.

Das Muster fordert für jeden Kanal eigene Sensoren und Aktuatoren, damit bei einem Sensor- oder Aktuatorausfall nicht beide Kanäle unbrauchbar werden. Es ist aber auch denkbar, zuverlässige kostspielige Sensoren und Aktuatoren für beide Kanäle zu verwenden.

Hinweise zur Implementierung

Die zusätzliche Implementierungsarbeit resultiert aus der Fehlererkennung und der Realisierung des Kontrollwechsels zwischen den Kanälen. Die Fehlererkennung erfolgt in der Daten-Validierung und der Aktuatorüberwachung und kann in gleicher Weise wie in der *Closed-Loop*-Variante des PSC-Musters umgesetzt werden. Die Implementierung des Kontrollwechsel hängt davon ab, ob die Kanäle parallel ausgeführt werden oder nicht. Bei einer parallelen Ausführung

17.3 Architekturmuster für Eingebettete Systeme I

geht beim Kontrollwechsel kein Berechnungsschritt verloren, da im Fehlerfall sofort zum alternativen Kanal gewechselt werden kann und dort das aktuelle Ergebnis schon vorliegt. Im Fall einer seriellen Ausführung der Kanäle gibt es zwei Strategien mit dem verlorenen Berechnungsschritt umzugehen. Entweder der aktiv gewordene Kanal verwirft die fehlerhafte Berechnung und rechnet mit den neuen Sensordatenwerten weiter, oder er stellt die alten Sensordaten wieder her und wiederholt die Berechnung. Bei der letzteren Variante muss besonders in zeitkritischen Systemen die Zeit für die Neuberechnung berücksichtigt werden.

18 Das Subsystem Applikation

Bei einer systematischen objektorientierten Softwareentwicklung ist der Ausgangspunkt für die Architektur des Subsystems »Applikation« immer das objektorientierte Analysemodell, das die fachliche Lösung der Fachdomäne wiedergibt.

Zunächst ist zu überlegen, inwieweit das OOA-Modell unter Berücksichtigung der zu verwendeten Programmiersprache verfeinert und präzisiert werden kann. Ist die Entscheidung für eine Programmiersprache noch nicht gefallen, dann muss sie jetzt getroffen werden.

Wenn im OOA-Modell die Bezeichner und die Datentypen der verwendeten Programmiersprache noch nicht berücksichtigt wurden, dann muss dies im OOD-Modell erfolgen. Die Attribute müssen daraufhin überprüft werden, ob Datentypen und Aufzählungen ausgelagert werden können. Konstruktoren und Methoden müssen ergänzt werden.

In einem OOA-Modell werden Hilfsklassen für die Verwaltung von erzeugten Objekten in der Regel *nicht* aufgeführt. Da jedoch in den meisten Anwendungen Listen von erzeugten Objekten benötigt werden, ist ein Objektverwaltung nötig. Dies geschieht in der Regel durch die Ergänzung des OOA-Modells um Containerklassen.

Im OOA-Modell werden Geschäftsprozesse zum Beispiel durch eEPKs (erweiterte ereignisgesteuerte Prozessketten) oder UML-Aktivitätsdiagramme beschrieben. *Use Cases* werden durch UML-Use-Case-Diagramme oder Use-Case-Schablonen dargestellt. Im OOD-Modell muss überlegt werden, wie diese Beschreibungen zum Beispiel in Klassen mit zugeordneten Zustandsautomaten umgesetzt werden.

Eines der zentralen Prinzipien der Softwaretechnik ist die lose Kopplung von Subsystemen (siehe »Architekturenprinzipien«, S. 29). Es ist daher zu überlegen, durch welche Maßnahmen die Kopplung zum Subsystem »Benutzeroberfläche« und zum Subsystem »Persistenz« möglichst lose gestaltet werden kann.

Besitzen die nichtfunktionalen Anforderungen Wartbarkeit (siehe »Wartbarkeit«, S. 116) und Weiterentwickelbarkeit (siehe »Weiterentwickelbarkeit«, S. 119) eine hohe Priorität, dann muss überlegt werden, ob zu den angrenzenden Subsystemen zusätzliche Subsysteme zwischengeschaltet werden, um die Abhängigkeiten zu reduzieren.

Von OOA zu OOD

Beispiel

Container-
klassenGeschäfts-
prozesse & Use
Cases

Lose Kopplung

NFAs

I 18 Das Subsystem Applikation

- Verteilt Soll es sich um eine verteilte Architektur handeln (siehe »Verteilte Architekturen«, S. 191), dann müssen Maßnahmen getroffen werden, um die Netzkommunikation, d. h. den Rundreise-Aufwand (*round trip cycle*), zu optimieren (siehe »Entwurfskonzepte für verteilte Anwendungen«, S. 306).
- | Soll es sich bei der Anwendung um eine Web-Architektur handeln, dann gibt es Vorschläge für eine geeignete »Binnen«-Architektur:
- »Web-Architektur für das Subsystem Applikation«, S. 414

18.1 Web-Architektur für das Subsystem Applikation

Eine bewährte Web-Architektur für das Subsystem Applikation, das sich auf dem Web-Server befindet, ist die Anwendung des MVC-Musters (Abb. 18.1-1). Damit ist die »Binnen«-Architektur festgelegt.

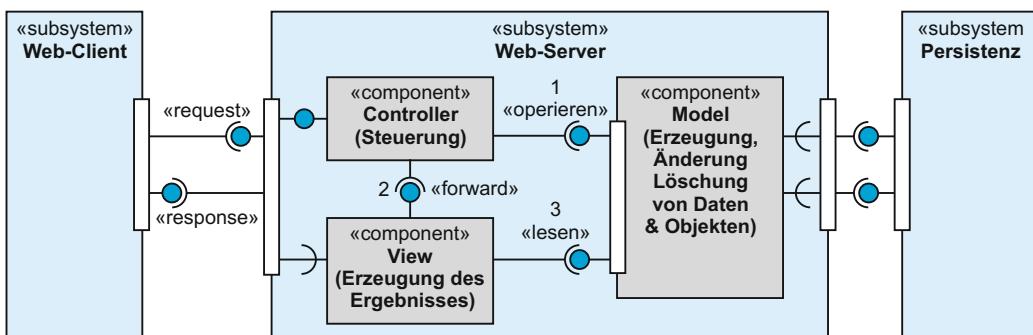


Abb. 18.1-1: Web-Architektur nach dem MVC-Muster.

Eine weitere Verbesserung der Architektur bieten Entwurfsmuster für die einzelnen Komponenten, die sich aus dem MVC-Muster ergeben. Die wichtigsten Entwurfsmuster für Web-Anwendungen heißen:

- Front Controller-Muster
- Command and Controller Strategy-Muster
- View Helper-Muster
- Transfer Object-Muster
- Service to Worker-Muster

Diese Entwurfsmuster wurden im Rahmen der Java EE-Plattform entwickelt und heißen nach dem früheren Namen der Java EE-Plattform J2EE-Muster, siehe Core J2EE Patterns (<http://java.sun.com/blueprints/corej2eepatterns/index.html>).

Die oben aufgeführten Entwurfsmuster werden nachfolgend kurz beschrieben.¹

¹Teile der folgenden Texte wurden mit freundlicher Genehmigung des W3L-Verlags aus [Wißm09, S. 270 ff.] entnommen.

18.1 Web-Architektur für das Subsystem Applikation I

Das Front Controller-Muster

Das Front Controller-Muster bezieht sich auf den Controller-Teil im MVC-Muster. Alle Anfragen an die Web-Anwendung werden an eine einzige, zentrale Komponente geleitet.

Der Controller-Teil der Web-Anwendung, der alle Anfragen entgegennimmt, besteht aus einem einzigen Servlet oder einer einzigen JSP-Seite (Abb. 18.1-2).

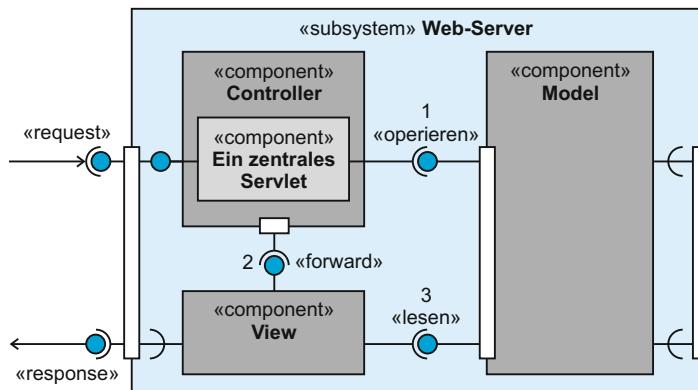


Abb. 18.1-2:
Entwurfsmuster
Front Controller-
Muster.

Im zentralen Servlet wird das empfangene Ereignis, dargestellt durch die aufgerufene URL, auf eine Funktion der Web-Anwendung abgebildet. Die Durchführung der Methoden zur Funktion wird im zentralen Servlet vorgenommen.

- + Neue Ereignis-Funktions-Paare oder neue oder geänderte Abbildungen von Ereignissen auf Funktionen werden zentral an einer Stelle durchgeführt. Dadurch wird die Übersichtlichkeit und Wartbarkeit unterstützt. Vorteil
- Bei vielen Funktionen, die in einem zentralen Servlet implementiert werden, wird diese Komponente sehr groß und damit auch unübersichtlich. Nachteil

Command and Controller Strategy-Muster

Dieses Entwurfsmuster ergänzt das Front Controller-Muster. Es wird das Command Design-Muster mit dem Front Controller-Muster kombiniert. Das Command Design-Muster besagt, dass die Methoden einer Funktion in einer abgeschlossenen Einheit gekapselt werden. Alle Komponenten zu Funktionen besitzen eine **einheitliche Schnittstelle** (Abb. 18.1-3).

Das zentrale Servlet nimmt die Anfragen und die damit verbundenen Ereignisse entgegen. Das Ereignis wird analysiert und auf eine Funktion abgebildet. Danach wird eine funktionsspezifische Komponente aufgerufen. Da alle Funktionskomponenten eine einheitli-

I 18 Das Subsystem Applikation

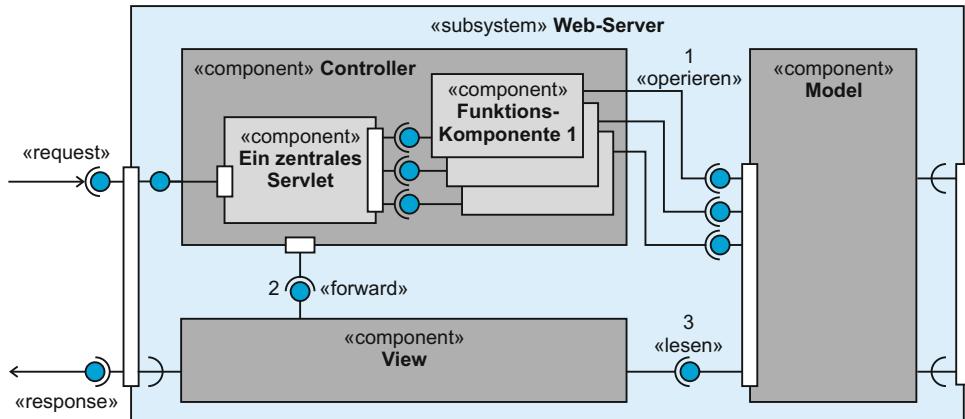


Abb. 18.1-3:
Entwurfsmuster
Command and
Controller
Strategy-Muster.

Vorteil

che Schnittstelle bereitstellen, kann der Aufruf im Servlet immer auf die gleiche Art und Weise erfolgen. Die gerufene Funktionskomponente arbeitet jetzt die Methoden zur Funktion unter Benutzung der Model-Komponente ab und delegiert anschließend die weitere Bearbeitung der Anfrage an die View-Komponente der Web-Anwendung.

+ Durch die Auslagerung der Implementierung der Funktionen in je eine gekapselte Einheit wird der Nachteil des Front-Controller-Musters behoben und die Wartbarkeit und Erweiterbarkeit des zentralen Servlets sowie die Wiederverwendbarkeit von Funktionen unterstützt.

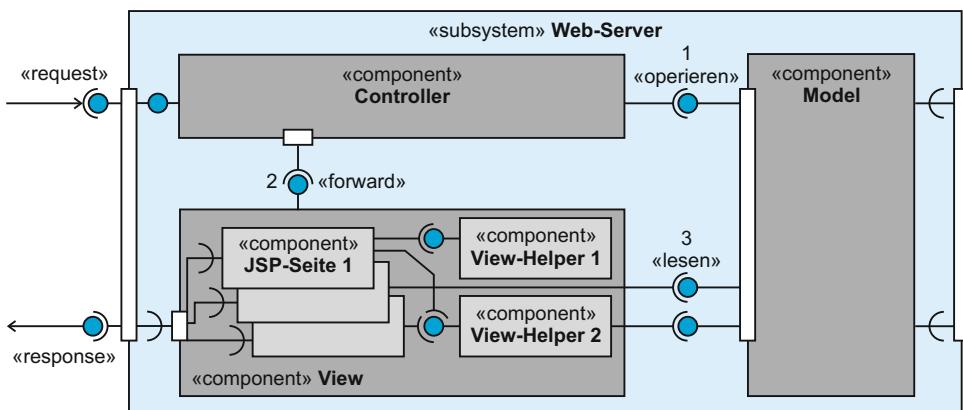
View Helper-Muster

Die View-Komponente hat die Aufgabe, ein Ergebnisdokument für den Client zu erzeugen. Da in der Regel, abhängig von der Benutzer-Anfrage, als Ergebnisdokument verschiedene Webseiten generiert werden, ist es zweckmäßig, für jede Webseite eine eigene View-Teilkomponente zu programmieren. Ist die zu erzeugende Website bezüglich des Layouts in Bereiche – wie Kopfbereich, Fußbereich, Navigationsbereich, Hauptinformationsbereiche etc. – aufgeteilt, so treten diese Bereiche auf allen Webseiten der Website auf. Es ist dann vorteilhaft für die View-Komponente Hilfskomponenten zu definieren, die jeweils die Inhalte der Bereiche erzeugen und in verschiedenen Teilkomponenten verwendet werden können. Ein weiterer Grund, Hilfskomponenten zu definieren, ist die Notwendigkeit, Hilfslogik in der View-Komponente zu programmieren. Die View-Komponente nutzt die Daten der Model-Komponente, realisiert aber *keinerlei* anwendungsspezifische Logik im Sinne von Änderungen der Daten in den Model-Komponenten. Trotzdem ist es oft notwendig, in den View-Komponenten Logik zu implementieren.

18.1 Web-Architektur für das Subsystem Applikation I

- Wenn eine Liste von Daten aus der Model-Komponente sortiert angezeigt werden soll und die Daten in der Model-Komponente nur unsortiert vorliegen, muss das Sortieren der Daten nach einem evtl. vom Benutzer gewählten Sortierkriterium implementiert werden. Beispiele
- Nach vom Benutzer definierten Vorgaben sind Daten zu formatieren oder zu selektieren.

Diese Logikanteile besitzen auch hohes Potenzial für die Wiederverwendung in verschiedenen View-Teilkomponenten. Das View-Helper-Muster besagt, dass solche Aufteilungen in Hilfsfunktionalitäten identifiziert werden und der zugehörige Programmcode in eigenen Einheiten als Hilfskomponenten gekapselt wird (Abb. 18.1-4).



Die Controller-Komponente delegiert den Kontrollfluss an die View-Komponente bzw. an eine View-Teilkomponente, bei JSP-Anwendungen an eine JSP-Seite. Im Code der JSP-Seite wird das übergeordnete Layout des Ergebnisdokuments definiert. Teile, die für die Anzeige bestimmter Bereiche zuständig sind oder komplexe logische Operationen benötigen, werden gekapselt und in View-Helper-Komponenten ausgelagert. Die View-Helper-Komponenten können dann in mehreren JSP-Seiten benutzt werden. Sie greifen ggf. auch auf die Daten der Model-Komponente zu.

Eine View-Helper-Komponente kann verschieden implementiert sein. Eine Möglichkeit der Implementierung ist einfach eine JSP-Seite, die per `<jsp:include>` in die von der Controller-Komponente beauftragte JSP-Hauptseite der View-Komponente eingebettet wird. Eine alternative Implementierung ist, Funktionalitäten in *Custom Tags* zu realisieren und dadurch als JSP-Anweisung wiederverwendbar zu machen. Der Kontrollfluss kehrt schließlich immer zur JSP-Hauptseite der View-Komponente zurück, die die Erstellung des Ergebnisdokuments abschließt und die Request-Bearbeitung beendet.

Abb. 18.1-4:
Entwurfsmuster
View
Helper-Muster.

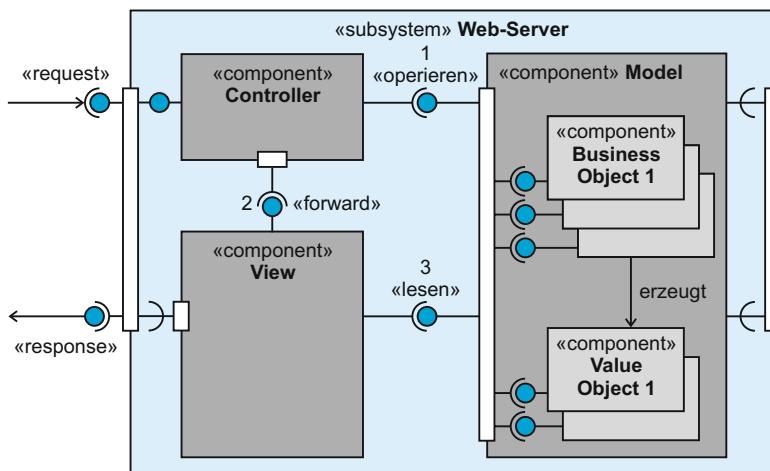
I 18 Das Subsystem Applikation

- Vorteil + Die Code-Struktur der View-Komponente wird übersichtlicher und die Wiederverwendung von Code wird verbessert.

Transfer Object-Muster

Die Funktionen, die von der Controller-Komponente ausgeführt werden, bewirken Änderungen in den Daten der Model-Komponente. Außerdem besitzt eine Funktionsausführung in der Regel ein Ergebnis, das von der Controller-Komponente an die View-Komponente weitergegeben wird und die Erzeugung des Ergebnisdokuments in der View-Komponente maßgeblich beeinflusst. Wie wird nun das Ergebnis dargestellt und an die View-Komponente weitergeleitet? Hier setzt das Transfer-Object-Muster an. Die Idee zu diesem Entwurfsmuster zeigt die Abb. 18.1-5.

Abb. 18.1-5:
Entwurfsmuster
Transfer Objekt-
Muster.



Temporäre Objekte werden in der Model-Komponente erzeugt, die ausschließlich für den Datentransfer von Ergebnissen innerhalb einer Request-Bearbeitung benutzt werden. Diese temporären Objekte werden *Value Objects* oder manchmal auch *Data Transfer Objects* genannt (siehe »Wert-Objekte«, S. 309). Die Controller-Komponente übergibt bei der Weitergabe des Kontrollflusses an eine View-Komponente in der Regel eine Referenz auf ein *Value Object*. Das *Value Object* kapselt das ggf. komplexe Ergebnis der Funktionsausführung. Die View-Komponente wertet das *Value Object* für die Erzeugung des Ergebnisdokuments aus. Mit Abschluss der Request-Bearbeitung in der View-Komponente wird dann das *Value Object* gelöscht.

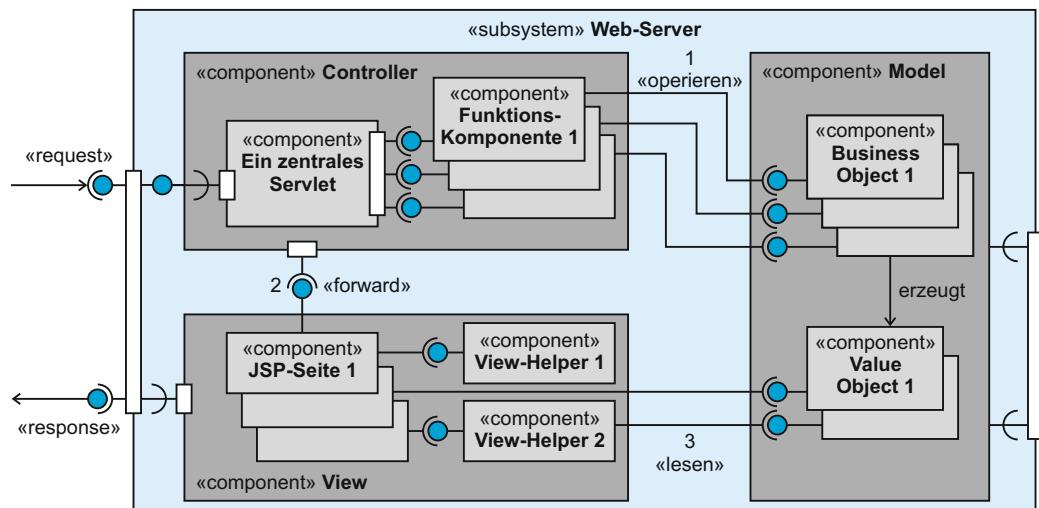
- Vorteil + Durch die Darstellung der Ergebnisse einer Funktionsausführung in einem *Value Object*, von welchem die Controller-Komponente einfach eine Referenz durchreicht, wird die Controller-Komponente vom Inhalt der Model-Komponente entkoppelt. Änderungen

18.1 Web-Architektur für das Subsystem Applikation I

im Informationsgehalt der Model-Teilkomponenten können daher ohne Änderungen in der Controller-Komponente vorgenommen werden, was die Erweiterbarkeit und Änderungsfreundlichkeit der Web-Anwendung erhöht.

Service to Worker-Muster

Das Service to Worker-Muster ist kein Entwurfsmuster, das die Architektur einer einzelnen Komponente betrifft, sondern die Kombination der gerade behandelten anderen Entwurfsmuster darstellt. Es bietet also eine Zusammenfassung der einzelnen Entwurfsmuster, die das MVC-Muster verfeinern. Das Service to Worker-Muster ist in der Abb. 18.1-6 dargestellt.



- + Der Vorteil dieses Entwurfsmusters ergibt sich aus der Summe der Vorteile der einzelnen Entwurfsmuster.

Abb. 18.1-6:
Entwurfsmuster
Service to Worker-
Muster.

19 Das Subsystem Persistenz

In fast allen Anwendungen müssen Daten langfristig gespeichert werden. Statt von langfristiger Speicherung spricht man häufig auch von (Daten-) **Persistenz**. Gemeint ist, dass die Daten das Beenden des dazu gehörenden Anwendungsprogramms »überleben« sollen, also unabhängig vom laufenden Programm weiter zur Verfügung stehen sollen. Zur Realisierung der Persistenz bestehen heute folgende Möglichkeiten:

- Speicherung in »flachen« Dateien (*flat files*) unter Nutzung des jeweiligen Betriebssystems.
- Speicherung in **relationalen Datenbanken** (RDB), d. h. Speicherung in Tabellenform.
- Speicherung in **XML-Datenbanken**, d. h. Speicherung in textueller Form mit semantischen Markierungen, ähnlich wie HTML- bzw. XHTML-Dokumente.
- Speicherung in **objektorientierten Datenbanken**, d. h. Speicherung in Form von Objektnetzen.
- Speicherung in NoSQL-Datenbanken [Ling11], [Müll10].

Mit über 90 Prozent Marktanteil dominieren heute relationale Datenbanken.

Langfristige Speicherung

Im Laufe der Zeit hat man vom direkten Zugriff auf eine herstellerspezifische relationale Datenbank immer mehr abstrahiert:

- »Vom Direktzugriff bis zum JPA«, S. 422

Bei der Verwendung einer objektorientierten Softwareentwicklung und dem Einsatz einer relationalen Datenbank müssen Klassen auf Tabellen abgebildet werden. Bei der Abbildung müssen eine Reihe von Regeln beachtet werden:

- »Exkurs: ORM – Objektrelationale Abbildung«, S. 426

Java stellt mit dem *Java Persistence API* eine komfortable Möglichkeit zur Verfügung, auf relationale Datenbanken zuzugreifen:

- »JPA – Java Persistence API«, S. 431

Die Fallstudie wird mit JPA realisiert:

- »Fallstudie: KV – JPA«, S. 436

.NET stellt mehrere Möglichkeiten für die Persistenz zur Verfügung:

- »Persistenz in .NET«, S. 442

Markt: RDB

I 19 Das Subsystem Persistenz

19.1 Vom Direktzugriff bis zum JPA

Direktzugriff

Auf eine Datenbank kann immer über die vom Hersteller bereitgestellte Schnittstelle zugegriffen werden (Abb. 19.1-1). Um die Abhängigkeit von einer speziellen Datenbank eines Herstellers zu reduzieren, hat man schon frühzeitig sogenannte Treiber-APIs erfunden, um Datenbanken austauschen zu können, ohne die Zugriffe neu zu programmieren.

ODBC

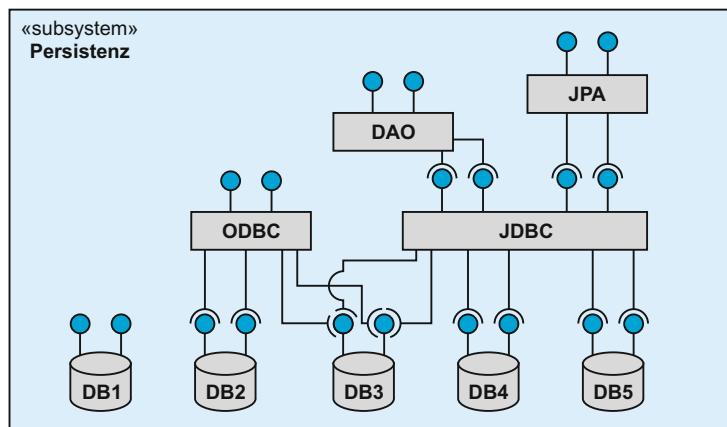
Will man von einer klassischen Programmiersprache auf eine relationale Datenbank zugreifen, dann verwendet man in der Regel einen sogenannten **ODBC**-Treiber.

JDBC

Java Wird die Programmiersprache Java verwendet, dann nutzt man in der Regel einen **JDBC**-Treiber (*Java Database Connectivity*), um auf relationale Datenbanken lesend und schreibend zuzugreifen. Nach Einbindung des Treibers (*Driver*) kann eine Verbindung (*Connection*) zur Datenbank aufgebaut werden. Über diese können Anweisungen in Form von SQL-Befehlen geschickt werden. Die Ausführung einer lesenden Anweisung liefert in der Regel eine Ergebnismenge (*Result Set*), welche die gewünschten Datensätze enthält.

Nachteilig beim Einsatz von JDBC ist, dass der Entwickler die normalen SQL-Befehle benutzen muss, um Tabellen anzulegen, Einträge vorzunehmen, Abfragen zu erstellen oder Datensätze zu löschen.

Abb. 19.1-1: Die verschiedenen Abstraktionsebenen im Subsystem »Persistenz«.



Das einfache DAO-Muster

Das **DAO-Muster** ermöglicht es, auf einem höheren Abstraktionsniveau mit einer relationalen Datenbank zu kommunizieren. Das DAO-Muster gehört zu den *Core J2EE Patterns*, siehe Website Sun DAO (<http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>).

Zwischen den JDBC-Treiber und einer Fachkonzeptklasse wird jeweils eine DAO-Fachkonzeptklasse »zwischengeschaltet«. DAO steht für *Data Access Object*: DAO-Klasse

»The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject« [Sun02].

Die jeweilige DAO-Klasse stellt für die jeweilige Fachkonzeptklasse die sogenannten CRUD-Methoden zur Verfügung: `create()`, `read()`, `update()` und `delete()`. Damit kann von der Fachkonzeptklasse mit normalen Methoden (ohne SQL) auf die Datenbank zugegriffen werden (Abb. 19.1-2). CRUD



Abb. 19.1-2:
Klassendiagramm
zum einfachen
DAO-Muster.

Das komplexe DAO-Muster

Das komplexe DAO-Muster erweitert das einfache DAO-Muster um zwei Aspekte:

- 1 Die Fachkonzeptklasse greift auf eine DAO-Schnittstelle und nicht auf eine DAO-Klasse zu. Dadurch können Implementierungen der Schnittstelle leicht ausgetauscht werden, ohne dass die Fachkonzeptklasse davon betroffen ist. Soll beispielsweise statt einer relationalen Datenbank eine XML-Datenbank verwendet werden, dann muss nur die Implementierung der Schnittstelle ausgetauscht werden. Die Fachkonzeptklasse bleibt unverändert.
- 2 Bei einer verteilten Architektur müssen Informationen über das Netz übertragen werden. Daher ist es sinnvoll, nur die Informationen zu übertragen, die auch gespeichert werden sollen. Bei einer Fachkonzeptklasse sollen u. U. nicht alle Attribute persistent gespeichert werden. Außerdem sind die Methoden der Klasse für die Speicherung nicht relevant. Daher ist kein Transport von Objek-

I 19 Das Subsystem Persistenz

ten einer komplexen Fachkonzeptklasse erforderlich, sondern es könnte stattdessen auf »schlankere Stellvertreter« ausgewichen werden.

Technisch werden diese beiden Aspekte wie folgt realisiert:

- 1 Wenn eine Fachkonzeptklasse gegen die zugehörige DAO-Schnittstelle programmiert werden soll, dann stellt sich das Problem, dass aus einer Schnittstelle keine Objekte erzeugt werden können. Die nutzende Fachkonzeptklasse benötigt aber Zugriff auf Objekte, mit denen die Methoden der Schnittstelle ausgeführt werden sollen. Das Problem wird durch die Verwendung einer sogenannten **Fabrikmethode** gelöst (siehe auch »Schnittstellen, Fabriken und Komposition«, S. 503). Der Aufrufer glaubt ein Objekt aus der Schnittstelle zu erzeugen, in Wirklichkeit erhält er von der Fabrikmethode aber ein Objekt einer Klasse, die die Schnittstelle passend implementiert.
- 2 Der zweite Aspekt wird durch die Einführung einer weiteren Klasse gelöst, deren Objekte – sogenannte **Transfer-Objekte** – ausschließlich für den Transfer der relevanten Attribute aus Objekten der Fachkonzeptklassen zwischen dem Subsystem »Applikation« und dem Subsystem »Persistenz« genutzt werden (siehe auch »Wert-Objekte«, S. 309).

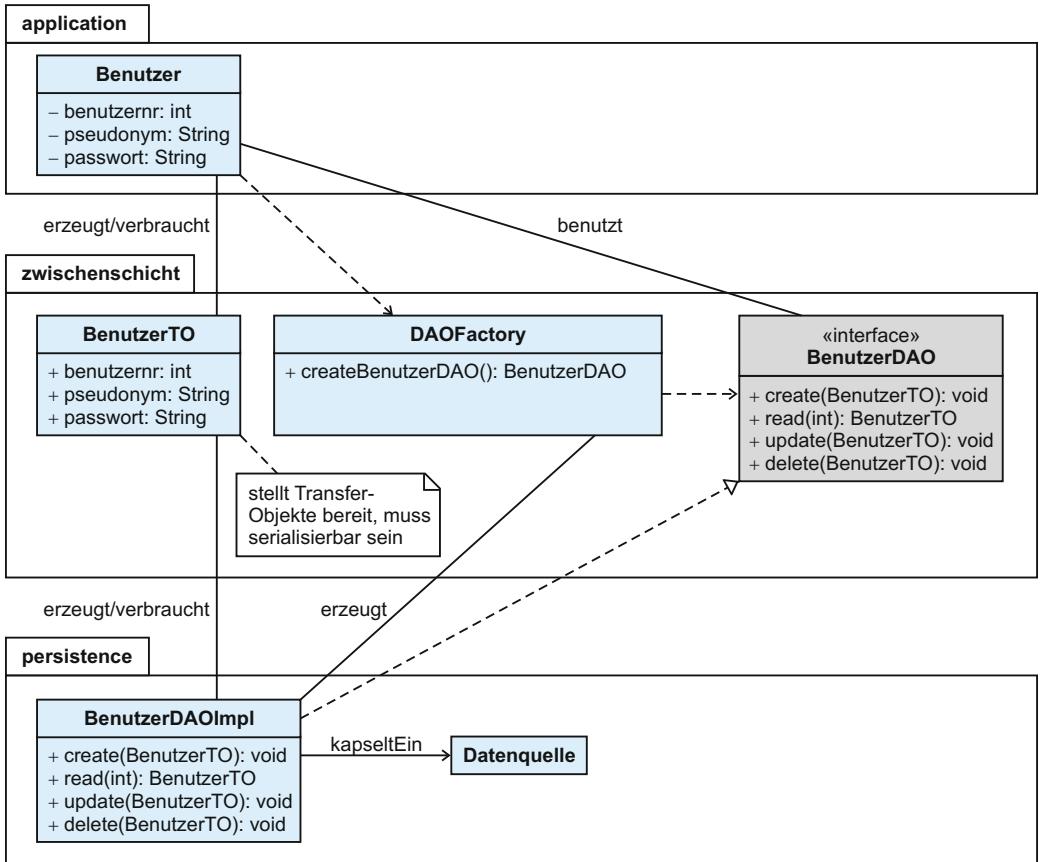
Beispiel 1a Die Objekte einer Fachkonzeptklasse Benutzer sollen in einer relationalen Datenbank gespeichert werden. Zur Fachkonzeptklasse Benutzer gibt es eine DAO-Implementierung BenutzerDAOImpl. Allerdings kennt Benutzer diese Implementierung *nicht* direkt, sondern nur die Schnittstelle BenutzerDAO. Über die Fabrik DAOFactory wird dafür gesorgt, dass die Fachkonzeptklasse und die DAO-Implementierung in Kontakt kommen (Abb. 19.1-3). Über Transferobjekte werden Daten zwischen der Fachkonzeptklasse und dem Subsystem Persistenz hin und her geschoben.

JPA (*Java Persistence API*)

Java Um dem Entwickler die Arbeit zu ersparen, das DAO-Muster jedes Mal selbst zu implementieren, stellt Java **JPA** zur Verfügung. Zusätzlich übernimmt das JPA die Aufgabe, eine objekt-relationale Abbildung vorzunehmen (siehe »Exkurs: ORM – Objektrelationale Abbildung«, S. 426). Außerdem wird mit **JPQL** eine eigene Abfragesprache – analog zur SQL – zur Verfügung gestellt.

@Entity In JPA wird eine Fachkonzeptklasse durch die Annotation **@Entity** als eine Klasse gekennzeichnet, die persistent gehalten werden soll.
entity manager In JPA verwaltet ein sogenannter *entity manager* den Zustand und den Lebenszyklus von Klassen, die als **@Entity**-Klassen annotiert sind. Der Vorteil von JPA liegt darin, dass Klassen als *normale*

19.1 Vom Direktzugriff bis zum JPA I



Klassen – oft als POJO-Klassen bezeichnet (*Plain Old Java Objects*) – verwendet werden können. Wenn sie vom *entity manager* verwaltet werden, dann können Daten in der Datenbank gespeichert oder aus ihr gelesen werden. Der *entity manager* stellt eine Schnittstelle *EntityManager* zur Verfügung, die von einem *persistence provider* implementiert wird, der daraus SQL-Anweisungen erzeugt und ausführt.

Die Abb. 19.1-4 zeigt das Beispiel 1a unter Verwendung von JPA. Es ist deutlich zu sehen, dass der Aufwand durch JPA drastisch reduziert wird.

Abb. 19.1-3:
Beispiel für die Anwendung des komplexen DAO-Musters.

I 19 Das Subsystem Persistenz

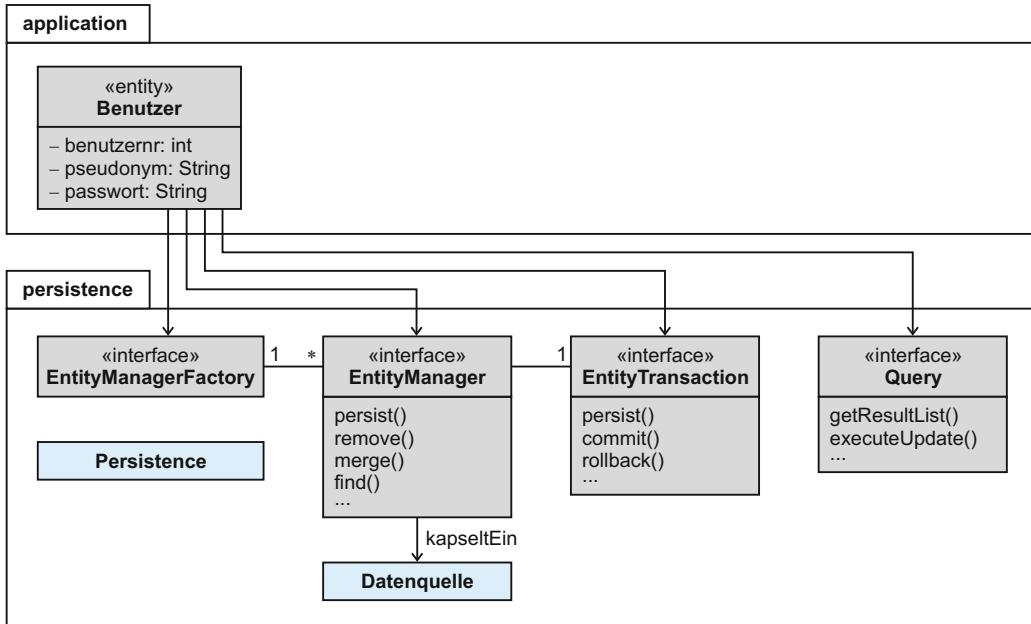


Abb. 19.1-4:
Beispiel für die
Benutzung des JPA.

19.2 Exkurs: ORM – Objektrelationale Abbildung

Objektorientierung gilt heute als Stand der Technik bei der Softwareentwicklung. Andererseits speichern die meisten Firmen ihre Anwendungsdaten in relationalen Datenbanken. Es ist also eine Brücke von der objektorientierten zur relationalen Welt notwendig. Sie wird als objektrelationale Abbildung (*object relational mapping, ORM*) bezeichnet.¹

Wichtige Begriffe

In relationalen Datenbanken werden Daten in Form von **Tabellen** gespeichert. Jede Tabelle besteht aus mehreren Zeilen, die auch als **Tupel** bezeichnet werden.

Alle **Tupel** einer Tabelle enthalten die gleichen Attribute, jedoch im Allgemeinen unterschiedliche Attributwerte. Jedes Tupel muss durch einen eindeutigen **Schlüssel** identifizierbar sein.

Oft verwendet man hierfür ein spezielles Attribut, z.B. **kundennummer**, das als **Schlüsselattribut** oder **Primärschlüssel** bezeichnet wird.

Um Tabellen miteinander zu verknüpfen, verwendet man **Fremdschlüssel**. Für jedes Attribut einer Tabelle muss analog zu dem Attribut einer Klasse der Typ definiert werden.

¹Teile der folgenden Texte und Abbildungen wurden mit freundlicher Genehmigung des W3L-Verlags aus [Balz09b, S. 117 ff.] entnommen.

19.2 Exkurs: ORM – Objektrelationale Abbildung I

Jede Tabelle wird bei der objektrelationalen Abbildung – unabhängig davon, ob ein fachliches Schlüsselattribut vorhanden ist oder nicht – um ein **OID-Attribut** erweitert, das die Rolle des Schlüsselattributs spielt.

OID-Attribut –
Schlüsselattribut
ohne fachliche
Bedeutung

Die Abkürzung OID bedeutet **Objektidentität** (*object identity*). Ein OID-Attribut darf *keinesfalls* eine fachliche Bedeutung besitzen, denn erfahrungsgemäß ändert sich diese Semantik.

Würde beispielsweise als OID-Attribut die Kundennummer gewählt und ist eine Erweiterung des Nummernkreises notwendig, dann müssen alle Datensätze, in denen diese Kundennummer als Primär- und Fremdschlüssel vorkommt, aktualisiert werden.

Sollen objektorientierte Systeme mit relationalen Datenbanken arbeiten, dann ist die Abbildung der objektorientierten Datentypen auf SQL-Datentypen notwendig. Für die Modellierung des Klassendiagramms werden in der Regel UML-Typen und selbstdefinierte Datentypen verwendet.

Von OOA- zu SQL-Typen

Erfolgt die Datenspeicherung beispielsweise mit einer Microsoft Access-Datenbank, dann ist eine Abbildung wie in der Tab. 19.2-1 dargestellt, durchzuführen.

| Analyse-Datentyp | Access-Datentyp |
|------------------|------------------------|
| Integer | Zahl (long integer) |
| String | Text (max. 50 Zeichen) |
| Boolean | Ja/Nein |
| Date | Datum/Uhrzeit |
| Currency | Währung |

Tab. 19.2-1:
Datentypen des
OOA-Modells & der
Access-Datenbank.

Abilden einer Klasse

Im einfachsten Fall wird eine Klasse auf eine Tabelle abgebildet, wobei ein OID-Attribut hinzugefügt wird. Abb. 19.2-1 zeigt die Abbildung der Klasse Kunde, die nur primitive Datentypen enthält, auf eine Tabelle.

Primitive
Datentypen

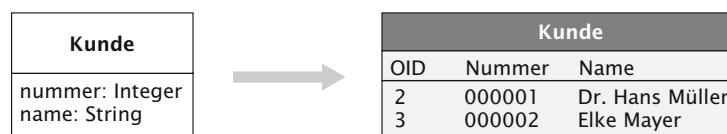


Abb. 19.2-1:
Abbildung einer
einfachen Klasse
auf eine Tabelle.

I 19 Das Subsystem Persistenz

Strukturen – integrieren oder extra Tabelle

Liegt ein Attribut von einem **strukturierten Datentyp** vor, dann gibt es für die Abbildung auf Tabellen zwei Möglichkeiten. Wie Abb. 19.2-2 zeigt, kann das strukturierte Attribut `adresse` entweder in die elementaren Komponenten zerlegt und in die Tabelle `Kunde` integriert oder auf eine eigene Tabelle abgebildet werden.

Beim Integrieren in die Tabelle `Kunde` geht die ursprüngliche Struktur verloren. Bei der Alternative besteht der Nachteil darin, dass beim Zugriff auf ein Kundenobjekt immer eine zusätzliche Tabellenverknüpfung (*join*) von der Tabelle `Kunde` zur Tabelle `Adresse` durchzuführen ist.

Neuere SQL-Standards ermöglichen auch die direkte Abbildung von strukturierten Datentypen. Sie wird allerdings nicht von allen Datenbanksystemen unterstützt.

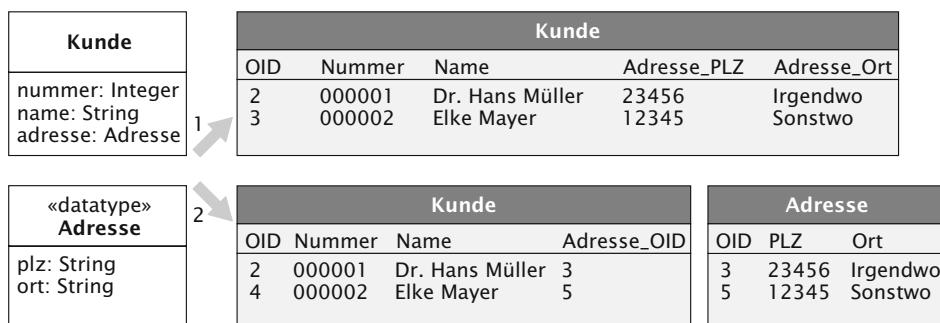


Abb. 19.2-2: Abbildung eines strukturierten Attributs auf eine oder zwei Tabellen.

Abilden einer Assoziation

Schlüssel-Fremdschlüssel-Beziehung

In der objektorientierten Welt »kennen sich« Objekte über ihre Objektbeziehungen (*links*). Bei relationalen Datenbanken werden diese Verbindungen durch Schlüssel-Fremdschlüssel-Beziehungen realisiert. Das bedeutet, dass die Tabellen um entsprechende Fremdschlüssele erweitert werden müssen.

1:m-Assoziation

Betrachten Sie zunächst, wie die **1:m-Assoziation** zwischen `Artikel` und `Lieferant` auf Tabellen abgebildet wird. Dabei ist zu berücksichtigen, dass alle Sätze in einer Tabelle die gleiche Länge besitzen. In der Abb. 19.2-3 wird zu jedem Artikel das OID-Attribut des jeweiligen Lieferanten als Fremdschlüssel gespeichert.

19.2 Exkurs: ORM – Objektrelationale Abbildung I

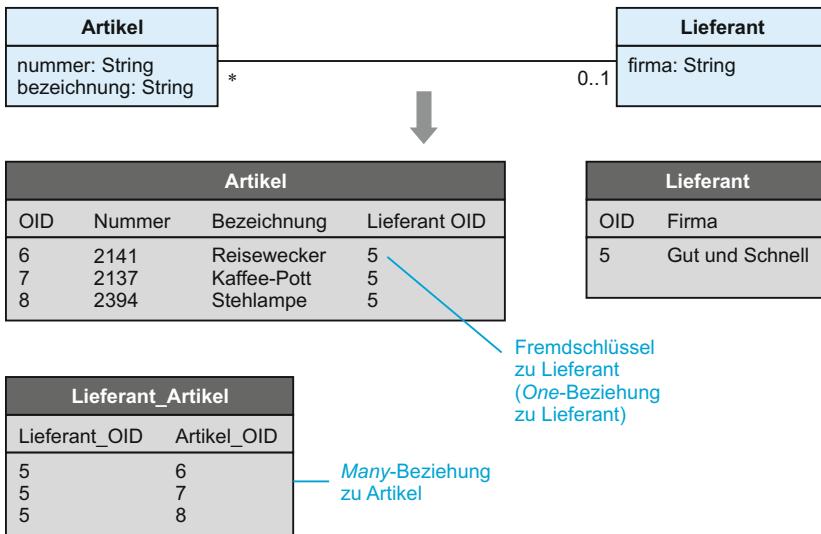


Abb. 19.2-3:
Abbildung einer
1:m-Assoziation
auf Tabellen.

Könnten auch in der Tabelle Lieferant die OID-Attribute aller gelieferten Artikel gespeichert werden?

Frage

Die Antwort heißt nein, denn zu jedem Lieferanten gibt es eine unbekannte Menge von Artikeln. Das hätte zur Folge, dass die Länge eines Datensatzes in der Tabelle nicht konstant ist. Neuere SQL-Standards bieten den *Collection*-Typ an, der eine Menge von Werten aufnehmen kann. Auch dieser Datentyp wird nicht von allen Datenbanksystemen unterstützt.

Antwort

Prinzipiell gibt es zwei Möglichkeiten, um alle Artikel zu einem Lieferanten zu ermitteln. Man kann alle Sätze der Tabelle *Artikel* lesen und alle Artikel des jeweiligen Lieferanten »herausfiltern«. Die andere Möglichkeit besteht darin, die Beziehungen zwischen Lieferant und Artikel in einer eigenen Assoziations-Tabelle zu speichern (Tabelle *Lieferant_Artikel*, Abb. 19.2-3). Diese zweite Alternative ist bei umfangreichen Tabellen wesentlich effizienter.

Die Abb. 19.2-4 modelliert eine **m:m-Assoziation** zwischen den Klassen *Kunde* und *Artikel*. In diesem Fall ist für die Abbildung der Assoziation eine zusätzliche Tabelle *Kunde_Artikel* notwendig, deren einziger Zweck es ist, die Beziehung zwischen den Tabellen zu speichern. Sie wird als **Assoziationsstabelle** (*associative table*) bezeichnet und enthält die beiden Fremdschlüssele *Kunde_OID* und *Artikel_OID*. Der Primärschlüssel setzt sich aus diesen beiden Attributen zusammen.

m:m-Assoziation

I 19 Das Subsystem Persistenz

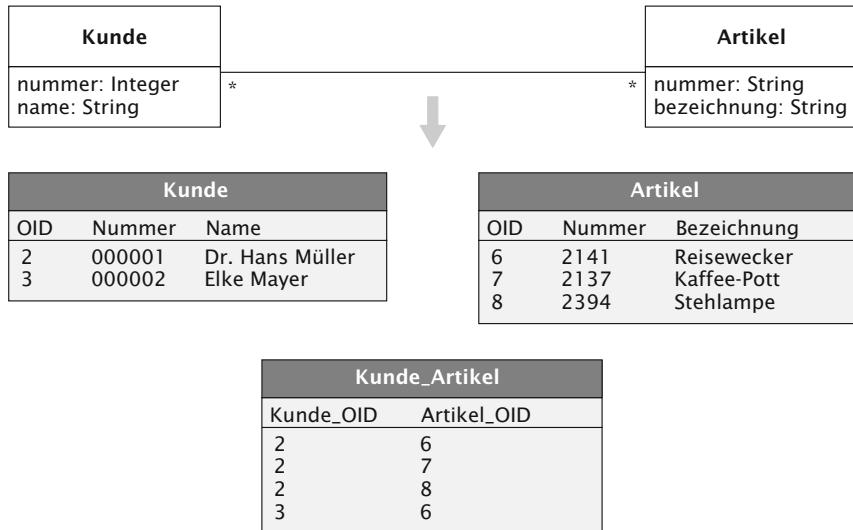


Abb. 19.2-4:
Abbildung einer
m:m-Assoziation
auf Tabellen.

Abbildung der Vererbung

Es gibt drei Möglichkeiten, um eine Vererbungsstruktur auf Tabellen abzubilden (Abb. 19.2-5).

Eine Tabelle für alle Klassen

Bei der ersten Variante werden alle Objekte aus allen Klassen der Vererbungshierarchie in einer einzigen Tabelle gespeichert. Das zusätzliche Attribut **Geschäftspartner_Typ** ist notwendig, um für jeden Tabelleneintrag angeben zu können, ob es sich um einen Kunden oder einen Lieferanten handelt. Der Vorteil dieses Ansatzes liegt in seiner Einfachheit. Ein Nachteil ist, dass die entstehende Tabelle »durchlöchert« ist. Beispielsweise können die mit einem Balken markierten Felder in der Abb. 19.2-5 niemals einen Wert annehmen. Dieser Nachteil ist jedoch bei Vererbungsstrukturen von geringem Umfang vernachlässigbar.

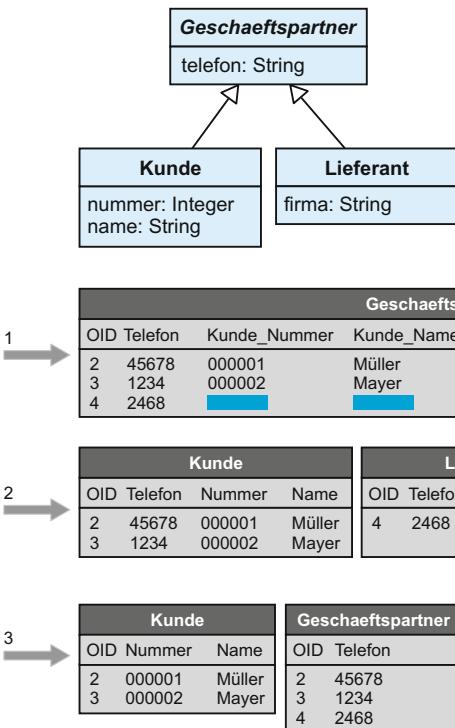
Eine Tabelle für jede konkrete Klasse

Bei der zweiten Variante wird jede konkrete Klasse auf eine Tabelle abgebildet. Sie enthält außer ihren eigenen Attributen auch alle Attribute ihrer Oberklassen. Jede Tabelle erhält als Primärschlüssel das **OID**-Attribut. Nachteilig ist, dass die Attribute der abstrakten Oberklasse in mehreren Tabellen vorhanden sind. Wenn diese Attribute modifiziert werden, dann sind alle betroffenen Tabellen zu aktualisieren. Zusätzlich muss bei einer Selektion über alle Objekte der Oberklasse ein Join von allen Tabellen gebildet werden!

Eine Tabelle für jede Klasse

Bei der dritten Variante wird jede Klasse – auch eine abstrakte – auf eine Tabelle abgebildet. Die Daten eines Kunden-Objekts sind dann beispielsweise auf die Tabelle **Kunde** und **Geschäftspartner** aufgeteilt. Der Zusammenhang wird durch das gemeinsame **OID**-Attribut hergestellt. Es fungiert in der Tabelle **Geschäftspartner** als Pri-

19.3 JPA – Java Persistence API I



märschlüssel, während es in der Tabelle **Kunde** sowohl Primär- als auch Fremdschlüssel ist. Der Hauptvorteil dieses Ansatzes ist, dass er am besten dem objektorientierten Konzept entspricht. Änderungen in der Oberklasse sind mit minimalem Aufwand durchführbar und neue Attribute können in allen Klassen einfach ergänzt werden. Dem stehen jedoch auch Nachteile gegenüber: Es entstehen viele Tabellen in der Datenbank und die Zugriffe auf Objekte dauern länger, weil mehrere Tabellen betroffen sind.

19.3 JPA – Java Persistence API

JPA (*Java Persistence API*) ist ein Bestandteil von *Java EE (Java Enterprise Edition)* und dient dort dazu, *Enterprise Java Beans* zu persistieren (siehe »Die Java EE-Plattform«, S. 321). JPA kann aber auch in Anwendungen der *Java SE (Java Standard Edition)* genutzt werden. Allerdings ist es dafür erforderlich, zunächst zusätzliche Software zu installieren, damit ein JPA-Provider zur Verfügung steht. Eine einfache Möglichkeit besteht in der Nutzung einer speziellen JPA-Implementierung, z.B. der Open-Source-Lösung von Apache, siehe Website Apache OpenJPA (<http://openjpa.apache.org>).

I 19 Das Subsystem Persistenz

- ☞ Beachten Sie, dass Quellcode, der JPA-Elemente enthält, nur erfolgreich kompiliert werden kann, wenn die entsprechende Bibliothek aus dem verwendeten JPA-Provider (z. B. openjpa-1.0.2.jar für OpenJPA von Apache) in den Erstellungspfad eingefügt ist.

Annotationen

Annotationen
 @Entity Aus einer Klasse »ganz gewöhnlicher« nicht-persistenter Java-Objekte, oft mit dem Akronym »POJO« (*Plain Old Java Objects*) bezeichnet, wird eine »Entity-Klasse«, deren Objekte persistent sind, indem man die Klasse mit der **Annotation** `@Entity` (bzw. mit qualifiziertem Namen `@javax.persistence.Entity`) kennzeichnet.

Beispiel 1: Die einfache Klasse Benutzer

Java

```
package de.w3l.db.jpa;
public class Benutzer
{
    private Long benutzernr;
    private String pseudonym;
    private String password;
}
```

wird nach Kennzeichnung mit `@Entity` zu einer persistenten Klasse. Die Annotation muss zuvor aus dem Paket `javax.persistence` importiert werden.

```
package de.w3l.db.jpa;
import javax.persistence.Entity;

@Entity
public class Benutzer
{ ... }
```

Mit anderen Annotationen kann bestimmt werden, wie die einzelnen Attribute des Java-Objekts auf Spalten in der relationalen Datenbank abzubilden sind.

JPA setzt voraus, dass die persistente Klasse ein **Schlüsselattribut** bzw. einen Primärschlüssel enthält. Eine wichtige Annotation ist daher die Kennzeichnung dieses Attributs. Im Beispiel 1 ist dies `benutzernr`. Hat eine Klasse noch *kein* Schlüsselattribut, dann sollte man an dieser Stelle die Klasse um ein entsprechendes Attribut vom Typ `Long` erweitern.

 @Id Durch die Annotation `@Id` wird das Schlüsselattribut oder die zum Schlüsselattribut gehörende get-Methode markiert. Da JPA auch auf private Attribute zugreifen kann, ist an dieser Stelle die Kennzeichnung des Attributs ausreichend.

Werden die Werte des Schlüsselattributs *nicht* von der Anwendung vorgegeben, sondern sollen durch das Subsystem »Persistenz« generiert werden, dann muss dies durch die zusätzliche Annotation @GeneratedValue deutlich gemacht werden.

```
@Id  
//@GeneratedValue zusätzlich, wenn Wert durch das  
//Subsystem Persistenz generiert werden soll.  
public Long getBenutzernr()  
{  
    return benutzernr;  
}  
  
public void setBenutzernr(Long benutzernr)  
{  
    this.benutzernr = benutzernr;  
}
```

Beispiel 1b
Java

Es gibt insgesamt 63 JPA-Annotationen, siehe Java Persistence API (<http://java.sun.com/javaee/5/docs/api/index.html?javax/persistence/package-summary.html>).

Weitere
Annotationen

EntityManager

Nachdem gekennzeichnet worden ist, welche Klassen bzw. Attribute auf welche Art gespeichert werden sollen, braucht man jetzt allerdings noch jemanden, der sich um die Speicherung selbst kümmert. Diese Aufgabe wird durch den EntityManager erledigt. Der EntityManager definiert eine Schnittstelle, die Methoden für alle Funktionalitäten spezifiziert, die im Zusammenhang mit der Abbildung von Laufzeitobjekten auf Datensätze in einer relationalen Datenbank benötigt werden. Wesentliche Methoden sind:

- persist(..), um ein Objekt dauerhaft zu speichern,
- merge(..), um Änderungen an einem Objekt in die Datenbank zu übernehmen,
- remove(..), um ein Objekt aus der Datenbank zu entfernen und
- find(..), um ein Objekt über seinen Primärschlüssel in der Datenbank zu suchen.

Ein konkreter EntityManager muss durch den JPA-Provider bereitgestellt werden.

EntityManagerFactory

Ein Objekt, das den Anforderungen der Schnittstelle EntityManager genügt, kann man von einer EntityManagerFactory erhalten. Auch hier handelt es sich aber wieder nur um eine Schnittstelle. Eine entsprechende Fabrik liefert die Klasse Persistence, die übrigens die einzige konkrete Klasse im gesamten Paket javax.persistence ist.

I 19 Das Subsystem Persistenz

Durch Aufruf von Persistence.createEntityManagerFactory(..) erhält man zunächst ein EntityManagerFactory-Objekt, mit dessen Hilfe man dann anschließend durch Aufruf von createEntityManager einen neuen EntityManager erzeugen kann. Dieser kann nun zum Speichern und Abrufen von Objekten verwendet werden.

Abfragen

JPQL JPA beinhaltet eine Abfragesprache mit dem Namen **JPQL** (*Java Persistence Query Language*). Diese verfügt über ähnliche Möglichkeiten wie **SQL** und lehnt sich auch in der Syntax an SQL an. Damit wird es z.B. möglich, auf persistente Attribute über Nichtschlüsselattribute zuzugreifen. Ist em ein EntityManager, so kann mittels

```
javax.persistence.Query abfrage = em.createQuery(...);  
ein Query-Objekt abfrage erzeugt werden. Über  
List ergebnisliste = abfrage.getResultList(); bzw.  
Object ergebnis = abfrage.getSingleResult();  
kann dann anschließend eine Liste von Ergebnisobjekten bzw. ein  
einziges Ergebnisobjekt abgerufen werden.
```

Eine Sprachreferenz finden Sie unter JPQL-Referenz (http://edocs.bea.com/kodo/docs41/full/html/ejb3_langref.html).

Beispiel 1: Der gesamte Quellcode der Klasse Benutzer sieht folgendermaßen aus:

```
Java  
Benutzer package de.w3l.db.jpa;  
import javax.persistence.Entity;  
import javax.persistence.EntityManager;  
import javax.persistence.EntityManagerFactory;  
import javax.persistence.Id;  
import javax.persistence.Persistence;  
import javax.persistence.Query;  
  
@Entity  
public class Benutzer  
{  
    private Long benutzernr;  
    private String pseudonym;  
    private String password;  
  
    @Id  
    // @GeneratedValue zusätzlich, wenn Wert durch  
    // Persistenzschicht generiert werden soll.  
    public Long getBenutzernr()  
    {  
        return benutzernr;  
    }  
  
    public void setBenutzernr(Long benutzernr)  
    {  
        this.benutzernr = benutzernr;
```

```
}

public String getPassword()
{
    return passwort;
}

public void setPassword(String passwort)
{
    this.passwort = passwort;
}

public String getPseudonym()
{
    return pseudonym;
}

public void setPseudonym(String pseudonym)
{
    this.pseudonym = pseudonym;
}

public static void main(String[] args)
{
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("benutzer");
    EntityManager em = emf.createEntityManager();

    /* Jetzt soll ein Objekt erzeugt ... */
    Benutzer neuerBenutzer = new Benutzer();
    neuerBenutzer.setBenutzernr(4711);
    neuerBenutzer.setPseudonym("meier2");
    neuerBenutzer.setPassword("geheim");

    /*
     * ... und jetzt persistent gespeichert werden Zunächst
     * wird eine neue Transaktion geöffnet. Anschließend
     * wird das Objekt gespeichert. Am Ende wird die
     * Transaktion abgeschlossen.
     */
    em.getTransaction().begin();
    em.persist(neuerBenutzer);
    em.getTransaction().commit();

    /* Jetzt wird das Passwort geändert */
    neuerBenutzer.setPassword("vergessen");

    /* und die Datenbank wird aktualisiert */
    em.getTransaction().begin();
    em.merge(neuerBenutzer);
    em.getTransaction().commit();

    /*
     * Jetzt wird ein Benutzer über sein Pseudonym gesucht.
     * Dazu wird JPQL verwendet. Als Ergebnis wird ein
    
```

I 19 Das Subsystem Persistenz

```
* einzelnes Objekt erwartet, sodass getSingleResult()
* verwendet werden kann.
*/
neuerBenutzer = null;
Query abfrage =
em.createQuery(
    "SELECT b FROM Benutzer b
     WHERE b.pseudonym = 'meier2'");
neuerBenutzer = (Benutzer) abfrage.getSingleResult();

/*
* Am Ende sollte auch der EntityManager geschlossen
* werden
*/
em.close();

/*
* Wenn man sicher ist, dass kein EntityManager
* mehr gebraucht wird, kann auch die Factory
* geschlossen werden.
*/
emf.close();
}
}
```

Bewertung Mit JPA können prinzipiell beliebige Java-Objekte persistiert werden. Die Klassen, deren Objekte dauerhaft gespeichert werden sollen, müssen sich *nicht* aus einer Oberklasse ableiten, sodass die Verwendung von JPA nicht zu einer Einschränkung von Vererbungshierarchien führt. Die persistenten Klassen müssen auch keine Schnittstelle implementieren.

Mit den Annotationen, die JPA bereitstellt, können Klassen als persistente Entitäten markiert werden. Dabei werden alle Attribute der Klasse *automatisch* persistent gehalten. Für die Persistierung ist kein weiterer Code notwendig, er wird durch die JPA hinzugefügt.

Die Kennzeichnung, welche Klassen bzw. Attribute wie gespeichert werden sollen, findet in Form von Annotationen im Quellcode der zu speichernden Klasse selbst statt.

Literatur [Gonc09, S. 41 ff.]

19.4 Fallstudie: KV – JPA

Die Fallstudie »Kundenverwaltung-Mini« mit der Fachkonzeptklasse Kunde (siehe »Fallstudie: KV – Überblick«, S. 15) soll mithilfe von JPA (*Java Persistence API*) realisiert werden (Abb. 19.4-1). Die Daten sollen in der relationalen Datenbank Derby gespeichert werden. Als JPA-Implementierung soll die Open-Source-Lösung von Apache OpenJPA (<http://openjpa.apache.org>) verwendet werden.

19.4 Fallstudie: KV – JPA I

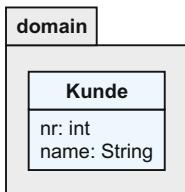


Abb. 19.4-1: OOA-Paketdiagramm für die vereinfachte Kundenverwaltung.

Bei der Verwendung von JPA mit der Java SE (*Standard Edition*) Java SE werden folgende Bibliotheken benötigt:

- Von der Website Apache OpenJPA – Downloads (<http://openjpa.apache.org/downloads.html>) ist folgende Datei herunterzuladen: apache-openjpa-2.0.0-binary.zip. Nach dem Entpacken ist die Bibliothek openjpa-all-2.0.0.jar (oder eine aktuellere Version) in das Projekt einzubinden.
- Von der Website Apache Derby – Downloads (http://db.apache.org/derby/derby_downloads.html) ist folgende Datei herunterzuladen: db-derby-10.5.3.0-bin.zip (oder eine aktuellere Version). Nach dem Entpacken ist die Bibliothek /lib/derby.jar in das Projekt einzubinden.

Bei der Verwendung der IDE Eclipse sollte im Projekt ein neuer Ordner (New->Folder) mit der Bezeichnung lib angelegt werden. Die jar-Dateien können dann mit Drag-and-Drop in diesen Ordner gezogen werden.

Das UML-Diagramm der Abb. 19.4-2 zeigt den Aufbau der Fallstudie. Die Klassen Start, Kundenfenster und KundenContainer bleiben unverändert.

Eclipse

Struktur

Subsystem Applikation

Das Paket javax.persistence definiert die *Java Persistence API*. Die Klasse Kunde wird durch die Annotation @Entity als Klasse gekennzeichnet, deren Attribute in der Datenbank gespeichert werden soll (siehe unten). Im Folgenden werden weitere Annotationen beschrieben (in der Fallstudie »Kundenverwaltung-Mini« wird zusätzlich zu @Entity noch die Annotation @ID genutzt).

- Mit @ID wird angegeben, dass das Attribut zum Primärschlüssel gehört. Bei zusammengesetzten Primärschlüsseln wird @ID vor alle zugehörigen Attribute geschrieben.
- Die Annotation @Embedded gibt an, dass eine Klasse als Komponente eingebunden und in der gleichen Tabelle gespeichert wird. Diese Klasse selber muss dann mit @Embeddable gekennzeichnet werden. Ein Spezialfall sind Aufzählungen. Beispielsweise würde bei der Enumeration

```
public enum Stadt { Berlin, Hamburg, München };
```

nicht die Definition von enum, sondern das Attribut annotiert, dessen Typ enum ist:

I 19 Das Subsystem Persistenz

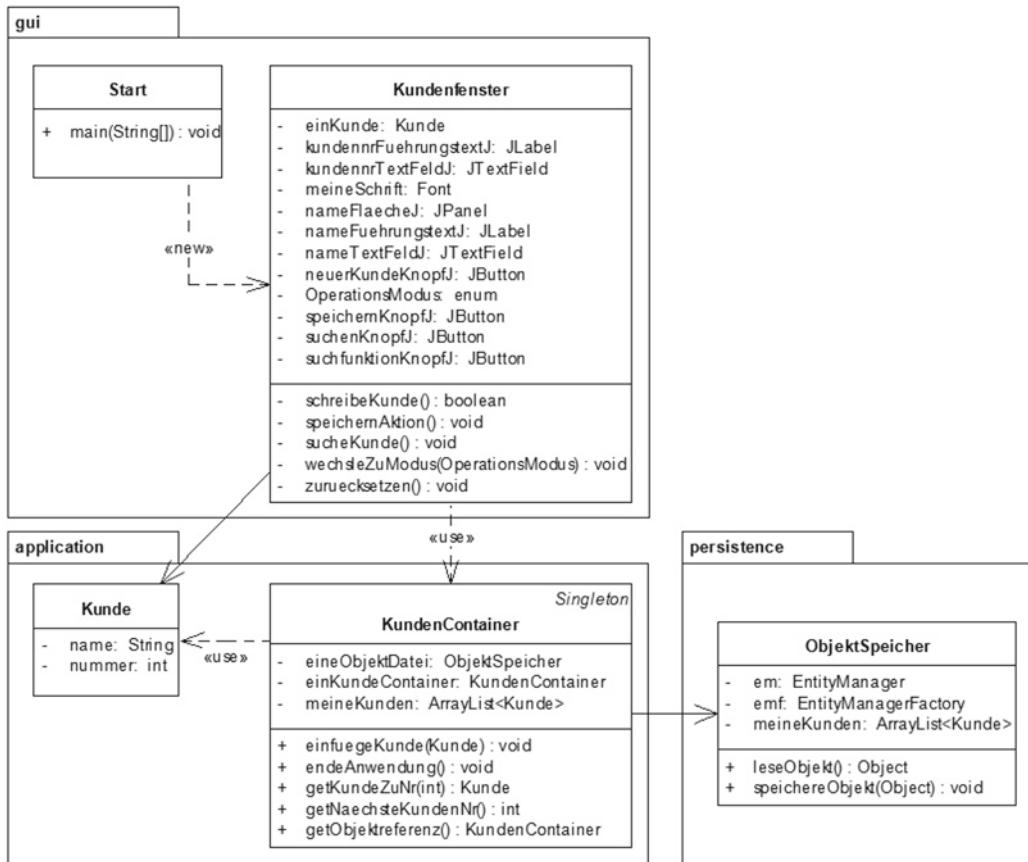


Abb. 19.4-2: OOD Klassendiagramm der Fallstudie KV - JPA.

```

@Enumerated(EnumType.STRING)
private Stadt stadt;

```

- Die Annotation `@Column(name="full_described_field")` legt den Namen der Tabellenspalte fest.
- Die Annotation `@OneToMany` spezifiziert eine 1: \ast - bzw. eine 1:n-Beziehung. `orphanRemoval=true` gibt an, dass bei einem Entfernen eines Objekts dieser Klasse auch das damit verbundene Objekt gelöscht wird. `mappedBy="otherSideProperty"` zeigt an, durch welches Attribut auf der Gegenseite die Assoziation realisiert wird.
- Die Annotation `@ManyToOne` definiert eine *:1- bzw. eine n:1-Beziehung.
- Die Annotation `@JoinColumn(name="field_name")` definiert einen Fremdschlüssel und gibt den Spaltennamen an.

Kunde

```

import javax.persistence.Entity;
import javax.persistence.Id;

```

```

@Entity
public class Kunde

```

19.4 Fallstudie: KV – JPA I

```
{  
    private String name;  
  
    @Id  
    private int nummer;  
  
    public String getName()  
    {  
        return name;  
    }  
  
    public int getNummer()  
    {  
        return nummer;  
    }  
  
    public void setName(String name)  
    {  
        this.name = name;  
    }  
  
    public void setNummer(int nummer)  
    {  
        this.nummer = nummer;  
    }  
}
```

Die JPA-Version der Klasse ObjektSpeicher besitzt zusätzlich die ObjektSpeicher folgenden Attribute :

```
private EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("kunde");  
private EntityManager em = emf.createEntityManager();
```

Beim Parameter für createEntityManagerFactory sieht man die Referenz zu der Datei persistence.xml: kunde ist der Name der persistence-unit (siehe unten).

Da keine dedizierte Sortierung in der Anwendung implementiert ist, soll beim Lesen aus der Datenbank direkt sortiert werden. Dies geschieht mit der JPQL-Anweisung order by ... Die Anfragen werden an EntityManager weitergereicht:

```
//Klasse: ObjektSpeicher  
public Object leseObjekt() throws Exception  
{  
    try  
{  
        Query query = em.createQuery(  
            "select k from Kunde k order by k.nummer");  
  
        //ArrayList<Kunde> wird erwartet  
        meineKunden = new ArrayList<Kunde>();  
        Iterator<Kunde> iter = query.getResultList().iterator();  
        while (iter.hasNext())  
        {
```

Java

I 19 Das Subsystem Persistenz

```
Kunde kunde = (Kunde) iter.next();
meineKunden.add(kunde);
}

if(meineKunden == null)
    meineKunden = new ArrayList<Kunde>();
}
catch (Exception e)
{
    System.out.println("Es wurde eine neue " +
        "Datenbasis angelegt");
    meineKunden = new ArrayList<Kunde>();
}
return meineKunden;
}
```

Auch das Speichern wird weitergereicht. In der Fallstudie werden beim Start der Anwendung alle bereits vorhandenen Datensätze aus der Datenbank gelesen und beim Beenden werden die neuen Datensätze gespeichert:

```
public void speichereObjekt(Object einObjekt)
{
    meineKunden = (ArrayList<Kunde>) einObjekt;

    Iterator<Kunde> iter = meineKunden.iterator();
    while (iter.hasNext())
    {
        Kunde kunde = iter.next();
        em.getTransaction().begin();
        em.persist(kunde);
        em.getTransaction().commit();
    }
    em.close();
    emf.close();
}
```

Subsystem Persistenz

`persistence.xml` Die Klasse ObjektSpeicher greift auf die Datenbank (über JPA) zu. Für die JPA-Konfiguration wird die Datei `persistence.xml` benötigt.

Es wird darin eine `persistence-unit` definiert, womit im Programm dann ein EntityManager konfiguriert wird:

```
<persistence-unit name="kunde"
    transaction-type="RESOURCE_LOCAL">
```

In der `persistence-unit` werden als Erstes alle Klassen aufgelistet, die persistent gespeichert werden sollen:

```
<class>application.Kunde</class>
..
```

19.4 Fallstudie: KV – JPA I

Die Klassen bekommen entsprechende JPA-Annotationen (siehe oben).

Unter <properties> werden verschiedene <property>-Angaben gespeichert. Folgende Angaben zur Datenbank sind nötig:

```
<property name="openjpa.ConnectionURL"
  value="jdbc:derby:c:/db/testDB;create=true" />
<property name="openjpa.ConnectionDriverName"
  value="org.apache.derby.jdbc.EmbeddedDriver" />
<property name="openjpa.ConnectionUserName" value="user1" />
<property name="openjpa.ConnectionPassword" value="user1" />
```

Die Angabe jdbc:derby:c:/db/testDB;create=true bedeutet, dass die als Datei vorliegende Derby-Datenbank testDB verwendet wird (im Ordner C:\db), und dass sie erstellt werden soll, wenn sie nicht vorhanden ist (create=true). Als ConnectionDriver wird der EmbeddedDriver eingesetzt. Benutzername wie Passwort lauten user1. Mit diesen Angaben ist es also nicht nötig, einen externen Derby-Server zu starten, sondern der Derby-Server ist sozusagen mit eingebettet in das Programm, und die Daten werden im angegebenen Ordner gespeichert.

Als letzte Eigenschaft wird gesetzt:

```
<property name="openjpa.jdbc.SynchronizeMappings"
  value="buildSchema"/>
```

Damit wird angegeben, dass OpenJPA automatisch alle Tabellen, Fremdschlüssel usw. in der Datenbank anlegt.

Start der Anwendung und Datensätze in der Datenbank

Der Start der Anwendung erfolgt mit der Klasse Start: Run Eclipse As->Run Configurations... Im Reiter (x) = Arguments ist bei VM arguments folgendes einzutragen: -javaagent:openjpa-all-2.0.0.jar. Wenn die jar-Datei in einem Unterverzeichnis liegt, z.B. in lib, dann ist der Pfad zu jar entsprechend anpassen, z.B. -javaagent:lib/openjpa-all-2.0.0.jar.

Die Resources-Ordner müssen in Eclipse als Source eingetragen sein (use as source folder). Darin gibt es den Ordner META-INF in den die Datei persistence.xml (siehe oben) gelegt werden muss.

Für den Test kann wie in der Abb. 19.4-3 über das GUI ein Kunde Meier angelegt werden.

Der Inhalt der Kundentabelle in der Datenbank sieht danach wie in der Abb. 19.4-4 aus.

Die Programme zu dieser Fallstudie können Sie im E-Learning- OOP Kurs herunterladen.

Installieren Sie die notwendige Software und führen Sie die Fallstudie auf Ihrem Computersystem aus.



I 19 Das Subsystem Persistenz

Abb. 19.4-3:
Eintragung eines
Kunden in der
Fallstudie »KV –
JPA«.

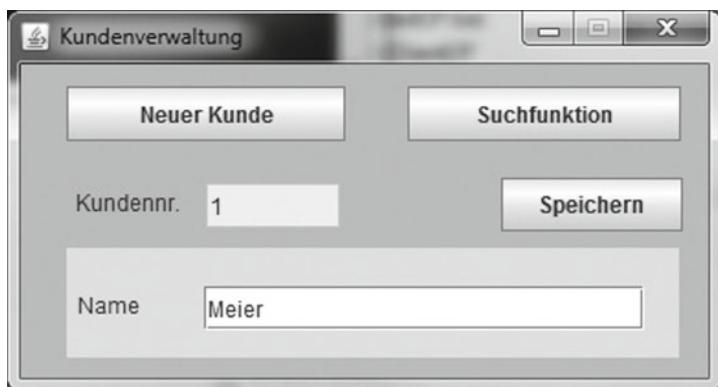


Abb. 19.4-4:
Datenbankabfrage
der Kundentabelle
in der Fallstudie
»KV – JPA«.

```
iJ Version 10.6
ij> connect 'jdbc:derby:c:/db/testDB;user=user1;password=user1';
ij> select * from kunde;
NUMMER      !NAME
-----
1          !Meier
1 Zeile ausgewählt
ij> -
```

19.5 Persistenz in .NET

Beim Datenzugriff unterstützt .NET sowohl relationale Datenbanken als auch XML. Beide Welten sind in .NET eng verknüpft. Ein Entwickler kann relationale Daten mit einer Zeile Programmcode in XML umwandeln oder XML-Daten (mit bestimmten Einschränkungen) ebenso einfach in eine verknüpfte Menge von Tabellen überführen.

Die XML-Unterstützung ist im Namensraum `System.Xml` realisiert und bietet neben den W3C-Standards für XML, XML, XSD, XPATH und XSLT auch einige Microsoft-eigene (aber meist schnellere) Zugriffsverfahren auf XML-Dokumente, z. B. `XMLReader` und `XMLWriter`.

Der Zugriff auf Datenbanken (Namensraum `System.Data`) ist unter dem Namen ADO.NET bekannt.

Innerhalb von ADO.NET gibt es zwei verschiedene Architekturansätze. Auf der einen Seite gibt es den klassischen tabellenorientierten Datenzugriff unter Einsatz der Standardsprache SQL, vergleich-

bar mit JDBC in der Java-Welt (siehe »Vom Direktzugriff bis zum JPA«, S. 422). Auf der anderen Seite steht die objektrelationale Abbildung mit dem ADO.NET Entity Framework, bei der die Tabellenstrukturen auf Objektmodelle abgebildet werden, vergleichbar mit JPA in der Java-Welt (siehe »JPA – Java Persistence API«, S. 431).

Tabellenorientierter Datenzugriff

Innerhalb des tabellenorientierten Datenzugriffs in ADO.NET ist wieder zu differenzieren zwischen einem verbindungslosen Datenzugriff und verbindungsorientierten Datenzugriff. Grundsätzlich ist für den Datenbankzugriff zunächst immer der Aufbau einer Verbindung (`DbConnection`) mit Hilfe eines ADO.NET-Datenbanktreibers zu einem Datenbankmanagementsystem notwendig. Danach folgt die Ausführung eines DML-Befehls (`Insert`, `Update`, `Delete`) über ein `DbCommand`-Objekt oder die Ausführung einer SQL-Abfrage, die Daten liefert. Hier ist zu unterscheiden, ob die angeforderten Daten dann sukzessive nach Anforderung zeilenweise vom Client ausgelesen werden (Hier ist die Verbindung solange offen zu halten, wie Daten gelesen werden, daher spricht man hier von »verbindungsorientierten Datenzugriff«) oder direkt komplett in den Hauptspeicher des Clients geladen werden. Bei diesem zweiten Weg kann die Verbindung dann sofort getrennt werden, daher spricht man hier von »verbindungslosem Datenzugriff«. Die Bezeichnungen »verbindungslos« / »verbindungsorientiert« beziehen sich also auf die Sicht des Clients bei der Verwendung der Daten. Die Abb. 19.5-1 veranschaulicht den verbindungsorientierten und den verbindungslosen Datenzugriff.

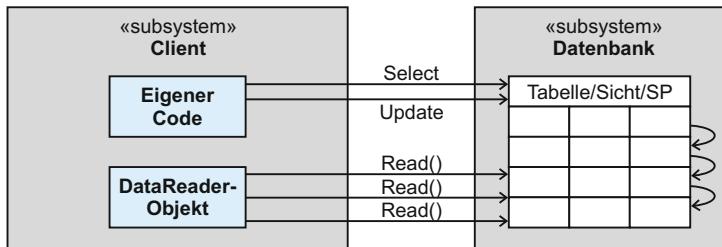
Für den verbindungsorientierten Datenzugriff stellt ADO.NET `DbDataReader`-Objekte bereit. Für den verbindungslosen Zugriff das `DataSet`-Objekt. `DbDataReader`-Objekte bieten keine Unterstützung für DML-Zugriffe, hier muss der Entwickler selbst DML-Befehle erzeugen, in `DbCommand`-Objekte verpacken und zur Datenbank senden. Das Lesen von Daten mit einem `DbDataReader`-Objekt ist aber wesentlich schneller als mit einem `DataSet`-Objekt. Das `DataSet` bietet aber mehr Komfort und stellt Funktionen einer In-Memory-Datenbank zur Verfügung wie das Filtern, Sortieren und Verknüpfen im RAM. In konkreten Anwendungsfällen sind Datenmenge und Nutzungshäufigkeit sowie die daraus resultierende Netzwerklast, den RAM-Verbrauch und die Rechenzeit abzuwägen für die Entscheidung zwischen `DbDataReader` und `DataSet`.

Die verschiedenen ADO.NET-Datenbanktreiber bieten Ableitungen für alle mit dem Kürzel `Db` beginnenden Klassen, z.B. `OracleCommand` für die Ausführung von Befehlen in Oracle-Datenbanksystemen und `SqlCommand` für die Ausführung von Befehlen in Microsoft SQL Server

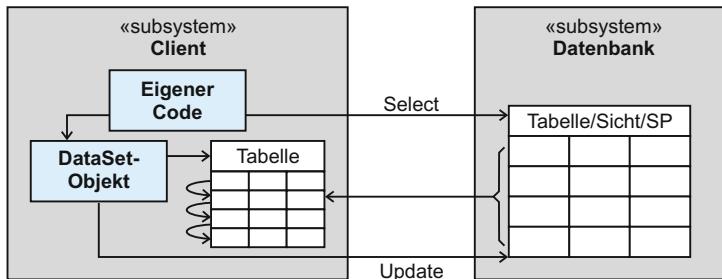
I 19 Das Subsystem Persistenz

Abb. 19.5-1: Verbindungsorientierter Datenzugriff (Server Cursor)
Vergleich der Zugriffsmodelle in ADO.NET

| |
|------------------------------|
| «subsystem» Client |
|------------------------------|



Verbindungsloser Datenzugriff (Client Cursor)



(hier ist zu beachten, dass »SQL« von Microsoft oftmals als Abkürzung für sein Datenbankmanagementsystem Microsoft SQL Server verwendet wird statt für die Standardsprache SQL).

Objektrelationale Abbildung

Während in der Java-Welt die objektrelationale Abbildung (ORM) schon eine lange Tradition besitzt, hat Microsoft diesen Trend lange verschlafen bzw. es nicht vermocht, ein geeignetes Produkt zur Marktreife zu führen. Drittanbieter und Open-Source-Projekte, oftmals Portierungen aus der Java-Welt (z. B. nHibernate zu Hibernate), füllten diese Lücke.

Erst mit dem .NET Framework Version 3.5 führte Microsoft die Bibliothek »ADO.NET Entity Framework« ein.

Das »ADO.NET Entity Framework« ist nicht als Ersatz, sondern als Ergänzung zum tabellenorientierten Datenzugriff zu sehen. Intern basiert das »ADO.NET Entity Framework« auf `DbDataReader` und `DbCommand`. Für Massenoperationen ist zu empfehlen, auf die zusätzliche Abstraktion zu verzichten.

Kernkonzept des Entity Framework ist das Entity Data Model (EDM), das die Abbildung von Tabellen, Sichten und *Stored Procedures* auf Klassen, Assoziationen und Vererbungsstrukturen beschreibt. Das Objektmodell kann dabei auch Vererbung nutzen.

Das Entity Framework bietet verschiedene Strategien, dies auf das relationale Datenbankmodell abzubilden. Eigenarten des relationalen Datenbankmodells – wie N:M-Zwischentabellen – werden durch das Entity Framework eliminiert.

Das XML-basierte EDM besteht aus drei Teilen:

- Die *Store Schema Definition Language* (SSDL) beschreibt das Schema der Datenbank.
- Die *Conceptual Schema Definition Language* (CSDL) beschreibt das Objektmodell.
- Die *Mapping Specification Language* (MSL) dient zur Abbildung des Datenbankschemata auf das Objektmodell.

Das EDM entsteht entweder zur Entwicklungszeit mit Hilfe eines grafischen Designers oder Ad-Hoc zur Laufzeit durch Programmcode. Es besteht die Wahl, eine Datenbank aus dem EDM erzeugen zu lassen (*Forward Engineering*) oder eine bestehende Datenbank in ein EDM einzulesen (*Reverse Engineering*).

Als Abfragesprache bietet das Entity Framework neben SQL zwei Sprachen an, die spezifischer mit dem erzeugten Objektmodell arbeiten. Entity SQL (eSQL) ist eine textbasierte Abfragesprache, die große Ähnlichkeit zu SQL besitzt, aber datenbankneutral ist und um zusätzliche Sprachelemente erweitert wurde. Die Hauptabfragesprache ist jedoch LINQ-to-Entities. LINQ-to-Entities gehört zum Konzept der *Language Integrated Query* (LINQ), einer Innovation auf Ebene der Sprachsyntax, die Microsoft im Jahr 2008 einführt.

LINQ ist ebenfalls SQL-ähnlich, wird jedoch anders als klassische SQL-Befehle vom Sprachcompiler nicht als Zeichenkette, sondern als komplizierbare Anweisung betrachtet. Die Sprachen wurden dafür um Schlüsselwörter wie `select`, `from`, `where`, `orderby`, `groupby` usw. erweitert.

Die kompilierten Abfragen können zur Laufzeit durch LINQ-Provider auf ganz unterschiedliche Datenspeicher ausgeführt werden. Neben Datenbanken sind Abfragen auch auf Objektmengen im Hauptspeicher, XML-Dokumente, Excel-Tabellen, SAP-Systeme und Websites wie Amazon, Google und Twitter möglich. Der jeweilige LINQ-Provider implementiert dabei die Umsetzung von LINQ in die vom jeweiligen Datenspeicher verwendete Abfragesprache.

Damit bietet LINQ eine einheitliche Abfragesyntax für zahlreiche heterogene Datenspeicher verbunden mit dem weiteren Vorteil, dass der Sprachkomplier die Syntax prüfen kann.

Die Abb. 19.5-2 zeigt ein relationales Datenmodell für eine Fluggesellschaft und Abb. 19.5-3 das daraus resultierende EDM. Beispiel

I 19 Das Subsystem Persistenz

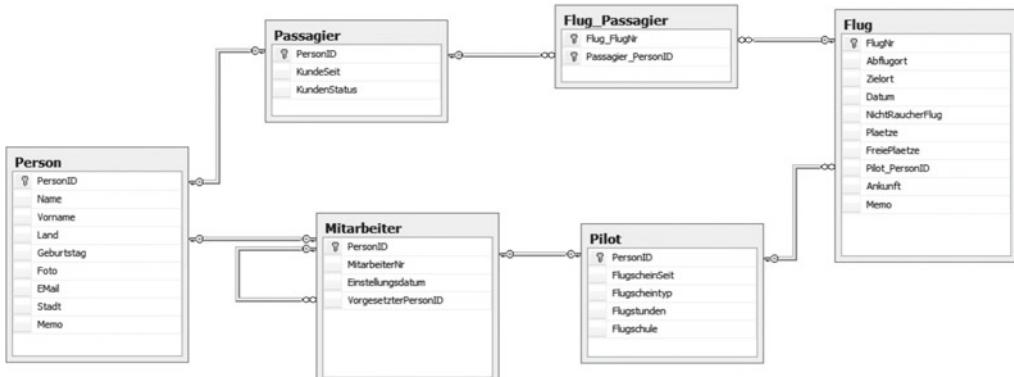


Abb. 19.5-2:
Relationales
Datenmodell für
eine
Fluggesellschaft.

Im Folgenden wird eine LINQ-to-Entities-Abfrage auf dem obigen EDM durchgeführt. Die Abfrage liefert alle Flüge von einem Abflugort, die mindestens einen Passagier haben und auf denen mindestens ein Passagier mit einem bestimmten Nachnamen gebucht ist. Für diese Abfrage werden also die Vererbungs- und Assoziationsbeziehungen zwischen den Entitäten ausgenutzt. Das Entity Framework erzeugt automatisch einen SQL-Befehl mit den notwendigen Joins über die vier Tabellen Flug, Flug_Passagier, Passagier und Person.

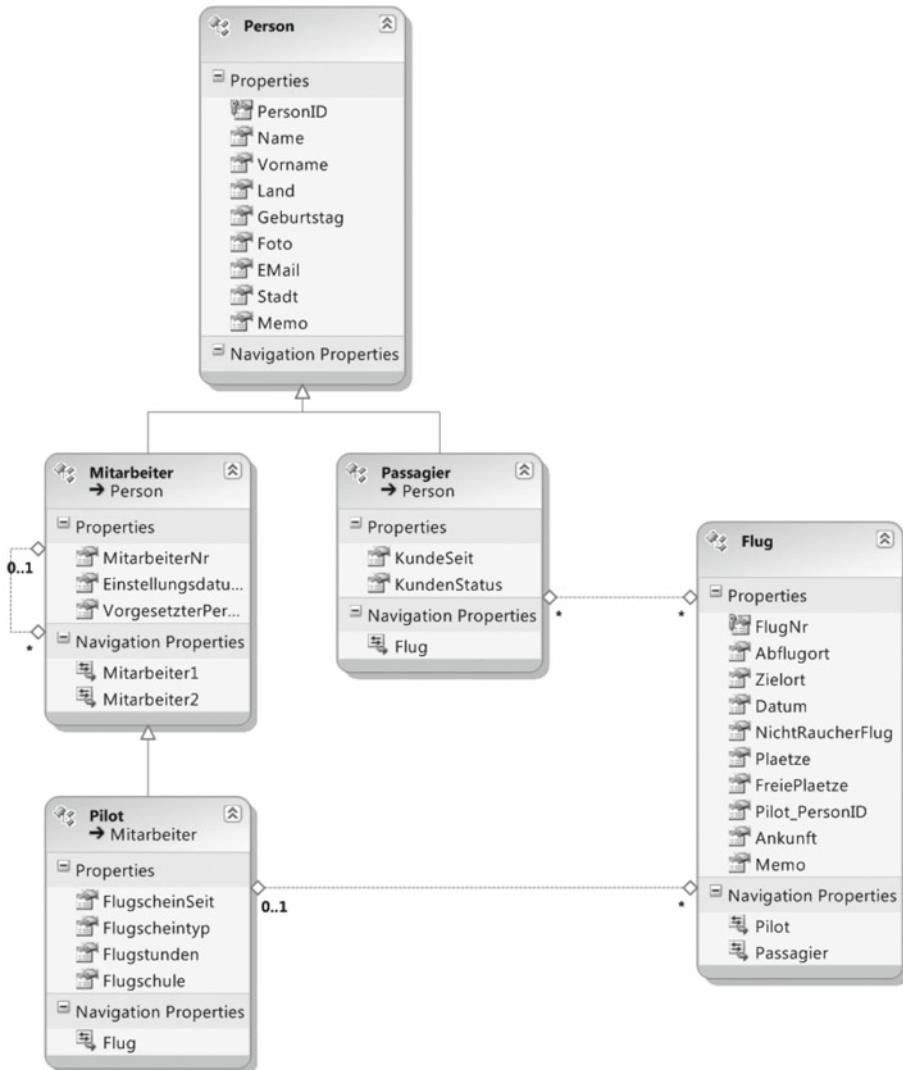
Das Beispiel zeigt außerdem die Veränderung der Objekte. Der gesuchte Passagier wird von allen Flügen entfernt, die Platzanzahl entsprechend um eins erhöht. Am Ende speichert das Programm alle Änderungen mit dem Aufruf von SaveChanges(). Die Methode SaveChanges() erzeugt automatisch eine Datenbanktransaktion, d. h. es werden entweder alle Änderungen oder keine einzige Änderung gespeichert.

```

///<summary><br/>
///Beispiel für die Verwendung des ADO.NET Entity Framework
///zum Lesen und Verändern von Daten
///</summary>
static void Beispiel_EF()
{
    //Modell instanziieren unter Verwendung der
    //Verbindungszeichenfolge aus Konfigurationsdatei
    using (WwWings6Entities modell = new WwWings6Entities())
    {
        string ort = "Rom";
        string name = "Müller";

        //Abfrage definieren
        var fluege = from f in modell.Flug.Include("Passagier")
                     where f.Abflugort == ort
                     && f.Passagier.Count > 0
                     && f.Passagier.Any(p => p.Name == name)
                     select f;
    }
}
  
```

19.5 Persistenz in .NET I



```
//Ergebnis ausgeben
foreach (Flug f in fluege)
{
    Console.WriteLine
        ("Flug Nr {0} von {1} nach {2} hat {3} freie Plätze!",
        f.FlugNr, f.Abflugort, f.Zielort, f.FreiePlaetze);

    //Passagiere ausgeben
    foreach (Passagier ps in f.Passagier.ToList())
    {

```

Abb. 19.5-3: Entity Data Model, abgeleitet aus dem relationalen Datenmodell für eine Fluggesellschaft.

I 19 Das Subsystem Persistenz

```
Console.WriteLine(" - " + ps.Name);

//Entferne den Passagier mit gesuchtem Namen
if (ps.Name == name)
{
    f.Passagier.Remove(ps);
    Console.WriteLine("    gelöscht!");
}

//Platzanzahl erhöhen
f.FreiePlaetze++;

}

//Alle Änderungen speichern
model1.SaveChanges();
} //Ende using-Block -> Dispose() wird aufgerufen
}
```

Optimistische Sperren von Datensätzen

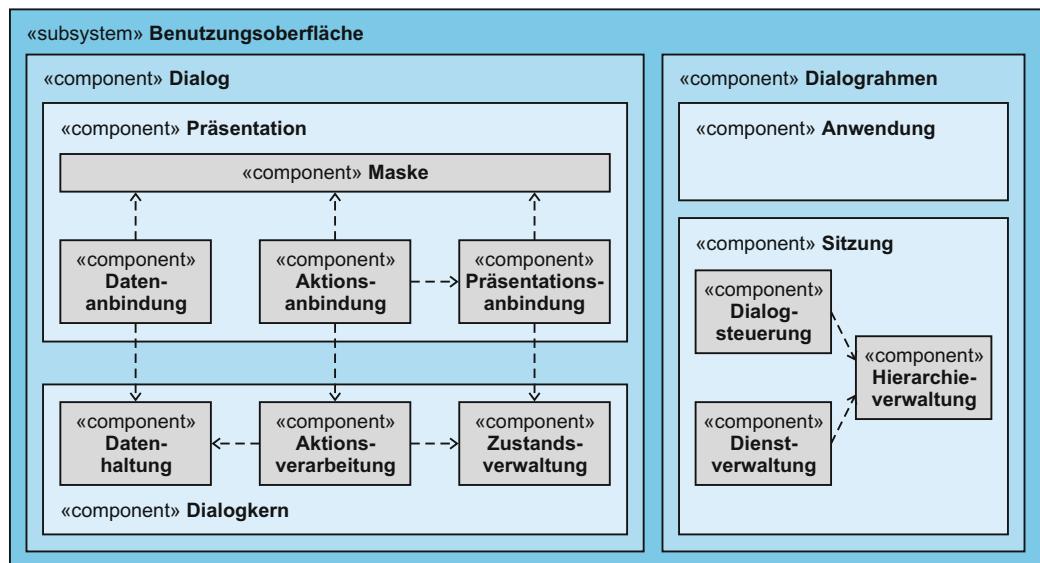
Unter dem Begriff Sperren (*Locking*) werden Mechanismen zusammengefasst, die die häufig mit Änderungskonflikten behafte gleichzeitige Bearbeitung eines Datensatzes durch mehrere Benutzer verhindern. Sowohl ADO.NET als auch das ADO.NET Entity Framework unterstützen das sogenannte »optimistische Sperren«. Optimistisches Sperren ist dabei ein Euphemismus, denn es verhindert die gleichzeitige Bearbeitung eines Datensatzes durch mehrere Benutzer gar nicht, sondern löst nur im Nachhinein einen Fehler aus. Dies geschieht, wenn zwei oder mehrere Benutzer einen Datensatz gelesen haben, einer von diesen (egal welcher) eine Änderung in die Datenbank schreibt und nun ein zweiter Benutzer diesen Datensatz auch schreiben will. Der zweite Benutzer bemerkt also erst beim Speichern, dass ein Speichern nicht möglich ist. In der Regel waren seine Änderungen dann vergebens, außer wenn die Software explizit Mechanismen vorsieht, um die Änderungen des ersten Benutzers zu überschreiben. Echte Sperrmechanismen sind in ADO.NET und ADO.NET Entity Framework nur im Rahmen von Datenbanktransaktionen verfügbar (siehe »Transaktionen in .NET«, S. 186).

20 Das Subsystem Benutzeroberfläche

Die »Binnen«-Architektur des Subsystems »Benutzeroberfläche« wird in der Regel durch die verwendeten *Frameworks* implizit vorgegeben. Die Architekturen der *Frameworks* sind oft auf den Kontext zugeschnitten, z.B. ob es sich um eine Desktop- oder Web-Oberfläche handelt. Oft spielt auch die Art der Netzkomunikation eine Rolle, z.B. ob über HTTP oder RMI die Daten übertragen werden.

Im Folgenden wird in Anlehnung an [HaOl07] eine allgemeine, komponentenbasierte Architektur für Benutzeroberflächen betrieblicher Informationssysteme skizziert (Abb. 20.0-1). Basis für die Architektur sind die zu erledigenden Aufgaben, die unabhängig vom Kontext, der Kommunikationsart und dem genutzten UI-*Framework* sind. Die Abb. 20.0-1 gibt einen Überblick über die Architektur.

Allgemeine Architektur



Zu jedem Dialograhmen kann es eine oder mehrere Dialog-Komponenten geben. Ziel ist es, Dialoge und Dialograhmen möglichst stark zu entkoppeln. Dadurch ist es möglich, dass Dialoge unterschiedlicher Komplexität auch eine unterschiedliche Architektur besitzen können. Zwischen Dialograhmen und Dialogen gibt es folgende Abhängigkeiten:

Abb. 20.0-1:
Binnen-Architektur
des Subsystems
Benutzerober-
fläche nach
[HaOl07].

I 20 Das Subsystem Benutzungsoberfläche

- Jeder Dialog muss für den Dialograhmen identifizierbar sein.
- Der Lebenszyklus von Dialogen muss vom Dialograhmen gesteuert werden, z. B. das Öffnen und das Schließen von Dialogen.
- Der Dialograhmen erlaubt eine Kommunikation zwischen den Dialogen.
- Jeder Dialog kann auf die Infrastruktur des Dialograhmens zugreifen.

Die Dialog-Architektur

Ein Dialog muss folgende Aufgaben erledigen:

- Erstellung der statischen Benutzungsoberfläche (Maske).
- Eingaben entgegennehmen und prüfen, Daten verwalten und auf der Benutzungsoberfläche darstellen.
- Steuerung der dynamischen Teile der Benutzungsoberfläche.
- Steuerung eigener Unterdialoge und Koordination mit anderen Dialogen.
- Kommunikation mit einem Server, mit einem Dateisystem usw.

Die Erledigung dieser Aufgaben erfolgt in zwei Schichten, der **Präsentationsschicht** und dem **Dialogkern** (siehe Abb. 20.0-1, linke Seite).

Präsentationsschicht

Die Präsentationsschicht erledigt folgende Aufgaben:

- Interaktion mit der verwendeten GUI-Bibliothek, z. B. Swing, JSF, Winforms.
- Präsentation von Daten des Dialogkerns.
- Präsentationsereignisse auf Ereignisse im Dialogkern abbilden.

Die Präsentationsschicht ist von der gewählten GUI-Bibliothek abhängig, aber nicht vom Serverzugriff.

Schicht Dialogkern

Der Dialogkern kümmert sich um folgende Aufgaben:

- Verwaltung der fachlichen Daten und Zustände.
- Ausführen der fachlichen Logik und der Dialoglogik.
- Kommunikation mit dem Server.

Der Dialogkern ist *unabhängig* von der gewählten GUI-Bibliothek.

Der Dialogkern besteht aus den Komponenten

- Datenhaltung,
- Aktionsverarbeitung und
- Zustandsverwaltung.

Datenhaltung Die Datenhaltung wird von der Präsentations-Komponente benutzt, die Daten darstellt und Eingaben zurückschreibt. Außerdem nutzt die fachliche Logik im Dialogkern, die Daten manipuliert und sie mit anderen Dialogen oder einem Server austauscht, die Datenhaltung.

20 Das Subsystem Benutzungsoberfläche I

Die Daten werden in der Datenhaltung *nicht* in der Benutzerrepräsentation abgelegt, sondern in einem logisch verarbeitbarem Format.

- Die PLZ ist kein String, sondern eine ganze Zahl.
- Das Datum ist kein String, sondern ein Datumsobjekt (Date-Objekt in Java). Beispiel

Auf den Objekten können logische Operationen ausgeführt werden.

Am einfachsten kann die Datenhaltung durch Klassen realisiert werden, deren Methoden auf Datenobjekte zugreifen. Bei Web-Architekturen geschieht dies oft durch *Backing Beans* (JSP, JSF). Damit die Oberfläche bei feingranularen Änderungen unmittelbar aktualisiert werden kann, ist es oft sinnvoll, mithilfe des Beobachter-Musters über Änderungen der Datenhaltung informiert zu werden.

Die Abb. 20.0-2 zeigt, wie im einfachsten Fall die Schnittstelle einer Datenhaltung aussehen kann. Zur Anzeige der Daten müssen diese in der Regel in eine für den Benutzer lesbare Form konvertiert werden. Die Formatierung ist dabei von der Lokalität des Benutzers (*locale*) (siehe »Globalisierung von Software«, S. 143) und evtl. von anderen Benutzereinstellungen abhängig, z. B. Sprache, Schriftgröße usw. Umgekehrt müssen vom Benutzer eingegebene Daten auf Korrektheit geprüft werden, bevor sie weiterverarbeitet werden. Diese Aufgaben werden in der Datenanbindung erledigt. Bei der Initialisierung wird per Konfiguration festgelegt, welches Datum der Datenhaltung an welches Element der Oberfläche gebunden wird und welche Konvertier- und Prüfroutinen dabei benutzt werden. Die Abb. 20.0-2 zeigt einen einfachen Adapter ohne Konvertierung und Prüfroutinen für eine native GUI-Bibliothek. Beispiel

Die Datenhaltung kann auch weitere Aufgaben übernehmen:

- Sicherstellung eines thread-sicheren Datenzugriffs.
- Bereitstellung eines Transaktionskonzepts.
- Meta-Informationen über Daten halten.

Die Aktionsverarbeitung setzt die fachliche Logik und die Dialoglogik um. Diese fachlichen Aktionen können mit Daten arbeiten, Serveraufrufe durchführen oder Steuerungsfunktionen übernehmen, beispielsweise einen anderen Dialog öffnen. Sie werden der Präsentationskomponente oder anderen Dialogkomponenten auf einer grobgranularen fachlichen Ebene zur Verfügung gestellt. Technisch kann dies durch Methoden einer Schnittstelle gelöst werden, z. B. durch eine *Backing Bean* wie in JSF. Eine stärkere Entkopplung ermöglicht das Kommando-Muster (siehe »Das Kommando-Muster (command pattern)«, S. 75) oder das Aktions-Muster. Kommandos

Aktions-verarbeitung

I 20 Das Subsystem Benutzeroberfläche

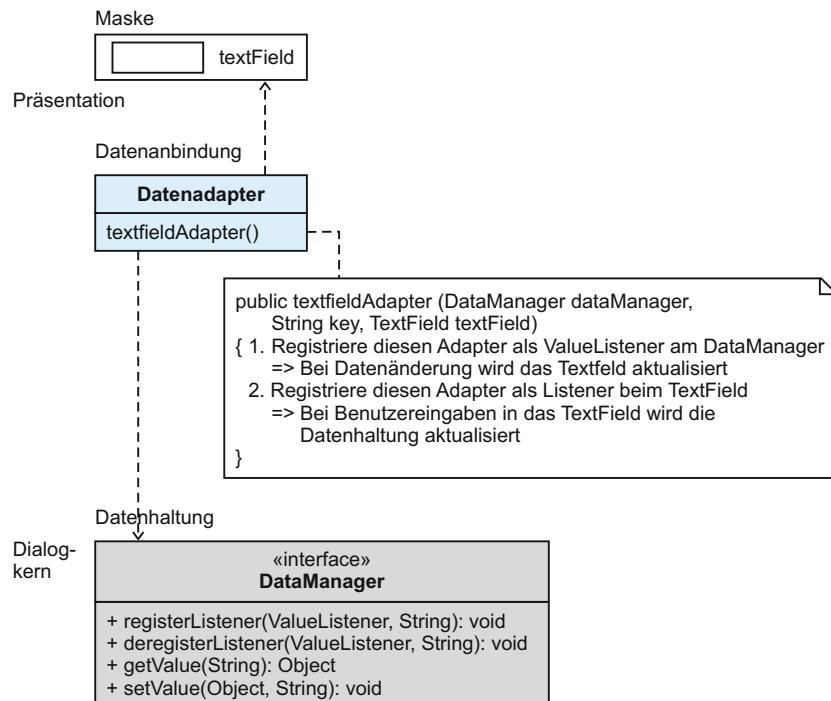


Abb. 20.0-2: oder Aktionen können dabei abhängig von ihrer Aufgabe ausführbar oder gesperrt sein. Die Änderung des Zustands kann über Beobachter verfolgt werden.

Die Abb. 20.0-3 zeigt eine einfache Version der Schnittstellen einer Aktionsverarbeitung auf der Basis des Kommando-Musters. Ein Aktionsadapter verbindet die Komponente »Maske« mit der Komponente »Aktionsverarbeitung«. Das technische Ereignis, z. B. Drücken des Druckknopfes »Speichern«, wird in eine fachliche Aktion, z. B. »Prüfen und Speichern der Daten«, gewandelt.

Zustandsverwaltung Die fachliche Logik bzw. die Dialoglogik kann auf einen Zustandsautomaten abgebildet werden, der durch Ereignisse gesteuert wird und je nach Zustand unterschiedliche Aktionen ausführt. Die fachlichen Aktionen sind als Zustandsübergänge integriert.

Präsentationssteuerung Auf der Benutzeroberfläche gibt es oft feingranulare Zustände. Es gibt Zustände, die der Benutzer ändern kann, z. B. Auswahl einer Tabellenzeile. Und es gibt Zustände, die auf die Benutzeroberfläche wirken, z. B. ob ein Druckknopf aktiviert oder deaktiviert ist. Diese Präsentationszustände hängen oft voneinander und von Zuständen im Dialogkern ab. Die Abb. 20.0-4 zeigt die exemplarische Verarbeitung einer Benutzeraktion durch ein Sequenzdiagramm [HaOl07, S. 153].

20 Das Subsystem Benutzeroberfläche I

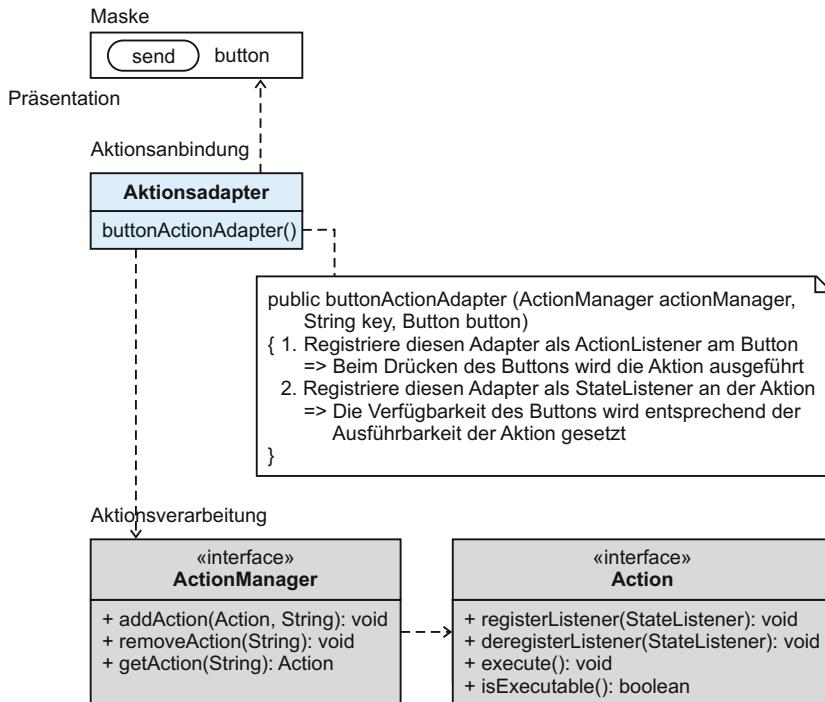


Abb. 20.0-3:
Zusammenspiel
zwischen Maske,
Aktionsanbindung
und Aktionsverar-
beitung.

Die Architektur des Dialograhmens

Der Dialograhmen stellt die Laufzeitumgebung für die Dialoge bereit. Er besteht aus folgenden zwei Komponenten (Abb. 20.0-1, rechte Seite):

- Sitzungs-Komponente: Eine Sitzung ist dialogübergreifend und benutzerspezifisch.
- Anwendungs-Komponente: Eine Anwendung ist sitzungsübergreifend und benutzerunabhängig.

Die Anwendungs-Komponente

Der Lebenszyklus der Anwendungs-Komponente ist identisch mit dem Lebenszyklus des Subsystems Benutzeroberfläche. Die Anwendungs-Komponente erledigt folgende Aufgaben:

- Benutzungsunabhängige Konfiguration des Subsystems.
- Sicherstellung, dass globale Ressourcen oder Dienste zur Verfügung stehen.
- Erzeugung und Beendigung ihrer Sitzungen.

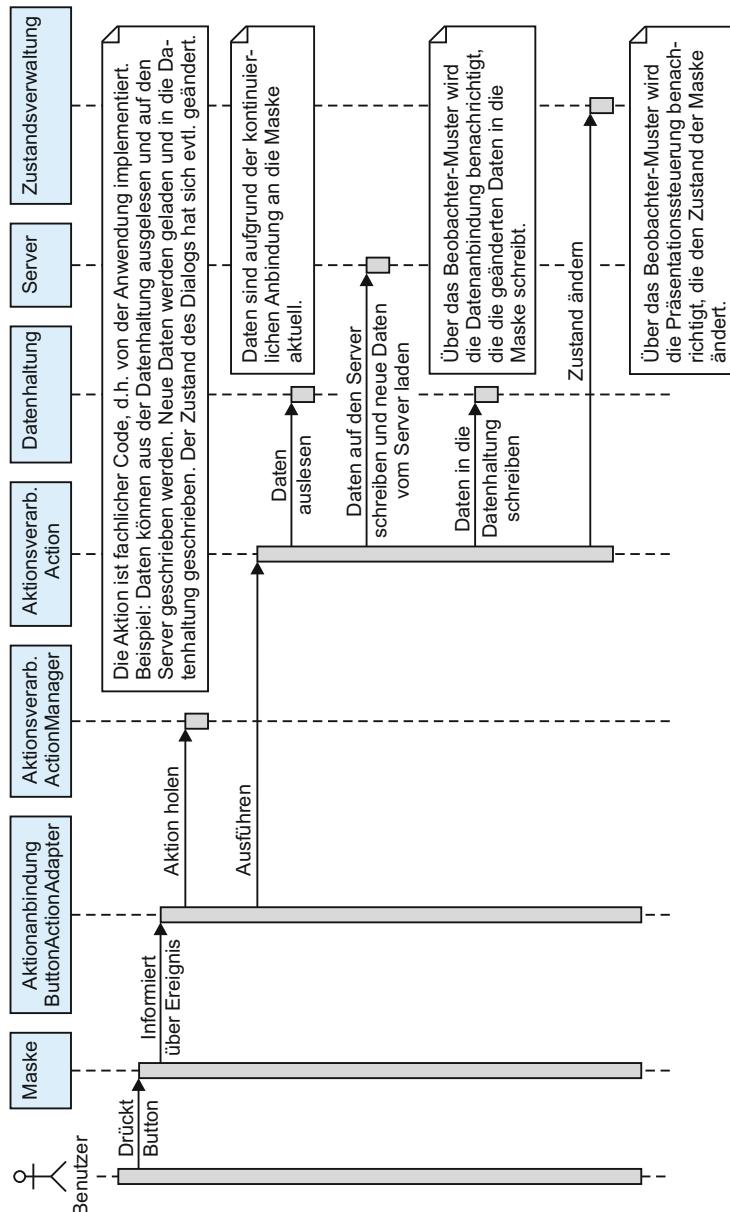
Die Sitzungs-Komponente

Die Sitzungs-Komponente erledigt folgende Aufgaben:

- Benutzerspezifische Interaktion.

I 20 Das Subsystem Benutzeroberfläche

Abb. 20.0-4:
Verarbeitung einer
Benutzeraktion.

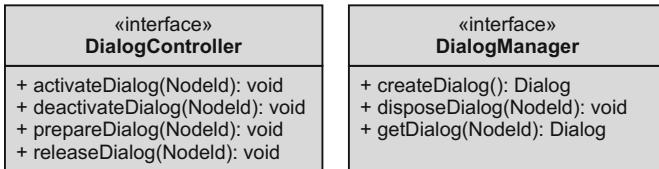


- An- und abmelden der Benutzer.
- Steuert die Lebenszyklen ihrer Dialoge.

Jede Sitzung wird in eine Dialogsteuerung, eine Hierarchieverwaltung und eine hierachische Dienstverwaltung untergliedert. Die Komponenten der Anwendung liegen an der Wurzel der Dialoghierarchie. Ihre Kinder sind die Sitzungs-Komponenten. Darunter liegen die Hauptdialoge, die Unterdialoge besitzen können.

20 Das Subsystem Benutzeroberfläche I

Die Dialogsteuerung ist für den Lebenszyklus der Dialoge zuständig. Sie erzeugt und zerstört die Dialogexemplare und organisiert sie in der Hierarchie. Die Abb. 20.0-5 zeigt eine mögliche Schnittstelle einer Dialogsteuerung, die Abb. 20.0-6 zeigt wie die Schnittstelle einer Hierarchieverwaltung aussehen kann.



Dialog-
steuerung

Abb. 20.0-5:
Mögliche
Schnittstelle einer
Dialogsteuerung.

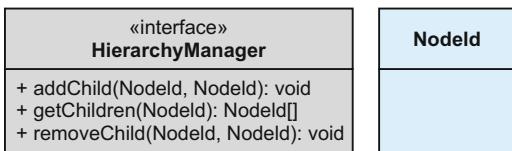


Abb. 20.0-6:
Mögliche
Schnittstelle einer
Hierarchieverwal-
tung.

Setzt sich ein Dialog aus mehreren Teildialogen zusammen, dann müssen die Darstellungen der Teildialoge integriert werden. Abhängigkeiten zwischen den einzelnen Teildialogen müssen durch einen Lebenszyklus spezifiziert werden (Abb. 20.0-7).

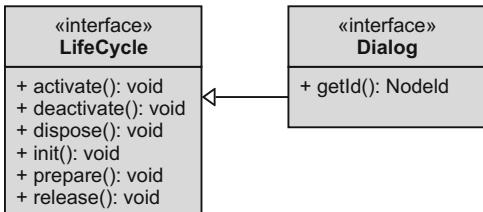


Abb. 20.0-7:
Mögliche
Lebenszyklus- und
Dialog-
Schnittstelle.

Die Abb. 20.0-8 zeigt beispielhaft das Zusammenspiel der Komponenten des Dialograhmens beim Deaktivieren eines Dialogs.

Hierarchisch organisierte Dienste ermöglichen die Kommunikation zwischen Dialogen untereinander und mit den Dialograhmen.

Dazu wird das Service-Locator-Muster verwendet, siehe Service-Locator-Muster (<http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html>).

Ein Dienst wird durch seine Schnittstelle definiert und durch Komponenten innerhalb der Dialoghierarchie, der Sitzung oder der Anwendung implementiert.

Kommunikation &
Dienste

I 20 Das Subsystem Benutzeroberfläche

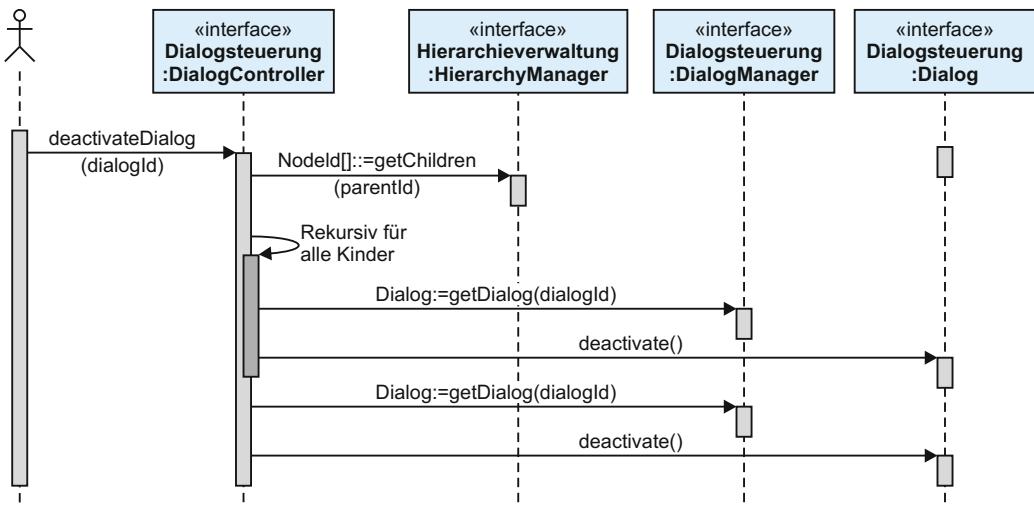


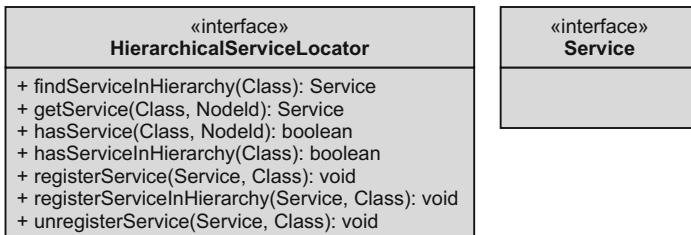
Abb. 20.0-8: Innerhalb des Subsystems Benutzeroberfläche können folgende Dienste bereitgestellt werden:
für das
Deaktivieren eines
Dialogs.

- Zugriff auf Benutzerdaten.
- Anzeigen einer Meldung in der Statuszeile eines übergeordneten Dialogs.
- Einbetten einer Menüoption in ein Menü.

Auf Dienste kann wie folgt zugegriffen werden:

- Beim hierarchischen Zugriff wird die erste Implementierung des Dienstes in der Hierarchie auf dem Weg zur Wurzel zurückgeliefert. Diese Zugriffe werden für die Kommunikation von einer Komponente zu ihrer Umgebung, d. h. ihren übergeordneten Komponenten, verwendet. Annahmen, die eine Komponente über ihre Umgebung trifft, werden kanalisiert und ohne starre Bindung behandelt (Abb. 20.0-9).

Abb. 20.0-9:
Mögliche
Schnittstelle eines
hierarchischen
ServiceLocators
der "Dienst-
verwaltung".



- Beim direkten Zugriff wird die Implementierung des Dienstes des gewünschten Knotens zurückgeliefert. Diese Zugriffe dienen zur Kommunikation mit (direkten) Kindern einer Komponente.
Eine solche Dienstarchitektur ermöglicht eine starke Entkopplung der Komponenten.

20.1 GUI-Entwurfsmuster MVP I

Das MVP-Muster ist eine Weiterentwicklung des MVC-Musters:

- »GUI-Entwurfsmuster MVP«, S. 457

Die Umsetzung dieser Architektur wird am Beispiel der Fallstudie »Kundenverwaltung« gezeigt:

- »Fallstudie: Kundenverwaltung – GUI«, S. 458

In .NET gibt es mehrere Möglichkeiten der GUI-Programmierung:

- »GUIs in der .NET-Plattform«, S. 467

20.1 GUI-Entwurfsmuster MVP

Name(n)

Das GUI-Entwurfsmuster **MVP** (*model view presenter pattern*) modifiziert das **MVC-Muster** (siehe »Das MVC-Muster (*model view controller pattern*)«, S. 62). Es trennt das Modell (*model*) von der Ansicht (*view*). Beide werden durch einen Präsentator (*presenter*) miteinander verbunden.

Grundidee

Im Gegensatz zum MVC-Muster ist die *view* im MVP-Muster auch für die Behandlung der Benutzerereignisse verantwortlich, wie *mouseDown*, *keyDown* usw. Im MVC-Muster ist dies die Aufgabe des *controllers*.

Anwendungsbereich(e)

Das MVP-Muster wird zur Architektur der »Benutzungsoberfläche« verwendet. Die .NET-Plattform unterstützt das Muster. In Java SE kann das Muster mithilfe des Biscotti-Frameworks implementiert werden. Die Frameworks Vaadin und Google Guice erlauben die Anwendung des Musters in Web-Architekturen.

Problem

Das Layout, die Implementierung und der Test der Benutzungsoberfläche soll vollständig unabhängig von der Anwendung erfolgen.

Lösung

Durch folgende Aufgabenteilung wird das Problem gelöst:

- **model**: Repräsentiert die Anwendungslogik, Geschäftslogik bzw. fachliche Logik. Es kennt weder die *view* noch den *presenter*.
- **view**: Zeigt die Daten (des *model*) an und gibt die Benutzerereignisse an den *presenter* weiter. Die *view* erzeugt ein *presenter*-Objekt und speichert dort eine Referenz auf sich selbst. Sie hat *keinen* Zugriff auf den *presenter* und das *model*.
- **presenter**: Kommuniziert mit den Schnittstellen des *model* und der *view*. Greift auf die Daten des *model* zu, persistiert sie und formatiert sie für die Anzeige durch die *view*. Steuert die *view*.

I 20 Das Subsystem Benutzungsoberfläche

- view* Die Funktionalität der *view* kann unterschiedlich ausgeprägt sein:
- Sie kann vollständig passiv sein und alle Benutzerinteraktionen an den *presenter* weiterleiten. Bei einem Benutzerereignis tut die *view* nichts außer einem Methodenaufruf (ohne Parameter und ohne Ergebniswert) beim *presenter*. Der *presenter* greift über Methoden, die die *view*-Schnittstelle zur Verfügung stellt, auf die Daten der *view* zu. Anschließend ruft der *presenter* Methoden auf dem *model* auf und aktualisiert die *view* mit den Ergebnissen dieser Methodenaufrufe.
 - Sie kann selbstständig Interaktionen, Ereignisse oder Kommandos verarbeiten. Dies ist oft bei Web-Architekturen der Fall. Die *view* wird im Browser ausgeführt und ist daher am besten geeignet, einzelne Interaktionen oder Kommandos zu bearbeiten.
- Struktur In einer Mehr-Schichten-Architektur kann der *presenter* zur Applikationsschicht gehören. Er kann aber auch eine eigene Schicht zwischen der Applikationsschicht und der Benutzungsoberflächen-Schicht bilden.
- Vorteile**
- Das MVP-Muster bringt folgende Vorteile mit sich:
- + Verbesserte Trennung der Zuständigkeiten.
 - + Erleichterung des automatischen Testens von Benutzungsoberflächen.
 - + *model* und *view* können ausgetauscht und wieder verwendet werden.
- Literatur [Leis11], [Fowl06]

20.2 Fallstudie: Kundenverwaltung – GUI

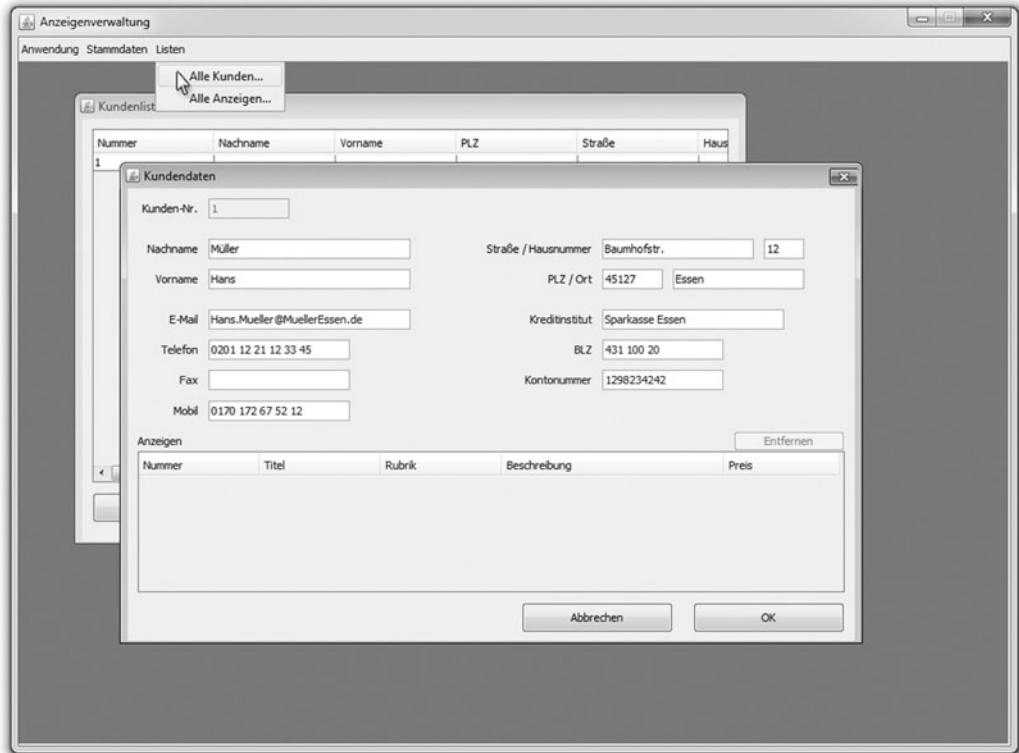
Die Benutzungsoberfläche der Fallstudie »Kundenverwaltung« (siehe »Fallstudie: KV – Überblick«, S. 15) wird im Folgenden in erweiterter Form als »Anzeigenverwaltung« mithilfe der in [HaOl07] vorgestellten Architektur realisiert (siehe »Das Subsystem Benutzungsoberfläche«, S. 449). Der Fokus liegt auf der GUI-Architektur. Die Anwendungslogik und die Persistierung der Objekte treten in den Hintergrund.

Der Aufbau der Benutzungsoberfläche

Die Benutzungsoberfläche besteht aus fünf verschiedenen Dialogen, um Kunden und Anzeigen zu verwalten (Abb. 20.2-1):

- HauptDialog: Wird initial angezeigt und besitzt ein Menü:
- Stammdaten: Das jeweilige Untermenü öffnet die Erfassungssicht für die entsprechende Klasse (Kunde, Anzeige).

20.2 Fallstudie: Kundenverwaltung – GUI I



- Listen: Das jeweilige Untermenü öffnet die entsprechende Listenansicht.
- KundenDialog: Editiermaske für Kunden.
- AnzeigenDialog: Editiermaske für Anzeigen.
- KundenListenDialog: Listensicht aller Kunden.
- AnzeigenListenDialog: Listensicht aller Anzeigen.

In den Listendialogen kann der Anwender mittels einer Tabelle die entsprechenden Elemente überblicken, löschen und die Editiermasken aufrufen.

Abb. 20.2-1:
Benutzeroberfläche der Fallstudie »Kundenverwaltung« in erweiterter Form als »Anzeigenverwaltung«.

Die Dialogarchitektur

Die Dialoge gliedern sich in zwei Schichten, nämlich die Schicht **Präsentation** und die Schicht **Dialogkern** (siehe »Das Subsystem Benutzungsoberfläche«, S. 449).

Erzeugung der Maske in der Schicht »Präsentation«

Die Erzeugung der Maske wird im Lebenszyklus des Dialogs in der Phase **prepare** durchgeführt. Über eine Fabrik können neue Steuerelemente erstellt werden, um anschließend deren Attributwerte (Breite, Höhe, Beschriftung usw.) anzupassen. Nach der Anpassung

I 20 Das Subsystem Benutzungsoberfläche

können die Steuerelemente auf dem zum Dialog zugehörigen Anzeigerafahmen (bspw. Fenster) platziert werden. Die typischen Schritte demonstriert folgendes Code-Beispiel, in dem ein Führungstext und eine Textbox zur Darstellung der Anzeigennummer erzeugt werden:

```
Label lblKundennummer = (Label) createComponent(Label.class);
lblKundennummer.initControl
(10, 10, 80, 20, getWindow(), "Anzeigen-Nr.");
lblKundennummer.setAlignment(Label.RIGHT);

Textbox textfeldAnzNummer =
(Textbox) createComponent(Textbox.class);
textfeldAnzNummer.initControl(100, 10, 80, 20, getWindow(), "");
textfeldAnzNummer.setEnabled(false);
```

Es ist hierbei zu beachten, dass die Methode `createComponent` die in der Sitzung hinterlegte Fabrik anspricht und eine kompatible Implementierung des gewünschten Steuerelements zurückgibt.

Entkopplung der Präsentation von Frameworks

Bei der Umsetzung der Architektur im Rahmen der Fallstudie wurde besonderer Wert auf die vollständige Entkopplung von konkreten *Frameworks* gelegt. Insbesondere das *Framework*, das für die Präsentation zuständig ist, soll austauschbar und für den Entwickler der Anwendung nicht von Belang sein.

Zwischenschicht

Um diese Trennung zu ermöglichen, wurde eine Zwischenschicht realisiert, die auf dem Muster der **Abstrakten Fabrik** basiert. Jedes konkrete Präsentations-*Framework* muss mittels einer solchen Fabrik und einer entsprechenden Verbindungsschicht an diese abstrakte Zwischenschicht angebunden werden.

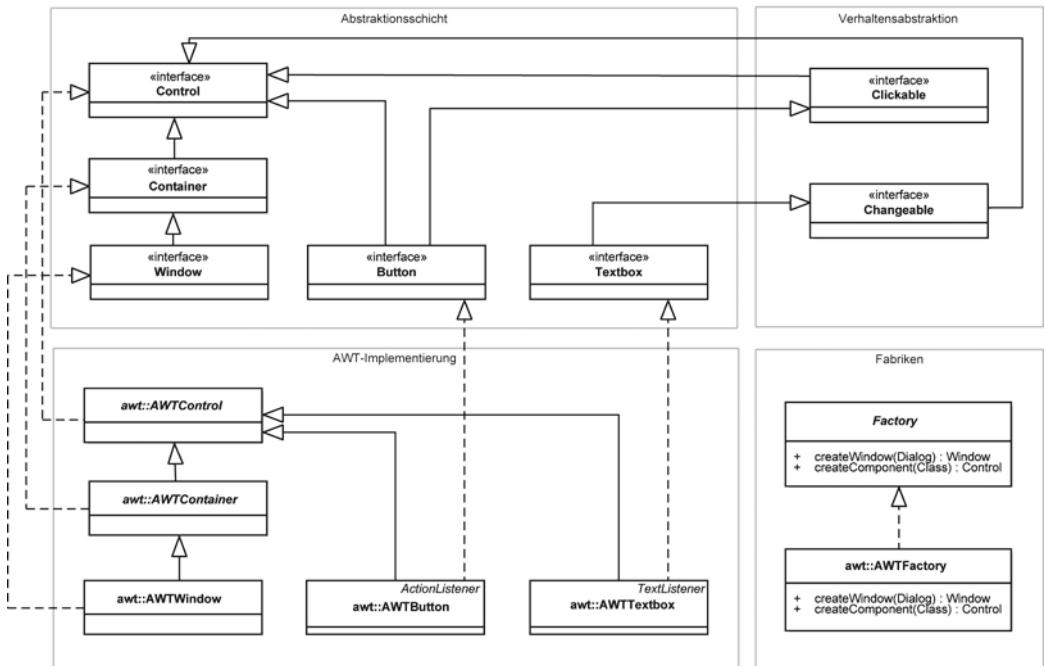
Ein Teil der abstrakten Zwischenschicht und die entsprechenden Klassen einer exemplarischen Verbindungsschicht für das AWT-*Framework* werden in der Abb. 20.2-2 präsentiert. Hierbei ist zu erkennen, dass die Steuerelemente verschiedene Schnittstellen realisieren. Beispielsweise ist eine konkrete Schaltfläche des AWT-Frameworks (`AWTButton`) gleichzeitig ein `Control` (Steuerelement), ein `Button` (Schaltfläche) und `Clickable` (Klickbar). Um Steuerelemente ineinander zu schachteln bieten sich Steuerelemente an, die die Schnittstelle `Container` realisieren.

Im implementierten Fallbeispiel wurden zwei Fabriken exemplarisch realisiert.

Für die Darstellung der Anwendung als Desktop-Anwendung wurde eine Zwischenschicht für das Swing-Framework implementiert.

Für die Darstellung als Web-Anwendung, welche einen eigenen Ansatz mittels AJAX/HTML verfolgt, wurde eine zweite Verbindungs geschicht samt Fabrik eingesetzt. Hierbei wurde versucht, eine möglichst ähnliche visuelle Darstellung zu erreichen.

20.2 Fallstudie: Kundenverwaltung – GUI I



Der eigentliche Code der entwickelten Anwendung ist somit bei der Web- und der Desktop-Anwendung nahezu identisch und unterscheidet sich nur an wenigen Stellen. Vor allem der Ort, an dem die Sitzung erzeugt wird, und die zu verwendende Fabrik differieren hier.

Datenanbindung

Nachdem die Maske erzeugt worden ist, muss eine Anbindung der Präsentation an den Dialogkern – und somit indirekt auch an das Fachkonzept der Anwendung – realisiert werden. Die Datenanbindung geschieht zwischen einem oder mehreren Steuerelementen und dem entsprechenden Attribut bzw. Dateneintrag. Jeder Dialog besitzt einen DataManager, in welchem hierzu Daten unter fachlichen Schlüsseln abgelegt werden können.

Mittels der Datenanbindung können anschließend Steuerelemente mit Attributen bzw. Einträgen aus dem DataManager verknüpft werden, so dass sie gegenseitig und automatisch synchronisiert werden. Es handelt sich hierbei um reinen Konfigurationscode, so dass kein weiterer Code zur Ereignisbehandlung eingesetzt werden muss.

Im folgenden Beispiel wird bei der Initialisierung des Dialogs zur Bearbeitung der Anzeigen (AnzeigeDialog) ein Textfeld erzeugt, ein Attribut des Fachkonzepts in den DataManager überführt und anschließend der neu erzeugte Eintrag mit dem Steuerelement verknüpft (siehe auch »Das Subsystem Benutzungsoberfläche«, S. 449):

Abb. 20.2-2:
Realisierung der
Abstraktions-
schicht
(Exemplarisch für
AWT).

I 20 Das Subsystem Benutzeroberfläche

```
getDataManager().setValue(meineAnzeige.getPreis(),
    "anzeige.preis");
Textbox textfeldAnzPreis =
    (Textbox)createComponent(Textbox.class);
textfeldAnzPreis.initControl(180, 10, 100, 20, getWindow(), "");
new IntegerTextboxAdapter(getDataManager(), "anzeige.preis",
    textfeldAnzPreis);
```

Die Klasse `IntegerTextboxAdapter` übernimmt gleichzeitig auch eine konvertierende Funktion, indem sie den eingegebenen Wert eines Textfelds vom Typ `String` automatisch in einen `Integer` konvertiert.

Klickt der Anwender auf eine Schaltfläche zur Speicherung der Daten, so müssen die Einträge aus dem `DataManager` in das entsprechende Objekt des Fachkonzepts überführt werden, welches dann beispielsweise persistiert werden kann.

Aktionsverarbeitung im Dialogkern

Bestimmte Steuerelemente, wie z. B. Schaltflächen oder Menüeinträge, werden in einem Dialogfenster häufig dazu verwendet, eine fachliche Aktion auszulösen. Beispiele für eine fachliche Aktion sind Neue Anzeige erstellen, Kunden löschen oder Daten speichern. Die Maske des Dialogs `AnzeigenListenDialog` (siehe Abb. 20.2-3) beinhaltet eine Tabelle, die eine Übersicht über alle Anzeigen bietet, sowie u. a. die Schaltfläche Bearbeiten.

Die Schaltfläche Bearbeiten soll die Aktion Anzeige bearbeiten auslösen, wenn sie geklickt wird. Diese Aktion soll jedoch nur dann ausführbar sein, wenn vorher in der Tabelle eine Zeile ausgewählt wurde. Für die Realisierung dieser Funktionalität müssen zuerst die Steuerelemente erstellt und initialisiert werden:

```
sfBearbeiten = (Button)createComponent(Button.class);
sfBearbeiten.initControl(330, 370, 150, 30, getWindow(),
    "Bearbeiten");

tabelle = (Table)createComponent(Table.class);
tabelle.initControl(10, 10, 630, 350, getWindow(), "");
```

Die Tabelle bietet eine Schnittstelle zur Ereignisbehandlung des Ereignisses Zeile gewählt (`onItemSelected`) an, an dem sich der Dialog registrieren muss.

```
tabelle.onItemSelected().addListener(this);
```

Im nächsten Schritt kann eine Aktion an das Click-Ereignis der Schaltfläche gebunden werden, wofür der Dialog eine Hilfsmethode `bindClickEvent` anbietet:

```
protected Action bindClickEvent(Clickable pControl,
    String pKeyAction, String pMethodName, Object pParam)
{
```

20.2 Fallstudie: Kundenverwaltung - GUI I



```
Action action = getActionManager().getAction(pKeyAction);
if (action== null)
{
    action = new MethodAction(this, pMethodName, pParam);
    getActionManager() .addAction(action, pKeyAction);
}

ClickActionAdapter clickAdapt =
    new ClickActionAdapter(getActionManager(),
    pKeyAction, pControl);
return action;
}
```

Diese Methode erhält das Steuerelement, einen eindeutigen Schlüssel für die Aktion, eine auszuführende Methode und einen zusätzlichen Parameter als Eingabe. Es wird zuerst geprüft, ob eine Aktion mit benanntem Schlüssel im ActionManager des Dialogs existiert. Falls dies nicht der Fall ist, so wird die Aktion angelegt und beim Aktionsmanager des Dialogs registriert. Es wird ein Objekt vom Typ MethodAction als konkrete Implementierung der Schnittstelle Action verwendet. Diese Implementierung führt bei ihrer Ausführung eine angegebene Methode aus.

Die Aktion und das Steuerelement müssen anschließend noch miteinander assoziiert werden. Dies erfolgt über einen ClickActionAdapter. Dieser reagiert auf das Click-Ereignis des Steu-

Abb. 20.2-3:
Darstellung der
Anzeigenliste.

I 20 Das Subsystem Benutzungsoberfläche

erelements und führt die Aktion aus. Der Adapter sorgt gleichzeitig aber auch noch dafür, dass das mit der Aktion verbundene Steuerelement deaktiviert wird, wenn die Aktion selbst nicht ausführbar ist.

Wenn mehrere Steuerelemente auf diese Weise an eine Aktion gebunden werden, beispielsweise eine Werkzeugleisten-Schaltfläche, ein Menüeintrag und eine gewöhnliche Schaltfläche, so reicht es später aus, alleine die Aktion zu aktivieren/deaktivieren. Die Steuerelemente werden dann automatisch synchronisiert.

Der konkrete Aufruf der Methode sieht in diesem Falle wie folgt aus:

```
bindClickEvent(sfBearbeiten, "bearbeiten", "bearbeiteAnzeige").  
    setExecutable(false);
```

Hierbei wird die Aktion deaktiviert, da bei der Anzeige des Dialogs noch kein Eintrag in der Tabelle gewählt ist. Die Deaktivierung sorgt dafür, dass die Schaltfläche ebenfalls deaktiviert wird. Um die Aktion ausführbar und somit die Schaltfläche zu aktivieren, muss nun noch überwacht werden, wenn ein Element der Tabelle geklickt wird. Dies könnte über einen speziellen Adapter erfolgen oder über die direkte Verarbeitung des Ereignisses:

```
@Override  
public void onItemSelected(Table sender)  
{  
    getActionManager().getAction("bearbeiten").  
        setExecutable(sender.getSelectedItem() != -1);  
}
```

Die Aktion wird über ihren Schlüssel referenziert (bearbeiten) und in Abhängigkeit, ob ein Element in der Tabelle gewählt worden ist, aktiviert.

Der Dialograhmen

Der Dialograhmen stellt die äußere Ablaufumgebung für die einzelnen Dialoge dar. Beim Start der Anwendung oder bei der Verbindung eines neuen Benutzers muss ein neuer Dialograhmen initialisiert werden. Soll eine Kommunikation zwischen den Dialogen erfolgen oder sollen neue Dialoge angezeigt werden, so muss mit den Komponenten des Dialograhmens kommuniziert werden.

Aufbau des Dialograhmens

Desktop-Anwendung Ein Desktop-Client wird in der Regel über eine ausführbare Datei gestartet (exe, jar). Bei einer Java-Anwendung wird bei der Ausführung die main-Methode aufgerufen und erstellt ein Objekt vom Typ Application und eine entsprechende Sitzung (Session). Ferner ist hierbei zu beachten, dass eine zum Anwendungstyp (Desktop-Anwendung) passende Fabrik an die Sitzung gebunden wird.

20.2 Fallstudie: Kundenverwaltung – GUI I

```
public class SwingAnzVerwRunner
{
    public static void main(String[] args)
    {
        AnzVerwApplication app = new AnzVerwApplication();
        Session s = app.createSession(new SwingFactory());
        createHauptDialog(s);
    }
    ...
}
```

Für Web-Anwendungen bietet es sich an, diese Initialisierung aufzuteilen. Zuerst kann beim Start der Web-Anwendung auf dem Server das entsprechende Application-Objekt erstellt werden. Eine Session sollte anschließend erst dann geöffnet werden, wenn tatsächlich ein Webbrowser auf die Anwendung bzw. den Server zugreift. Dies kann zum Beispiel über ein Servlet oder einen SessionListener realisiert werden.

```
HashMap<String, Session> mapping =
    new HashMap<String, Session>();

protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
{
    String sessionid = request.getSession().getId();

    Session s = mapping.get(sessionid);

    if (s != null)
    {
        s= myApp.createSession(new WebFactory());
        mapping.put(sessionid, s);
        createHauptDialog(s);
    }
    ServletContext sc=this.getServletContext();
    RequestDispatcher r =
        sc.getRequestDispatcher("/desktop.htm");
    r.include(request,response);
}
```

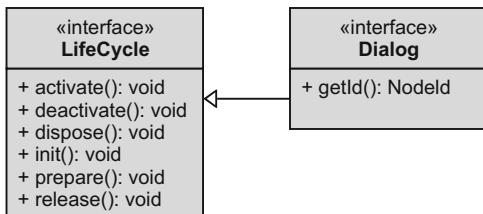
Nach dem Ausführen der frameworkspezifischen Initialisierung verwenden beide Anwendungsarten (Web und Desktop) den gleichen Code.

Anzeige eines neuen Dialogs

Für die Anzeige eines neuen Dialogs müssen bestimmte Schritte durchgeführt werden. Hierbei muss vor allem auf den Lebenszyklus (Abb. 20.2-4) des Dialogs geachtet werden:

I 20 Das Subsystem Benutzeroberfläche

Abb. 20.2-4:
Mögliche
Lebenszyklus- und
Dialog-
Schnittstelle.



- 1 Es muss ein neues Objekt der entsprechenden Dialogklasse erzeugt werden. Hierbei sollte bis auf die Übergabe von Parametern noch keine weitere Initialisierung, beispielsweise GUI-Erzeugung oder Kommunikation mit anderen Dialogen, erfolgen.
- 2 Dem DialogManager der Sitzung muss mitgeteilt werden, dass ein neuer Dialog erstellt worden ist. Der Dialog wird somit in der Dialoghierarchie verankert und kann von anderen Dialogen angesprochen werden.
- 3 Der Dialog erhält vom DialogManager eine Nachricht zur Initialisierung.
- 4 Der DialogController muss darüber informiert werden, dass ein neuer Dialog seine Maske vorbereiten soll. Der DialogController sendet hierzu anschließend eine Nachricht an den Dialog.
- 5 Zuletzt wird der Dialog aktiviert und zeigt seine Maske an.
Mittels einer Hilfsmethode können diese Schritte auch zusammengefasst werden:

```
public void showDialog(Dialog d, Session s)
{
    s.getDialogManager().registerDialog(d);
    //Sendet Botschaft init() an den Dialog
    s.getDialogController().prepareDialog(d.getId())
    //Sendet Botschaft prepare() an den Dialog
    s.getDialogController().activateDialog(d.getId())
    //Sendet Botschaft activate() an den Dialog
    //Dialog ist nun sichtbar
}
```

Falls der Dialog geschlossen wird, müssen entsprechende Schritte zur Deinitialisierung durchgeführt werden (deactivateDialog, releaseDialog, disposeDialog).

OOP Das vollständige Programm finden Sie im kostenlosen E-Learning-Kurs zu diesem Buch.

 Laden Sie das Programm auf Ihr Computersystem und führen Sie es aus. Sehen Sie sich die einzelnen Klassen im Detail an.

20.3 GUIs in der .NET-Plattform

.NET bietet vier verschiedene Oberflächentechniken für grafische Benutzungsoberflächen:

- ASP.NET + JavaScript/AJAX
- Windows Forms
- Windows Presentation Foundation (WPF)
- Silverlight

Die Tab. 20.3-1 zeigt die Einsatzgebiete der Techniken.

| | Win-dows-Desktop | Unix / Linux / Mac OS-Desktop | Browser (Sandbox) | Browser (volle Rechte) | Microsoft Office-Anwendungen | Windows-basierte Telefone |
|----------------------------|--------------------|-------------------------------|--------------------------------------|--------------------------------------|--|--------------------------------|
| ASP.NET + JavaS-cript/AJAX | Nein | Nein | Ja, alle Browser | Nein | Nein | Ja, im Browser des Mobilgeräts |
| Windows Forms | Ja | Ja | Ja, nur IE, nur bis .NET 1.0 bis 3.5 | Ja, nur IE, nur bis .NET 1.0 bis 3.5 | Ja | Ja, bis Windows Mobile 6.x |
| WPF | Ja | In Arbeit | Ja, nur IE & Firefox | Ja, optional, nur IE & Firefox | Nur indirekt über Integration zwischen WPF & Windows Forms | Nein |
| Silver-light | Ja, ab Version 4.0 | teilweise | Ja, viele Browser | Ja, ab Version 5.0 | Nein | Ja, ab Windows Phone 7 |

Tab. 20.3-1: Anwendungsbereiche der vier .NET-GUI-Techniken.

ASP.NET

ASP.NET ist seit dem .NET Framework 1.0 eine der Stärken von .NET. ASP steht für *Active Server Pages* und drückt aus, dass es sich (primär) um ein serverseitiges *Framework* für Webanwendungen handelt. Allerdings war ASP.NET von Anfang an nicht rein serverseitig, denn einige Funktionen (z. B. Eingabeprüfungen und Hyperlinks, die sich verhalten wie Schaltflächen) erforderten die Ausführung von JavaScript im Browser. Entscheidend ist aber, dass das .NET Framework nur auf dem Server installiert sein muss. Als Client wird nur ein Webbrowswer benötigt, der HTML, CSS und JavaScript versteht. Bei deaktiviertem JavaScript können einige ASP.NET-Funktionen *nicht* korrekt funktionieren, aber im Kern kann man ASP.NET auch ohne JavaScript verwenden. Hinsichtlich der Browser-Kompatibilität hat

I 20 Das Subsystem Benutzeroberfläche

sich ASP.NET über die .NET-Versionen hinweg gebessert, sodass die meisten Funktionen inzwischen auch kompatibel zu anderen Browsern sind.

ASP.NET bietet eine signifikante Abstraktion von den Eigenarten der Webprogrammierung. Entwickler arbeiten mit Steuerelementen, wie sie dies aus der Entwicklung von Desktop-Anwendungen kennen. Diese Steuerelemente, die sich »Webserversteuerelemente« nennen, erzeugen dann zur Ausführungszeit auf dem Webserver in einem *Rendering* genannten Vorgang eine von dem jeweiligen Browser verstandene Markup-Sprache. Im Standard sind dies HTML, CSS und JavaScript. Der Webserver sendet das Ergebnis des *Rendering* zum Browser.

In versteckten Feldern wird die Webseite automatisch durch Zustandsinformationen angereichert, die die Zustandslosigkeit von HTTP aus der Sicht des Entwicklers zum Teil transparent macht. Die Umsetzung der Steuerelemente in die vom Browser verwertbare Seitenbeschreibungssprache kann browserspezifisch erfolgen (*Adaptive Rendering*), um Unterschiede in der Interpretation zwischen den Browsern auszugleichen. Die Oberflächenbeschreibung erfolgt in ASPX-Dateien, die neben HTML, CSS und JavaScript auch XML-Tags enthalten, die die Webserversteuerelemente repräsentieren. Im Rahmen des *Rendering* ersetzt das ASP.NET Page Framework diese XML-Tags durch HTML, CSS und JavaScript, sodass diese XML-Tags nicht zum Browser gelangen. Die Abb. 20.3-1 veranschaulicht das *Adaptive Rendering*.

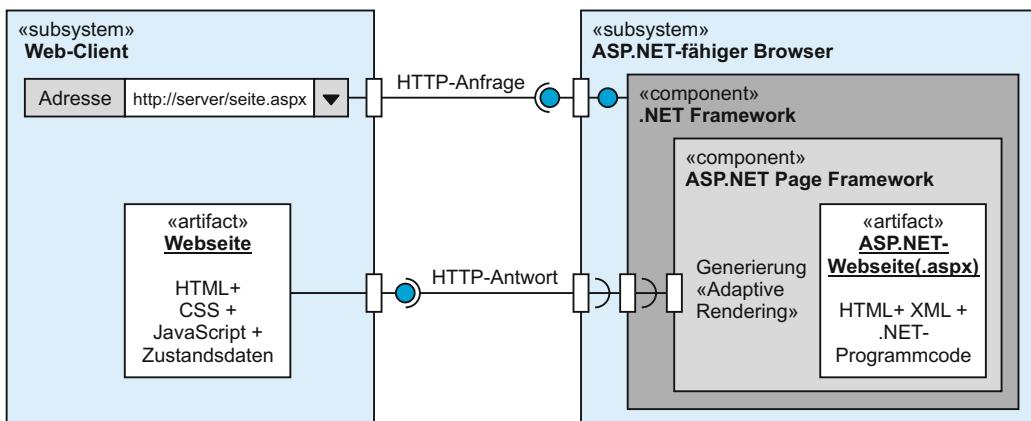


Abb. 20.3-1: Web-Architektur unter Einsatz von ASP.NET.

Der folgende Programmcode zeigt Ausschnitte aus der Erstellung einer Webseite zu einer Flugbuchung mit ASP.NET. Das Ergebnis zeigt die Abb. 20.3-2.

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Buchung.aspx.cs" Inherits="Buchung"%>
```

20.3 GUIs in der .NET-Plattform I

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>ASP.NET-Beispiel: Flugbuchung</title>
    <!-- CSS-Stylesheets einbinden: externe Datei &
    Inline-CSS--%>

    <link href=".\\WwWings.css" rel="stylesheet"
 type="text/css"/>
    <style type="text/css">
        .Tabelle
        {
            width: 100%;
            background-color: #87CEFA;
        }
        .Spalte1
        {
            background-color: #B0E2FF;
        }
        .Spalte2
        {
            background-color: #B0D2FF;
        }
    </style>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            
            <br />
            <br />
            <span class="style1">Flugbuchung<br />
                <br />
                <br />
            </span>
            <span class="style2">
                Hier können Sie einen Flug bei uns buchen.<br />
                <br />
                <p>
                    <br />
                    <asp:ScriptManager ID="ScriptManager1"
                        runat="server">
                    </asp:ScriptManager>
                </p>
                <asp:Panel ID="Panel1" runat="server"
                    BorderStyle="None" BorderWidth="1px"
                    GroupingText="Passagier">
                    &nbsp;
                    <br />
                    <table class="Tabelle">
                        <tr>
                            <td class="Spalte1">
                                Passagier-ID
                            </td>
```

I 20 Das Subsystem Benutzungsoberfläche

```
<td class="Spalte2">
    <asp:TextBox ID="C_PassagierID" runat="server"
        AutoPostBack="True"
        OnTextChanged="C_PassagierSuchen_Click">
    </asp:TextBox>
    <asp:Button ID="C_PassagierSuchen"
        runat="server" Text="Suchen"
        OnClick="C_PassagierSuchen_Click" />
</td>
</tr>
<tr>
    <td class="Spalte1">
        Name
    </td>
    <td class="Spalte2">
        <asp:UpdatePanel ID="P_Passagier"
            runat="server"
            UpdateMode="Conditional">
            <ContentTemplate>
                <asp:Label ID="C_Name" runat="server">
                    </asp:Label>
            </ContentTemplate>
            <Triggers>
                <asp:AsyncPostBackTrigger
                    ControlID="C_PassagierSuchen"
                    EventName="Click" />
                <asp:AsyncPostBackTrigger
                    ControlID="C_PassagierID"
                    EventName="TextChanged" />
            </Triggers>
        </asp:UpdatePanel>
    </td>
</tr>
</table>
<br />
</asp:Panel>
<p>
...
<br />
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate>
        <asp:Button ID="C_Buchen" runat="server"
            OnClick="C_Buchen_Click"
            Text="Buchen" Enabled="False" />
    </ContentTemplate>
</asp:UpdatePanel>
<br />
</asp:Panel>
</div>
</form>
</body>
</html>
```

20.3 GUIs in der .NET-Plattform I



Abb. 20.3-2:
Darstellung der
ASP.NET-Webseite
im Browser.

Innerhalb der Webseite kommt auch AJAX zum Einsatz. Hiermit wird erreicht, dass bei einem Klick des Benutzers auf einen der Suchen-Schaltflächen nicht die ganze Webseite neu vom Webserver geladen werden muss, sondern nur der jeweils relevante Bereich (Kasten Passagier oder Kasten Flug). Die neu zu ladenden Bereiche definiert der Entwickler in der ASPX-Seite durch das Webserversteuer-Element `<asp:UpdatePanel>`. Mit Triggern legt er fest, unter welchen Bedingungen dieser Teil der Seite neu beim Webserver angefordert werden soll.

Zu einer ASPX-Datei gehört jeweils auch eine Programmcode-Datei in einer .NET-Sprache, in der das Verhalten der Websteuerelemente implementiert ist. Der folgende Programmcode zeigt das Verhalten bei Klick auf die Passagiersuche und die Schaltfläche Buchen. Im Sinne einer Mehrschichtenarchitektur erfolgt hier kein direkter Zugriff auf die Datenbank, sondern es werden Klassen der Geschäftslogikschicht verwendet. Aufgrund der Zustandslosigkeit von HTTP ist es notwendig, die erfassten Daten in Session-Variablen zu speichern, die beim nächsten Seitenaufruf auf Basis eines Sitzungs-Cookies wiederhergestellt werden.

I 20 Das Subsystem Benutzeroberfläche

```
Beispiel 1b using System; // Einbinden der .NET-System-Bibliothek
using WWings_GO; //Einbinden der Geschäftsobjekte

//Einbinden der selbstentwickelten Geschäftslogik
using WWings_GL;

public partial class Buchung : System.Web.UI.Page
{
    Passagier Passagier;
    Flug Flug;

    protected void C_PassagierSuchen_Click(
        object sender, EventArgs e)
    {
        //Zustand zurücksetzen!
        Passagier = null;
        int PID;
        if (Int32.TryParse(this.C_PassagierID.Text, out PID))
        {
            PassagierBLManager PM = new PassagierBLManager();
            Passagier = PM.GetPassagier(PID);
            Session["Passagier"] = Passagier;
            if (Passagier != null)
            {
                this.C_Name.Text = Passagier.GanzerName;
            }
        }
        RefreshDisplay();
    }

    ...
    protected void C_Buchen_Click(object sender, EventArgs e)
    {
        Flug = (Flug)Session["Flug"];
        Passagier = (Passagier)Session["Passagier"];

        BuchungsBLManager BM = new BuchungsBLManager();
        string Buchungscode =
            BM.NewBuchung(Flug.FlugNr, Passagier.PersonID);
        //JavaScript erzeugen für Dialogfenster
        ClientScript.RegisterStartupScript(typeof(Buchung),
            "Start", "<script>alert('Ihr Buchungscode = "
            + Buchungscode + "'");</script>");
    }
}
```

Windows Forms vs. WPF

Während Java-Entwickler schon lange die Wahl zwischen verschiedenen GUI-Bibliotheken (z. B. AWT, SWT, Swing) haben, war die Welt für .NET-Entwickler in .NET 1.0 und 2.0 noch sehr einfach, denn hier gab es für Windows-Desktop-Anwendungen nur die Windows Forms.

Mit der Windows Presentation Foundation (WPF) hat Microsoft in .NET 3.0 eine alternative Bibliothek eingebracht, die wesentlich mehr grafische Möglichkeiten bietet und nicht nur, aber auch Oberflächen für Windows-Anwendungen bereitstellt. WPF unterstützt verschiedene Arten von Benutzungsoberflächen in einer durchgängigen Bibliothek, insbesondere:

- Klassische Desktop-Fenster
- 2-D-Grafiken (vgl. GDI)
- 3-D-Grafiken (vgl. DirectX)
- Dokumente (vgl. Postscript und PDF)
- Browser-basierte Anwendungen (vgl. Macromedia Flash und Java Applets)
- Animationen und Videos

WPF bietet nicht nur mehr Anwendungsarten, sondern viel reichhaltigere Visualisierungsmöglichkeiten. Dazu gehören Farbübergänge, Verformungen (Transformationen), Bildveränderungen (*Pixel Shader*) wie z. B. Schatten- und Spiegeleffekte, Überblendeffekte und Bewegungen (Animation). So kann WPF jedes beliebige Element in jedem beliebigen Winkel kippen oder rotieren. Elemente lassen sich beliebig kombinieren, z. B. kann ein Kontrollkästchen Teil einer Auswahlfeldes sein oder ein Video Hintergrund für ein Eingabefeld. Die Anzeige in WPF ist vektorbasiert und bietet daher eine gute Darstellung unabhängig von der Größe des Anzeigegeräts, auch bei sehr kleinen Bildschirmen. Durch die Definition wiederverwendbarer Formatvorlagen (*Styles*) lassen sich einheitliche Gestaltungsmerkmale auf visuelle Elemente anwenden. Durch den Austausch der Formatvorlagen sind schnell gestalterische Anpassungen möglich – wie man es von CSS im Web kennt. WPF bietet gegenüber Windows Forms ein wesentlich ausgeprägteres Ereignissystem und eine Abstraktion bei der Bindung von Oberflächenelementen an Befehle, durch das im aktuellen Kontext nicht verfügbare Elemente sofort ausgeblendet werden.

Im Gegensatz zu Windows Forms kann bei WPF die Beschreibung der Oberfläche durch XML-Elemente erfolgen. Microsoft hat dafür eine XML-Sprache mit Namen *Extensible Application Markup Language* (XAML) entwickelt. XAML ist eine XML-basierte Sprache mit der einzelne .NET-Objekte und ganze Objektbäume in XML-Form ausdrückbar sind. Zwar kann man in WPF eine Benutzungsoberfläche auch noch in Form von Programmcode erstellen, dies ist aber nicht der bevorzugte Weg, denn durch XAML ist eine Trennung von Code und Gestaltung einfacher. In Windows Forms wurde die Benutzungsoberfläche durch Code beschrieben, der – auch wenn in einer eigenen »Designer-Datei« getrennt – doch sehr eng mit dem Programmcode verzahnt war. In WPF kann man durch XAML viele Dinge deklarativ ausdrücken (z. B. Datenbindung, Animationen), die in Windows Forms gar nicht in der Designer-Datei untergebracht werden können.

I 20 Das Subsystem Benutzeroberfläche

ten. Für Windows Forms gab es als einziges Gestaltungswerkzeug den in Visual Studio eingebauten Designer. Für WPF gibt es eigene Produkte für die Oberflächengestalter, z. B. Microsoft Expression Blend. Dadurch können Benutzeroberflächen von Anwendungen einfacher von ausgebildeten Gestaltern erstellt werden.

Nachteile

Es gibt einige Kritikpunkte an WPF:

- In den ersten Versionen von WPF fehlten typische Steuerelemente für Geschäftsprozessanwendungen, z. B. zur Darstellung von Daten in Tabellen (*Datagrid*) oder ein Auswahlfeld für Datum/Uhrzeit (*Date-Time-Picker*). WPF war in den ersten Versionen nur auf grafische Spielereien gerichtet. Erst in .NET 4.0 hat sich dies gebessert.
- Microsoft liefert immer noch keine adäquaten grafischen Werkzeuge zur Erstellung von WPF-Oberflächen für Softwareentwickler. Das Werkzeug *Expression Blend* richtet sich an Grafikdesigner. Die Unterstützung in der Entwicklungsumgebung Visual Studio ist auch in Version 2010 unzureichend.
- WPF läuft nur auf modernen Windows-Betriebssystemen ab Windows XP. WPF ist für Unix/Linux/Mac OS noch nicht verfügbar.
- WPF braucht bei der Hardware unbedingt eine moderne Grafikkarte mit Unterstützung ab Direct3D Version 9. Außerdem sollte man deutlich mehr RAM haben als Microsoft in den Mindestanforderungen des jeweiligen Betriebssystems angibt. WPF läuft in Leistungsprobleme, wenn man viele Elemente in einer Oberfläche verwendet, selbst wenn diese gerade gar nicht sichtbar sind.
- Der RAM-Hunger wird gerade beim Einsatz in *Terminal Services* über das *Remote Desktop Protocol* (RDP) zum Problem. Auch verursachen die grafischen Effekte von WPF deutlich mehr Netzwerkklast, was sich an einem Beispiel verdeutlichen lässt: Ein Farbverlauf lässt sich bei der Übertragung so optimieren wie eine einfarbige Fläche. Erst ab Windows 7 und Windows Server 2008 R2 jeweils mit Service Pack 1 ist dies entschärft durch eine Erweiterung von RDP mit Namen RemoteFX.
- Der Lernaufwand für WPF ist hoch im Vergleich zu Windows Forms.

Beispiel Das folgende Listing zeigt ein WPF-Beispiel in XAML und die Abb. 20.3-3 die zugehörige Ausgabe.

```
<Window x:Class="WWings_WPF.F_Buchungsmaske_Webservice"
    xmlns="http://schemas.microsoft.com/winfx
    /2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Buchungsmaske" Height="472" Width="432"
    Icon="/WWings_WPF;component/Images/WWings.ico">
    <Canvas>
        <!--Passagier-Sektion-->
        <Rectangle Canvas.Left="12" Canvas.Top="23" Height="114"
```

```

        Name="rectangle1" Stroke="Black" Width="386" />
<Rectangle Canvas.Left="22" Canvas.Top="12"
    Fill="#FFF6FAD2" Height="24" Name="rectangle4"
    Stroke="Black" Width="80" />
<Label Canvas.Left="25" Canvas.Top="10"
    Content="Passagier" Height="28" Name="label9" />
<TextBox Name="C_PassagierID" Canvas.Left="138"
    Canvas.Top="49" Height="23" Width="120" />
<TextBox Name="C_Name" Canvas.Left="138"
    Canvas.Top="86" Height="23" Width="237" />
<Label Canvas.Left="25" Canvas.Top="44" Content=
"Passagier-ID" Height="28" Name="label1" />
<Label Canvas.Left="25" Canvas.Top="86"
    Content="Name" Height="28" Name="label2" />

<!--Flug-Sektion-->
<Rectangle Canvas.Left="12" Canvas.Top="171" Height="208"
    Name="rectangle2" Stroke="Black" Width="386" />
<Rectangle Canvas.Left="23" Canvas.Top="160" Height="24"
    Name="rectangle3" Stroke="Black" Width="79"
    Fill="#FFF6FAD2" />
<Label Canvas.Left="26" Canvas.Top="158" Content="Flug"
    Height="28" Name="label8" />
<TextBox Name="C_FlugNr" Canvas.Left="138"
    Canvas.Top="196" Height="23" Width="120" />
<TextBox Name="C_Abflugort" Canvas.Left="138"
    Canvas.Top="232" Height="23" Width="237" />
<Label Canvas.Left="25" Canvas.Top="193"
    Content="Flugnummer" Height="28" Name="label3" />
<Label Canvas.Left="25" Canvas.Top="232"
    Content="Abflugort" Height="28" Name="label4" />
<TextBox Name="C_Zielort" Canvas.Left="138"
    Canvas.Top="268" Height="23" Width="237" />
<Label Canvas.Left="25" Canvas.Top="268"
    Content="Zielort" Height="28" Name="label5" />
<TextBox Name="C_Datum" Canvas.Left="138"
    Canvas.Top="302" Height="23" Width="237" />
<Label Canvas.Left="25" Canvas.Top="302"
    Content="Datum" Height="28" Name="label6" />
<TextBox Name="C_FreiePlaetze" Canvas.Left="138"
    Canvas.Top="331" Height="23" Width="237" />
<Label Canvas.Left="25" Canvas.Top="331"
    Content="Freie Plätze" Height="28" Name="label7" />
<Button Canvas.Left="273" Canvas.Top="195"
    Content="Flug suchen" Height="23" Name="button3"
    Width="102" />
<Button Canvas.Left="323" Canvas.Top="398"
    Content="Abbrechen" Height="23" Name="button1"
    Width="75" />
<Button Canvas.Left="242" Canvas.Top="398"
    Content="Buchen" Height="23" Name="button2"
    Width="75" />
</Canvas>
</Window>
```

I 20 Das Subsystem Benutzeroberfläche

Abb. 20.3-3:
Buchungsfenster.



WPF-Oberflächen laufen – genau wie Windows Forms – als eigenständige Windows-Fenster oder im Fenster eines Webbrowsers als sogenannte *Web Browser Application* (WBA). Sowohl WPF als auch Windows Forms erfordern jedoch beim Einsatz im Web ein vollständiges .NET Framework – zumindest das .NET Framework Client Profile – auf jedem Zielsystem. WBAs sind daher ebenso wie Windows Forms im Browser kaum zu finden, allenfalls in Intranet-Szenarien.

Silverlight

- RIA Silverlight ist eine .NET-Technik zur Erstellung von *Rich Internet Applications* (RIAs), die im Vergleich zu HTML, CSS und JavaScript reichhaltigere Möglichkeiten zur Gestaltung der Benutzeroberfläche, Animationen und eine bessere Benutzerinteraktion (z. B. Unterstützung für Ziehen und Fallenlassen von Objekten mit der Maus) bieten, siehe »Web-Architektur«, S. 196. Viele der in Silverlight leicht realisierbaren Effekte sind in HTML, CSS und JavaScript nur mit sehr großem Aufwand möglich und bergen immer die Gefahr, dass sie nicht in allen Browsern funktionieren.

Silverlight basiert auf einem Browser-Plug-in (wie auch Java Applets und Adobe Flash), das eine Mini-Version des .NET Frameworks realisiert. Im Gegensatz zu den 60 MB für ein vollständiges .NET Framework ist das Installationspaket von Silverlight nur rund 5 MB groß (Microsoft unterstützt Windows und Mac OS, Mono auch Unix und Linux). Silverlight-Anwendungen können im Internet Explorer auch ohne die Browser-Menüs gestartet werden, sodass sie dem Benutzer wie eigenständige Desktop-Anwendungen erscheinen.

Beim Einsatz von Silverlight im Internet ist die Verbreitung des Plug-ins abzuwägen (vgl. <http://www.riastats.com/>). Beim Aufruf einer Silverlight-Anwendung erhalten Benutzer, die das Plug-in noch nicht installiert haben, die Aufforderung zur Installation. Benutzer, die dies verweigern, sind von der Anwendung ausgeschlossen.

Silverlight verwendet eine abgespeckte und syntaktisch veränderte Version von XAML, das auch in WPF zum Einsatz kommt. Silverlight läuft im Standard in der *Sandbox* des Browser und hat keinen direkten Zugriff zu Systemressourcen und Datenbanken. Es ist lediglich möglich, über Webservices mit einem Webserver zu kommunizieren.

Daraus ergibt sich immer eine Architektur mit drei physikalischen Schichten, während bei WPF und ASP.NET auch Architekturen mit zwei physikalischen Schichten möglich sind (Abb. 20.3-4).

Beim Vergleich von Silverlight und WPF ist stets zu beachten, dass die beiden Begriffe unterschiedliche Reichweiten besitzen. WPF ist eine Oberflächenbibliothek innerhalb des .NET Frameworks, die XAML verwendet. Silverlight ist im engeren Sinne ebenfalls eine Oberflächenbibliothek, die XAML verwendet, andererseits wird der Name Silverlight im weiteren Sinne aber auch als kleineres Pendant zum .NET Framework gesehen und umfasst in diesem Sinne auch Bibliotheken, die nichts mit Benutzungsoberflächen zu tun haben.

Silverlight vs. WPF

Das Model-View-ViewModell-Muster

Microsoft hat für WPF und Silverlight eine Anpassung des MVC-Musters erschaffen (siehe »Das MVC-Muster (*model view controller pattern*)«, S. 62), das Model-View-ViewModell (MVVM) genannt wird. *Model* und *View* entsprechen dabei der Bedeutung im klassischen MVC-Entwurfsmuster.

Das *ViewModel* kann als eine spezielle Implementierungsform eines *Controllers* gesehen werden, das die Aufgabe hat, die im Modell enthaltenen Informationen in einer Form aufzubereiten (»adaptieren«), die die Datenbindungstechniken in WPF und Silverlight direkt nutzen können.

I 20 Das Subsystem Benutzeroberfläche

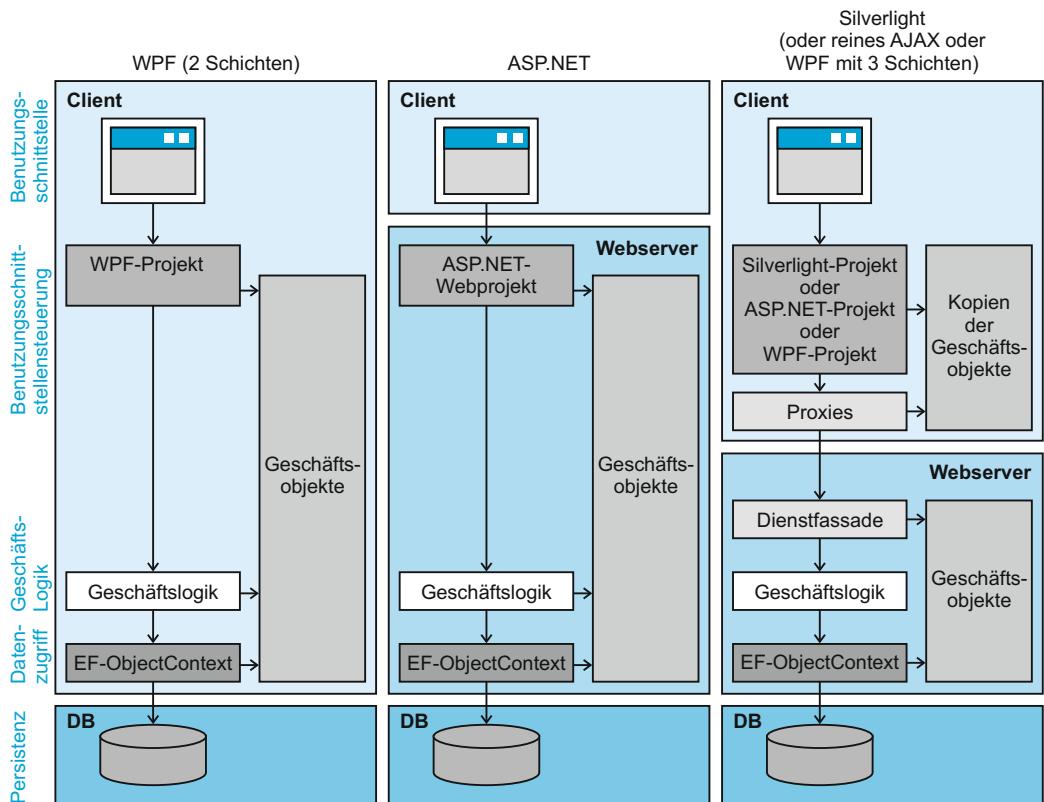


Abb. 20.3-4: Die Architekturen WPF, ASP.NET und Silverlight im Vergleich.

Das *ViewModel* hat auch die Aufgabe, sogenannte *Commands* zu implementieren, die an Ereignisse der Benutzeroberfläche gebunden werden. Das oberste Ziel von MVVM ist es, die Arbeit von Grafikern/Designern (die *Views* erstellen) und die Arbeit der Entwickler (die *Models* und *ViewModels* erstellen) zu trennen.

Ein *View* enthält im reinen MVVM-Muster keinen Programmcode, sondern besteht nur aus einem XAML-Markup. Den Aufbau einer Architektur nach dem MVVM-Muster zeigt die Abb. 20.3-5.

Das Subsystem Webserver ist optional bei WPF, aber verpflichtend bei Silverlight. Eine WPF-Anwendung kann direkt das Subsystem Persistenz verwenden.

20.3 GUIs in der .NET-Plattform I

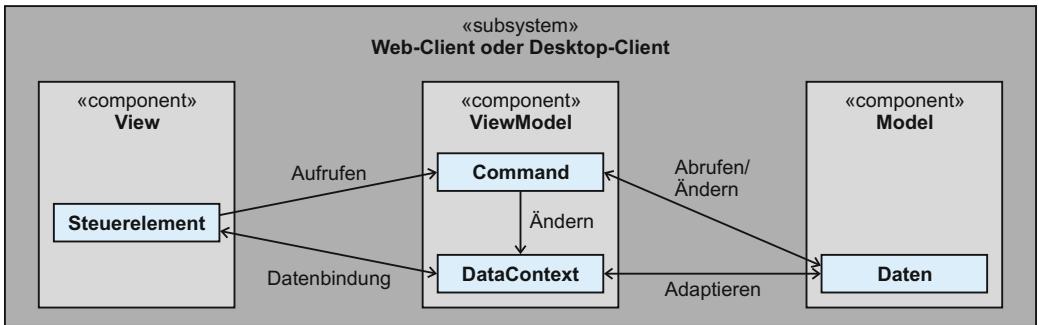


Abb. 20.3-5:
Architektur für
WPF- und
Silverlight-
Anwendungen
nach dem MVVM-
Muster.

21 Der Entwurfsprozess

Aufgabe des **Entwerfens** (*design*) ist es, aus den gegebenen Anforderungen an ein Softwaresystem eine softwaretechnische Lösung im Sinne einer Softwarearchitektur zu entwickeln, d.h. das Softwaresystem in seine Hauptbestandteile auf der obersten Ebene zu zerlegen einschl. der Beschreibung ihrer Beziehungen und Interaktionen.

Definition

Wie die Abb. 21.0-1 zeigt, ist der Ausgangspunkt für den Entwurf die fachliche Lösung. Bei einer objektorientierten Softwareentwicklung ist dies in der Regel ein objektorientiertes Analysemodell (OOA) der Domäne. Das Ergebnis ist die technische Lösung.

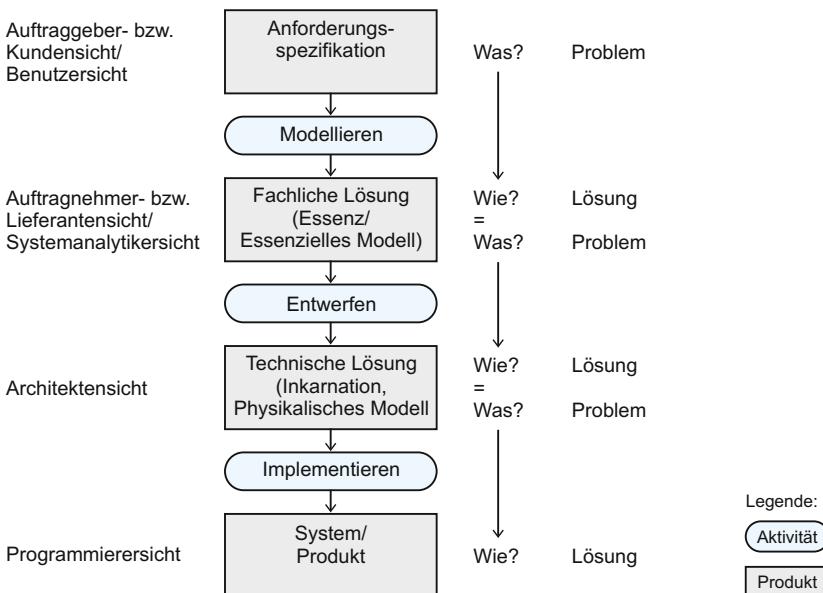


Abb. 21.0-1: Was vs. Wie im Entwicklungsprozess.

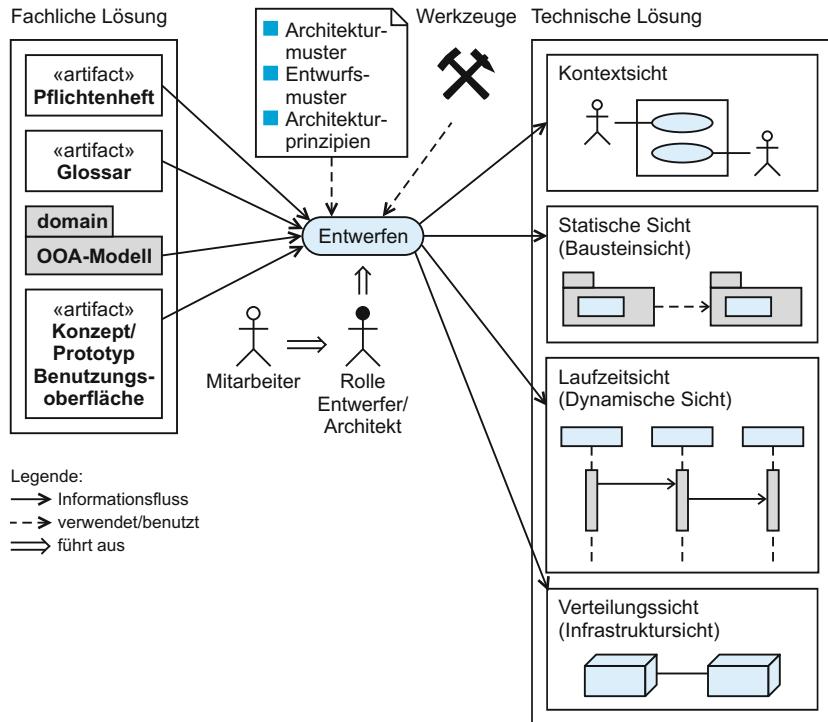
Die Abb. 21.0-2 zeigt einen beispielhaften Entwurfsprozess.

Es ist *nicht* möglich, einen allgemeingültigen Entwurfsprozess mit detaillierten Schritten darzustellen, da neben vielfältigen Randbedingungen insbesondere das jeweilige verwendete Prozess- und Qualitätsmodell eine wesentliche Rolle spielt.

Der Entwurfsprozess wird unterschiedlich ablaufen, in Abhängigkeit davon, ob nur ein Basis-Prozessmodell (z.B. inkrementelles Modell), einen Rahmen-Prozessmodell (z.B. CMMI, ISO 15504-Modell)

I 21 Der Entwurfsprozess

Abb. 21.0-2:
Beispielhafter
Entwurfsprozess.



oder ein agiles Modell (z.B. XP, Scrum) eingesetzt wird. Einen Vergleich dieser Modelle wird in dem »Lehrbuch der Softwaretechnik – Softwaremanagement« vorgenommen [Balz08, S. 515 ff.].

Dokumentation

Auch der Aufbau und der Umfang der Architekturdokumentation hängen stark vom verwendeten Prozess- und Qualitäts-Modell ab. Als Minimum sollte mithilfe der UML die »Statische Sicht«, die »Verteilungssicht«, die »Laufzeitsicht« und die »Kontextsicht« dokumentiert werden (siehe »Was ist eine Softwarearchitektur?«, S. 23). Für die Beschreibung von Architekturen wird die UML jedoch *nicht* als ausreichend anerkannt. Es fehlen beispielsweise Möglichkeiten zur Definition von Mustern, zur Beschreibung von Nebenläufigkeit und zur Spezifikation der Fehlerbehandlung [WoEm10].

Präskriptiv vs. Deskriptiv

Zu beachten ist, dass die Architekturdokumentation zwei Aufgaben zu erfüllen hat [Star05, S. 75]:

- Während der Entwicklung schreibt die Architekturdokumentation vor, wie das Softwaresystem zu realisieren ist (präskriptiv).
- In der Betriebsphase dient die Architekturdokumentation dazu, Entwurfsentscheidungen und Zusammenhänge zu verdeutlichen, um die Wartbarkeit und Weiterentwickelbarkeit zu unterstützen (deskriptiv). Um diese Aufgabe optimal zu ermöglichen, müssen folgende Anforderungen erfüllt werden:

- Vom Quellcode muss es Rückverweise auf die Architekturdokumentation geben (Nachverfolgbarkeit).
- Die Architekturdokumentation muss aktuell sein, das heißt mit dem Quellcode übereinstimmen.

Eine ausführliche Behandlung der Dokumentation von Softwarearchitekturen ist z. B. in [Star05, S. 71 ff.] zu finden.

Unabhängig von dem verwendeten Prozess-, Qualitäts- und Dokumentations-Modell kann jedoch angegeben werden, auf welche Reihenfolge der Entscheidungen zu achten ist.

Reihenfolge der Entscheidungen

Beim Entwurf einer Softwarearchitektur müssen eine Vielzahl von Entscheidungen getroffen werden.

Zuerst sollten die Entscheidungen getroffen werden, die die größten Auswirkungen auf die Softwarearchitektur haben. Dabei sind die Einflussfaktoren und die Wechselwirkungen zwischen ihnen zu berücksichtigen (siehe »Einflussfaktoren auf die Architektur«, S. 135).

Im Folgenden wird eine mögliche Reihenfolge vorgestellt. Je nach Anwendungsfall kann es sinnvoll sein, die Reihenfolge zu variieren.

1 Anwendungsart festlegen

Zweifellos die größte Auswirkung hat die Anwendungsart (siehe »Einflussfaktoren auf die Architektur«, S. 135). Eine falsche Entscheidung bei der Anwendungsart kann gravierende Auswirkungen auf die Softwarearchitektur haben, wenn sich später herausstellt, dass die Anwendungsart falsch gewählt wurde.

Überlegen Sie, welche Änderungen notwendig sind, um aus einer Einzelplatz-Anwendung eine Mehrplatz-Anwendung zu machen. Wie sieht es umgekehrt aus? Frage

Um aus einer Einzelplatz-Anwendung eine Mehrplatz-Anwendung zu machen, muss u. U. eine Benutzer- und Rechteverwaltung ergänzt werden. Außerdem muss zumindest ein Teil der Anwendung auf einem Server laufen, sodass von Clients aus zugegriffen werden kann. Aus einer Mehrplatz-Anwendung eine Einzelplatz-Anwendung zu machen ist einfach. Der Server wird auf dem Arbeitsplatz des Benutzers installiert und der Client greift lokal auf den Server zu. Antwort

2 Nichtfunktionale Anforderungen priorisieren

Da die nichtfunktionalen Anforderungen die meisten anderen Entwurfsentscheidungen beeinflussen, sollten sie frühzeitig priorisiert werden (siehe »Nichtfunktionale Anforderungen«, S. 109).

I 21 Der Entwurfsprozess

3 Verteilungsart festlegen

Ergibt sich aus der Anwendungsart, dass die Anwendung auf mehrere Computersysteme verteilt werden muss, dann ist als nächstes zu entscheiden, welche Verteilung gewählt werden soll (siehe »Verteilte Architekturen«, S. 191). Dabei sind die nichtfunktionalen Anforderungen und der Kontext des zu entwickelnden Systems zu berücksichtigen.

4 Grobentwurf erstellen

Auf Basis der Anwendungsart und der Verteilungsart lässt sich die grobe Architektur (Grobentwurf) des Anwendungssystems entwerfen. Der Grobentwurf dient als Grundlage für die Festlegung der softwaretechnischen Infrastruktur, weil sich mit dessen Hilfe die benötigten Infrastrukturkomponenten identifizieren lassen (siehe »Softwaretechnische Infrastrukturen«, S. 319).

5 Nichtfunktionale Anforderungen verfeinern und priorisieren

Wurden die nichtfunktionalen Anforderungen bisher nicht detailliert und priorisiert, sollte dies spätestens vor der Auswahl der softwaretechnischen Infrastruktur geschehen.

6 Auswahl der softwaretechnischen Infrastruktur

Nachdem die Anwendungs- und Verteilungsart festgelegt und die nichtfunktionalen Anforderungen detailliert und priorisiert wurden, sollte die softwaretechnische Infrastruktur ausgewählt werden. Zunächst sollte anhand des Grobentwurfs analysiert werden, welche Infrastrukturkomponenten benötigt werden. Unter Berücksichtigung der nichtfunktionalen Anforderungen, der Kompatibilität der zur Auswahl stehenden Techniken untereinander und weiterer Einflussfaktoren wie Kostenvorgaben und dem bereits vorhandenen Know-How, sollten dann konkrete Produkte ausgewählt werden.

Ausnahme

Falls die zu verwendende softwaretechnische Infrastruktur durch den Kunden vorgegeben wird, entfällt diese Entscheidung und die oben beschriebenen Entscheidungen hängen von der vorgegebenen softwaretechnischen Infrastruktur ab.

7 Nichtfunktionale Anforderungen überprüfen

Nach der Auswahl der softwaretechnischen Infrastruktur ist zu prüfen, ob die nichtfunktionalen Anforderungen mit der gewählten Lösung zu realisieren sind. Unter Umständen müssen Zugeständnisse bei einzelnen Zielvorgaben gemacht werden.

Beispiel

Die gewählte softwaretechnische Infrastruktur eignet sich zur Erfüllung aller funktionalen Anforderungen und unterstützt bis auf eine Leistungsanforderung alle anderen nichtfunktionalen Anforde-

rungen. Weil die Leistungsanforderung nur eine durchschnittliche Priorität hat, kann der Auftragnehmer sich mit dem Kunden auf eine Abschwächung der Anforderung einigen.

8 Kontext des Anwendungssystems

Nach dem Grobentwurf und der Festlegung der softwaretechnischen Infrastruktur sind die Schnittstellen zu den Anwendungssystemen, mit denen die zu entwickelnde Software später interagieren soll, zu konkretisieren. Die Art der Verteilung hängt auch vom Kontext des Anwendungssystems ab. Muss das System mit anderen Anwendungssystemen (intern oder extern) kommunizieren, dann ist zu prüfen, über welche Schnittstellen die anderen Anwendungssysteme angesprochen werden können. Müssen insbesondere externe Anwendungssysteme angesprochen werden, dann ist davon auszugehen, dass der Einfluss auf die Schnittstellen der externen Anwendungssysteme nicht oder nur in geringem Umfang vorhanden ist. In einem solchen Fall müssen daher standardisierte oder flexible Schnittstellen verwendet werden.

9 Feinentwurf

Unter Berücksichtigung der funktionalen Anforderungen und aller zuvor gemachten Entscheidungen und Einflussfaktoren kann der Grobentwurf der Softwarearchitektur verfeinert werden.

10 Evaluation

Bevor mit der Implementierung der Anwendung begonnen wird, sollte geprüft werden, ob die entworfene Softwarearchitektur sich zur Erfüllung der nichtfunktionalen Anforderungen tatsächlich eignet.

22 Qualitätssicherung der Architektur

Wie alle Artefakte, die im Laufe einer Softwareentwicklung entstehen, sollte bzw. muss auch die Softwarearchitektur einer Qualitäts sicherung unterzogen werden. Wegen der überragenden Bedeutung der Architektur sollte eine Qualitätssicherung nicht erst am Ende, d. h. nach der kompletten Fertigstellung der Architektur, erfolgen, sondern sie sollte die Architektur bei ihrer Entstehung begleiten.

Es gibt eine Reihe von Bewertungsverfahren für Architekturen und Architekturentscheidungen, die sich wie folgt kategorisieren lassen [Maie11]:

■ Quantitative Verfahren

- Metriken
- Nachweisverfahren (Simulationen, Experimente, Prototypen)

■ Fragebasierte Verfahren

- Fragebogenbasierte Verfahren
- Checklistenbasierte Verfahren
- Szenariobasierte Verfahren

Bei den quantitativen Verfahren wird versucht, Artefakte auf Zahlenwerte abzubilden, die dann als Maß für das Erreichen einer Qualitätsanforderung interpretiert werden.

Verfahren

quantitativ

Bei den fragebasierten Verfahren erfolgt eine Analyse anhand von Fragen und Antworten. Fragebogenbasierte Verfahren verwenden standardisierte Fragenkataloge. Diese Kataloge enthalten Fragen, die in der Vergangenheit dazu beigetragen haben, problematische Bereiche zu entdecken. Während fragebogenbasierte Verfahren grundsätzliche Fragestellungen enthalten, gehen Checklisten detailliert auf einzelne Qualitätsanforderungen ein. Fragebögen und Checklisten lenken den Blick auf typische Probleme, berücksichtigen aber nicht spezifische Anforderungen des betrachteten Systems. Szenariobasierte Verfahren beschreiben Szenarien. Ein QS-Team geht jedes Szenario Schritt für Schritt durch und analysiert, wie sich das auf dieser Architektur basierende System verhalten würde.

fragebasiert

Szenariobasierte Verfahren

Szenarien beschreiben, wie ein System auf das Eintreten eines oder mehrerer Ereignisse reagiert.

I 22 Qualitätssicherung der Architektur

Für die Überprüfung funktionaler Anforderungen eignen sich **Use Cases** und **User Storys** (bei der agilen Softwareentwicklung) gut als Szenarien zu Architekturanalyse.

Für die Überprüfung nichtfunktionaler Anforderungen (siehe »Nichtfunktionale Anforderungen«, S. 109) haben sich Qualitätsmerkmals-Szenarien (*Quality Attribute Scenarios*) bewährt. Die Tab. 22.0-1 zeigt die Bestandteile eines Qualitätsmerkmals-Szenarios [Maie11].

| Bestandteil | Kurzbeschreibung | Beispiele |
|----------------------|--|---|
| Quelle | Objekt, das ein Ereignis auslöst | Personen oder andere Systeme |
| Ereignis | Muss vom System in geeigneter Weise berücksichtigt werden | Benutzerinteraktion oder eine eintreffende Nachricht |
| Ausgangssituation | Situationen, in dem sich das System befindet, wenn das Ereignis eintritt | Normalbetrieb oder eingeschränkter Modus |
| Artefakt | Betroffenes Artefakt (ganzes System oder Teil davon) | Benutzungsoberfläche oder Datenbank |
| Reaktion | Was soll passieren, wenn das Ereignis eintritt? | Aktion wird ausgeführt bzw. Nachricht wird verarbeitet |
| Maß für die Reaktion | Die Reaktion wird damit objektiv testbar | Mit einer durchschnittlichen Reaktionszeit von 1 s, aber niemals mehr als 2 s |

Tab. 22.0-1: Für ein Lagerverwaltungssystem können für die nichtfunktionale Anforderung »Leistung« (siehe »Leistung und Effizienz«, S. 128) mehrere Szenarien erstellt werden. Ein Beispiel für ein solches Szenario zeigt die Tab. 22.0-2 [Maie11].

| Bestandteil | Szenario |
|----------------------|---|
| Quelle | Lagerverwaltungssystem |
| Ereignis | Nachricht zur Einlagerung einer Palette trifft ein |
| Ausgangssituation | Normalbetrieb, Lagerbefüllungsgrad $\leq 85\%$ |
| Artefakt | System |
| Reaktion | Paletten-Stellplatz wird gefunden und zurückgemeldet |
| Maß für die Reaktion | Alle Nachrichten werden beantwortet; durchschnittliche Latenz einer Antwort max. 500 ms (gemittelt über einen Tag); max. 2 % aller Antworten benötigen mehr als 2 s, keine mehr als 5 s |

Tab. 22.0-2: Es gibt verschiedene szenariobasierte Verfahren:
■ SAAM (*Software Architecture Analysis Method*): Ermöglicht die Analyse der nichtfunktionalen Anforderungen Modifizierbarkeit, Portabilität, Erweiterbarkeit sowie funktionaler Anforderungen. Wurde vor über 15 Jahren am SEI (*Software Engineering Institute*) entwickelt.

22 Qualitätssicherung der Architektur I

- ALMA (*Architecture-Level Modifiability Analysis*): Ermöglicht die Analyse der nichtfunktionalen Anforderung Modifizierbarkeit. Wurde 2004 von einem Team holländischer und schwedischer Forscher vorgestellt.
- ATAM (*Architecture Tradeoff Analysis Method*): Ermöglicht die Analyse sämtlicher nichtfunktionaler Anforderungen. Neben der Identifikation von Risiken werden auch Abhängigkeiten und Kompromisse sichtbar gemacht. Stellt eine Weiterentwicklung von SAAM dar.

Allen diesen Verfahren ist gemeinsam, dass in einer gemeinsamen Sitzung von **Stakeholdern**, Architekten des Systems und einem QS-Team die Szenarien durchgegangen werden. Alle diese Verfahren können schon in einer frühen Phase des Entwurfs eingesetzt werden.

Eine Analyse kann sich auf das gesamte System oder auf einzelne Subsysteme konzentrieren.

Auch die betrachteten nichtfunktionalen Anforderungen lassen sich situativ einschränken.

Der Einsatz eines szenariobasierten Verfahrens bringt folgende Vorteile mit sich:

- + Architekturrisiken und Schwachstellen können frühzeitig erkannt werden.
- + Die Analyse der Architektur zwingt die Entwerfer, die Architektur im Detail zu reflektieren.
- + Um eine Analyse vornehmen zu können, muss die Architektur geeignet dokumentiert sein.
- + Die Anforderungen müssen mithilfe von Szenarien formuliert werden.
- + Alle Beteiligten lernen eine Menge über das zu entwickelnde System.
- Alle szenariobasierten Verfahren erfordern einen nicht zu unterschätzenden personellen, finanziellen und zeitlichen Aufwand. Der Einsatz konzentriert sich daher im Wesentlichen auf risikobehaftete Softwaresysteme. Nachteil

Vergleich der Verfahren

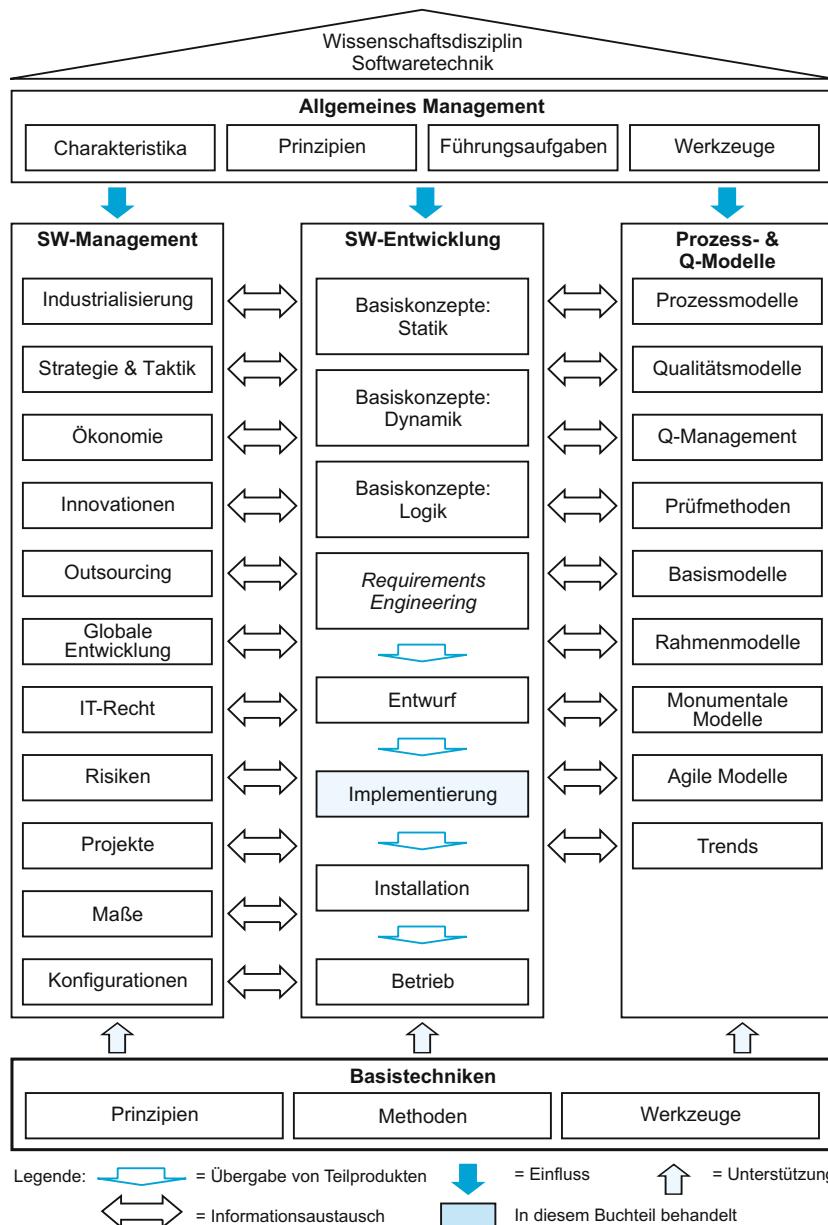
Die Tab. 22.0-3 zeigt den Vergleich einiger Verfahren zur Architekturanalyse [Maie11].

I 22 Qualitätssicherung der Architektur

| Verfahren → | Metriken | Nachweisverfahren | Szenariobasierte Verfahren |
|--|--|---|--|
| Abgedeckte nichtfunktionalen Anforderungen | Modifizierbarkeit, Zuverlässigkeit | Leistung, Benutzbarkeit, Sicherheit, Zuverlässigkeit, Funktionalität | SAAM: Modifizierbarkeit, Funktionalität ALMA: Modifizierbarkeit ATAM: alle |
| Verwendete Ansätze | Statische Codeanalyse | Experimente, Simulationen, Prototypen | Gedankenexperimente auf Basis systemspezifischer Szenarien |
| Zeit der Anwendung | Nachdem ein repräsentativer Teil des Systems realisiert wurde | Nach dem Architekturentwurf | Nach dem Architekturentwurf |
| Besondere Vorteile | Kann automatisiert werden, kostengünstig | Liefern messbare Ergebnisse, sind früh im Entwicklungsprozess anwendbar | Früh im Entwicklungsprozess anwendbar, Verifikation der nichtfunktionalen Anforderungen mit allen <i>Stakeholdern</i> , verbesserte Dokumentation & Kommunikation, Lerneffekte |
| Besondere Nachteile | Kein Bezug zu den konkreten Anforderungen des untersuchten Systems | Zeitlicher Aufwand & Kosten | Keine präzisen oder messbaren Ergebnisse. Zeitlicher Aufwand & Kosten |

Tab. 22.0-3:
Vergleich einiger Verfahren zur Architekturnalyse.

II Die Implementierung



II II Die Implementierung

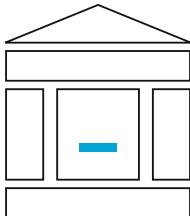
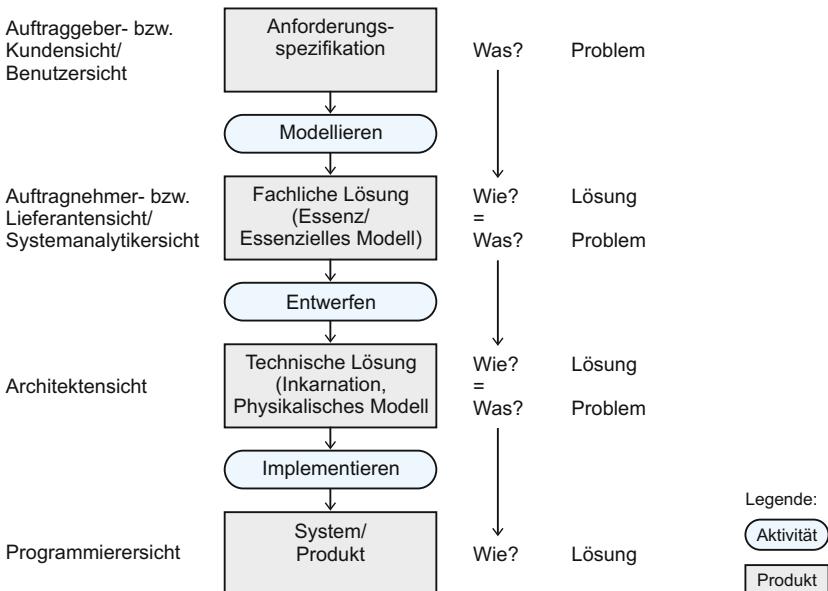


Abb. 22.0-2: Was vs. Wie im Entwicklungsprozess.



Die Abb. 22.0-3 zeigt einen beispielhaften Implementierungsprozess.

Es ist *nicht* möglich, einen allgemeingültigen Implementierungsprozess mit detaillierten Schritten darzustellen, da das jeweilige verwendete Prozess- und Qualitätsmodell eine wesentliche Rolle spielt.

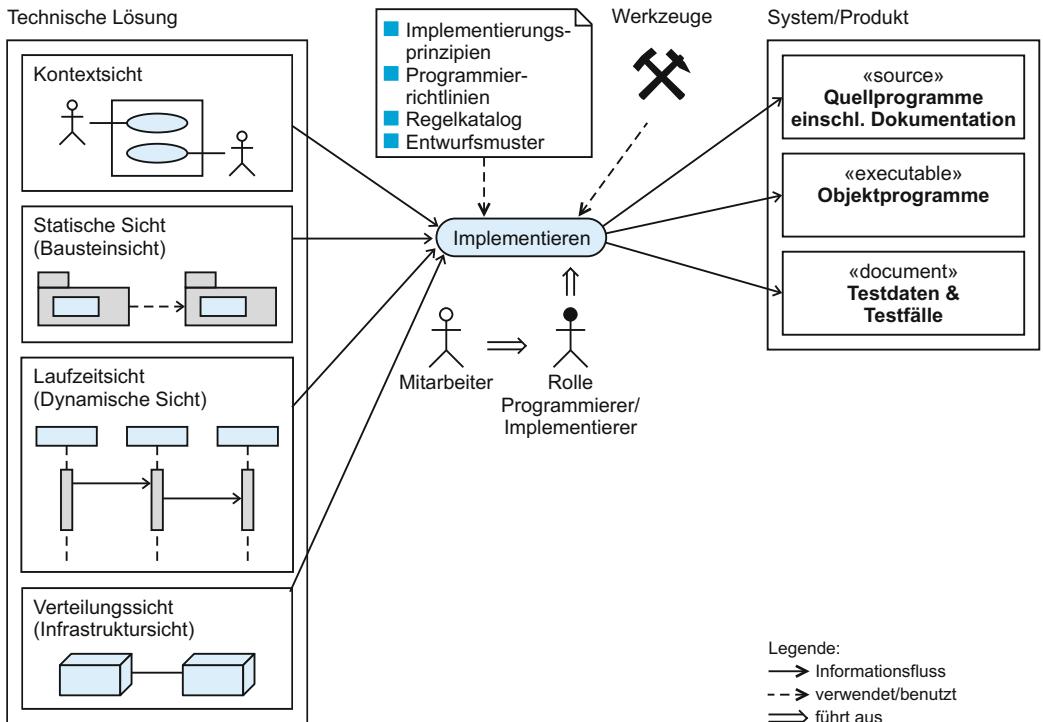
Prozess-Modell

Der Implementierungsprozess wird unterschiedlich ablaufen, in Abhängigkeit davon, ob nur ein Basis-Prozessmodell (z. B. inkrementelles Modell), ein Rahmen-Prozessmodell (z. B. CMMI, ISO 15504-Modell) oder ein agiles Modell (z. B. XP, Scrum) eingesetzt wird. Ein Vergleich dieser Modelle wird in dem »Lehrbuch der Softwaretechnik – Softwaremanagement« vorgenommen [Balz08, S. 515 ff.].

Die Tätigkeit **Implementieren** beinhaltet u. a. folgende Einzelaktivitäten:

- Konzeption von Datenstrukturen und Algorithmen,
- Strukturierung des Programms durch geeignete Verfeinerungsebenen,
- Dokumentation der Problemlösung und der Implementierungsentscheidungen durch geeignete Verbalisierung und Kommentierung,

II Die Implementierung II



- Umsetzung der Konzepte in die Konstrukte der verwendeten Programmiersprache,
- Angaben zur Zeit- und Speicherkomplexität des Programms in Abhängigkeit von den Eingabegrößen,
- Test oder Verifikation des entwickelten Programms einschließlich Testplanung und Testfallerstellung bei Anwendung einer Testmethode.

Alle Teilprodukte müssen integriert und einem Systemtest unterzogen werden.

Bei der Implementierung sollten eine Reihe von Prinzipien eingehalten werden:

- »Implementierungsprinzipien«, S. 495

Um die Wartbarkeit und die Weiterentwickelbarkeit zu verbessern, sollte unter Abwägung von Entwicklungsaufwand und Laufzeiteffizienz jeweils geprüft werden, ob anstelle von Klassen und Vererbung besser Schnittstellen, Fabriken und die Komposition verwendet werden:

- »Schnittstellen, Fabriken und Komposition«, S. 503

Programme werden oft iterativ in mehreren Schritten erstellt. Bei jedem Schritt sollte überprüft werden, ob Verbesserungen der bisherigen Programmstruktur möglich sind:

- »Restrukturieren (refactoring)«, S. 511

*Abb. 22.0-3:
Beispielhafter
Implementierungs-
prozess.*

23 Implementierungsprinzipien

Bei der Implementierung sollten folgende Prinzipien eingehalten werden:

- Prinzip der Verbalisierung,
- Prinzip der problemadäquaten Datentypen,
- Prinzip der integrierten Dokumentation,
- Prinzip des defensiven Programmierens.

Im Folgenden werden diese Prinzipien näher erläutert.

Prinzip der Verbalisierung

Verbalisierung bedeutet, Gedanken und Vorstellungen in Worten auszudrücken und damit ins Bewusstsein zu bringen. Bezogen auf die Entwicklung eines Programms soll Verbalisierung dazu dienen, die Ideen und Konzepte des Programmierers im Programm möglichst gut sichtbar zu machen und zu dokumentieren. Eine gute Verbalisierung kann erreicht werden durch:

- aussagekräftige, mnemonische Namensgebung,
- geeignete Kommentare,
- selbstdokumentierende Programmiersprache.

Für die Verständlichkeit eines Programms ist eine geeignete Wahl der **Bezeichner** (*identifier*) für Klassen, Attribute, Konstruktoren und Methoden entscheidend. Sie sollen die Funktion dieser Größe bzw. ihre Aufgabe zum Ausdruck bringen. Bezeichner, die problemfrei sind oder technische Aspekte z.B. der Repräsentation bezeichnen, sind zu vermeiden. Die in der Mathematik übliche Verwendung einzelner Buchstaben ist ebenfalls ungeeignet.

- | | |
|---|----------|
| ○ <code>feld1, feld2, zaehler: problemfreie, technische Bezeichner</code> | Beispiel |
|---|----------|
- Besser:
`messreihe1, messreihe2, anzahlZeichen: problembezogene Bezeichner`
- | | |
|---|--|
| ○ <code>p = g2 + z1 * d: Bezeichner ohne Aussagekraft, zu kurz</code> | |
|---|--|
- Besser:
`praemie = grundpraemie2 + zulage1 * dienstjahre;`
- | | |
|--|--|
| ○ <code>praemie = 50.0 + 10.0 * dienstjahre: unverständliche Konstanten</code> | |
|--|--|
- Besser:
`final double grundpraemie2 = 50.0;`
`final double zulage1 = 10.0;`
`praemie = grundpraemie2 + zulage1 * dienstjahre;`

II 23 Implementierungsprinzipien

Durch die Verwendung **benannter Konstanten** wird die Lesbarkeit eines Programms deutlich verbessert. Zusätzlich wird das Programm dadurch auch änderungsfreundlicher.

Kurze Bezeichner sind *nicht* aussagekräftig und außerdem wegen der geringen Redundanz anfälliger gegen Tippfehler und Verwechslungen. Der erhöhte Schreibaufwand durch lange Bezeichner wird durch die Vorteile mehr als ausgeglichen.

Kommentare Die Verbalisierung wird ebenfalls durch geeignete **Kommentare** unterstützt.

else-Teil Als vorteilhaft hat sich erwiesen, den **else**-Teil einer Auswahl zu kommentieren. Als Kommentar wird angegeben, welche Bedingungen im **else**-Teil gelten.

Beispiel

```
if (A < 5
    {...}
else //A >= 5
    {...}
```

Kurzkommentare sollten vermieden werden. Die in ihnen enthaltene Information ist meist besser in Namen unterzubringen.

Beispiel

```
i = i + 1; //i wird um Eins erhöht
Besser:
lagermenge = lagermenge + 1;
```

Besonders wichtige Kommentare können auch in einem Kommentarkasten untergebracht und damit hervorgehoben werden.

Selbstdokumentierende Sprache Der Grad der Kommentierung wird auch dadurch bestimmt, inwieweit die verwendete Programmiersprache selbstdokumentierende Programmierung fördert. Wichtige Anforderungen an Programmiersprachen sind daher die Selbsterklärung der Sprache und die Lesbarkeit der Programme.

Richtlinien, Konventionen für Bezeichner Für den Aufbau von Bezeichnern gibt es in jeder Programmiersprache eine festgelegte Syntax. Unabhängig von der erlaubten Syntax sind jedoch weitere **Regeln** und **Konventionen** einzuhalten, um gut lesbare Programme zu erhalten. Dadurch wird es nicht nur für den Autor, sondern auch für andere Personen, die sich in ein Programm einarbeiten wollen, leichter verständlich.

Vorteile Die Einhaltung des Prinzips der Verbalisierung bringt folgende Vorteile mit sich:

- + Leichte Einarbeitung in fremde Programme bzw. Wiedereinarbeitung in eigene Programme.
- + Erleichterung der Qualitätssicherung, der Wartung und der Pflege.
- + Verbesserte Lesbarkeit der erstellten Programme.

Prinzip der problemadäquaten Datentypen

Ein Problem soll so gelöst werden, dass sich die dem Problem innerwohnenden Daten- und Kontrollstrukturen in der programmiersprachlichen Lösung möglichst unverfälscht widerspiegeln. Dazu ist es erforderlich, dass die verwendete Programmiersprache bezogen auf Datenstrukturen

- ein umfangreiches Repertoire an Basistypen zur Verfügung stellt, Anforderungen an Sprache
- über geeignete Typkonstruktoren verfügt und
- benutzerdefinierbare Typen ermöglicht.

Die Aufgabe des Programmierers besteht darin, das Angebot an Konzepten einer Programmiersprache optimal zur problemnahen Lösungsformulierung zu verwenden.

- Können die Daten durch Basistypen beschrieben werden, dann ist der **geeignete Basistyp** auszuwählen. Der Wertebereich sollte so festgelegt werden, dass er möglichst genau das Problem widerspiegelt – unter Umständen durch Einschränkungen des Basistyps.

```
enum Familienstand {LEDIG, VERHEIRATET, GESCHIEDEN, VERWITWET};      Beispiele
enum Geschlecht {WEIBLICH, MAENNICH};
enum Ampel {ROT, GELB, GRUEN};
enum Steuerschlüssel {OHNE_STEUER, VORSTEUER, ER_MWST,
    VOLLE_MWST};
enum Weinpraedikate {KABINETT, SPAETLESE, AUSLESE,
    BEERENAUSLESE, TROCKENBEERENAUSLESE};
enum Bauteiltyp {R, L, C, U, I};
```

Bei der objektorientierten Softwareentwicklung sollten bereits in OOP der Spezifikation und im Entwurf die Typen von Attributen durch <<dataType>> und <<enumeration>> in der UML modelliert werden, wenn es sich um keine primitiven Datentypen handelt.

- Der Typkonstruktor **Feld** (array) ist zu verwenden, wenn möglichst viele der folgenden Merkmale zutreffen:
 - Zusammenfassung gleicher Datentypen.
 - Zugriff wird dynamisch berechnet (während der Laufzeit).
 - Hohe Komponentenanzahl möglich.
 - Feldgrenzen statisch, dynamisch oder unspezifiziert.
 - Mittlere Zugriffszeit auf eine Komponente, unabhängig vom Wert des Index.
 - Als zugehörige Kontrollstruktur wird die zählende Wiederholung eingesetzt.
- Der Typkonstruktor **Verbund** (struct) ist zu verwenden, wenn möglichst viele der folgenden Merkmale zutreffen:
 - Zusammenfassung logischer Daten mit unterschiedlichen Typen.
 - Zugriff wird statisch berechnet (zur Übersetzungszeit).
 - Anzahl der Komponenten ist immer fest.

II 23 Implementierungsprinzipien

- Jede Komponente ist einzeln benannt, daher ist der Umfang begrenzt (100 oder 1000 Komponenten nicht sinnvoll).
- Kurze Zugriffszeit auf eine Komponente erwünscht.
- Bei Varianten Verbunden ist die Mehrfachauswahl die geeignete Kontrollstruktur.

OOP In manchen objektorientierten Sprachen wie Java wird der Typkonstruktor »Verbund« durch Klassen realisiert.

- Vorteile Die Wahl problemadäquater Datentypen bringt folgende Vorteile:
- + Gut verständliche, leicht lesbare, selbstdokumentierende und wartbare Programme.
 - + Statische und dynamische Typprüfungen verbessern die Qualität des jeweiligen Programms.
 - + Die Daten des Problems werden 1:1 in Datentypen des Programms abgebildet, d.h., Wertebereiche werden weder über- noch unterspezifiziert.

Prinzip der integrierten Dokumentation

Integraler Bestandteil jedes Programms muss eine geeignete Dokumentation sein. Eine gute Dokumentation sollte folgende Angaben beinhalten:

- Kurzbeschreibung des Programms,
- Verwaltungsinformationen,
- Kommentierung des Quellcodes.

Programmvorspann Die ersten beiden Angaben können in einem Programmvorspann zusammengefasst werden:

- Programmname: Name, der das Programm möglichst genau beschreibt.
- Aufgabe: Kurzgefasste Beschreibung des Programms.
- Zeit- und Speicherkomplexität des Programms.
- Name des Programmautors bzw. der Programmautoren.
- Versionsnummer und Datum.
- Bearbeitungszustand des Programms, z.B. geplant, in Bearbeitung, vorgelegt, akzeptiert.

Quellcode-Kommentierung Neben dem Programmvorspann muss auch der Quellcode selbst dokumentiert werden. Besonders wichtig ist die geeignete Kommentierung der Methoden einer Klasse. Neben der Aufgabenbeschreibung jeder Methode ist die Bedeutung der Parameter zu kommentieren, wenn diese aus dem Parameternamen nicht eindeutig ersichtlich ist.

Hinweis In Java gibt es spezielle Dokumentationskommentare (`/** Kommentar */`), die durch das Werkzeug Javadoc ausgewertet werden. Es entsteht eine HTML-Dokumentation, die die Standarddokumentation für alle bereitgestellten Klassen ist.

23 Implementierungsprinzipien II

Zusätzlich wird eine gute Dokumentation durch eine geeignete Verbalisierung (siehe oben) unterstützt.

Die **Dokumentation** muss aus folgenden Gründen integraler Bestandteil der Softwareentwicklung sein:

- Bei einer Nachdokumentation am Ende der Codeerstellung sind wichtige Informationen, die während der Entwicklung angefallen sind, oft nicht mehr vorhanden.
- Entwicklungsentscheidungen (z. B.: Warum wurde welche Alternative gewählt?) müssen dokumentiert werden, um bei Modifikationen und Neuentwicklungen bereits gemachte Erfahrungen auswerten zu können.
- Der Aufwand für die Dokumentation wird reduziert, wenn zu dem Zeitpunkt, an dem die Information anfällt von demjenigen, der sie erzeugt oder verarbeitet, auch dokumentiert wird.

Integrierte Dokumentation

Die Beachtung des Prinzips der integrierten Dokumentation

Vorteile

- + reduziert den Aufwand zur Dokumentenerstellung,
- + stellt sicher, dass keine Informationen verloren gehen,
- + garantiert die rechtzeitige Verfügbarkeit der Dokumentation,
- + erfordert die entwicklungsbegleitende Dokumentation.

Eine gute Dokumentation bildet die Voraussetzung für

- leichte Einarbeitung in ein Softwaresystem bei Personalwechsel oder durch neue Mitarbeiter,
- gute Wartbarkeit und Weiterentwickelbarkeit des Softwaresystems.

Prinzip des defensiven Programmierens

Je umfangreiche und komplexer Programme werden, desto mehr Fehler werden die Programme enthalten. Daher ist es sehr wichtig, **defensiv zu programmieren**, d. h. potenzielle Fehlerquellen möglichst konstruktiv zu vermeiden.

- Jede mögliche Fehlersituation durch eine **Ausnahme-Behandlung** (*exception handling*) abfangen. Moderne Programmiersprachen wie Java stellen besondere Sprachkonstrukte zur Verfügung, um auf einfache und elegante Weise Fehler abzufangen und damit das defensive Programmieren zu erleichtern (try-catch-finally-Konstrukt in Java).

Regeln

Bei der Ausführung eines Programms kann es zu verschiedenen Fehlerarten kommen, die ein Weiterarbeiten unmöglich machen:

Beispiele

- Der Programmierer hat eine Fehlersituation übersehen, z. B. tritt bei einer mathematischen Berechnung eine Division durch Null auf.

II 23 Implementierungsprinzipien

- Der Benutzer des Programms gibt Werte ein, die nicht vorgesehen waren und nicht weiterverarbeitet werden können, z.B. wird ein Buchstabe statt einer Ziffer eingegeben.
- Beide Fehlerarten können durch defensives Programmieren vermieden werden.
- Um Fehler beim Programmieren zu vermeiden, sollten an allen Programmstellen, wo Werte oder Kombinationen von Werten gelten müssen, sogenannte **Zusicherungen** (*assertions*) eingefügt werden, die dann automatisch während der Laufzeit ausgewertet werden. Wird die Zusicherung verletzt, dann wird eine Fehlermeldung ausgegeben und das Programm abgebrochen. In Java erfolgt eine Zusicherung in einer assert-Anweisung.

Beispiel In einem Programm zur Prämienberechnung kann durch Zusicherungen sichergestellt werden, dass beim Programmieren keine Fehler in den Abfragen aufgetreten sind:

```
Java // Prämienberechnung in Abhängigkeit von den
      // Dienstjahren und dem Alter mit Zusicherungen
Praemie2
import inout.Console;
class Praemie2
{
    public static void main (String args[])
    {
        final int grundpraemie1 = 200, grundpraemie2 = 100,
zulage1 = 20, zulage2 = 100; //Konstanten
        int dienstjahre, alter; //Eingabe
        int praemie = 0; //Ausgabe
        System.out.println("Dienstjahre?");
        dienstjahre = Console.readInt();
        System.out.println("Alter?");
        alter = Console.readInt();
        if (dienstjahre > 5)
        {
            assert(dienstjahre > 5):
                "Dienstjahre ist nicht > 5";
            praemie = grundpraemie2 + zulage1 * dienstjahre;
            if (alter > 45)
            {
                assert(dienstjahre > 5 && alter > 50):
                    "Dienstjahre oder Alter falsch";
                praemie = praemie + zulage2;
            }
        }
        else //dienstjahre <= 5
        if (dienstjahre > 2)
        {
            assert(!(dienstjahre > 5)):
                "Dienstjahre ist nicht <= 5";
            praemie = grundpraemie1;
        }
    }
}
```

23 Implementierungsprinzipien II

```
System.out.println("Dienstjahre:\t" + dienstjahre);
System.out.println("Alter:\t\t" + alter);
System.out.println("Prämie:\t\t" + praemie);
}
}
```

Wenn Sie durch einen Tippfehler in der Zeile
if (Alter > 45) geschrieben haben, statt
if (Alter > 50), dann erhalten Sie bei folgender Programmausführung den Fehlerhinweis, dass die Zusicherung
assert(Dienstjahre > 5 && Alter > 50) : "Dienstjahre oder Alter falsch";
verletzt wurde:

```
Dienstjahre?
8
Alter?
46
Exception in thread "main" java.lang.AssertionError:
Dienstjahre oder Alter falsch
at Praemie2.main(Praemie2.java:25)
```

- Als Kommentar wird angegeben, welche Bedingungen im else-Teil gelten (siehe oben).

Die Orientierung an dem Prinzip des defensiven Programmierens bringt folgende Vorteile mit sich:

- + Programmierfehler werden im Programm abgefangen, ohne dass das Programm »abstürzt».
- + In dem Programm wird dafür gesorgt, dass alle Annahmen, die das Programm über seine Umgebung macht, z.B. Annahmen über Parameterwerte, selbst überprüft werden. Dadurch wird auch die Wiederverwendbarkeit eines Programmes erhöht.
- + Das Programm wird der robuster, d.h. die Zuverlässigkeit wird erhöht.

Dem stehen folgende Nachteile gegenüber:

Nachteile

- Der Programmieraufwand steigt.
- Die Laufzeiteffizienz wird schlechter. In manchen Programmiersprachen können die zusätzlichen Sicherungsmechanismen aber für den Produktivbetrieb abgeschaltet werden.

24 Schnittstellen, Fabriken und Komposition

Die nichtfunktionale Anforderung Weiterentwickelbarkeit mit seinen Merkmalen Änderbarkeit und Erweiterbarkeit (siehe »Weiterentwickelbarkeit«, S. 119) spielt bei vielen Softwaresystemen eine immer wichtigere Rolle. Diese Anforderung kann auf der Ebene des Entwurfs und der Implementierung durch die Reduzierung von Abhängigkeiten, durch die Vermeidung von Redundanz und durch die Trennung von Zuständigkeiten (siehe »Architekturprinzip: Trennung von Zuständigkeiten«, S. 31) unterstützt werden.

Eine Analyse der objektorientierten Konzepte »Klasse« und »Vererbung« zeigt, dass sie eine Weiterentwickelbarkeit eher verhindern als fördern.

Klassen & Vererbung

Das Architekturprinzip »Trennung von Zuständigkeiten« wird von Klassen *nicht* erfüllt. Das Erzeugen und das Benutzen von Objekten sind unterschiedliche Dinge. Wer Objekte benutzt, muss deshalb noch lange nicht in der Lage sein, sie auch erzeugen zu können. Eine Klasse ist jedoch für beides zuständig. Daraus ergeben sich unnötige Abhängigkeiten zwischen Objektbenutzern und Objektimplementierungen.

Das Konzept der Klasse ist überladen. Eine Klasse regelt einfach alles, was mit dem Aufbau, der Erzeugung und der Benutzung von Objekten zu tun hat. Daher gibt es viele Richtlinien und Muster, um Klassen sinnvoll einzusetzen.

Schnittstellen und Fabriken

Zur Entflechtung der Zuständigkeiten von Klassen können abstrakte Schnittstellen und abstrakte Fabriken eingesetzt werden.

Abstrakte Schnittstellen erlauben es, ein Objekt zu *benutzen*, ohne zu wissen, zu welcher Klasse es gehört. Viele Entwurfsmuster basieren auf dieser Trennung. Insbesondere bei verteilten Systemen werden Stellvertreter für Objekte in anderen Adressräumen genauso benutzt wie gleichartige Objekte im lokalen Adressraum (siehe »Das Proxy-Muster (*proxy pattern*)«, S. 83).

Abstrakte Fabriken erlauben es, ein Objekt zu *erzeugen*, ohne zu wissen, zu welcher Klasse das erzeugte Objekt gehört. Das verwendete Fabrikobjekt wird wiederum benutzt, ohne zu wissen, zu welcher Klasse es gehört.

II 24 Schnittstellen, Fabriken und Komposition

Probleme

Die Verwendung von Schnittstellen und Fabriken anstelle von Klassen ist mit einem erheblichen Mehraufwand und zusätzlichen Fehlerquellen verbunden. Bei der Verwendung von Fabriken kommt hinzu, dass sie nicht wirklich für die Objekterzeugung zuständig sind, sondern nur eine Fassade darstellen, hinter der die zuständigen Klassen verborgen sind.

Eine echte Entkopplung erfordert, dass die Funktionalität der Konstruktoren (und gegebenenfalls auch des Destruktors) vollständig in die Fabrik verlagert werden.

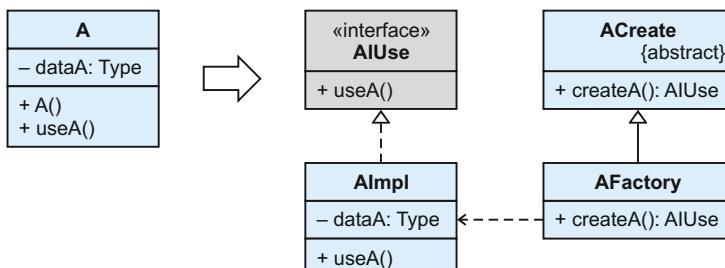
Trennung der Zuständigkeiten

Sollen die Zuständigkeiten einer Klasse getrennt werden, dann muss die Klasse in mehrere Konzepte aufgeteilt werden [Drac05, S. 138]:

- Wer nur Objekte einer Klasse *benutzen* will, muss nur öffentliche Methoden kennen. Diese Methoden werden unabhängig von der Klasse in Nutzungs-Schnittstellen deklariert.
- Wer Objekte einer Klasse *erzeugen* will, muss nur Konstruktoren kennen oder nur die entsprechenden Fabrikmethoden kennen. Die Fabrikmethoden werden abgesetzt von der Klasse in (Erzeugungs-)Schnittstellen deklariert. Diese Schnittstellen werden nicht von der Klasse, sondern von den »befreundeten« Fabriken implementiert.
- Aufgabe einer Klasse ist es, die Daten und die Nutzungs-Schnittstellen zu implementieren.

Im einfachsten Fall wird also eine Klasse in vier Teile zerlegt (Abb. 24.0-1) [Drac05, S. 137].

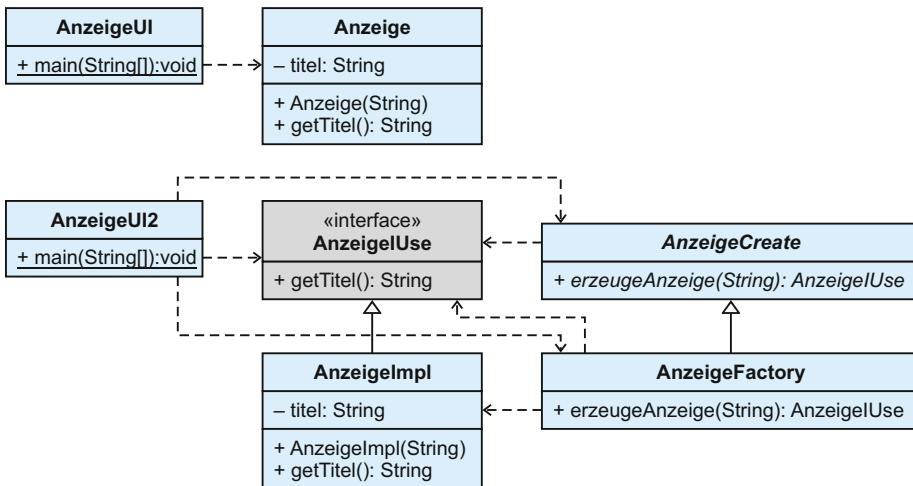
Abb. 24.0-1:
Zerlegung einer Klasse in eine Schnittstelle, eine Implementierungs-klasse, eine abstrakte Erzeugungsklasse und eine Fabrikklasse.



Anzeigenverwaltung

Beispiel: Für eine Anzeigenverwaltung von Zeitungsanzeigen wird eine Klasse Anzeige benötigt (hier in vereinfachter Form). Der obere Teil der Abb. 24.0-2 zeigt, wie von der Klasse AnzeigeUI auf die Klasse Anzeige zugegriffen wird. Im unteren Teil der Abb. 24.0-2 ist zu sehen, wie die Klasse Anzeige in eine Schnittstelle AnzeigeIUse, eine Klasse AnzeigeImpl, eine abstrakte Klasse AnzeigeCreate und eine Klasse AnzeigeFactory aufgeteilt wird.

24 Schnittstellen, Fabriken und Komposition II



Die Java-Programme für beide Varianten sehen wie folgt aus:

```

public class AnzeigeUI
{
    public static void main(String arg[])
    {
        //Erzeugung
        Anzeige eineAnzeige = new Anzeige("Gartentisch");
        //Nutzung
        System.out.println(eineAnzeige.getTitel());
    }
}

public class Anzeige
{
    private String titel;

    public Anzeige (String titel)
    {
        this.titel = titel;
    }

    public String getTitel()
    {
        return this.titel;
    }
}

public class AnzeigeUI2
{
    public static void main(String arg[])
    {
        AnzeigeCreate eineFabrik = new AnzeigeFactory();
        //Erzeugung
        AnzeigeIUse eineAnzeige =
            eineFabrik.erzeugeAnzeige("Gartentisch");
        //Nutzung
    }
}
  
```

*Abb. 24.0-2:
Zerlegung der
Klasse Anzeige in
Schnittstelle,
Implementierungs-
klasse, abstrakte
Erzeugungsklasse
und eine
Fabrikklasse.*

Java
Variante 2

II 24 Schnittstellen, Fabriken und Komposition

```
System.out.println(eineAnzeige.getTitle());
//Erzeugung
eineAnzeige =
    eineFabrik.erzeugeAnzeige("Staubsauger");
//Nutzung
System.out.println(eineAnzeige.getTitle());
}

}

public interface AnzeigeIUse
{
    //abstrakte Methode
    public abstract String getTitle();
}

public class AnzeigeImpl implements AnzeigeIUse
{
    private String titel;

    public AnzeigeImpl (String titel)
    {
        this.titel = titel;
    }

    public String getTitle()
    {
        return this.titel;
    }
}

public abstract class AnzeigeCreate
{
    public abstract AnzeigeIUse erzeugeAnzeige (String titel);
}

public class AnzeigeFactory extends AnzeigeCreate
{
    public AnzeigeIUse erzeugeAnzeige(String titel)
    {
        return new AnzeigeImpl(titel);
    }
}
```

 Spielen Sie verschiedene Änderungsszenarien durch und sehen Sie sich an, welcher Änderungsaufwand in der Variante 1 und welcher Änderungsaufwand in der Variante 2 erforderlich sind.

Komposition mit Weiterleitung

Durch das Konzept der Vererbung ist es möglich, redundanten Code zu vermeiden. Gemeinsame Attribute und Methoden werden in Oberklassen verlagert (Faktorisierung). Die reduzierte Redundanz führt aber zu zusätzlichen Abhängigkeiten.

24 Schnittstellen, Fabriken und Komposition II

Die ursprünglich unabhängigen Klassen, aus denen der redundante Code ausgelagert wurde, sind nun von einer gemeinsamen Oberklasse abhängig.

Notwendige Änderungen oder Fehlerbeseitigungen an der Oberklasse wirken automatisch auf die Unterklassen. Das ist ein Vorteil. Die Vererbung ist jedoch nachteilig, wenn Änderungen in der Oberklasse nur für einen Teil der Unterklassen relevant sind.

Die Beseitigung von Redundanz führt zu Abhängigkeiten und umgekehrt. Es muss im Einzelfall abgewogen werden, was wichtiger ist.

Regel

Arten von
Abhängigkeiten

- Es lassen sich unterschiedliche Abhängigkeiten unterscheiden:
- einseitige Abhängigkeiten
- zyklische Abhängigkeiten
- Abhängigkeiten von Schnittstellen
- Abhängigkeiten von konkreten Implementierungen

Zyklische Abhängigkeiten sind gravierender als einseitige Abhängigkeiten, Abhängigkeiten von konkreten Implementierungen gravierender als Abhängigkeiten von Schnittstellen.

Eine Vererbungsstruktur führt zu starken Abhängigkeiten, da die Unterklassen von der konkreten Implementierung der Oberklasse abhängig werden.

Werden Klassen in Schnittstellen und Implementierung aufgeteilt und dann die Objekte durch Komposition zusammengefügt, dann erben die Klassen keine konkrete Implementierung mehr, sondern leiten Aufgaben explizit an andere Objekte weiter.

Von diesen anderen Objekten kennen sie nur die Schnittstelle. Dadurch können verschiedene Implementierungen der beteiligten Schnittstellen frei miteinander kombiniert werden. Es ergeben sich schwächere Abhängigkeiten.

Wird die Klassenvererbung dazu genutzt, Code wieder zu verwenden, dann lässt sich die Klassenvererbung immer durch eine Objektkomposition ersetzen [GHJ+95].

Regel

Die Abb. 24.0-3 zeigt eine Immobilienverwaltung in drei Varianten. Beispiel
Die Variante (a) zeigt die klassische Vererbung, die Variante (b) zeigt die Weiterleitung und die Variante (c) zeigt eine Komposition mit Weiterleitung.

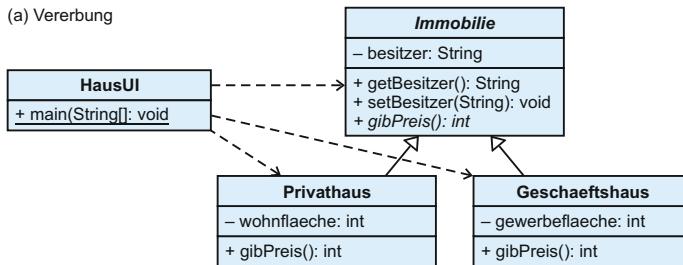
Den Code zu den Beispielen können Sie im E-Learning-Kurs zu diesem Buch herunterladen.

Laden Sie den Beispielcode auf Ihr Computersystem und führen Sie alle drei Varianten aus.

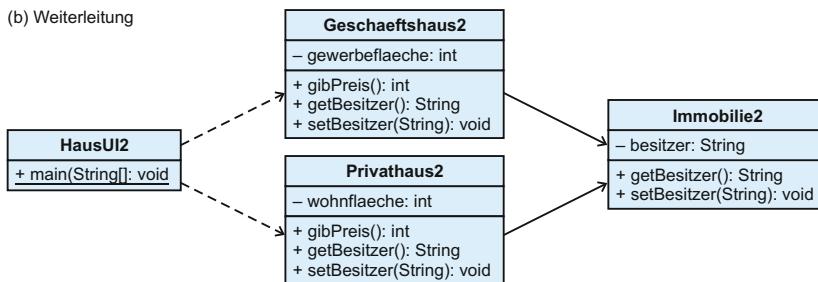


II 24 Schnittstellen, Fabriken und Komposition

(a) Vererbung



(b) Weiterleitung



(c) Komposition mit Weiterleitung

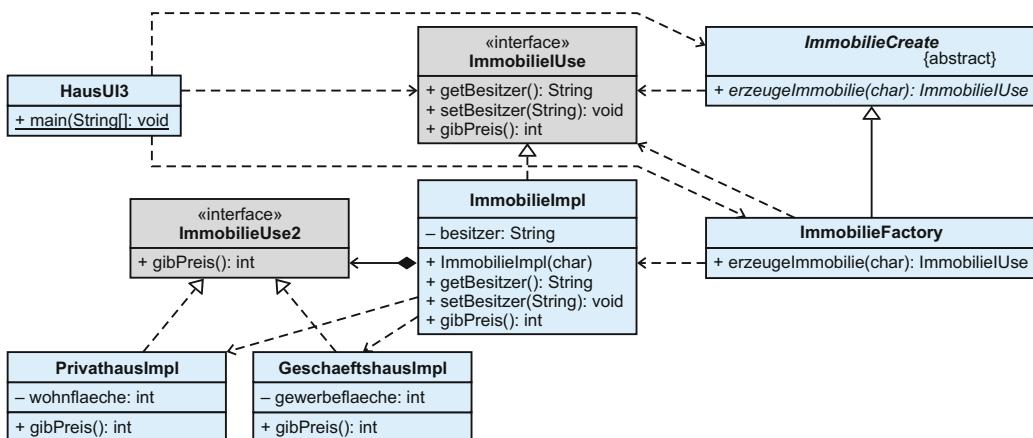


Abb. 24.0-3:
Beispiel für das
Ersetzen von
Vererbung durch
Objektkom-
position.

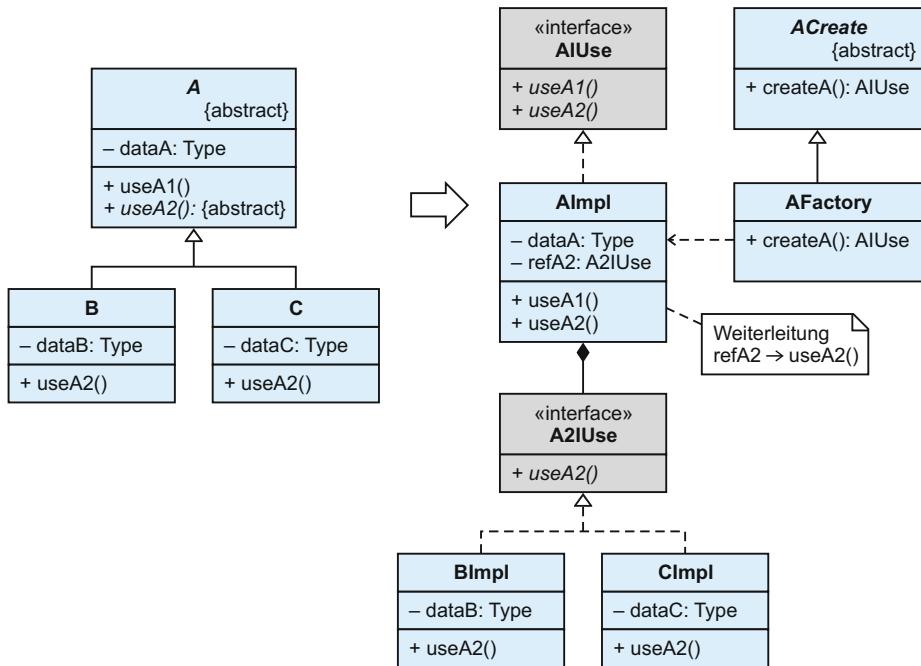
Spielen Sie verschiedene Änderungsszenarien durch und sehen Sie sich an, welcher Änderungsaufwand in der Variante (a) und welcher Änderungsaufwand in der Variante (c) erforderlich sind.

Die Abb. 24.0-4 zeigt das allgemeine Schema [Drac05, S. 138].

Die Verwendung der Klassenvererbung besitzt folgende Nachteile:

- Es wird das Denken in konkreten Implementierungen gefördert, statt das Denken in abstrakten Schnittstellen.

24 Schnittstellen, Fabriken und Komposition II



- In Kombination mit dynamischer Bindung können undurchschaubare zyklische Abhängigkeiten entstehen, bei denen die Oberklasse und die Unterklassen sich gegenseitig beeinflussen.

Dagegen ist die Komposition mit Weiterleitung eine gute Alternative mit weniger Risiken.

[Drac05], [GHJ+95], [Fowl05].

*Abb. 24.0-4:
Ersetzen einer Vererbungsstruktur
durch eine Objekt-
komposition.*

[Literatur](#)

25 Restrukturieren (*refactoring*)

Die Umsetzung einer Softwarearchitektur in Programme erfolgt in der Regel inkrementell und iterativ. Insbesondere bei einer agilen Softwareentwicklung ist es das Ziel, auf der Basis von Testfällen (*test-first*) Programme zu erstellen und anschließend sofort mit bereits vorhandenen Programmen zu integrieren (*continuous integration*) (siehe z.B. [Reif11]). Dabei wird die Funktionalität des Programms dann schrittweise erweitert.

Trotz aller Sorgfalt und Vorausschau stellt man bei der Weiterentwicklung von Programmen oft fest, dass durch eine Umstrukturierung die Programmstruktur verbessert werden kann. Mit dem Begriff *refactoring* – zu deutsch Umstrukturierung oder Restrukturierung – wird die Umstrukturierung existierender Programme bezeichnet, um ihre Struktur zu verbessern.

Restrukturierung (*refactoring*) bezeichnet den Änderungsprozess eines Softwaresystems in der Weise, dass das externe Verhalten des Codes nicht verändert, aber die interne Struktur verbessert wird (in Anlehnung an [Fowl05, S. xvi]).

Definition

Eine Restrukturierung ist natürlich *nicht* nur während einer Neuentwicklung von Programmen erforderlich, sondern insbesondere bei der Wartung und Pflege von im Betrieb befindlicher Softwaresysteme. Auf die besonderen Aspekte, die sich im Laufe eines Betriebs durch Änderungen der Anforderungen und Umgebungsbedingungen ergeben, wird in folgenden Kapiteln eingegangen:

Wartung & Pflege

- »*Reengineering* (Teil 2)«, S. 557
- »*Reengineering* (Teil 3)«, S. 561

Oft sind es viele kleine Änderungen, die in der Summe jedoch den Entwurf radikal verbessern können.

Restrukturierung von Klassen und Methoden¹

Während der Entwicklung werden häufig nach und nach zusätzliche Funktionen hinzugefügt, was dazu führt, dass die Methoden und die Klassen immer länger werden. Wird der Code immer mehr erweitert, dann erfüllt eine Methode oder Klasse irgendwann nicht mehr nur eine klar definierte Aufgabe, sondern mehrere. Der Bindungsgrad sinkt.

¹ Die folgenden Texte und Grafiken wurden mit freundlicher Genehmigung des W3L-Verlags aus [Balz11, S. 300 ff.] entnommen.

II 25 Restrukturieren (*refactoring*)

Das Restrukturieren von Klassen und Methoden bedeutet, Klassen und Methoden zu überdenken und neu zu entwerfen. Oft wird dabei eine Klasse in zwei Klassen aufgeteilt oder aus einer Methode entstehen mehrere. Das Zusammenfassen mehrerer Klassen oder Methoden zu einer ist ebenfalls möglich, kommt aber seltener vor.

- Beispiel Ein Seminarveranstalter bietet nur firmeninterne Veranstaltungen an. Zur Verwaltung der Veranstaltung gibt es die Klasse Veranstaltung mit folgenden Attributen:

```
Java import java.util.*;
//Firmeninterne Veranstaltungen
public class Veranstaltung
{
    private int nummer;
    private short dauer;
    private Calendar vom, bis;
    private String ort, adresse;
    private short teilnehmerMax;
    private float pauschalpreis;
    private boolean storniert = false;
    ...
}
```

Während der Entwicklung teilt der Auftraggeber mit, dass er in Zukunft auch öffentliche Veranstaltungen anbieten will. Die Klasse Veranstaltung wird durch den zuständigen Programmierer daher um folgende zusätzlichen Attribute ergänzt:

```
Java import java.util.*;
//Firmeninterne und öffentliche Veranstaltungen
public class Veranstaltung
{
    private int nummer;
    private short dauer;
    private Calendar vom, bis;
    private String ort, adresse;
    private short teilnehmerMax;
    private float pauschalpreis; //nur wenn firmenintern
    private boolean storniert = false;
    private short teilnehmerMin; //nur wenn öffentlich
    private short teilnehmerAktuell; //nur wenn öffentlich
    private float stornogebühr; //nur wenn öffentlich
    ...
}
```

Es entsteht eine Situation, in der je nach Seminarart einige Attribute *nicht* belegt werden. Das widerspricht dem Prinzip einer starken Klassenbindung, da *nicht mehr alle* Attribute immer benötigt werden, um den Zweck der Klasse zu erfüllen. Um eine starke Bindung sicherzustellen ist eine Umstrukturierung nötig. Es werden zwei Unterklassen FirmeninterneVeranstaltung und OeffentlicheVeranstaltung gebildet. Die Klasse Veranstaltung wird zur abstrakten Oberklasse:

```
import java.util.*;                                         Java

public abstract class Veranstaltung
{
    private int nummer;
    private short dauer;
    private Calendar vom, bis;
    private String ort, adresse;
    private short teilnehmerMax;
    private boolean storniert = false;
}

public class FirmeninterneVeranstaltung
    extends Veranstaltung
{
    private float pauschalpreis;
}

public class OeffentlicheVeranstaltung
    extends Veranstaltung
{
    private short teilnehmerMin;
    private short teilnehmerAktuell;
    private float stornogebühr;
}
```

In [Fowl05, S. 84] wird die temporäre Nutzung von Attributen – wie im obigen Beispiel – als *Temporary Field* bezeichnet.

In einem Verein wird zu jedem Mitglied neben den persönlichen Daten auch die Bankverbindung gespeichert, damit der Mitgliedsbeitrag per Lastschrift abgebucht werden kann:

```
public class Mitglied
{
    private String nachname, vorname;
    private String geldinstitut;
    private long blz, kontonr;
    ...
}
```

Bei einer Projektbesprechung teilt der Kassierer dem Projektleiter mit, dass er Fälle kennt, bei dem ein Mitglied dem Verein mitteilt, dass ab dem nächsten Jahr eine andere Bankverbindung gilt. In diesem Jahr soll noch von dem alten Konto abgebucht werden. Nachdem der Projektleiter diesen Sonderfall dem zuständigen Programmierer mitgeteilt hat, erweitert er die Klasse `Mitglied` wie folgt:

```
import java.util.*;

public class MitgliedErweitert
{
    private String nachname, vorname;
```

II 25 Restrukturieren (*refactoring*)

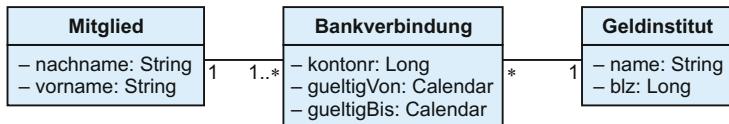
```
private String geldinstitut;
private long blz, kontonr;
private String geldinstitut2;
private long blz2, kontonr2;
private Calendar gueltigAb; //für Konto 2
...
}
```



Überlegen Sie, ob es eine bessere Lösung gibt.

Eine allgemeinere Lösung zeigt die Abb. 25.0-1. Die Daten des Geldinstituts werden in eine eigene Klasse ausgelagert. Zwischen die Klasse **Mitglied** und die Klasse **Geldinstitut** wird eine assoziative Klasse **Bankverbindung** gesetzt.

Abb. 25.0-1: UML-Klassendiagramm nach dem Refactoring.



Frage

Welche Vorteile hat diese Lösung?

Antwort

- Die Bankverbindung ist von der Klasse **Mitglied** gelöst, d.h. bei einer neu hinzukommenden Bankverbindung muss die Klasse **Mitglied** nicht mehr geändert werden.
- Dem Mitglied können beliebig viele Konten zugeordnet werden.
- Wenn zwei Mitglieder die gleiche Bankverbindung haben, können sie auf das gleiche Bankverbindung-Objekt verweisen, somit müssen Daten nicht redundant gespeichert werden.
- Für Bankverbindungen, die dem gleichen Geldinstitut zugehören, müssen die Daten des Geldinstituts ebenfalls nur einmal (und nicht mehrmals, und damit redundant) gespeichert werden.

Vorgehensweise

Das Umstrukturieren eines Programms bedeutet, ein lauffähiges Programm teilweise massiv zu verändern. Die Wahrscheinlichkeit, dass dabei Fehler entstehen ist groß. Es sollten daher zunächst Testfälle – wenn noch nicht vorhanden – aufgestellt werden und das bisherige Programm damit durchlaufen werden. Diese Testfälle müssen für **Regressionstests** aufbewahrt werden.

Wurden die Tests erfolgreich mit der bisherigen Programmversion durchlaufen, dann wird die Umstrukturierung vorgenommen – ohne die Funktionalität zu erweitern. Anschließend erfolgt der Regressionstest.

Erst nach dem erfolgreich verlaufenen Regressionstest werden die gewünschten Änderungen eingebaut und mit neuen Tests überprüft.

Restrukturierungs-Katalog

In [Fowl05] werden detailliert die Prinzipien des *Refactoring* sowie eine systematische Vorgehensweise beschrieben. Außerdem werden die verschiedenen Restrukturierungen in Form eines Katalogs zusammengestellt. Der Katalog wiederum fasst Restrukturierungen zu Kategorien zusammen. Um Ihnen eine Idee von möglichen Restrukturierungen zu geben, werden diese Kategorien mit ihren Restrukturierungen im Folgenden aufgelistet. Die deutsche Übersetzung wurde weitgehend aus dem Tutego-Refactoring-Katalog (<http://www.tutego.de/java/refactoring/catalog/>) übernommen. Dort werden auch die einzelnen Restrukturierungen erläutert.

■ Methoden zusammenstellen (*Composing Methods*)

Katalog

- Methode extrahieren (*Extract Method*)
- Methode inline setzen (*Inline Method*)
- Temporäre Variable entfernen (*Inline Temp*)
- Ersetze temporäre Variable durch Anfragemethode (*Replace Temp with Query*)
- Beschreibende Variable einführen (*Introduce Explaining Variable*)
- Trenne temporäre Variable (*Split Temporary Variable*)
- Entferne Zuweisung an Parametervariable (*Remove Assignments to Parameters*)
- Ersetze eine Methode durch ein Methoden-Objekt (*Replace Method with Method Object*)
- Ersetze Algorithmus (*Substitute Algorithm*)

■ Eigenschaften zwischen Objekten verschieben (*Moving Features Between Objects*)

- Verschiebe Methode (*Move Method*)
- Verschiebe Attribut (*Move Field*)
- Extrahiere Klasse (*Extract Class*)
- Setze Klasse inline (*Inline Class*)
- Verstecke den Delegierer (*Hide Delegate*)
- Entferne Klasse in der Mitte (*Remove Middle Man*)
- Führe fremde Methode ein (*Introduce Foreign Method*)
- Führe lokale Erweiterung ein (*Introduce Local Extension*)

■ Daten organisieren (*Organizing Data*)

- Kapsle eigene Attributzugriffe (*Self Encapsulate Field*)
- Ersetze eigenes Attribut durch Objektverweis (*Replace Data Value with Object*)
- Ersetze Wert durch Referenz (*Change Value to Reference*)
- Ersetze Verweis durch Wert (*Change Reference to Value*)
- Ersetze Feld durch ein Objekt (*Replace Array with Object*)
- Beobachtete Daten verdoppeln (*Duplicate Observed Data*)
- Unidirektionale Beziehung in bidirektionale Beziehung ändern (*Change Unidirectional Association to Bidirectional*)

II 25 Restrukturieren (*refactoring*)

- Bidirektionale Beziehung in unidirektionale Beziehung ändern (*Change Bidirectional Association to Unidirectional*)
- Ersetze magische Zahl durch symbolische Konstante (*Replace Magic Number with Symbolic Constant*)
- Kapsle Attribut (*Encapsulate Field*)
- Kapsle die Kollection (*Encapsulate Collection*)
- Ersetze einen Datensatz durch eine Datenklasse (*Replace Record with Data Class*)
- Ersetze Typschlüssel durch eine Klasse (*Replace Type Code with Class*)
- Ersetze typspezifischen Code durch Unterklassen (*Replace Type Code with Subclasses*)
- Ersetze typspezifischen Code durch Zustands-/Strategie-Muster (*Replace Type Code with State/Strategy*)
- Ersetze Unterklaasse durch Attribute (*Replace Subclass with Fields*)
- Bedingte Ausdrücke vereinfachen** (*Simplifying Conditional Expressions*)
 - Zerlege Bedingung (*Decompose Conditional*)
 - Bedingungen von Ausdrücken zusammenführen (*Consolidate Conditional Expression*)
 - Wiederholte Anweisungen aus Bedingungen zusammenführen (*Consolidate Duplicate Conditional Fragments*)
 - Kontroll-Schalter entfernen (*Remove Control Flag*)
 - Geschachtelte Bedingungen durch Wächter ersetzen (*Replace Nested Conditional with Guard Clauses*)
 - Ersetze Fallunterscheidungen durch Polymorphie (*Replace Conditional with Polymorphism*)
 - Null-Objekt einführen (*Introduce Null Object*)
 - Zusicherungen einführen (*Introduce Assertion*)
- Methodenaufrufe vereinfachen** (*Making Method Calls Simpler*)
 - Methode umbenennen (*Rename Method*)
 - Parameter hinzufügen (*Add Parameter*)
 - Parameter entfernen (*Remove Parameter*)
 - Trenne Anfrage vom Verändern (*Separate Query from Modifier*)
 - Parametrisiere Methode (*Parameterize Method*)
 - Ersetze Parameter durch explizite Methoden (*Replace Parameter with Explicit Methods*)
 - Vorhandenes Objekt übergeben (*Preserve Whole Object*)
 - Ersetze Parameter durch Methode (*Replace Parameter with Method*)
 - Parameterobjekt einführen (*Introduce Parameter Object*)
 - Entferne schreibende Attribut-Methode (*Remove Setting Method*)
 - Verstecke Methode (*Hide Method*)
 - Ersetze Konstruktor durch Fabrikmethode (*Replace Constructor with Factory Method*)

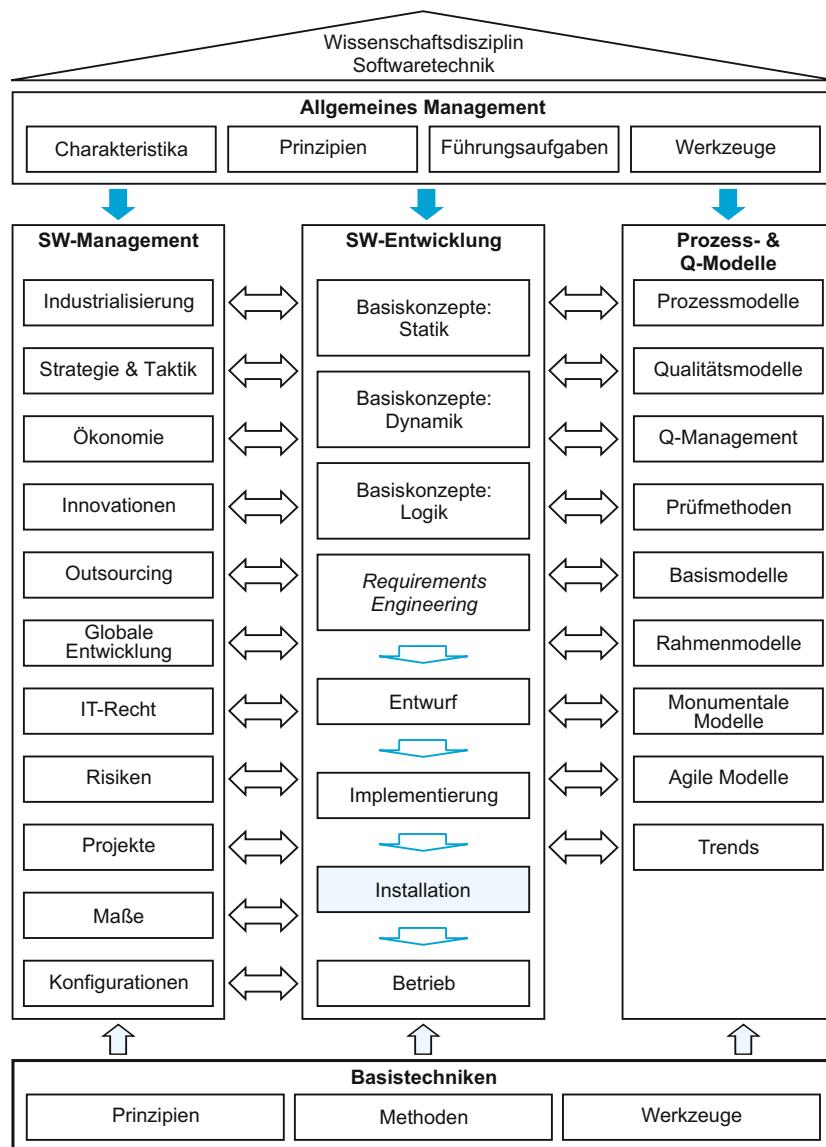
25 Restrukturieren (*refactoring*) II

- Explizite Typanpassung kapseln (*Encapsulate Downcast*)
- Ersetze Fehlerabfragen durch Ausnahmen (*Replace Error Code with Exception*)
- Ersetze Ausnahmen durch Tests (*Replace Exception with Test*)
- Umgang mit Generalisierung** (*Dealing with Generalization*)
 - Attribut hochziehen (*Pull Up Field*)
 - Methode hochziehen (*Pull Up Method*)
 - Konstruktorrumpf hochziehen (*Pull Up Constructor Body*)
 - Methode nach unten verlagern (*Push Down Method*)
 - Attribut nach unten verlagern (*Push Down Field*)
 - Extrahiere UnterkLASSE (*Extract Subclass*)
 - Extrahiere OberKLASSE (*Extract Superclass*)
 - Extrahiere Schnittstelle (*Extract Interface*)
 - Hierarchie reduzieren (*Collapse Hierarchy*)
 - Erstelle Schablonenmethode (*Form Template Method*)
 - Ersetze Vererbung durch Delegation (*Replace Inheritance with Delegation*)
 - Ersetze Delegation durch Vererbung (*Replace Delegation with Inheritance*)

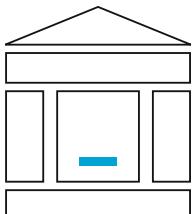
[SGS+10], [Cold04]

Literatur

III Verteilung, Installation, Abnahme und Einführung



III III Verteilung, Installation, Abnahme und Einführung



Wird ein Softwaresystem für einen individuellen Auftraggeber erstellt (**Individual-Software**), dann muss es nach der Fertigstellung auf die Infrastruktur beim Auftraggeber verteilt und installiert werden. Anschließend erfolgt eine Abnahme des Softwaresystems durch den Auftraggeber und die Einführung beim Auftraggeber.

Handelt es sich um eine **Standard-Software**, dann muss die Abnahme durch eine freigabeberechtigte Instanz innerhalb der Firma erfolgen. Anschließend wird in der Regel das Softwaresystem für Pilotkunden freigegeben, um Erfahrungen bei der Installation und bei der Einführung zu sammeln. Sind diese Erfahrungen positiv, dann wird das Softwaresystem für den Vertrieb freigegeben.

i In Abhängigkeit von der Art des Softwaresystems, sind unterschiedliche Aktivitäten bei der Verteilung und Installation durchzuführen:

■ »Verteilung und Installation«, S. 521

Ein verteiltes und installiertes Softwaresystem muss anschließend noch abgenommen und eingeführt werden:

■ »Abnahme und Einführung«, S. 525

26 Verteilung und Installation

Nach der Fertigstellung eines Softwaresystems muss es verteilt und installiert werden.

Vorbereitungen

Bevor ein Softwaresystem verteilt werden kann, müssen einige Vorbereitungen getroffen werden:

- Es muss überlegt werden, auf welche Art und Weise das Softwaresystem auf dem oder den Zielsystemen installiert werden soll:
 - Soll ein allgemeines **Installationsprogramm** (*installer*) oder ein spezielles verwendet werden?
 - Welches Datei- und welches Programmformat werden für das Installationsprogramm benötigt?
 - Soll der Benutzer oder ein Administrator die Installation vornehmen?
 - Lässt sich das Installationsprogramm nachträglich durch den Administrator anpassen?
 - Müssen die Installationsdateien vor unbefugten Zugriff geschützt werden?
 - Welche Rechte und Berechtigungen werden zur Durchführung der Installation benötigt?
 - Soll der Installationsstatus netzwerkweit überprüft werden können, z. B. mithilfe der verwendeten Verteilsoftware.
- Es muss überlegt werden, auf welche Art und Weise das Softwaresystem verteilt werden soll:
 - Soll eine **Verteilsoftware** eingesetzt werden?
 - Wie muss das zu verteilende Softwaresystem dafür paketiert werden (Erstellung von Archiven usw.)?
 - In einem Konfigurationsverwaltungssystem muss eine Liste der Pakete hinterlegt werden, die durch die Verteilsoftware installiert werden sollen, einschl. der Parameter zur Konfiguration.

Verteilung

Je nach Art der Software, wird unter Verteilung – auch Software-Distribution genannt – etwas Unterschiedliches verstanden (siehe auch »Einflussfaktoren auf die Architektur«, S. 135):

III 26 Verteilung und Installation

- Bei einer **Einzelplatz-Anwendung** muss das zu installierende Softwaresystem auf einem Datenträger (CD, DVD, USB-Stick) oder über das Netz (Download) dem Einzelplatz-Computer zur Verfügung gestellt werden.
- Bei einer **Client-Server-Anwendung** besteht die zu verteilende Software aus mindestens zwei Teilen. Ein Teil muss auf alle Clients verteilt und installiert werden, der andere Teil auf einem oder mehreren Servern.
- Bei einer **Web-Anwendung** wird die zu verteilende Software auf einen oder mehrere Servern verteilt. Die Clients greifen über Webbrowser auf das Softwaresystem zu.

Bei großen Unternehmen muss die Software u. U. auf mehr als 10.000 Clients verteilt werden.

Hinweis

Wie ein Softwaresystem auf unterschiedliche Rechnerknoten verteilt werden soll, kann in der UML durch ein sogenanntes **Verteilungsdiagramm** (*deployment diagram*) dargestellt werden (siehe »Verteilungsdiagramme«, S. 11).

Installation

Bei der **Installation** (*installation, setup*) wird ein neues Softwaresystem (einschl. Treibern, Plug-ins usw.) auf ein vorhandenes Computersystem kopiert, eventuell konfiguriert und so vorbereitet, dass es ausgeführt werden kann. Ein Installationsprogramm (*installer*) automatisiert dabei die meisten Tätigkeiten, die für die Installation durchgeführt werden müssen.

Zur Installation gehören folgende Tätigkeiten:

- | | |
|----------|---|
| Prüfung | <ul style="list-style-type: none">■ Prüfung, ob alte Softwareversionen auf dem Computersystem vorhanden sind und eventuelle Deinstallation.■ Prüfung, ob das zu installierende Softwaresystem für das Computersystem geeignet ist (Überprüfung der Hardwareausstattung, der Betriebssystemversion und anderer bereits installierter Komponenten).■ Prüfung, welche Dateien, Bibliotheken, Komponenten und Konfigurationsdaten benötigt werden.■ Prüfung der Integrität der Installationsdateien z. B. über eine Prüfsumme. |
| Kopieren | <ul style="list-style-type: none">■ Entpacken und Kopieren der Dateien in ein (neues) Verzeichnis auf dem Zielcomputer. Einige Dateien müssen oft auch in allgemeine Verzeichnisse oder Verzeichnisse des Betriebssystems kopiert werden.■ Bibliotheken und Komponenten, die durch mehrere Softwaresysteme genutzt werden, müssen gegebenenfalls mit installiert, registriert und ältere Versionen aktualisiert werden. |

26 Verteilung und Installation III

- Bei der Installation werden oft auch notwendige computerspezifische und/oder benutzerspezifische Einstellungen vorgenommen, z. B. Sprache und Land.
- Wird ein Verteilsystem verwendet, dann meldet der Client den Erfolg oder Misserfolg der Installation und Konfiguration.
- In einer Lizenzverwaltungs-Software wird gespeichert, welche Installationen auf welchem Computersystem vorgenommen wurden.

Neben der Erstinstallation haben sich verschiedene Verfahren zur Aktualisierung einer Installation etabliert:

- **Inkrementelle Installation:** Das Softwaresystem prüft periodisch oder durch einen Aufruf des Benutzers z. B. anhand einer Liste im Internet, ob die installierte Version noch aktuell ist. Sind Aktualisierungen verfügbar, dann fragt das Softwaresystem beim Benutzer nach und lädt dann gezielt diejenigen Dateien herunter, die zu aktualisieren sind.
- **Automatische Aktualisierung:** Bei jedem Start des Softwaresystems wird automatisch überprüft, ob die installierte Version noch aktuell ist. Wenn nicht, dann werden ohne Rückfrage des Benutzers Aktualisierungen automatisch heruntergeladen und installiert.
- **Serverinstallation:** Das Softwaresystem wird auf einem Server abgelegt. Über einen Link wird das Softwaresystem vom Client aus über das Netzwerk in den Arbeitsspeicher des Client geladen.

Terminologie

Der englische Begriff **Deployment** wird auch im Deutschen oft unverändert verwendet. Unter *Deployment* werden oft alle Aktivitäten verstanden, die ein Softwaresystem für einen Benutzer einsetzbar machen. Wird der Begriff so weit gefasst, dann umfasst er Verteilung, Installation und teilweise auch die Einführung.

Wird der Begriff im engeren Sinne verwandt, dann bezieht er sich auf die Verteilung der Software.

Unter dem Begriff **Rollout** (ausrollen) wird oft auch das Verteilen von Softwaresystemen verstanden, bisweilen aber auch die breite Einführung in einem Unternehmen oder die Markteinführung.

27 Abnahme und Einführung

Die Abnahme

Ein fertiggestelltes, verteiltes und installiertes Softwaresystem muss abgenommen werden. Folgende Tätigkeiten sind dafür auszuführen:

- **Übergabe** des Gesamtsystems einschließlich der gesamten Dokumentation an den Auftraggeber, falls das Produkt individuell für ihn erstellt wurde, oder an eine freigabeberechtigte Instanz innerhalb der Firma, falls das Softwaresystem für einen anonymen Markt hergestellt wurde.
- Mit der Übernahme verbunden ist im Allgemeinen ein **Abnahmetest**. Im Rahmen der Abnahme werden eine Reihe von Tests durchgeführt, die der Abnehmer ausgearbeitet hat und mit denen er prüft, ob sich das Softwaresystem entsprechend seinen ursprünglichen spezifizierten Anforderungen (in der Spezifikation festgehalten) verhält.
- Innerhalb einer Abnahme-Testserie ist es auch sinnvoll, **Belastungs-** oder **Stresstests** durchzuführen. Beim Stresstest eines Textverarbeitungssystems kann man z. B. überprüfen, ob auch bei sehr hohen Schreibgeschwindigkeiten kein Fehler auftritt. Insgesamt werden die Testfälle so zusammengestellt, dass ein komprimierter Test der Produktfunktionen – im Allgemeinen mit Echtdaten – möglich ist.

Das Ergebnis der Abnahme ist ein Abnahmeprotokoll. In ihm werden alle relevanten Eingabedaten, durchgeföhrten Tests und erhaltenen Ergebnisse festgehalten.

Nach erfolgreichen Tests erfolgt die Abnahme des Produkts durch den Auftraggeber. Die formale **Abnahme** ist die (schriftliche) Erklärung der Annahme (im juristischen Sinne) eines Produkts durch den Auftraggeber.

Handelt es sich um einen externen Auftraggeber, dann hängt der Abnahmetest auch davon ab, ob

- der Auftraggeber das Softwaresystem nur nutzt, aber nicht wartet und pflegt, oder ob
- der Auftraggeber das Softwaresystem nutzt und selbst wartet und pflegt.

Welche Alternative der Auftraggeber wählt, sollte natürlich bereits bei der Auftragsvergabe bekannt sein. Die für den Auftraggeber relevanten nichtfunktionalen Anforderungen hängen von der gewählten Alternative ab.

Abnahmeprotokoll

Abnahme

Wartung & Pflege
beim
Auftraggeber?

III 27 Abnahme und Einführung

Für die Produktnutzung sind die nichtfunktionalen Anforderungen Benutzbarkeit, Leistung und Effizienz sowie Sicherheit und Zuverlässigkeit wesentlich, während für die Wartung & Pflege die nichtfunktionalen Anforderungen Wartbarkeit, Weiterentwickelbarkeit und Portabilität hinzu kommen (siehe »Nichtfunktionale Anforderungen«, S. 109).

Beim Abnahmetest muss die Erfüllung der nichtfunktionalen Anforderungen natürlich so weit wie möglich überprüft werden.

Führt der Auftraggeber die Wartung & Pflege selbst durch, dann benötigt er natürlich auch die gesamte Spezifikations-, Entwurfs- und Implementierungsdokumentation sowie eine sorgfältige Einführung in die Softwarearchitektur.

Aus der Sicht des Auftraggebers ist es beim Abnahmetest *nicht* möglich, das Softwaresystem vollständig zu testen. Daher ist es umso wichtiger, dass bei der Auftragsvergabe geprüft wird, ob beim Auftragnehmer die Softwareentwicklung nach einem definierten und »gelebten« Softwareprozess durchgeführt wird.

Die Einführung

Bei einer Software für den anonymen Markt erfolgt nach der Abnahme die **Markteinführung**.

Handelt es sich um ein auftragsbezogenes Softwaresystem, dann erfolgt nach der Abnahme die Einführung des Softwaresystems beim Auftraggeber. Folgende Tätigkeiten sind durchzuführen:

- **Einrichtung** des Softwaresystems in dessen Zielumgebung zum Zwecke des Betriebs.
- **Schulung** der Benutzer und des Betriebspersonals. Nach der Installation des Softwaresystems sind die Benutzer in die Handhabung des Softwaresystems einzuleiten.
- **Inbetriebnahme** des Softwaresystems. Darunter versteht man den Übergang zwischen Installation und Betrieb.

Einführungsprotokoll
Alle Vorkommnisse, die bei der Einführung auftreten, werden in einem **Einführungsprotokoll** festgehalten. Die Einführung eines Softwaresystems muss sorgfältig geplant werden. Umfangreiche Produkteinführungen sind wie Innovationseinführungen zu behandeln. Dementsprechend sind die allgemeinen Charakteristika zu beachten, die bei Innovationseinführungen eine Rolle spielen.

Besonders wichtig ist die **Zeitplanung** der Umstellung. Netzpläne können dafür sinnvoll eingesetzt werden.

Umstellung der Datenbestände
Eine wichtige Aufgabe ist die Umstellung der Datenbestände. Manuelle Karteien müssen oft erst entsprechend aufbereitet oder zusammengestellt werden, bevor sie in der Form vorliegen, in der sie für die neue Datenverwaltung erfasst werden können. Bei umfangreichen Beständen muss für die manuelle Datenerfassung entspre-

27 Abnahme und Einführung III

chend Zeit eingeplant werden. Je größer der Umfang eines Datenbestandes ist, desto früher ist eine Umstellung erforderlich. Eine hohe Änderungsintensität spricht für eine späte Umstellung.

Da Daten heute in der Regel in digitaler Form vorliegen, müssen Transformationsprogramme und Importfilter geschrieben werden, um sie in dem neuen Softwaresystem verwenden zu können.

Das größte Problem ergibt sich bei der Übertragung »lebender« Datenbestände, z. B. Lagerdateien. Hier muss zu einem bestimmten Zeitpunkt oder zu mehreren Zeitpunkten umgestellt werden.

Es ist zu berücksichtigen, dass zur Erstellung neuer Bestände zum Teil eigene Programme erforderlich sind, die entwickelt werden müssen. Außerdem ist zu überlegen, wie die Richtigkeit der erstellten Datenbestände überprüft werden kann.

Die eigentliche Inbetriebnahme kann auf drei Arten erfolgen:

- direkte Umstellung,
- Parallelauf,
- Versuchslauf.

Inbetriebnahme

Bei der **direkten Umstellung** wird unmittelbar von dem alten auf das neue System übergegangen. Die Benutzung des alten Systems wird gestoppt, um das neue System sofort in Betrieb zu nehmen. Für die Umstellungsarbeiten wird ein Wochenende oder eine Feiertagsperiode gewählt. Die direkte Umstellung ohne weitere Vorkehrungen ist risikoreich und sollte vermieden werden.

Direkte Umstellung

Beim Parallelauf werden die Bewegungsdaten sowohl im alten als auch im neuen System verarbeitet, sodass die Ergebnisse miteinander verglichen werden können. Der Vorteil des Parallelaufs liegt darin, dass man Sicherheit hat, falls das neue System nicht funktioniert. Nachteilig sind die hohen Kosten und die Schwierigkeiten, die durch den Parallelauf zweier Systeme entstehen.

Parallelauf

Versuchsläufe können in zwei Formen durchgeführt werden:

- Bei dem einen Verfahren arbeitet das neue System mit Daten aus vergangenen Perioden, sodass die Ergebnisse bekannt sind und überprüft werden können. In der Zeit der Versuchsläufe wird der Benutzer aufgefordert, Beanstandungen und Fehler mitzuteilen. Die aktuelle Verarbeitung erfolgt noch im alten System.
- Die zweite Möglichkeit ist die Einführung des neuen Systems in einzelnen Stufen, in denen verschiedene Funktionsbereiche sukzessiv übernommen werden.

Versuchslauf

Wird ein Softwaresystem für den anonymen Markt hergestellt – dies ist in der Regel bei Standard-Software der Fall – dann erfolgen vor einer allgemeinen Vertriebsfreigabe eine Reihe von **Pilotinstallationen** bei Pilotkunden (Betatest).

Pilot-installationen

III 27 Abnahme und Einführung

Ende der Softwareentwicklung Nach erfolgreicher Einführung des Softwaresystems erfolgt die offizielle Freigabe des Produkts. Damit ist die Softwareentwicklung beendet. Das Softwaresystem wird sowohl in das Wartungs- als auch in das Produktarchiv übertragen.

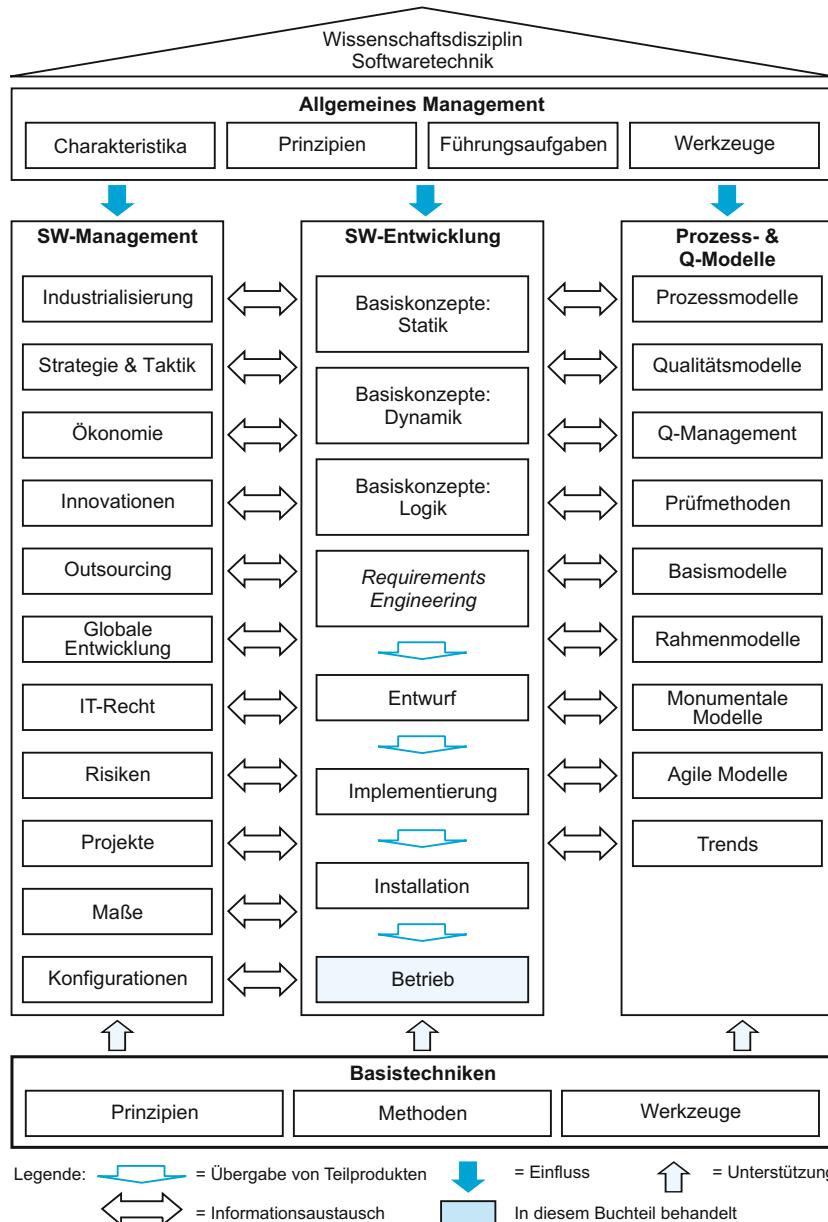
Nach der Abnahme und Einführung liegen also folgende Ergebnisse vor:

- Gesamtprodukt einschließlich Gesamtdokumentation,
- Abnahmeprotokoll,
- Einführungsprotokoll.

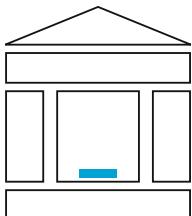
Gesamtprodukt Das **Gesamtprodukt** umfasst alle Produkte bzw. Teilprodukte, die in der Softwareentwicklung erstellt wurden. Um eine gute Grundlage für die spätere Wartung zu besitzen, ist es nötig, alle Produkte zu archivieren. Das Wartungsarchiv unterscheidet sich vom Produktarchiv vor allem darin, dass von jedem Softwaresystem verschiedene Versionen aufbewahrt werden müssen. Das Wartungsarchiv kann darüber hinaus noch Informationen über die installierten Versionen bei den einzelnen Kunden aufnehmen.

Einen Teil des Gesamtprodukts bilden die **Benutzerdokumentation** und das installierte System. Unter der Benutzerdokumentation ist die Gesamtheit der Unterlagen zu verstehen, die dem Anwender einen ordnungsgemäßen Betrieb des Systems ermöglichen.

IV Der Betrieb



IV IV Der Betrieb



Alterung

Mit der erfolgreichen Verteilung, Installation, Abnahme und Einführung eines Softwaresystems beginnt seine Wartung und Pflege. Nach der Inbetriebnahme eines Softwaresystems

- treten im täglichen Betrieb Fehler auf,
- ändern sich die Umweltbedingungen (neue Systemsoftware, neue Hardware, neue organisatorische Einbettung),
- entstehen neue Wünsche und Anforderungen (neue Funktionen, geänderte Benutzeroberfläche, erhöhte Geschwindigkeit).

Symptome

Software, bei der nicht ständig Fehler behoben und Anpassungen sowohl an die Umwelt als auch an neue Anforderungen vorgenommen werden, altert und ist irgendwann veraltet. Sie kann dann nicht mehr für den ursprünglich vorgesehenen Zweck eingesetzt werden. »Software veraltet in dem Maße, wie sie mit der Wirklichkeit nicht Schritt hält« [Snee83].

Folgende Symptome weisen auf potenzielle Probleme von »**Alt-systemen**« (*legacy system*) hin [DDN09, S. 4 f.]:

- Es gibt nur eine veraltete Dokumentation oder gar keine Dokumentation zu dem Softwaresystem.
- Es sind keine Testfälle vorhanden, um das System zu testen.
- Die ursprünglichen Entwickler und Benutzer des Systems sind nicht mehr verfügbar.
- Das Wissen über die Interna des Systems ist nicht mehr vorhanden – insbesondere wenn keine aktuelle und ausreichende Dokumentation vorhanden ist.
- Niemand hat einen Überblick über das gesamte System.
- Es dauert zu lange und ist zu aufwändig, um einfache Änderungen vorzunehmen.
- Es müssen ständig Fehler beseitigt werden.
- Wenn ein Fehler beseitigt ist, dann tritt an einer anderen Stelle ein neuer Fehler auf.
- Lange Übersetzungszeiten erschweren Änderungen.
- Es ist schwierig, kundenspezifische Anpassungen zu verwalten und in neuen *Release* zu berücksichtigen.
- Das System enthält zunehmend mehr duplizierten Code.
- Der Code »riecht schlecht« (*smells bad*), d. h. er enthält duplizierten Code, lange Methoden, große Klassen, lange Parameterlisten, *Switch*-Anweisungen und Datenklassen.

Aufgaben

Bei den im Betrieb durchzuführenden Aktivitäten lassen sich zwei unterschiedliche Tätigkeitsgruppen unterscheiden:

- **Wartungsaktivitäten**
 - Stabilisierung/Korrektur
 - Optimierung/Leistungsverbesserung
- **Pflegeaktivitäten**
 - Anpassung/Änderung
 - Erweiterung

In der amerikanischen Literatur werden Wartung und Pflege *nicht* unterschieden, sondern unter dem Begriff **Maintenance** subsumiert.

Hinweis

Betrachtet man den gesamten **Lebenszyklus** (*life cycle*) eines Softwaresystems, dann lässt sich der Aufwand für ein Softwaresystem aufteilen in den Entwicklungsaufwand und den Wartungs- & Pflegeaufwand. Verschiedene Untersuchungen erlauben folgende Aussagen:

- Der Aufwand für die Wartung & Pflege ist normalerweise größer als der Entwicklungsaufwand. »Faustregeln«
- Der Aufwand für die Wartung & Pflege ist typischerweise um einen Faktor von 2 bis 4 größer als der Entwicklungsaufwand für ein umfangreiches Softwaresystem.

Eine solche Aufwandsverteilung bedeutet, dass im Extremfall von 100 Mitarbeitern einer Software-Abteilung 80 Mitarbeiter mit der Wartung & Pflege »alter« Software beschäftigt sind und nur 20 Mitarbeiter neue Software entwickeln.

Die Wartung & Pflege als zeitlich der Entwicklung nachgeordnet, spürt die Auswirkungen einer guten oder schlechten Produktqualität natürlich am direktesten. Naheliegend sind zunächst zwei Maßnahmen, um den Wartungs- & Pflegeaufwand zu reduzieren:

- Nur ein Softwaresystem mit hoher Qualität freigeben – bezogen auf die nichtfunktionalen Anforderungen, die für die Wartung & Pflege relevant sind. D. h. Verbesserung des Softwareentwicklungsprozesses.
- Verbesserung der Produktivität der Wartung & Pflege.

Die Produktivität der Wartung & Pflege kann dadurch verbessert werden, dass Wartungsaktivitäten und Pflegeaktivitäten voneinander getrennt werden. Diese Trennung ist jedoch oft schwierig durchzuführen und durchzuhalten. Wenn ein Programm »angefasst« wird, dann werden in der Regel Fehlerkorrekturen, Optimierungen, Anpassungen und Erweiterungen in einem Durchgang ausgeführt. Da aber Wartungs- und Pflegeaktivitäten unterschiedliche Charakteristika besitzen, sollte auf jeden Fall eine Trennung erfolgen.

Trennung von
Wartung & Pflege

Anpassungen und Erweiterungen eines Softwaresystems sind auch charakteristisch für Weiterentwicklungen bzw. für neue Versionen von Produkten. Es kann daher sinnvoll sein – abgesehen von minimalen Änderungen – alle Pflegeaktivitäten den normalen Softwareentwicklungsprozess durchlaufen zu lassen.

Pflege = Weiter-
entwicklung

Insbesondere in agilen Prozessmodellen gibt es oft keine Pflegephase mehr, sondern Pflegeaktivitäten werden als Erstellung einer neuen Produktversion angesehen.

Untersuchungen haben gezeigt, dass der Wartungs & Pflegeaufwand sowohl mit dem Alter als auch mit dem Umfang des Softwaresystems zunimmt.

IV IV Der Betrieb

»Faustregel« Der Umfang wächst durchschnittlich um 10 Prozent pro Jahr. Die Bereitstellung zusätzlicher Merkmale und Funktionen trägt vor allem zu diesem Zuwachs bei.

Ältere Softwaresysteme tendieren daher dazu, umfangreicher und schwerer wartbar zu sein. Ab einem bestimmten Zeitpunkt stellen sich dann folgende Fragen:

- Soll das Softwaresystem weiter gewartet werden?
- Soll das Softwaresystem weiter gepflegt werden?
- Soll das Softwaresystem durch ein neues Softwaresystem ersetzt werden?

Ausschlaggebend für die Wahl der richtigen Alternative ist natürlich vor allem die Wirtschaftlichkeit, die wiederum im Wesentlichen davon abhängt, wie groß die »Lebenserwartung« des alten Softwaresystems noch ist.

i Die Wartung spielt eine entscheidende Rolle, damit ein in Betrieb befindliches Softwaresystem unter den vorgegebenen Bedingungen seine Leistung erfüllt bzw. beibehält:

- »Wartung«, S. 533

Da Softwaresysteme in der Regel länger im Einsatz sind als ursprünglich geplant und die Umgebung sich immer schneller ändert, nimmt die Bedeutung der Pflege zu:

- »Pflege«, S. 537

Um Softwaresysteme anpassen und erweitern zu können, ist bei Altsystemen in der Regel zunächst ein *Reverse Engineering* erforderlich:

- »Reverse Engineering«, S. 543 *Reverse Engineering*

Auf der Grundlage dieser Erkenntnisse ist es dann möglich, das Softwaresystem zu ändern. Nur durch eine Kultur des kontinuierlichen *Reengineering* ist es möglich, dauerhaft flexible und wartbare Systeme zu erhalten:

- »Reengineering (Teil 1)«, S. 551
- »Reengineering (Teil 2)«, S. 557
- »Reengineering (Teil 3)«, S. 561

28 Wartung

Wartung beschäftigt sich mit der Lokalisierung und Behebung von Fehlerursachen bei in Betrieb befindlichen Softwaresystemen, wenn die Fehlerwirkung bekannt ist.

Wartungsaktivitäten lassen sich folgendermaßen charakterisieren: Charakteristika

- Ausgangsbasis ist ein fehlerhaftes bzw. inkonsistentes Produkt.
- Abweichungen zwischen Subsystemen sind zu lokalisieren und zu beheben.
- Die Korrektur einzelner Fehler hat nur begrenzte Auswirkungen auf das Gesamtsystem.
- Die Fehlerkorrekturen konzentrieren sich im Allgemeinen auf die Implementierung, d. h. auf die Programme.
- Ereignisgesteuert, d.h. nicht vorhersehbar und daher schwer planbar und kontrollierbar.

Die Wartungsaktivitäten lassen sich gliedern in:

- Stabilisierung/Korrektur
- Optimierung/Leistungsverbesserung

Die Wartungsaktivitäten werden erleichtert, wenn ein Softwaresystem die nichtfunktionale Anforderung »Wartbarkeit« erfüllt (siehe »Wartbarkeit«, S. 116). Wartbarkeit

Stabilisierung/Korrektur

Unter Stabilisierung/Korrektur fallen alle Tätigkeiten, die dazu dienen, Fehler zu beheben. Es kann sich dabei um Fehler handeln, die bereits bei der Entwicklung in das Softwaresystem gelangt sind, oder um Fehler, die bei der Wartung neu entstehen.

Da während der Softwareentwicklung ein allumfassender Test aller Funktionen eines komplexen Produkts oft nicht wirtschaftlich vertretbar ist, wird vielfach nur eine minimale Testabdeckung erreicht.

Softwaresysteme werden daher mit durchschnittlich 0,2 bis 0,05 Prozent Defekten pro 1000 Anweisungen freigegeben. Bei einem Produkt mit 1 Million Zeilen sind das immerhin zwischen 50 und 200 Defekte. Nur ein Teil dieser Fehler wird vor der Inbetriebnahme entdeckt. Die meisten bleiben verborgen und werden erst im Betrieb festgestellt. Die Beseitigung dieser Fehler verursacht erhebliche Kosten. Restfehler i.e.S.

Die Lokalisierung und Behebung dieser Restfehler kann man als Wartung Wartung im engeren Sinne bezeichnen, obwohl dies eigentlich eine Restarbeit der Entwicklung ist.

IV 28 Wartung

Wartungsfehler Besonders schnell vermehren sich Wartungsfehler, die sogenannten *Second Level Defects*. Sie machen bald die Mehrzahl der Fehler aus. Ursache ist die schlechte Konstruktion und Fehleranfälligkeit des ursprünglichen Softwaresystems.

Beispiel Bei früheren Softwareentwicklungen wurden die Anforderungen nur in einem Pflichtenheft festgehalten. Eine formale Spezifikation wurde weder erstellt noch auf Vollständigkeit, Konsistenz und Eindeutigkeit überprüft. Sonderfälle wurden daher oft übersehen und dementsprechend auch nicht implementiert. Ein freigegebenes Softwaresystem »lief« daher so lange, wie Sonderfälle nicht auftraten. »Stürzte« das Softwaresystem beim ersten Sonderfall ab, dann kam der Wartungsprogrammierer und ergänzte die Implementierung um einen »Rucksack« der Art `if Sonderfall then ...` Trat ein weiterer Sonderfall auf, dann wurde ein weiterer »Rucksack« angehängt. Dadurch wurde das Programm immer unübersichtlicher und schlechter wartbar. An unerwarteten Stellen traten plötzlich Folgefehler auf.

Optimierung/Leistungsverbesserung

Frisch eingesetzte Software ist nicht nur unzuverlässig, sondern sie verbraucht auch mehr Zeit und Speicher, als zur Erfüllung ihrer Aufgaben erforderlich ist. Softwaresysteme werden selten vor der ersten Freigabe optimiert. Sobald ein Softwaresystem funktionsfähig ist, wird es oft freigegeben. Die Optimierung bleibt der Wartung vorbehalten.

Zur Optimierung gehören alle Aktivitäten, die dazu dienen, die Leistung des Softwaresystems zu verbessern. *Tuning*, *Monitoring* und Reduzierung des Speicherbedarfs sind entsprechend durchzuführende Aufgaben. Zum Teil sind auch Restrukturierungen erforderlich, um die Leistungsverbesserungen zu erreichen.

Organisation

Um eine geordnete Abwicklung der Wartungsaufgaben sicherzustellen, ist ein geeignetes **Konfigurations- und Änderungsmanagement** erforderlich. Im »Lehrbuch der Softwaretechnik – Softwaremanagement« werden im Kapitel 15 »Konfigurationen und Änderungen managen« diese Themen ausführlich behandelt.

Eine wichtige organisatorische Frage ist, ob die Wartung eigenständig oder Teil der Entwicklung sein soll.

Frage Überlegen Sie, welche Argumente für und gegen eine von der Entwicklung getrennte Wartungsorganisation sprechen.

Antwort Für eine von der Entwicklung getrennte Wartungsorganisation sprechen folgende Argumente:

- ⊕ Klare Zuordnung der Wartungs- und Entwicklungskosten.
- ⊕ Entlastung der Entwickler von Wartungsaufgaben und insbesondere von paralleler Durchführung unterschiedlicher Tätigkeiten.
- ⊕ Qualitativ besserer Abnahmetest durch das Wartungsteam.
- ⊕ Besserer Kundenservice durch Konzentration auf die Wartung.
- ⊕ Einstellung spezialisierter Mitarbeiter bzw. gezielte Ausbildung der Mitarbeiter.
- ⊕ Effizientere Kommunikation zwischen den Wartungsmitarbeitern.
- ⊕ Höhere Produktivität durch Spezialisierung und zusammenhängende Produktkenntnisse.

Gegen eine eigenständige Wartungsorganisation sprechen allerdings folgende Gründe:

- Wartungsarbeiten können ein »schlechtes Image« bekommen, wodurch die Motivation der Mitarbeiter sinkt.
- Beim Übergang von der Entwicklung zur Wartung geht wertvolles Wissen über das Softwaresystem verloren.
- Koordinationsprobleme zwischen Entwicklung und Wartung, insbesondere, wenn neue Softwaresysteme alte ersetzen.
- Die Entwickler müssen nicht die Konsequenzen ihrer Entwicklung tragen.
- Die Wartungsmitarbeiter müssen sich aufwändig in das Softwaresystem einarbeiten.
- Eine gleichmäßige Auslastung der Mitarbeiter ist schwierig zu erreichen.

Die Vor- und Nachteile zeigen, dass es keine perfekte Organisation gibt. Ein geeigneter Kompromiss kann darin bestehen, getrennte Organisationen zu haben, die Mitarbeiter aber zwischen beiden Organisationseinheiten »rotieren« zu lassen.

Insgesamt betrachtet hängt der Erfolg der Wartung weniger von der Softwaretechnik, sondern vor allem von der Organisation und dem Management ab.

29 Pflege

Pflege beschäftigt sich mit der Lokalisierung und Durchführung von Änderungen und Erweiterungen von in Betrieb befindlichen Softwaresystemen, wenn die Art der gewünschten Änderungen/Erweiterungen festliegt.

Pflegeaktivitäten besitzen folgende Charakteristika:

Charakteristika

- Ausgangsbasis ist ein konsistentes Softwaresystem, in das gezielt – unter Beibehaltung der Konsistenz – Änderungen und Erweiterungen einzubringen sind.
- Die Bandbreite bei Änderungen und Erweiterungen kann von kleinen bis zu großen Modifikationen gehen.
- Änderungen und Erweiterungen sind in allen Teilprodukten (System-Spezifikation, -Entwurf, -Implementierung) durchzuführen.
- Planbar.

Die Pflegeaktivitäten lassen sich gliedern in:

- Anpassung/Änderung
- Erweiterung

Die Pflegeaktivitäten werden erleichtert, wenn ein Softwaresystem die nichtfunktionale Anforderung »Weiterentwickelbarkeit« erfüllt (siehe »Weiterentwickelbarkeit«, S. 119).

Weiterentwickelbarkeit

Anpassung/Änderung

Anpassungen werden durch Wandlungen in der Umwelt erzwungen:

- Änderungen in der technischen Umgebung, z. B. neue Systemsoftware,
- Änderungen in den Benutzungsoberflächen, z. B. modifizierte Fenster oder Formulare,
- Änderungen in den Funktionen, z. B. können Gesetzesänderungen neue betriebliche Regelungen bedingen.

Erweiterung

Erweiterungen führen zu einer funktionalen Ergänzung des Softwaresystems. Funktionen, die bei der Erstentwicklung vorgesehen oder geplant, aber nicht implementiert wurden, werden eingebaut. Oder es ergeben sich neue Funktionen aus den Erfordernissen des Betriebs der Software. Alle Pflegeaktivitäten fasst man technisch unter dem Begriff *Reengineering* zusammen.

Reengineering

Forward Engineering Bei der Neuentwicklung eines Softwaresystems findet in der Regel ein **Forward Engineering** statt (Abb. 29.0-1). Ausgehend von einem abstrakten, objektorientierten Analysemodell (OOA) wird durch zunehmende Konkretisierung eine objektorientierte Entwurfsmodell (OOD) erstellt, das dann implementiert (OOP) wird.

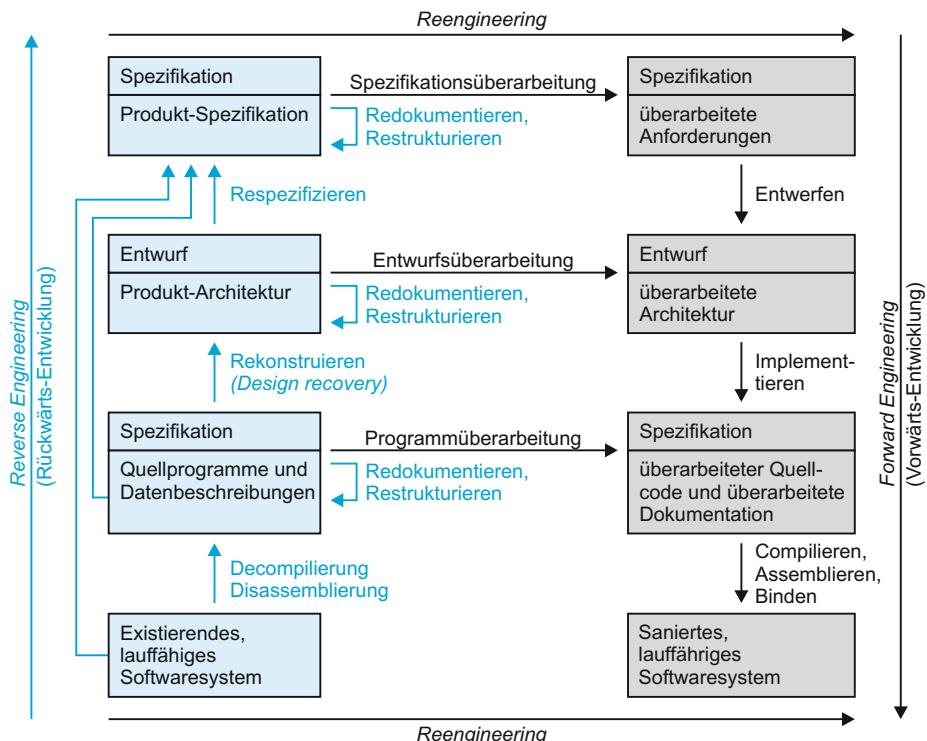


Abb. 29.0-1:
Forward-, Reverse-
und
Reengineering.

Reverse Engineering beschreibt den umgekehrten Vorgang. Ausgehend von einem konkreten, im Einsatz befindlichen Softwaresystem ist es das Ziel, durch die manuelle und/oder werkzeuggestützte Analyse des lauffähigen Systems oder des Quellcodes (unter Umständen nach einer Decompilierung bzw. Deassembly) die Systembeschreibung auf höheren Abstraktionsebenen oder die Verbesserung von Systembeschreibungen auf der jeweils gleichen Abstraktionsebene abzuleiten.

Rekonstruieren (Design Recovery) als Teil des Reverse Engineering ist der Versuch, aus vorliegendem Quellcode, existierenden Entwurfsdokumenten (wenn verfügbar), persönlichen Erfahrungen und allgemeinem Wissen über die Anwendungsdomäne eine Systemarchitektur wiederzugewinnen.

Die Rekonstruktion des Softwareentwurfs ist eine aufwändige, teure und kreative Tätigkeit. Durch ausschließlich automatische Analyse des Quellcodes lässt sich die Softwarearchitektur *nicht* ermitteln.

Respezifizieren ist der Versuch, aus dem unter Umständen rekonstruierten Entwurf und/oder dem Quellcode und/oder dem laufenden System eine Produkt-Spezifikation wiederzugewinnen. Diese Aufgabe ist in der Regel einfacher zu erledigen als eine Rekonstruktion, da das *Black Box*-Verhalten des Systems zu respezifizieren ist, während beim Rekonstruieren das System als *White Box* zu beschreiben ist.

In der Literatur wird Respezifizieren oft unter Rekonstruieren subsumiert, was aber von den Aufgaben her nicht gerechtfertigt ist.

Redokumentieren (*Redocumentation*) ist die Erstellung oder Revision einer semantisch äquivalenten Repräsentation innerhalb des selben Abstraktionsniveaus. Die neuen Dokumente enthalten in der Regel alternative Sichten, die dem Entwickler helfen, das System besser zu verstehen. Die Redokumentation ist die einfachste und älteste Form des *Reverse Engineering*.

- Um die Verschachtelung der Kontrollstrukturen besser zu überblicken, wird mit Hilfe eines Struktogramm-Generators von einem Quellprogramm ein Struktogramm erzeugt. Beispiele
- Die Programmdokumentation wird um mit einem Werkzeug gemessene Metrikwerte ergänzt.

Restrukturieren (*Restructuring*) ist die Transformation von einer Darstellungsform in eine andere auf demselben relativen Abstraktionsniveau, wobei das externe Verhalten des Systems (Funktionalität und Semantik) erhalten bleibt.

- Ein unstrukturiertes Programm mit Sprunganweisungen wird in ein strukturiertes Programm ohne Sprunganweisungen transformiert. Beispiele
- Ein nichtnormalisiertes Datenbankschema wird in ein normalisiertes Datenbankschema transformiert, das die ersten drei Normalformen erfüllt.

Restrukturierung ist oft eine Form der perfektionierenden Wartung, um den physikalischen Zustand des Altsystems bezogen auf bevorzugte Standards zu verbessern. Oft umfasst die Restrukturierung auch die Anpassung des Altsystems an neue Umgebungsbedingungen, die keine Veränderungen auf höheren Abstraktionsniveaus zur Folge haben.

IV 29 Pflege

Die Hauptaufgabe des *Reverse Engineering* ist es also, die Dokumente, die man bei einer Neuentwicklung als Standard erstellt, bei einem Altsystem nachträglich wiederzugewinnen, wenn sie verloren gegangen bzw. unvollständig sind, oder auf der Grundlage vorhandener Informationen neu zu erstellen.

Ziel dieser Aktivitäten ist es, das Verständnis des Altsystems und damit auch seine Qualität zu verbessern, um die Wartung zu erleichtern und die Weiterentwicklung und Wiederverwendung in anderen Systemen zu ermöglichen.

Durch das *Reverse Engineering* wird das Altsystem in seiner Funktionalität und Semantik selbst *nicht* verändert. Die Ergebnisse des *Reverse Engineering* bilden in der Regel die Voraussetzung für ein anschließendes *Reengineering*.

Reengineering

Reengineering umfasst alle Aktivitäten zur Änderung von Software-Altsystemen, um sie partiell oder komplett zu verbessern oder sie in einer neuen Form wieder implementieren zu können. Die Änderungen können bedeuten, dass neue Standards eingehalten, neue funktionale und nichtfunktionale Anforderungen ergänzt und gegebenenfalls auf eine neue Hardware- und/oder Systemplattform gewechselt wird.

Reengineering kann auf einer oder mehreren Ebenen stattfinden. Voraussetzung dazu ist eine vorangegangenes *Reverse Engineering*, wenn keine ausreichende Dokumentation auf allen Abstraktionsebenen zur Verfügung steht.

Die **Programmüberarbeitung** dient dazu, die Quellprogramme so zu überarbeiten, dass sie der neuen Form bzw. neuen Anforderungen entsprechen.

- Beispiele
- Aus einem Programm werden alle Ein-/Ausgabeanweisungen entfernt, und es wird so umstrukturiert, dass es ereignisgesteuert seine Aufgaben erledigt und den Informationsaustausch über Parameterlisten erledigt.
 - Ein Programm wird von C++ auf Java umgestellt und um generische Klassen und Methoden ergänzt.

Die **Entwurfsüberarbeitung** dient dazu, die Architektur des Altsystems an eine neue Form bzw. neue Anforderungen anzupassen.

- Beispiele
- Die Client-Server-Architektur wird in eine Web-Architektur geändert.
 - Die Benutzeroberfläche wird von Java-Swing auf Java-ServerFaces umgestellt.

Die **Spezifikationsüberarbeitung** dient dazu, die Produkt-Spezifikation des Altsystems an die neue Form bzw. neue Anforderungen anzupassen.

Das Entity-Relationship-Modell wird in ein OOA-Modell transformiert Beispiel und um Methoden ergänzt.

Ein Ziel des *Reengineering* kann auch darin bestehen, Komponenten des Altsystems für neue und/oder andere Systeme wiederverwendbar zu machen. Aus einer ungeplanten Wiederverwendung wird dann eine geplante Wiederverwendbarkeit hergestellt. Im Anschluß an ein *Reengineering* sind oft noch *Forward Engineering*-Aktivitäten durchzuführen.

Bisweilen wird für den Begriff *Reengineering* auch synonym der Begriff Software-**Sanierung** verwendet. Sanierung steht aber auch als Oberbegriff für alle notwendigen *Reverse Engineering*-, *Reengineering*- und *Forward Engineering*-Maßnahmen, um ein saniertes, lauffähiges System zu erhalten, das definierte neue Ziele erfüllt.

30 Reverse Engineering

Um ein vorhandenes Softwaresystem mit unzureichender oder fehlender Dokumentation zu verstehen, ist ein **Reverse Engineering** erforderlich, das in mehreren Schritten durchgeführt werden kann [DDN09, S. 17 ff.]:

- 1** Festlegung der Marschrichtung
- 2** Einarbeitung in das System
- 3** Erstes Verstehen des Systems
- 4** Erstellen eines detaillierten Modells

Festlegung der Marschrichtung

- Zunächst muss entschieden werden, welche Probleme zuerst angegangen werden sollen. Bewährt hat sich, mit den Aspekten zu beginnen, die dem Kunden am meisten nützen.
- Ziel ist es, sich auf die Problemursachen zu konzentrieren, nicht auf die Symptome.
- Es sollten nur die Teile des Systems betrachtet werden, die »kaputt« sind, d. h. die nicht mehr an geplante Änderungen angepasst werden können. *If it ain't broke, don't fix it.*
- Es sollte immer eine adäquate, aber einfache Lösung anstelle einer allgemeineren, aber komplexeren Lösung gefunden werden.
- Es müssen die Prioritäten für das Projekt festgelegt und die Prinzipien identifiziert werden, damit das Team die richtige Richtung beibehält.
- Eine Person sollte als »Navigator« dafür verantwortlich sein, dass die neue Architektur-Vision im Auge behalten wird.
- Damit das Team synchronisiert arbeitet, sollten kurze, reguläre *Round Table Meetings* abgehalten werden.

Einarbeitung in das System

- Einen ersten Einblick in die Historie und den Kontext des Systems erhält man durch **Diskussionen mit den Wartungsmitarbeitern**. Folgende Fragen helfen, das System zu verstehen:
- Welches war der leichteste und welches der schwerste Fehler, der im letzten Monat behoben wurde? Wie viel Zeit wurde benötigt, um den jeweiligen Fehler zu finden? Warum war es einfach oder so schwierig den Fehler zu lokalisieren?
- Wie gut ist der Code? Wie zuverlässig ist die Dokumentation?

IV 30 Reverse Engineering

- Der Zustand des Systems sollte durch ein kurzes, aber intensives **Code Review** abgeschätzt werden (Durchsicht des gesamten Codes in 1 Stunde). Anhand einer Checkliste sollten folgende Punkte beachtet werden:
 - Wenn das Entwicklungsteam Codierungs-Richtlinien eingehalten hat, dann sollte insbesondere auf Namenskonventionen geachtet werden, denn sie erlauben es, den Code schnell zu überfliegen.
 - Wenn funktionale Tests und Modultests vorhanden sind, dann erlauben sie Rückschlüsse auf die Funktionalität des Systems. Außerdem helfen sie beim Reengineering zu überprüfen, ob das System so funktioniert wie erwartet.
 - Abstrakte Klassen und Methoden geben Hinweise auf Entwurfs-Intentionen.
 - Hoch in der Hierarchie angeordnete Klassen definieren Abstraktionen der Domäne.
 - Taucht das Singleton-Muster auf, dann repräsentiert es oft Informationen, die für die gesamte Ausführung des Systems konstant sind.
 - Umfangreiche Strukturen weisen oft auf wichtige Funktionalitäts-Blöcke hin.
 - Kommentare lassen oft Entwurfs-Intentionen erkennen, können aber auch in die Irre führen.

Tipp ■ Es ist wichtig, das Vokabular der Entwickler zu erlernen, das innerhalb des Softwaresystems verwendet wird, um es zu verstehen und mit den Entwicklern zu kommunizieren.

Hinweis ■ Pro Stunde können ca. 10.000 Zeilen Code durchgesehen werden [DDN09, S. 55].

- Die zusätzlich vorhandene **Dokumentation** sollte in kurzer Zeit **überflogen** werden, um ihre Relevanz zu beurteilen. Dabei ist Folgendes zu beachten:
 - Inhaltsverzeichnisse geben einen schnellen Überblick über die Struktur und die dargestellte Information.
 - Versionsnummern und Datumsangaben geben Hinweise darauf, wie aktuell die Dokumentation ist.
 - Ein Abbildungsverzeichnis erlaubt es, schnell auf bestimmte Teile der Dokumentation zuzugreifen.
 - Bildschirmabzüge, Beispieldrucke, Beispielberichte und Kommandobeschreibungen geben Auskunft über die Funktionalität des Systems.
 - Formale Spezifikationen, wie Zustandsdiagramme, weisen – wenn vorhanden – auf wichtige Funktionalitäten hin.
 - Ist ein Index vorhanden, dann gibt er Hinweise auf die Terminologie der Autoren.

- Da es kostspielig ist, eine Dokumentation auf dem aktuellen Stand zu halten, werden oft nur die stabilen Teile eines Systems dokumentiert.
- Das laufende System sollte durch mehrere **Demonstrationen** unterschiedlicher Personen präsentiert werden und die Personen, die die Präsentation vorgenommen haben, sollten anschließend **interviewt** werden. Folgendes sollte dabei beachtet werden:
 - Ein Endbenutzer sollte verschiedene Szenarien aus der täglichen Praxis demonstrieren.
 - Ein Manager sollte darüber informieren, wie das System in die restliche Geschäftsdomäne passt.
 - Eine Person vom Vertrieb sollte die Software mit konkurrierenden Systemen vergleichen.
 - Eine Person vom *Help Desk* sollte demonstrieren, welche Eigenarten die meisten Probleme verursachen.
 - Ein Systemadministrator soll zeigen, was hinter den Kulissen abläuft (Start des Systems, Herunterfahren des Systems, Sicherungsverfahren, Archivieren von Daten usw.).
 - Ein Wartungs-Mitarbeiter und/oder ein Entwickler sollen demonstrieren, wie die Subsysteme arbeiten, miteinander kommunizieren und warum eine solche Architektur gewählt wurde.

Nach den Demonstrationen sollten in einem Bericht die Erkenntnisse zusammengefasst werden:

- Beschreibung typischer Benutzer-Szenarios.
- Auflistung der wichtigsten Eigenschaften des Systems und warum sie wertvoll oder nicht wertvoll sind.
- Skizzierung der Systemkomponenten und ihrer Zuständigkeiten.
- Das System sollte in einer »sauberer« Umgebung in einer beschränkten Zeit (maximal in einem Tag) **recompiliert, installiert und »gebaut«** werden. Wenn es einen Selbsttest des Systems gibt, dann sollte er durchgeführt werden. Ziel ist es, den Installations- und Bau-Prozess zu verifizieren, nicht das System komplett zu verstehen.

Tipp

Erstes Verstehen des Systems

Fehlerhafte Informationen sind das größte Problem, um ein erstes Verständnis über das System zu erhalten. Daher sollte die Basis für alle Informationen der Quellcode sein.

Prinzipiell kann man Top-down oder Bottom-up vorgehen. In der Praxis wird beides kombiniert. Beim Top-down-Vorgehen wird mit einer abstrakten Darstellung begonnen und diese gegen den Quellcode verifiziert.

IV 30 Reverse Engineering

Beim Bottom-up-Vorgehen ist der Startpunkt der Quellcode. Die relevanten Informationen werden gefiltert und auf einer höheren Abstraktionsebene dargestellt.

Bottom-up Beim Bottom-up-Vorgehen werden die persistenten Daten analysiert und außergewöhnliche Eigenschaften genauer betrachtet:

- Die **persistenten Daten werden analysiert**, indem die Datenbankschemata betrachtet und für die wichtigsten Daten die Strukturen herausgefiltert und als Klassendiagramm dargestellt werden.
- Die **wichtigsten Strukturen**, die das Softwaresystem ausmachen (Vererbungshierarchien, Pakete, Klassen, Methoden), werden **vermessen**. Auffälligkeiten in den gesammelten, quantitativen Daten werden manuell daraufhin verifiziert, ob sie Anomalien von Entwurfsproblemen darstellen. Folgende Hinweise sollten beachtet werden:
 - Zunächst ist zu überlegen, welche Metriken erfasst werden sollen.
 - Anschließend ist zu prüfen, mit welchen Werkzeugen die Metriken ermittelt werden können.
 - Die Interpretation der Messdaten muss sorgfältig erfolgen, da nicht jede Anomalie problematisch ist. Es ist hilfreich, simultan verschiedene Maße für denselben, auffälligen Codebereich heranzuziehen.
 - Zur schnellen Identifikation von Anomalien ist es hilfreich, dass die Metrik-Werkzeuge Visualisierungsmöglichkeiten zur Verfügung stellen.
 - Durch Werkzeuge identifizierte Anomalien müssen manuell im Code überprüft werden.
 - Identifizierte potenzielle Probleme müssen daraufhin überprüft werden, ob sie ein ernstes Problem darstellen. In Reengineering-Projekten werden mehr Probleme identifiziert, als in der gegebenen Zeit gelöst werden können. Daher ist eine Priorisierung umgänglich.

Top-down Beim Top-down-Vorgehen wird versucht, den Entwurf »wieder zu entdecken«:

- Ein erstes **hypothetisches Klassendiagramm** wird erstellt, das den Entwurf repräsentiert. Es wird überprüft, ob die Namen im Klassendiagramm auch im Quellcode erscheinen. Das Modell wird schrittweise verfeinert und der Prozess wiederholt, bis das Klassendiagramm sich stabilisiert. Bei Diskrepanzen zwischen dem Klassendiagramm und dem Quellcode sollten folgende Adaptationen vorgenommen werden:
 - *Renaming*, wenn die Namen im Quellcode nicht mit der Hypothese übereinstimmen.

- Remodelling*, wenn die Quellcode-Repräsentation des Entwurfskonzepts nicht mit dem eigenen Modell übereinstimmt. Beispielsweise kann eine Methode in eine Klasse oder ein Attribut in eine Methode transformiert werden.
- Erweiterung, wenn wichtige Elemente im Quellcode nicht im eigenen Klassendiagramm erscheinen.
- Alternativen suchen, wenn ein eigenes Entwurfskonzept nicht im Quellcode zu finden ist.

Das Top-down-Vorgehen ist bei großen objektorientierten Programmen (über 100 Klassen) erfolgversprechender, da das Bottom-up-Vorgehen bei großen Programmen *nicht* praktikabel ist.

Tipp

Erstellen eines detaillierten Modells

Nach dem ersten Überblick über das System ist es nun das Ziel, ein detailliertes Modell von den Teilen des Systems zu erstellen, die für das *Reengineering* wichtig sind. In diesem Schritt ist umfangreicheres technisches Wissen erforderlich, mehr Werkzeuge müssen eingesetzt werden und der Aufwand nimmt zu. Folgende Aktivitäten müssen durchgeführt werden, um die Entwurfskonzepte zu ermitteln, die im Code verborgen sind:

- Beim Durcharbeiten des Codes sollten **Fragen direkt und sofort beim Code annotiert** werden. Folgendes sollte beachtet werden:
 - Die Annotationen können Fragen, Hypothesen, To-Do-Listen oder einfach Beobachtungen über den Code sein.
 - Um Annotationen identifizieren zu können, sollten Konventionen beachtet werden. Beispielsweise können der Entwickler, der den Kommentar gemacht hat, und das Datum des Kommentars angegeben werden.
 - Wenn eine Antwort auf eine Frage gefunden wurde, dann muss die Annotation für spätere Leser aktualisiert werden. Oder sie wird gelöscht, wenn sie nicht weiter relevant ist.
 - Fragen im Code werden oft durch *Refactoring* gelöst.
- Teile des Systems werden iterativ einem **Renaming** und **Refactoring** unterzogen, um sicherzustellen, dass der Code das reflektiert, was das System aktuell tut. Folgendes ist dabei zu beachten:
 - Primäres Ziel ist es, das System zu verstehen, *nicht* den Code zu verbessern.
 - Folgende Aktivitäten verbessern die Lesbarkeit des Codes:
 - Attributnamen so umbenennen, dass sie die Rollen verdeutlichen, für die sie stehen. Auf Attribute mit kryptischen Namen konzentrieren. Um die Rollen zu identifizieren, sollten alle Zugriffe auf die Attribute betrachtet werden.

IV 30 Reverse Engineering

- Methodenamen so umbenennen, dass ihr Zweck sichtbar wird. Alle Aufrufe und benutzten Attribute sind zu analysieren, um die Aufgabe der Methode abzuleiten.
- Klassennamen so umbenennen, dass ihr Zweck sichtbar wird. Zu prüfen ist, welche Clients welche Methoden der Klasse aufrufen und wer Objekte der Klasse erzeugt.
- Entfernen von dupliziertem Code.
- Die Alternativen von umfangreichen bedingten Verzweigungen sollten durch neue (private) Methoden ersetzt werden.
- Umfangreiche Methoden sollten durch kleinere Methoden auf einheitlichem Abstraktionsniveau ersetzt werden.
- Für das *Refactoring* sollten Werkzeuge eingesetzt werden.
- **Beispielhafte Szenarien** sollen in einem Debugger durchlaufen werden, um zu verstehen, wie Objekte zur Laufzeit zusammenwirken und wie sie erzeugt werden. Die dabei gewonnenen Erkenntnisse sollten verallgemeinert und dokumentiert werden, z. B. als Code-Annotationen.
- Es sollte geprüft werden, ob Clients die **Schnittstellen von Klassen** richtig benutzen. Die Beobachtungen sollten in Form von Verträgen (*contracts*) verallgemeinert werden, damit sichtbar wird, was Klassen von ihren Clients erwarten. Folgende Aktivitäten helfen, Abhängigkeiten zwischen Klassen zu identifizieren:
 - Werkzeuge, insbesondere *Design Extraction*- und *Round-Trip-Engineering*-Werkzeuge, ermöglichen es, Beziehungen zwischen Klassen zu ermitteln. Ein Entwurfsüberblick hilft dabei, Schlüsselklassen in der Hierarchie, z. B. abstrakte Klassen, Aggregatklassen usw. zu identifizieren.
 - Schlüsselmethoden können anhand der Methodenamen und der Parametertypen ermittelt werden. Parameter repräsentieren temporäre Assoziationen zwischen Objekten. Wenn derselbe Parametertyp wiederholt in Methodensignaturen auftritt, dann repräsentiert er wichtige Assoziationen.
 - Um zu verstehen, wie und wann Objekte erzeugt werden, sollte nach Methoden in anderen Klassen geschaut werden, die Konstruktoren aufrufen. Aufrufe von Konstrukturen können auf eine *Whole-Part*-Beziehung hinweisen. Auch könnte es sich um eine Fabrikmethode oder eine abstrakte Fabrik handeln (siehe »Schnittstellen, Fabriken und Komposition«, S. 503).
- Durch den **Vergleich aufeinander folgender Versionen** des Systems erhält man Einblicke in den Entwurf. Ziel ist es, festzustellen, welche Teile des System stabil sind und welche nicht. Das Hauptinteresse besteht darin, festzustellen, was mit der »alten« Funktionalität geschehen ist, nicht, welche Funktionalität neu hinzugekommen ist. Folgende Hinweise sollten beachtet werden:

- Mithilfe von Metrik- oder Konfigurations-Management-Werkzeugen kann festgestellt werden, welche Funktionalität entfernt wurde. Daran erkennt man einen konsolidierten Entwurf. Außerdem sollte geprüft werden, welche Teile oft geändert wurden, da sie für einen instabilen Entwurf sprechen.

31 Reengineering (Teil 1)

Die Weiterentwicklung eines Alt-Systems kann nach [DDN09, S. 149 ff.] in fünf Schritten erfolgen:

- 1 Tests einsetzen**
- 2 Das Alt-System migrieren**
- 3 Duplizierten Code aufspüren**
- 4 Neuverteilung von Zuständigkeiten**
- 5 Bedingungen in Polymorphie transformieren**

Im Folgenden werden die ersten drei Schritte behandelt.

Tests einsetzen

Um die Risiken im Rahmen der *Reengineering*-Änderungen zu reduzieren, müssen Tests effizient eingesetzt werden. Folgende Aktivitäten sind nötig:

- **Schreiben von Tests**, um die Evaluation zu ermöglichen. Es muss ein Testprozess eingeführt werden, der auf automatisierten, wiederholbaren und gespeicherten Tests basiert. Tests ermöglichen es, Änderungen vorzunehmen und dabei sicherzustellen, dass das Verhalten des Softwaresystems gleich bleibt.
- **Inkrementeller Ausbau der Testbasis**. Um die Kosten und die Vorteile von Tests auszubalancieren, sollten Tests immer dann inkrementell eingeführt werden, wenn sie gerade benötigt werden. Am Anfang sollten Tests nur für die kritischen Komponenten und nur für die Teile des Systems entwickelt werden, für die eine Änderung geplant ist.
- **Verwendung eines Test-Frameworks**. Ein Test-Framework sollte benutzt werden, um Regressionstests leicht zu entwickeln und zu organisieren. Außerdem sollte es möglich sein, die Tests leicht ablaufen zu lassen, und Testsuites aus individuellen Testfällen zusammenstellen zu können. Ein Beispiel für ein Test-Framework ist JUnit.
- **Schnittstellen testen, nicht Implementierungen**, auch Black-Box-Testen genannt. Die Tests sollten sich auf das externe Verhalten beziehen, nicht auf Implementierungsdetails. Dadurch sind diese Tests auch nach Änderungen des Systems einsetzbar. Insbesondere sollten auch die Grenzwerte von Methodenparametern überprüft werden, da hier oft Fehler gemacht werden. Ein separates Überdeckungs-Werkzeug sollte eingesetzt werden, um zu prüfen, ob die Tests den gesamten Code überdecken.

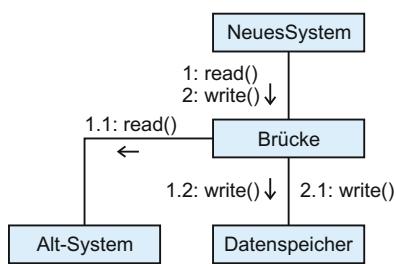
IV 31 Reengineering (Teil 1)

- **Geschäftsregeln in Form von Tests dokumentieren.** Dadurch ist sichergestellt, dass die Geschäftsregeln mit der Implementierung synchronisiert sind.
- **Hypothesen und Schlussfolgerungen über das Alt-System in Form von ausführbaren Tests beschreiben.** Dadurch können bestimmte Aspekte des Systems exakt spezifiziert werden. »Tests cannot be fuzzy« [DDN09, S. 179].

Das Alt-System migrieren

- **Benutzer einbeziehen**, um eine maximale Akzeptanz der Änderungen zu erreichen. Den Benutzern sollten die Prioritäten erläutert werden. Die Prioritäten sollten soweit herunter gebrochen werden, dass Änderungen in kleinen Inkrementen eingeführt werden können.
- Durch die frühzeitige Demonstration positiver Ergebnisse sollte bei den Benutzern **Vertrauen aufgebaut** werden. In kurzen Intervallen sollten anschließend weitere Ergebnisse ausgeliefert werden.
- Es sollte ein **Prototyp des neuen Systems** entwickelt und evaluiert werden, um die Risiken der Migration abzuschätzen. Dies kann in folgenden Schritten geschehen:
 - Es ist das größte technische Risiko für das Reengineering-Projekt zu ermitteln. Typische Risiken sind: Wahl einer neuen Systemarchitektur, Migration der Alt-Daten in das neue System, Erzielen einer adäquaten Performanz.
 - Es ist zu entscheiden, ob ein explorativer Prototyp zur Prüfung der Durchführbarkeit einer technischen Option entwickelt wird und der anschließend »weggeworfen« wird. Oder ob ein evolutionärer Prototyp entwickelt wird, der sich eventuell zu dem neuen System weiterentwickelt.
- Das **System sollte inkrementell migriert** werden. Es sollte ein erstes Update des Alt-Systems so früh wie möglich installiert werden. Anschließend sollte inkrementell zum Zielsystem hin migriert werden. Ein kompletter Umstieg zu einem Zeitpunkt sollte aus Risikogründen vermieden werden (*Big Bang*). Es kann wie folgt vorgegangen werden:
 - Zerlegen des Alt-Systems in Teile.
 - Zu einer Zeit sollte immer nur ein Teil in Angriff genommen werden.
 - Für den jeweiligen Teil sollten Tests bereitgestellt werden und für die Teile, die davon abhängen.
 - Die Komponente des Alt-Systems sollte durch geeignete Schritte »eingepackt« (*wrapping*), einem *Reengineering* unterzogen oder ersetzt werden.

- Die geänderte Komponente sollte installiert und ein Feedback angefordert werden.
- Es sollte **jederzeit eine lauffähige Version** vorhanden sein. Daher sollte in regelmäßigen Abständen das System neu »gebaut« werden. Ziel sollte es sein, neue Änderungen auf einer täglichen Basis zu integrieren. Um Risiken zu vermeiden, sollte eine kontinuierliche Integration (*continuous integration*) angestrebt werden. Da große Systeme oft eine lange »Bauzeit« (*build time*) haben, kann es nötig sein, zunächst das Alt-System zu überarbeiten, um kürzere »Bauzeiten« zu erreichen.
- **Nach jeder Änderung** sollte ein **Regressionstest** durchgeführt werden, um sicher zu gehen, dass die letzte Änderung das System nicht »beschädigt« hat. In einem komplexen System können kleine Änderungen zu unerwarteten Seiteneffekten führen, die sonst nicht sofort erkannt werden.
- Die **Daten vom Alt-System** sollten **über eine Daten-Brücke** in das parallel laufende neue System **migrirt** werden (Abb. 31.0-1). Folgende Schritte sollten durchgeführt werden:
 - Es sind die Komponenten im Alt-System und im neuen System zu identifizieren, die mit denselben logischen Daten arbeiten.
 - Es ist eine »Daten-Brücke« zu implementieren, die dafür verantwortlich ist, Lesezugriffe von der neuen Komponente auf die Datenquellen des Alt-Systems umzuleiten, wenn die Daten noch nicht migriert sind. Die »Brücke« ist verantwortlich für jede notwendige Datenkonversion. Die neue Komponente sollte nichts von der »Brücke« wissen.
 - Die Komponente des Alt-Systems ist so zu adaptieren, dass schreibende Zugriffe auf die neue Komponente umgeleitet werden, damit die neuen Daten aktuell bleiben.
 - Wenn alle Daten transferiert sind, sind die »Brücke« und die Komponente des Alt-Systems zu entfernen.



*Abb. 31.0-1:
Kommunikations-
diagramm einer
»Daten-Brücke«
[DDN09, S. 204].*

- Die **Schnittstellen für das neue System identifizieren** und das alte System so »verpacken«, dass die neuen Schnittstellen emuliert werden. Dadurch ist es möglich, sich von der Architektur des Alt-Systems zu lösen und sich auf alternative Ansätze zu konzentrieren.

IV 31 Reengineering (Teil 1)

- Um die Migration von Schnittstellen des Alt-Systems zu Schnittstellen des neuen Systems zu erleichtern, sollte zwischen **stabilen »öffentlichen« Schnittstellen und instabilen »zu veröffentlichten« Schnittstellen** unterschieden werden. Dadurch wird sichtbar, dass die zu veröffentlichten Schnittstellen sich noch in Entwicklung befinden und Änderungen noch möglich sind.
- Nicht mehr zu verwendende, **veraltete Schnittstellen sollten gekennzeichnet** werden (*deprecated*). Dadurch ist es möglich, dass neue und alte Schnittstellen eine Zeit lang koexistieren können.
- **Radikale Änderungen sollten vermieden** werden, um den Benutzer nicht zu verunsichern. Es sollten daher nur wenige Änderungen zwischen neuen Releases eingeführt werden.
- Bevor ein ineffizienter Teil des Systems optimiert wird, sollte **mit Hilfe eines Profiler-Werkzeugs bestimmt werden, wo der aktuelle Engpass liegt**. Es sollte erst dann eine Optimierung vorgenommen werden, wenn der Profiler anzeigt, dass signifikante Verbesserungen möglich sind. Es sollte immer folgende Reihenfolge eingehalten werden: »*Do it, then do it right, then do it fast.*«

Duplizierten Code aufspüren

Studien haben gezeigt, dass zwischen 8% und 12% der industriellen Software aus duplizierten Code besteht. Duplizierter Code behindert die Einführung von Änderungen, da jede implementierte Variante geändert werden muss. Da leicht einige Varianten übersehen werden, tauchen Fehler dann an anderer Stelle auf. Außerdem wiederholt und verstreut duplizierter Code die Logik eines Systems, anstelle die Logik in identifizierbaren Artefakten zu gruppieren.

- **Jede Zeile des Quellcodes sollte mithilfe von Software-Werkzeugen mit allen anderen Codezeilen verglichen** werden, um duplizierten Code zu finden. Durch das Entfernen aller Variablenbezeichner oder durch ihre Abbildung auf einen allgemeinen Bezeichner können ähnliche Codemuster entdeckt werden.
- **Duplizierter Code sollte in einer Matrix visualisiert** werden (*Dotplots*). Jede Achse repräsentiert eine Code-Quelle. Punkte in der Matrix zeigen, wo Quellcodezeilen dupliziert sind. Dadurch ist es leicht möglich, Einsichten in den Umfang und die Natur der Codeduplikation zu erhalten. Die Abb. 31.0-2 zeigt mögliche Sequenzen von Punkten und ihre Interpretationen [DDN09, S. 234]. Mithilfe solcher Darstellungen können auch große Bedingungskonstruktionen erkannt werden.

31 Reengineering (Teil 1) IV

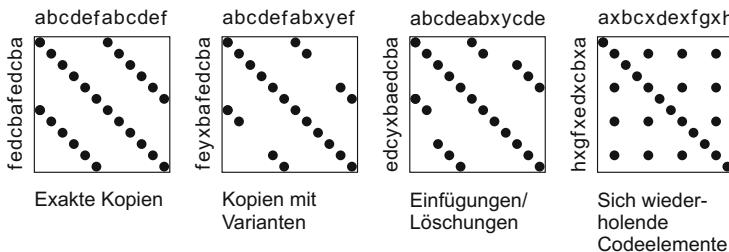


Abb. 31.0-2:
Mögliche
Sequenzen von
dupliciertem Code
und ihre
Interpretation
[DDN09, S. 234].

Ein Überblick und eine Klassifikation von Werkzeugen zur Visualisierung von Softwaresystemen werden in [TVS10] gegeben.

32 Reengineering (Teil 2)

Die Weiterentwicklung eines Alt-Systems kann nach [DDN09, S. 149 ff.] in fünf Schritten erfolgen:

- 1 Tests einsetzen
 - 2 Das Alt-System migrieren
 - 3 Duplizierten Code aufspüren
 - 4 Neuverteilung von Zuständigkeiten**
 - 5 Bedingungen in Polymorphie transformieren
- Im Folgenden wird der vierte Schritt behandelt.

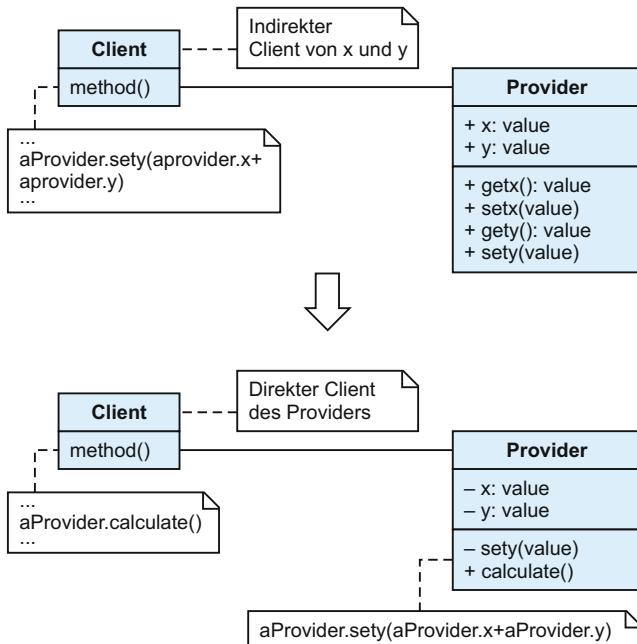
Neuverteilung von Zuständigkeiten

In vielen Alt-Systemen sind Zuständigkeiten falsch verteilt. Es gibt zwei Extreme, die häufig auftreten. Datencontainer enthalten nichts anderes als Datenstrukturen ohne identifizierbare Zuständigkeiten. »Gott«-Klassen sind prozedurale Monster, die zu viele Zuständigkeiten umfassen. Beide Extreme sind Kennzeichen einer fragilen Architektur. Ziel muss es daher sein, Datencontainer und »Gott«-Klassen durch eine Neuverteilung von Zuständigkeiten und eine Verbesserung der Bindung zu reduzieren.

- Das **Verhalten**, das oft bei Clients angeordnet ist, muss den **Daten zugeordnet** werden. Datencontainer stellen oft nur Zugriffsmethoden zur Verfügung und zwingen dadurch die Clients, das Verhalten selbst zu definieren, anstatt es nur zu benutzen. Neue Clients müssen daher das Verhalten reimplementieren. Wenn die interne Repräsentation der Daten im Datencontainer sich ändert, müssen viele Clients aktualisiert werden. Folgende Schritte sind durchzuführen (Abb. 32.0-1):
 - Das Verhalten in einem Client ist zu identifizieren, das zu einer Service-Anbieterklasse transferiert werden soll.
 - Es ist eine korrespondierende Methode in der Anbieterklasse zu erzeugen.
 - Die neue Methode sollte einen Namen erhalten, der die Intention der Methode wiedergibt.
 - In dem Client ist die neue Methode aufzurufen.
 - Der Client-Code ist zu bereinigen.
 - Diese Schritte sind für mehrere Clients zu wiederholen.
- **Navigationscode sollte eliminiert werden.** Wenn ein Client sich durch einen Objektgraph navigiert, dann sind bei Änderungen von Schnittstellen einer Klasse nicht nur die direkten Clients betroffen, sondern auch alle indirekten Clients. Nach dem Gesetz

IV 32 Reengineering (Teil 2)

Abb. 32.0-1:
Datencontainer werden in Service-Anbieter transformiert [DDN09, S. 244].



von Demeter (*Law of Demeter*) [LHR88] sollten Objekte nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren, um die Kopplungen zu reduzieren. Folgende Schritte sind durchzuführen (Abb. 32.0-2):

- Es ist der Navigationscode zu identifizieren, der an eine andere Stelle gebracht werden soll.
- Es ist das oben beschriebene Verfahren (Abb. 32.0-1) anzuwenden, das Verhalten den Service-Anbietern zuzuordnen, um eine Ebene der Navigation zu entfernen.
- Wenn notwendig, sind die beiden vorherigen Schritte zu wiederholen.
- »Gott«-Klassen sollten aufgeteilt werden. Anstelle einer Klasse mit vielen Zuständigkeiten entstehen viele kleine, in sich gebundene Klassen. Folgende Schritte sind iterativ durchzuführen (Abb. 32.0-3):
 - Die Zuständigkeiten sind auf Datencontainer zu verteilen oder es sind neue Klassen zu erzeugen, bis nichts mehr übrig bleibt als eine Fassade (siehe »Das Fassaden-Muster (*facade pattern*)«, S. 69).
 - Die Fassade ist zu entfernen.

Literatur

In [Fowl05] wird detailliert die Neuverteilung von Zuständigkeiten im Rahmen des *Refactoring* beschrieben (siehe auch »Restrukturieren (*refactoring*)«, S. 511).

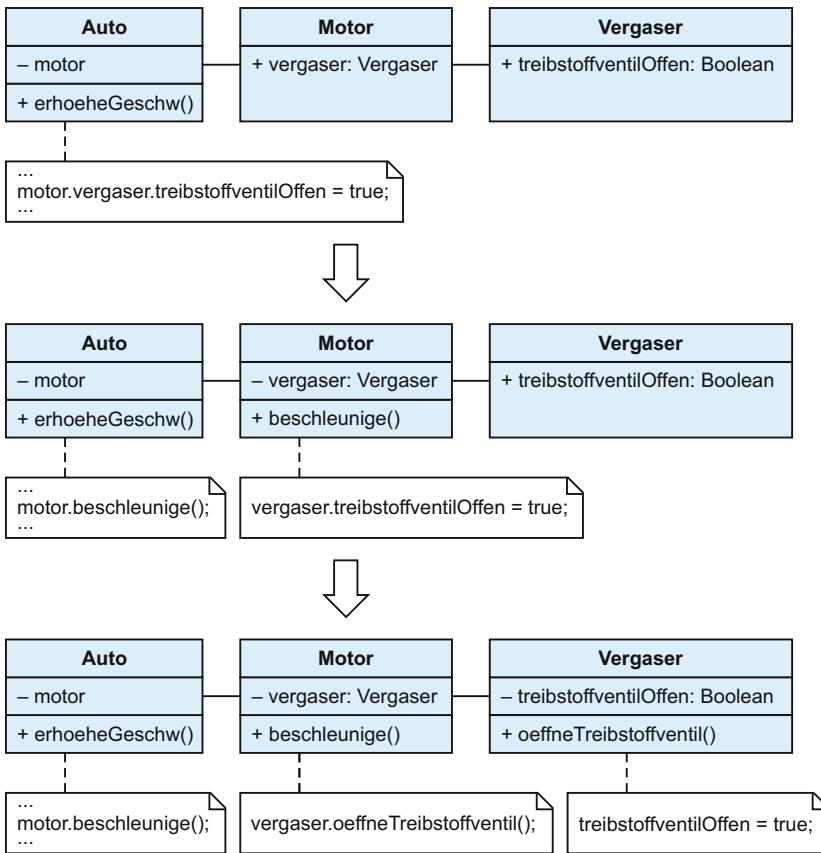
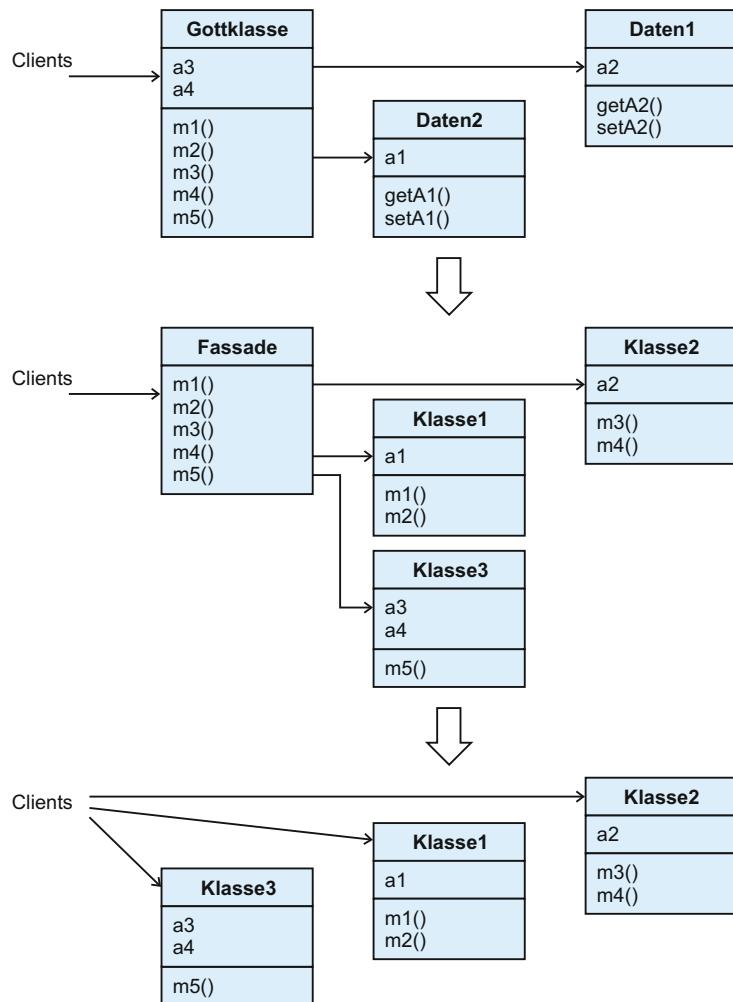


Abb. 32.0-2:
Ketten von
Datencontainern
werden in Service-
Anbieter
transformiert
[DDN09, S. 256].

IV 32 Reengineering (Teil 2)

Abb. 32.0-3: Die Zuständigkeiten einer »Gott«-Klasse werden auf neue Klassen verteilt [DDN09, S. 265].



33 Reengineering (Teil 3)

Die Weiterentwicklung eines Alt-Systems kann nach [DDN09, S. 149 ff.] in fünf Schritten erfolgen:

- 1 Tests einsetzen
- 2 Das Alt-System migrieren
- 3 Duplizierten Code aufspüren
- 4 Neuverteilung von Zuständigkeiten
- 5 **Bedingungen in Polymorphie transformieren**

Im Folgenden wird der fünfte Schritt behandelt.

Bedingungen in Polymorphie transformieren

Ein auffälliges Zeichen für falsch zugeordnete Zuständigkeiten in objektorientierten Systemen sind umfangreiche Methoden, die fast vollständig aus switch-Anweisungen bestehen, um den Typ von Argumenten zu testen. Diese Anweisungen röhren daher, dass es bei einer Anpassung oft am einfachsten ist, einen Spezialfall durch eine neue Bedingung in einer Methode einzufügen. Um die Kopplung zwischen Klassen zu reduzieren und die Flexibilität für weitere Änderungen zu erhöhen, ist es notwendig, diese umfangreichen Bedingungen zu eliminieren. Dazu gibt es mehrere Möglichkeiten.

- Um die Erweiterbarkeit einer Klasse zu verbessern, wird **eine komplexe, bedingte Anweisung durch den Aufruf einer Methode ersetzt**, die in Unterklassen implementiert wird (Abb. 33.0-1). In jeder UnterkLASSE wird diese Methode mit dem Code implementiert, der in der Original-Case-Anweisung enthalten war. Durch die Transformation solcher Klassen in eine Klassenhierarchie, die explizit die verschiedenen Datentypen repräsentiert, wird die Bindung verbessert, da aller Code der zu einem einzelnen Datentyp gehört, zusammengefasst wird. Der Entwurf wird dadurch transparenter und leichter zu warten.

Code im Alt-System:

```
public class Client
{
    public static void main(String[] args)
    {
        Aktion meineAktion = new Aktion();

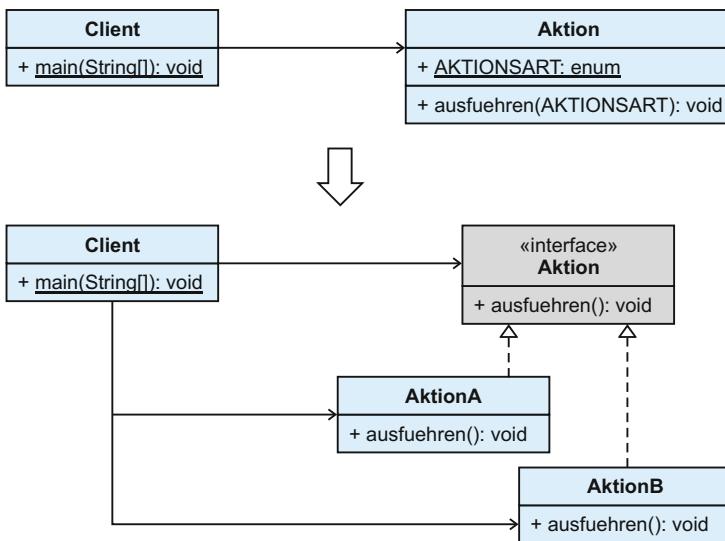
        //Aktion A
        meineAktion.ausfuehren(Aktion.AKTIONSART.A);

        //Aktion B
    }
}
```

Java

IV 33 Reengineering (Teil 3)

*Abb. 33.0-1:
Transformation
expliziter
Typüberprüfungen
in polymorphe
Methodenaufrufe.*



```
meineAktion.ausfuehren(Aktion.AKTIONSART.B);  
}  
}  
  
public class Aktion  
{  
    public static enum AKTIONSART {A,B};  
  
    public void ausfuehren(AKTIONSART aktionsart)  
    {  
        switch(aktionsart)  
        {  
            case A:  
                System.out.println("A");  
                break;  
            case B:  
                System.out.println("B");  
                break;  
            default:  
        }  
    }  
}
```

Nach dem Reengineering:

```
Java public class Client
{
    public static void main(String[] args)
    {
        //Schnittstelle Aktion
        Aktion meineAktion = null;

        //Aktion A
        meineAktion = new AktionA();
        meineAktion.ausfuehren();
```

```

//Aktion B
meineAktion = new AktionB();
meineAktion.ausfuehren();
}
}

public interface Aktion
{
    public void ausfuehren();
}

public class AktionA implements Aktion
{
    @Override
    public void ausfuehren()
    {
        System.out.println("A");
    }
}

public class AktionB implements Aktion
{
    @Override
    public void ausfuehren()
    {
        System.out.println("B");
    }
}

```

- Um die Kopplung zwischen Client und Dienstanbieter zu reduzieren, wird ein **bedingter Code, der den Typ des Dienstanbieters prüft, in einen polymorphen Aufruf zu einer neuen Dienstanbieter-Methode transformiert**. Clients werden von Aktionen, die in die Verantwortlichkeit des Dienstanbieters fallen, entlastet. Dazu ist eine neue Methode in der Dienstanbieter-Hierarchie einzuführen. Die neue Methode wird in jeder Unterkasse der Dienstanbieter-Hierarchie implementiert. Die korrespondierenden Bedingungen des Clients werden in die entsprechenden Unterklassen »verschoben« (Abb. 33.0-2).

Code im Alt-System:

Java

```

public class Client
{
    public static void main(String[] args)
    {
        A obj = new B(); init(obj);

        obj = new C(); init(obj);

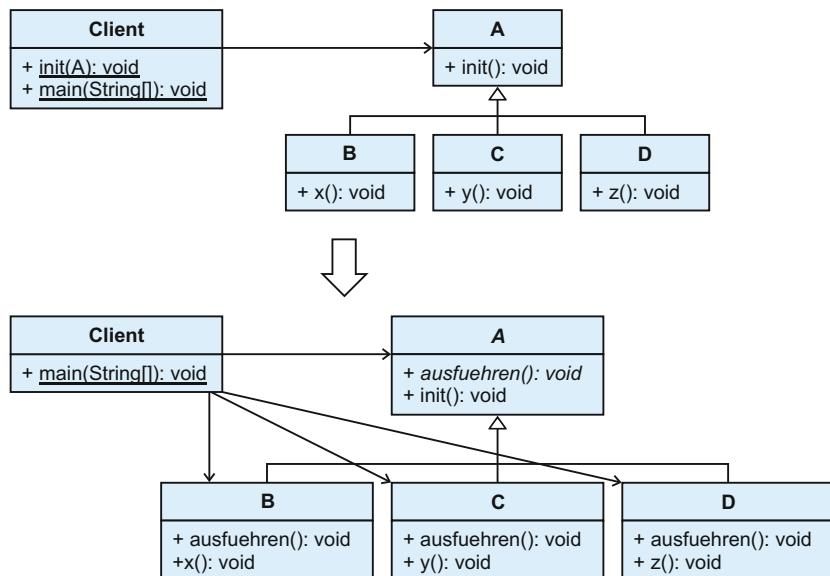
        obj = new D(); init(obj);
    }

    public static void init(A obj)
    {

```

IV 33 Reengineering (Teil 3)

Abb. 33.0-2:
Transformation
expliziter
Typüberprüfungen
im Client in
polymorphe
Methodenaufrufe.



```

if(obj instanceof B)
{
    obj.init(); //A
    ((B) obj).x(); //B
}
else if(obj instanceof C)
{
    obj.init(); //A
    ((C) obj).y(); //C
}
else if(obj instanceof D)
{
    ((D) obj).z(); //D
}
}
  
```

```

public class A
{
    public void init()
    {
        System.out.println("A");
    }
}
  
```

```

public class B extends A
{
    public void x()
    {
        System.out.println("B");
    }
}
  
```

```

}

public class C extends A
{
    public void y()
    {
        System.out.println("C");
    }
}

public class D extends A
{
    public void z()
    {
        System.out.println("D");
    }
}

```

Nach dem *Reengineering*:

```

public class Client
{
    public static void main(String[] args)
    {
        A obj = new B();
        obj.ausfuehren();

        obj = new C();
        obj.ausfuehren();

        obj = new D();
        obj.ausfuehren();
    }
}

public abstract class A
{
    public void init()
    {
        System.out.println("A");
    }

    public void ausfuehren() {}
}

public class B extends A
{
    public void ausfuehren()
    {
        this.init();
        this.x();
    }

    public void x()
    {

```

IV 33 Reengineering (Teil 3)

```
        System.out.println("B");
    }

public class C extends A
{
    public void ausfuehren()
    {
        this.init();
        this.y();
    }

    public void y()
    {
        System.out.println("C");
    }
}

public class D extends A
{
    public void ausfuehren()
    {
        this.z();
    }

    public void z()
    {
        System.out.println("D");
    }
}
```

- Ist in einer Methode der Zustandswechsel eines Objekts durch bedingte Anweisungen implementiert, dann sollte das **zustandsabhängige Verhalten in separate Objekte ver kapselt** werden (Anwendung des Zustands-Musters), siehe Abb. 33.0-3. Da auf die Zustände über Delegation vom Originalobjekt zugegriffen wird, sind Clients nicht betroffen. Erweiterungen sind dadurch leicht möglich.

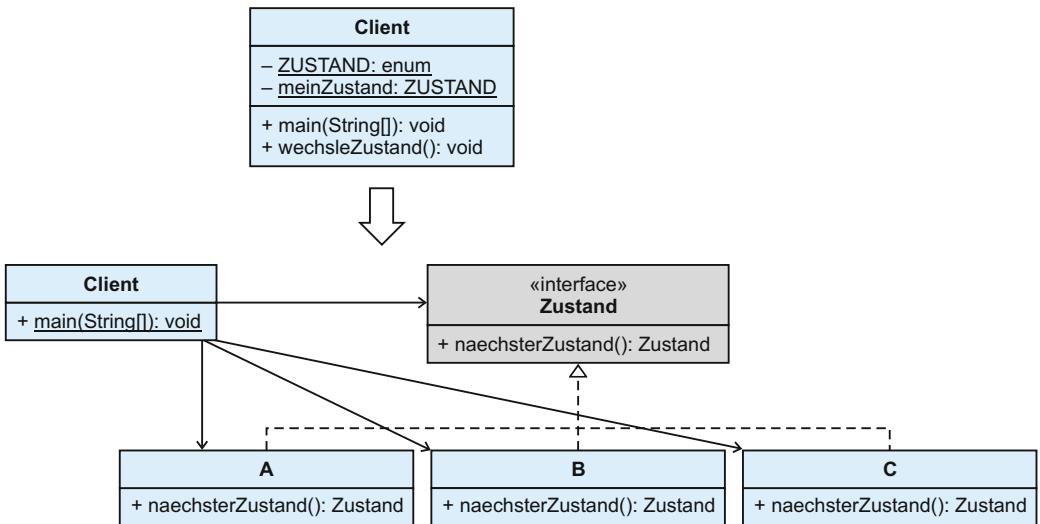
Java Code im Alt-System:

```
public class Client
{
    private static enum ZUSTAND {A,B,C};
    private static ZUSTAND meinZustand=null;

    public static void main(String[] args)
    {
        //Initialzustand
        meinZustand = ZUSTAND.A;

        //Zustandswechsel
        wechsleZustand(); //A->B

        //Zustandswechsel
```



```

wechsleZustand(); //B->C
//Zustandswechsel (kein Wechsel, da bereits in C)
wechsleZustand(); //C
}

public static void wechsleZustand()
{
    switch(meinZustand)
    {
        case A:
            System.out.println("A->B");
            meinZustand = ZUSTAND.B;
            break;
        case B:
            System.out.println("B->C");
            meinZustand = ZUSTAND.C;
            break;
        case C:
            System.out.println("C");
            break;
        default:
    }
}
  
```

Abb. 33.0-3:
Transformation
einer expliziten
Zustandsverwal-
tung im Client in
das
Zustandsmuster.

Nach dem Reengineering:

```

public class Client
{
    public static void main(String[] args)
    {
        //Initialzustand
        Zustand meinZustand = new A();
  
```

IV 33 Reengineering (Teil 3)

```
//Zustandswechsel
meinZustand = meinZustand.naechsterZustand(); //A->B

//Zustandswechsel
meinZustand = meinZustand.naechsterZustand(); //B->C

//Zustandswechsel (kein Wechsel, da bereits in C)
meinZustand = meinZustand.naechsterZustand(); //C
}

public interface Zustand
{
    public Zustand naechsterZustand();
}

public class A implements Zustand
{
    @Override
    public Zustand naechsterZustand()
    {
        System.out.println("A->B");
        return new B();
    }
}

public class B implements Zustand
{
    @Override
    public Zustand naechsterZustand()
    {
        System.out.println("B->C");
        return new C();
    }
}

public class C implements Zustand
{
    @Override
    public Zustand naechsterZustand()
    {
        System.out.println("C");
        return new C();
    }
}
```

- Bedingte Anweisungen, die dafür sorgen, dass ein passender Algorithmus ausgewählt wird, fallen durch **Anwendung des Strategie-Musters** weg (siehe »Das Strategie-Muster (*strategy pattern*)«, S. 96). Dazu wird das algorithmusabhängige Verhalten in separate Objekte mit polymorphen Schnittstellen gekapselt. Aufrufe werden an diese Objekte delegiert. Dadurch können Algorithmen unabhängig von ihrem Kontext variiert werden.

- Bedingte Anweisungen, die auf Null-Werte testen, werden durch Anwendung des **Null-Objekt-Musters** eliminiert. Das Null-Verhalten wird in eine separate Dienstanbieterklasse verpackt, so dass die Client-Klasse keine Null-Tests mehr ausführen muss (Abb. 33.0-4). Der Code in der Client-Klasse wird dadurch vereinfacht.

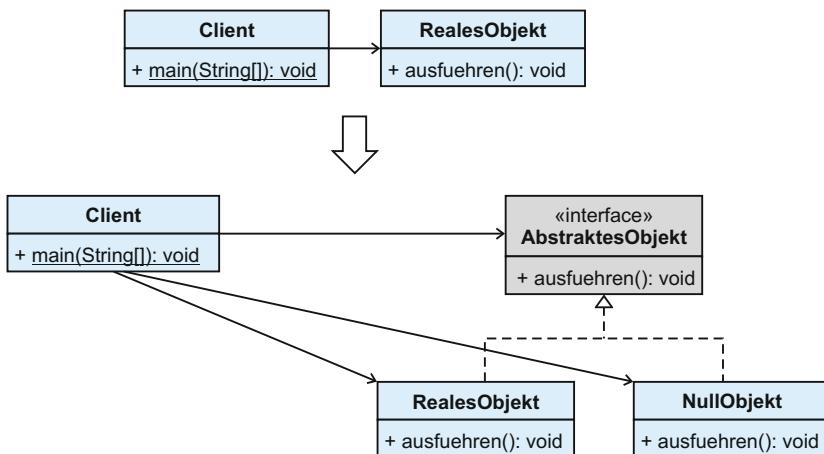


Abb. 33.0-4:
Transformation
einer Abfrage auf
einen Null-Wert in
eine Architektur
mit einem Null-
Objekt.

Code im Alt-System:

```
public class Client
{
    @SuppressWarnings("unused")
    public static void main(String[] args)
    {
        RealesObjekt obj = null;

        //Objekt ist null, Methode wird nicht ausgefuehrt
        if(obj!=null)
            obj.ausfuehren();

        obj = new RealesObjekt();

        //Objekt ist nicht null, Methode wird ausgefuehrt
        if(obj!=null)
            obj.ausfuehren();
    }
}

public class RealesObjekt
{
    public void ausfuehren()
    {
        System.out.println("Ausfuhrung");
    }
}
```

Java

IV 33 Reengineering (Teil 3)

Nach dem *Reengineering*:

```
Java public class Client
{
    public static void main(String[] args)
    {
        AbstraktesObjekt obj = new NullObjekt();

        //Objekt ist vom Typ NullObjekt, Methodenrumpf leer
        obj.ausfuehren();

        obj = new RealesObjekt();

        //Objekt ist vom Typ RealesObjekt,
        //Methode wird ausgefuehrt
        obj.ausfuehren();
    }

    public interface AbstraktesObjekt
    {
        public void ausfuehren();
    }

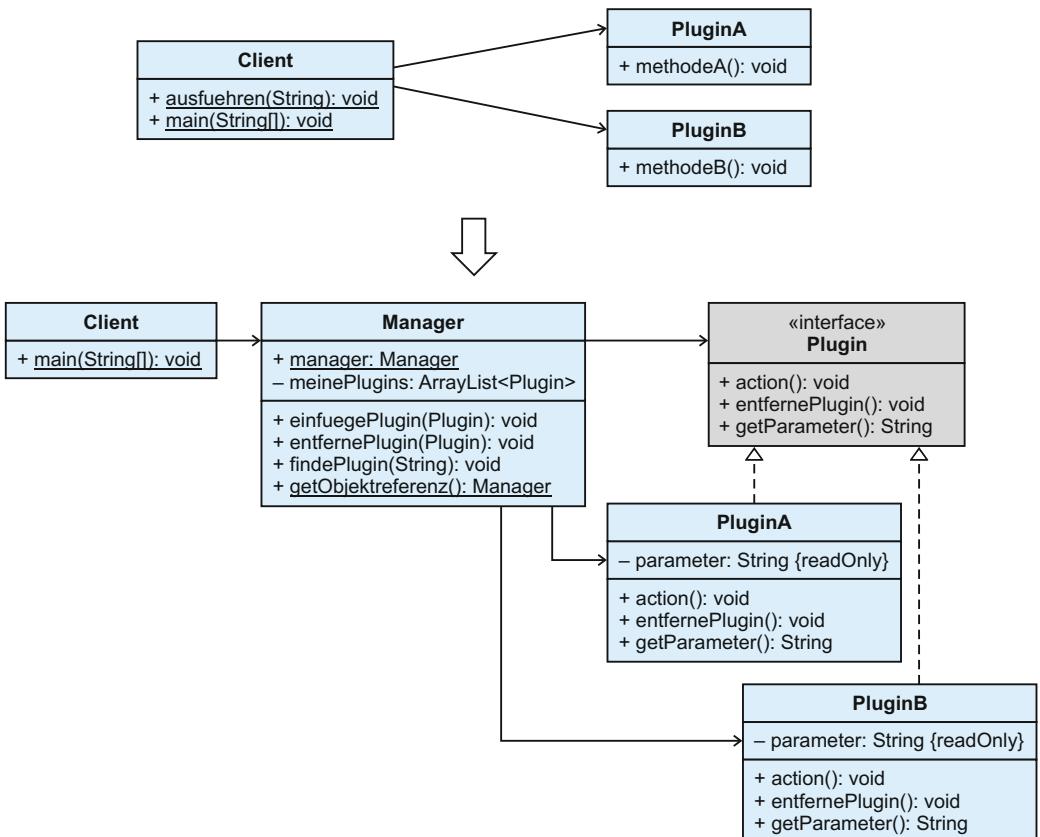
    public class RealesObjekt implements AbstraktesObjekt
    {
        @Override
        public void ausfuehren()
        {
            System.out.println("Ausführung");
        }
    }

    public class NullObjekt implements AbstraktesObjekt
    {
        @Override
        public void ausfuehren()
        {
        }
    }
}
```

- Die Kopplung zwischen Clients, die Werkzeuge benutzen, und Werkzeugen, die Services bereitstellen, kann reduziert werden, wenn sich **jedes Werkzeug über einen Registrierungsmechanismus selbst registriert** und die Clients, die die Werkzeuge benutzen, das Registrierungsverzeichnis abfragen anstatt Bedingungen auszuführen (Abb. 33.0-5).

Code im Alt-System:

```
Java public class Client
{
    public static void main(String[] args)
    {
        //Plugin A
        ausfuehren("A");
    }
}
```



```

//Plugin B
ausfuehren("B");
}

public static void ausfuehren(String plugin)
{
    //Pruefung, welches Plugin in Frage kommt
    if(plugin.equals("A"))
    {
        PluginA pluginA = new PluginA();
        pluginA.methodeA();
    }
    else if(plugin.equals("B"))
    {
        PluginB pluginB = new PluginB();
        pluginB.methodeB();
    }
}
}

public class PluginA

```

Abb. 33.0-5:
Ablösung von
Bedingungen zur
Werkzeugauswahl
durch Einführung
eines
Registrierungsme-
chanismus.

IV 33 Reengineering (Teil 3)

```
{  
    public void methodeA()  
    {  
        System.out.println("Methodenrumpf A");  
    }  
}  
  
public class PluginB  
{  
    public void methodeB()  
    {  
        System.out.println("Methodenrumpf B");  
    }  
}
```

Nach dem *Reengineering*:

```
Java public class Client  
{  
    public static void main(String[] args)  
    {  
        //Plugin A  
        Manager.getObjektreferenz().findePlugin("A");  
  
        //Plugin B  
        Manager.getObjektreferenz().findePlugin("B");  
    }  
}  
  
import java.util.ArrayList;  
import java.util.Iterator;  
  
public class Manager  
{  
    //Singleton  
    public static Manager manager = null;  
  
    private ArrayList<Plugin> meinePlugins;  
  
    public Manager()  
    {  
        meinePlugins = new ArrayList<Plugin>();  
    }  
  
    @SuppressWarnings("unused")  
    public static Manager getObjektreferenz()  
    {  
        if (manager == null)  
        {  
            manager = new Manager();  
  
            //Die vorhandenen Plugins hinzufuegen  
            PluginA pluginA = new PluginA();  
            PluginB pluginB = new PluginB();  
        }  
        return manager;  
    }  
}
```

```

}

public void einfuegePlugin(Plugin einPlugin)
{
    meinePlugins.add(einPlugin);
}

public void entfernePlugin(Plugin einPlugin)
{
    meinePlugins.remove(einPlugin);
}

public void findePlugin(String param)
{
    Iterator<Plugin> iter = meinePlugins.iterator();
    while (iter.hasNext())
    {
        Plugin plugin = iter.next();
        if(plugin.getParameter().equals(param))
            plugin.action();
    }
}

public interface Plugin
{
    //Die Methode, die der Manager ausfuehrt.
    public void action();

    //Zum Pruefen, ob ein Plugin fuer den Parameter
    public String getParameter();

    //Zum Entfernen des Plugins beim Manager.
    public void entfernePlugin();
}

public class PluginA implements Plugin
{
    private final String parameter="A";

    public PluginA()
    {
        //Beim Erstellen soll ein Objekt direkt beim
        //Manager registriert werden.
        Manager.getObjektreferenz().einfuegePlugin(this);
    }

    @Override
    public void action()
    {
        System.out.println("Methodenrumpf A");
    }

    @Override
    public String getParameter()
    {
}

```

IV 33 Reengineering (Teil 3)

```
        return parameter;
    }

@Override
public void entfernePlugin()
{
    Manager.getObjektreferenz().entfernePlugin(this);
}

public class PluginB implements Plugin
{
    private final String parameter="B";

    public PluginB()
    {
        //Beim Erstellen soll ein Objekt direkt
        //beim Manager registriert werden.
        Manager.getObjektreferenz().einfuegePlugin(this);
    }

    @Override
    public void action()
    {
        System.out.println("Methodenrumpf B");
    }

    @Override
    public String getParameter()
    {
        return parameter;
    }

    @Override
    public void entfernePlugin()
    {
        Manager.getObjektreferenz().entfernePlugin(this);
    }
}
```

Literatur In [Fowl05] wird im Rahmen des *Refactoring* detailliert beschrieben, wie Bedingungen in Polymorphie transformiert werden können (siehe auch »Restrukturieren (*refactoring*)«, S. 511).

Glossar

Abnahme (*approval*) Die Abnahme ist der Zeitpunkt, zu dem ein Kunde seinem Lieferant erklärt, dass das gelieferte Produkt keine schwerwiegenden Mängel hat und deshalb von ihm akzeptiert wird. In vielen europäischen Ländern wird die Abnahme im Zusammenhang mit einem Werkvertrag gesetzlich gefordert.

Abstrakte Fabrik (*abstract factory*) Entwurfsmuster, welches das Ziel hat, zu einer Menge von zusammenhängenden abstrakten Oberklassen auf klar definierte Art und Weise Exemplare von konkreten Unterklassen zu erzeugen. Auf diese Weise kann ein Programm Funktionalität einbinden, ohne von einer konkreten Implementierung dieser Funktionalität abhängig zu sein.

ActiveX (*ActiveX*) Bezeichnet eine von Microsoft eingeführte Technik, die zur Realisierung von sog. aktiven Elementen benutzt werden kann. Aktive Elemente können in Form von *ActiveX controls* in einem Web-Browser geladen und ausgeführt werden.

Adobe Flash (*Adobe Flash*) Adobe Flash ist eine proprietäre Entwicklungsumgebung von Adobe Systems zur Erstellung multimedialer, interaktiver Inhalte, der so genannten Flash-Filme. Bekannt und umgangssprachlich gemeint ist aber meist das Dateiformat (SWF) bzw. der Flash Player, eine Softwarekomponente zum Betrachten dieser SWF-Dateien im Webbrower.

AJAX (*AJAX; Asynchronous JavaScript and XML*) Ermöglicht es, dass Teile einer Webseite ausgetauscht werden können, ohne dass die gesamte Seite neu geladen werden muss. Dies wird im Wesentlichen durch JavaScript und ein Konzept der asynchronen Datenübertragung zwischen Browser und Webserver erreicht.

Altsystem (*Legacy System*) Softwaresysteme, die aus dem Blickwinkel der Gegenwart, mit veralteten Softwareme-

thoden und -konzepten entwickelt wurden, und ohne eine Sanierung bzw. ein *Reengineering* für den vorgesehnen Einsatzzweck und die vorgesehne Einsatzumgebung nicht oder nicht mehr wirtschaftlich und/oder fachlich verwendet werden können.

Annotation (*annotation*) Liefert Angaben über das Programm, in welchem sie sich befinden, sind aber nicht Teil des Programms. Annotationen haben keinen Einfluss auf die Funktion des Programms, welches sie annotieren. In Java sind @Deprecated und @Override Beispiele für Annotationen.

Artefakt (*artifact*) Ein Artefakt repräsentiert in der UML ein physisch vorhandenes Modell-Element, z. B. eine Datei. Artefakte realisieren logische Modell-Elemente, z. B. realisiert eine class-Datei eine Klasse.

ASP.NET (*Active Server Pages .NET*) Serverseitige Technik von Microsoft zum Erstellen von Web-Anwendungen auf Basis des .NET-Frameworks von Microsoft. Web-Anwendungen können mit allen vom .NET-Framework unterstützten Programmiersprachen erstellt werden, einschließlich C# und VB.NET.

Authentifizierung (*authentication*) Authentifizierung bedeutet aufgaben- und benutzerabhängige Zugangs- und/oder Zugriffsberechtigung. Sie hat den Zweck, Systemfunktionen vor Missbrauch zu schützen. Bei der Kommunikation stellt die Authentifizierung sicher, dass der Kommunikationspartner auch derjenige ist, für den er sich ausgibt. Es wird zwischen einseitiger und gegenseitiger Authentifizierung unterschieden. In der Praxis wird meistens die einseitige Authentifizierung verwendet, wobei beispielsweise beim Login der Benutzer sein Passwort eingibt und damit nachweist, dass er wirklich der angegebene Benutzer ist. Als Sicherheitsdienst für die einseitige Identifikation dient der Empfängernachweis,

Glossar

durch den die Benutzer-Identität und damit auch die Benutzungsberechtigung gegenüber dem System nachgewiesen wird. Dazu dienen hauptsächlich Passwörter, persönliche ID-Nummern, kryptografische Techniken sowie Magnet- oder Chip-Ausweiskarten. Eine strenge Authentifizierung kann mit der Vergabe von Einmalpasswörtern (OTP = One Time Passwords) erfolgen. Darüber hinaus gibt es Authentisierungssysteme, die mit biometrischen Daten arbeiten, und Mehrfaktorsysteme, die unter anderem auf so genannte USB-Token (Stecker oder Dongle, der zur persönlichen Authentifizierung des Anwenders dient) setzen. Sicherer als die einseitige Authentifizierung ist die gegenseitige, bei der alle Kommunikationspartner ihre Identität beweisen müssen, bevor untereinander vertrauliche Daten ausgetauscht werden. So sollte beispielsweise beim Abheben von Bargeld an einem Terminal (Ein-/Ausgabegerät) dieses vor Eingabe der PIN beweisen, dass es sich bei dem Gerät um ein echtes Geldausgabegerät handelt und nicht um einen Nachbau oder eine Konsole, welche nur die eingegebenen Daten aufzeichnet (*Keylogging*).

Authentisierung (*authentication*) Feststellung der berechtigten Verwendung einer Benutzerkennung durch Angabe eines geheimen Passwortes.

Autorisierung (*authorization*) Als Autorisierung wird in der Informationstechnik die Zuweisung und Überprüfung von Zugriffsrechten auf Daten und Funktionen eines IT-Systems an Benutzer bezeichnet.

Betriebssicherheit (*security*) Kennzeichnet die Qualität einer Software, einen unberechtigten Zugriff auf Daten oder Programme zu unterbinden.

Chat (*chat*) Erlaubt Internet-Benutzern gemeinsame interaktive Gespräche in Echtzeit. (Syn.: Plaudern im Internet)

Client-Server-Architektur

(*client/server architecture*) Ein Client-Server-System besteht aus Clients, die Verbindungen zu Diensten anbietenden Servern aufbauen. Der Client bietet die Benutzungsoberfläche bzw. die Benutzungsschnittstelle der Anwendung an. Durch Client-Server-Architekturen wird

u. a. eine Verteilung der Rechnerlast auf unterschiedliche Systemkomponenten erreicht.

CORBA (*CORBA; common object request broker architecture*) Es handelt sich um eine objektorientierte *Middleware*, die ein Protokoll und entsprechende Dienste zur Verfügung stellt, um eine plattformübergreifende Interprozesskommunikation zu realisieren. CORBA eignet sich insbesondere bei der verteilten Programmierung in heterogenen Umgebungen, bei der die beteiligten Programme in verschiedenen Programmiersprachen entwickelt und auf unterschiedlichen Plattformen betrieben werden. Die Spezifikation wird durch die OMG entwickelt.

CSS (*CSS; Cascading Style Sheets*) *Style-sheet*-Sprache, die festlegt, wie Elemente in einem strukturiertem Dokument dargestellt werden. CSS wird vor allen in XHTML- und XML-Dokumenten eingesetzt. Es ist möglich, die Darstellung an das spezifische Ausgabemedium anzupassen.

DAO-Muster (*DAO pattern; data access object pattern*) Entwurfsmuster, das zu einer Entkopplung der Fachkonzept-Schicht (auch Applikationslogik oder Geschäftslogik) und der Persistenz- bzw. Datenschicht führt.

Deployment (*deployment*) Vorgehensweise, um ein fertiggestelltes Software-System an die Kunden auszuliefern, zu installieren und in Betrieb zu nehmen.

Drei-Schichten-Architektur (*three-tier architecture*) Besteht aus der GUI-Schicht (Schicht der Benutzeroberfläche), der Fachkonzeptschicht – auch Applikationsschicht genannt – und der Datenhaltungsschicht.

Effizienz (*efficiency*) Effizienz umfasst nach DIN ISO 9126 alle Teilmerkmale, die ein Software-Produkt zeitlich charakterisieren. Hierzu gehören das Zeitverhalten und auch das Verbrauchsverhalten einer Software.

Entwurfsmuster (*design pattern*) Gibt eine bewährte, generische Lösung für ein immer wiederkehrendes Entwurfsproblem an, das in bestimmten Situationen auftritt. Es lassen sich u. a. klassen- und objektbasierte Muster unterscheiden. Klassenbasierte Muster wer-

den durch Vererbungen ausgedrückt. Objektbasierte Muster beschreiben in erster Linie Beziehungen zwischen Objekten.

ESB (enterprise service bus) Verknüpft Services in einer serviceorientierten Architektur (SOA) flexibel und dynamisch. Es müssen keine Verbindungen programmiert werden. Die Kopplung erfolgt im Wesentlichen asynchron auf Basis eines Nachrichtenkonzepts.

Fabrik-Dienst (factory service) Ein von entfernten Clients aus zugreifbares Objekt auf einem Server, das Operationen zur Erzeugung von Fachkonzept-Objekten, zur Suche nach vorhandenen und zur Löschung nicht mehr benötigter Objekte zur Verfügung stellt.

Fassaden-Klasse (fassade class) Eine Fassaden-Klasse stellt eine Sicht auf eine andere Klasse dar. Die Methoden der Fassaden-Klasse sind für die Zugriffe durch bestimmte Clients optimiert.

Forward Engineering (forward engineering) Vorgehensweise, um ein neues Softwaresystem zu entwickeln. (Syn.: Vorfwärts-Entwicklung)

Framework (framework) Besteht aus einer Menge von zusammenarbeitenden Klassen, die einen wiederverwendbaren objektorientierten Entwurf für einen bestimmten Anwendungsbereich implementieren. Es besteht aus konkreten und insbesondere aus abstrakten Klassen, die Schnittstellen definieren. Die abstrakten Klassen enthalten sowohl abstrakte als auch konkrete Operationen. Im Allgemeinen wird vom Anwender (= Programmierer) des Frameworks erwartet, dass er Unterklassen definiert, um das Framework zu verwenden und anzupassen. (Syn.: Rahmenwerk)

Funktionsicherheit (safety) Maß, in dem Gefährdungen bzw. Bedrohungen von einem System für seine Umwelt ausgehen.

Gebrauchstauglichkeit (usability) Grad, in dem ein Softwareprodukt von einem Benutzer in seinem Nutzungs-kontext effektiv, effizient und zufriedenstellend eingesetzt werden kann, um festgelegte Ziele zu erreichen.

HTTP (HTTP; Hypertext Transfer Protokoll) Standardprotokoll, mit dem Webseiten vom Webserver zum Webbrower übertragen werden. HTTP ist ein *zustandsloses* Protokoll. Nach jeder Anfrage und der zugehörigen Übertragung der Webseite an den Webbrower wird die Verbindung vom Client zum Server wieder getrennt. HTTP hat sich bei der Übertragung von HTML- und XHTML-Seiten als Standard etabliert.

IETF (Internet Engineering Task Force) Organisation, welche die Standardisierung im Internet vorantreibt.

Installation (installation) Die Installation von Software ist der Vorgang, bei dem neue Programme auf einen vorhandenen Computer kopiert und dabei eventuell konfiguriert werden. (Syn.: setup)

Instant Messaging (Instant Messaging) Dienst im Internet, der es erlaubt, auf dem eigenen Computersystem eine Liste mit Freunden, Arbeitskollegen usw. zu führen. Immer wenn ein Mitglied dieser Liste online ist, wird dies angezeigt. Man kann auch einen privaten Chat führen, aber auch Dateien austauschen usw. Auch Audio-Chats sind möglich. Eine Art E-Mail in Echtzeit. Als Software wird ein *Instant Messenger* benötigt. Es gibt mehrere Systeme im Markt, die aber nicht kompatibel sind. (Abk.: IM)

Internationalisierung (internationalization) Im Rahmen einer Softwareentwicklung wird eine Anwendung so konstruiert, dass eine Lokalisierung, d.h. eine Anpassung an unterschiedliche Sprachen, Währungen, Datumsformate etc. möglich ist. Meist wird die gleichzeitige Nutzbarkeit unterschiedlicher nationaler Aspekte angestrebt. Für Währungen bedeutet dies z.B., dass das System mit mehreren Währungen parallel arbeiten und die Währungen ineinander umwandeln kann. (Abk.: I18N)

Internet-Telefonie (internet telephony, voice over IP) Übertragung von Telefongesprächen über IP-basierte Netzwerke – insbesondere das Internet. Auch *Voice over IP* (VoIP) genannt. (Abk.: VoIP, VON; Syn.: voice over the net)

Glossar

Java Collection Framework (*Java collection framework*) Eine Sammlung von Schnittstellen und Klassen im Paket `java.util`, die leicht zu nutzende und trotzdem effiziente Containerklassen für viele Probleme des täglichen Programmierer-Lebens zur Verfügung stellt. (Abk.: JCF)

Java-Applet (*Java applet*) Programm, geschrieben in der Programmiersprache Java, das in einem Webbrowser abläuft.

JavaScript (*JavaScript*) Am meisten verbreitete Skriptsprache zur Verknüpfung von Programmcode mit statischen HTML-Seiten. Sie ermöglicht es, Webseiten dynamisch zu verändern. Obwohl es der Name vermuten lässt, handelt es sich *nicht* um eine Teilmenge von Java. JavaScript gilt als *unsicher*, da Webseiten »bösertige« JavaScript-Programme enthalten können, die dann auf dem Web-Client ausgeführt werden.

JDBC (*JDBC*; Java Database Connectivity) Standard, um aus Java-Programmen heraus auf relationale Datenbanksysteme zugreifen zu können. JDBC ist objektorientiert und plattformunabhängig.

JPA (*JPA*; Java Persistence API) Schnittstellendefinition (API), um eine Abbildung zwischen Java-Objekten und Daten in relationalen Datenbanken herzustellen. Durch den Einsatz von JPA können Objekte über die Laufzeit eines Programms hinaus dauerhaft (persistent) gespeichert werden.

JPQL (*JPQL*; Java Persistence Query Language) Abfragesprache innerhalb der JPA (Java Persistence API). Lehnt sich an SQL an und verfügt über ähnliche Möglichkeiten.

JSF (*JSF*; JavaServer Faces) JSF (*Java Server Faces*) ist ein Web-Framework zur Entwicklung von Benutzeroberflächen für Web-Anwendungen. Es basiert auf Java-Servlets und JavaServer Pages.

JSP (*JSP*; JavaServer Pages) HTML-Dokument, das eingebettete Java-Anweisungen enthält, um dynamische Web-Inhalte zu erzeugen. Die Java-Anweisungen werden durch besondere Markierungen (*tags*) innerhalb von HTML ge-

kennzeichnet. Eine JSP wird von einem JSP-Server in ein Java-Servlet übersetzt und anschließend ausgeführt.

JVM (*JVM*; Java Virtuelle Maschine) Bezeichnung für Java-Interpreter, die den Java-Bytecode zur Laufzeit analysieren und interpretieren (Java-Laufzeitumgebung). (Syn.: Java virtual machine, VM, Virtuelle Maschine)

Komponente (*component*) 1. Anwendungsorientierter, in sich abgeschlossener, wiederverwendbarer, binärer Softwarebaustein, der nach außen eine Schnittstelle mit Funktionen zur Verfügung stellt, die semantisch zusammengehören. 2. In der UML 2 ist eine Komponente eine Spezialisierung der Klasse. Ihr Verhalten wird durch bereitgestellte und benötigte Schnittstellen spezifiziert. (Syn.: Halbfabrikat)

Lebenszyklus (*life cycle*) Gesamte Lebensdauer eines Produkts von seiner Entwicklung (Geburt) über seinen Betrieb bis hin zu seiner »Außer-Betriebnahme« (Tod).

Lokalisierung (*localisation*) Vorgang im Rahmen der Softwareentwicklung: Anpassung und Übersetzung von Software unter Berücksichtigung der für die Zielgruppe relevanten lokalen Verhältnisse (Abk.: L10N)

Maintenance (*maintenance*) Oberbegriff für Wartung und Pflege.

MVC-Muster (*MVC pattern*) Entwurfsmuster, das häufig bei der Realisierung von dialogorientierten Anwendungen verwendet wird. Das Muster trennt zwischen der anwendungsinternen Repräsentation der darzustellenden Daten (*Model*), der Darstellung der Benutzeroberfläche (*View*) und der Komponente zur Steuerung der Interaktion des Benutzers mit der Oberfläche (*Controller*). Durch die Trennung wird eine größtmögliche Unabhängigkeit zwischen der Anwendungslogik, der Darstellung von Daten auf der Benutzeroberfläche und Steuerung der Interaktion ermöglicht.

Namensdienst (*naming service*) Eine Anwendung, die ein Objekt unter einem eindeutigen Namen (*String*) registriert. Ein anderes Objekt kann über den Namen eine Referenz auf das registrierte Objekt erhalten. Namensdienste kom-

men vorwiegend in verteilten Systemen zum Einsatz. Dort liefern Sie Referenzen auf Objekte, die meist auf einem entfernten Computer-System liegen.

Objektidentität (*object identity*) Jedes Objekt besitzt eine Identität, die es von allen anderen Objekten unterscheidet. Selbst wenn zwei Objekte zufällig dieselben Attributwerte besitzen, haben sie eine unterschiedliche Identität. Im Speicher wird die Identität durch unterschiedliche Adressen realisiert.

Objektorientiertes Datenbanksystem (*objectoriented database system*) Speichert Daten entsprechend dem objektorientierten Datenmodell. Objekte werden mit ihren Attributen, Beziehungen (Vererbung, Assoziation, Aggregation) und Operationen gespeichert, wobei Attribute beliebige, auch selbstdefinierte Typen besitzen können.

ODBC (*ODBC; Open Database Connectivity*) Standardisierte Schnittstelle für den Zugriff auf relationale Datenbanksysteme. Wurde ursprünglich von Microsoft spezifiziert, hat sich aber zu einen betriebssystemübergreifenden, allgemein akzeptierten de facto-Standard entwickelt. Siehe auch: JDBC.

OID-Attribut (*OID attribute*) Schlüsselattribut in der Tabelle einer relationalen Datenbank, das keinerlei fachliche Bedeutung besitzt. OID ist die Kurzform für Objektidentität (*object identity*).

OMG (*Object Management Group*) Konsortium von über 800 Mitgliedern zur Schaffung von Industriestandards für objektorientierte Anwendungen.

Orchestrierung (*orchestration*) Unter Orchestrierung versteht man die Organisation des Zusammenspiels der einzelnen, von verschiedenen Systemen bereit gestellten Services (i.d.R. Webservices) im Rahmen eines Prozesses.

Paket (*package*) Fasst Modellelemente (z.B. Klassen) zusammen. Ein Paket kann selbst Pakete enthalten. Es wird benötigt, um die Systemstruktur auf einer hohen Abstraktionsebene zu modellieren.

Persistenz (*persistence*) Persistenz bezeichnet die Fähigkeit, Daten oder Objekte auf nicht-flüchtigen Speichermedien zu speichern, so dass diese unab-

hängig von einem Programm und dessen Arbeitsspeicherbelegung zur Verfügung stehen. Die Daten »überleben« also das Programm.

Pflege Lokalisierung und Durchführung von Änderungen und Erweiterungen in Softwaresystemen, die in Betrieb sind, wenn die Art der gewünschten Modifikationen festliegt.

Portabilität (*portability*) Fähigkeit eines Softwaresystems, sich an andere Systeme anzupassen, mit anderen Systemen zusammenzuarbeiten, Informationen anderen Systemen auszutauschen und in definierten Umgebungen installierbar zu sein. (Syn.: Übertragbarkeit)

Prinzip (*principle*) Ein Prinzip ist ein Grundsatz, den man seinem Handeln zugrunde legt.

Reengineering (*reengineering*) Alle Aktivitäten zur Änderung von Software-Altsystemen, um sie entsprechend vorgegebenen Zielsetzungen zu überarbeiten. Voraus gehen muss in der Regel ein *Reverse Engineering*. (Syn.: Renovierung, Renovation, Reclamation)

Regressionstest (*regression testing*) Wiederholung der bereits durchgeführten Tests nach Änderung des Programms. Er dient zur Überprüfung der korrekten Funktion eines Programms nach Modifikationen, z.B. Fehlerkorrekturen.

Relationales Datenbanksystem (*relational database system*) Speichert Daten entsprechend dem relationalen Datenmodell, d.h. in Form von Tabellen mit Primär- und Fremdschlüsseln. (Abk.: RDBS)

Restrukturierung (*refactoring*) Bei der Restrukturierung wird ein Softwaresystem so geändert, dass das externe Verhalten des Codes nicht geändert, aber die interne Struktur verbessert wird.

Reverse Engineering (*reverse engineering*) Vorgehensweise, mit der versucht wird, ein Software-Altsystem anhand des laufenden Systems und der vorhandenen Unterlagen zu verstehen. Fehlende Informationen, Dokumente und Modelle werden durch manuelle und/oder automatische Analysen ab-

Glossar

geleitet. Das Altsystem wird in seiner Funktionalität und Semantik nicht verändert. (Syn.: Rückwärts-Entwicklung)

RMI-Compiler (*RMI compiler*) Ein Generator, der Java-Bytecode-Dateien analysiert und für Klassen, die eine Remote-Schnittstelle implementieren, Stummel-Objekte erzeugt.

Rolle (*role*) Eine Rolle stellt eine Zusammenfassung von Aufgaben dar, die in einem bestimmten Kontext (z. B. bei der Durchführung eines Geschäftsprozesses) i.d.R. von einer Person durchgeführt werden. Rollen müssen Mitarbeitern nicht fest zugeordnet sein. Ein Mitarbeiter kann je nach Situation unterschiedliche Rollen einnehmen – auch mehrere Rollen gleichzeitig.

Rollout (*rollout*) Von einem Rollout spricht man, wenn eine in einem Unternehmensbereich eingeführte und getestete Software von diesem kleinen Bereich in das gesamte Unternehmen, d. h. alle anderen Bereiche oder Niederlassungen, verbreitet wird.

Rundreise (*round trip*) Wenn ein Client über eine Netzverbindung auf einen Server zugreift, sendet er die Anfrage als Datenpaket an den Server. Dieser wertet das Datenpaket aus, bearbeitet die darin enthaltene Anfrage und schickt die Antwort wieder als Datenpaket über die Netzverbindung zum Client zurück. Dieser wertet das Paket aus und kann mit der Antwort weiterarbeiten. Dieser Zyklus wird als Rundreise bezeichnet.

Servlet (*servlet*) Java-Programm, das auf einem Webserver läuft, Anfragen von Clients entgegennimmt und HTML-Code als Ausgabe erzeugen kann. JSPs werden von einem JSP-Server in Servlets übersetzt. (Syn.: Java-Servlet)

Single Sign-On (*single sign-on*) Der Begriff bezeichnet den Vorgang einer einmaligen Anmeldung an einer Softwareanwendung, die dann während der gesamten Verbindung der Anwender auch über Systemebenen hinweg Gültigkeit hat. Als Beispiel kann die Anmeldung über einen Webbrower an einer verteilten Java EE-Anwendung gelten, die dann nach erfolgreicher Authentifizierung Gültigkeit sowohl für den Web-Container als auch für den EJB-Contai-

ner und für die dabei genutzten Datenquellen hat. (Abk.: SSO; Syn.: Einmalanmeldung)

Singleton-Muster (*singleton pattern*) Objektbasiertes Entwurfsmuster, das sicherstellt, dass eine Klasse genau ein Objekt besitzt. Ermöglicht einen globalen Zugriff auf dieses Objekt.

SOA (*service oriented architecture*; Serviceorientierte Architektur) Ansatz zum Aufbau komplexer, verteilter Informationssysteme mit lose gekoppelten Komponenten. Die beteiligten Systeme und Komponenten stellen hierbei ihre Funktionalitäten zumeist in Form von Webservices zur Verfügung. Häufig erfolgt die Koordination der Webservice-Aufrufe über einen *Enterprise Service Bus* (ESB) oder ein *Business Process Management-System* (BPMS) als gemeinsame Plattform.

SOA-Service (*SOA service*) Ein Service im Rahmen einer SOA(rchitektur) ist ein lose gekoppelter Anwendungsbaustein, der eine klar umrissene fachliche Aufgabe erledigt.

SOAP (*SOAP*; Simple Object Access Protocol) Protokoll, das Programmen die Kommunikation ermöglicht. Dabei spielt es keine Rolle, ob beide unter dem gleichen oder verschiedenen Betriebssystemen laufen. SOAP basiert auf dem HTTP-Protokoll und XML für den Informationsaustausch. Es legt genau fest, wie ein *HTML-Header* und eine XML-Datei aufgebaut sein müssen, damit sich Programme aufrufen und Daten austauschen können. Ein Vorteil von SOAP ist, dass es *Firewalls* durchdringen kann.

Softwarearchitektur (*software architecture*) Strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen.

SQL (*SQL*; structured query language) Sprache der 4. Generation, heute bei relationalen Datenbanksystemen am weitesten verbreitet und standardisiert. Umfasst Kommandos zur Datendefinition (DDL) und zur Datenmanipulation (DML) sowie Kommandos zum Vergeben von Zugriffsberichtigungen. SQL 99 erweitert das relationale Datenmodell zu einem objekt-relationalen Datenmodell.

Stakeholder (*stakeholder*) Person oder Organisation, die eigene Interessen bei einer Softwareentwicklung oder beim späteren Einsatz vertritt. (Syn.: Akteure, Anspruchsberechtigte, Interessenvertreter)

Stereotyp (*stereotype*) In der UML werden Modell-Elemente mit einem Stereotyp markiert, um eine spezielle Ausprägung des eigentlichen Modell-Elements anzuzeigen. Eine Klasse kann z.B. mit dem Stereotyp Bean bezeichnet werden. Die Klasse ist dann eine spezielle Art von Klasse, eben ein Bean.

Stummel-Objekt (*stub*) Ein lokaler Stellvertreter für ein Objekt auf einem anderen Computer-System oder in einer anderen JVM. Über das Stummel-Objekt können Operationen des entfernten Objekts aufgerufen werden. Für einen Client sieht der Aufruf wie ein lokaler Aufruf aus.

Transaktion (*transaction*) Eine Transaktion bezeichnet eine komplett an einem Stück auszuführende Aktion in einer Datenbank oder einem Anwendungssystem. Kann eine Transaktion nicht erfolgreich abgeschlossen werden, so wird der Zustand wiederhergestellt, der vor Beginn der Transaktion vorlag.

UDDI (*UDDI*; Universal Description, Discovery and Integration) Konzept für die Entwicklung von Verzeichnisdiensten für Komponenten, die als Webservices über das Internet angeboten werden. Vorstellbar ist dies als eine Art »Telefonbuch« für Webservices. Darin werden Informationen zur Anbieter-Firma, zu den Serviceleistungen und technische Informationen geliefert. XML ist die zugrundeliegende Datenstruktur.

UML (*Unified Modeling Language*) Notation zur grafischen Darstellung von objektorientierten Konzepten. Zur grafischen Darstellung gehören unter anderem Klassendiagramme, Sequenzdiagramme und Aktivitätsdiagramme.

URI (*uniform ressource identifier*) Name zur eindeutigen Identifizierung von Ressourcen. Syntaktische Struktur des Namens standardisiert durch IETF in RFC 3986. Spezielle URIs sind URLs zur Adressierung von Webseiten im Web.

URL (*uniform ressource locator*) Im Web verwendete standardisierte Darstellung von Internetadressen. Eine URL enthält das verwendete Zugriffsprotokoll (z.B. HTTP) und den Ort der Ressource. Aufbau: protokoll://domain-Name/Dokumentpfad. URLs sind eine Unterart der allgemeineren URIs (*Uniform Ressource Identifier*). (Syn.: Adresse, Web-Adresse)

Use Case (*use case*) Sequenz von zusammengehörenden Transaktionen, die von einem Akteur im Dialog mit einem System ausgeführt werden, um ein Ergebnis von messbarem Wert zu erstellen. Messbarer Wert bedeutet, dass die durchgeführte Aufgabe einen sichtbaren, quantifizierbaren und/oder qualifizierbaren Einfluss auf die Systemumgebung hat. Eine Transaktion ist eine Menge von Verarbeitungsschritten, von denen entweder alle oder keiner ausgeführt werden. (Syn.: Anwendungsfall)

Verbalisierung (*verbalization*) Gedanken und Vorstellungen in Worten ausdrücken und damit ins Bewusstsein bringen. In der Softwaretechnik bedeutet dies, aussagekräftige Namen und geeignete Kommentare zu wählen und selbstdokumentierende Konzepte, Methoden und Sprachen einzusetzen.

Voice over IP (*voice over IP*) Übertragung von Telefongesprächen über IP-basierte Netzwerke – insbesondere das Internet. Kurzform: VoIP. Auch Internet-Telefonie genannt.

Wartbarkeit (*Maintainability*) Änderungsfähigkeit eines Softwareprodukts während der Betriebsphase. Die Wartbarkeit wird beeinflusst durch die Analysierbarkeit, die Änderbarkeit, die Stabilität und die Testbarkeit eines Softwaresystems. Der Begriff ist in der Norm ISO/IEC 9126-1 als Qualitätsmerkmal definiert.

Wartung Beseitigung, d.h. Lokalisierung und Behebung, von Fehlerursachen in Softwaresystemen, die in Betrieb sind, wenn die Fehlerwirkung bekannt ist.

Web-Architektur (*web architecture*) Verteilung einer – in mehrere logische Software-Schichten (Schichtenarchitektur) gegliederten – Anwendung auf ein Netzwerk (Internet, Extranet, Intranet),

Glossar

das aus vielen Web-Clients (Clients, auf denen ein Web-Browser läuft), mindestens einem Web-Server sowie Anwendungs- und Daten-Server besteht. Die Verbindung zwischen Web-Client und Web-Server erfolgt über das HTTP-Protokoll, das pro Anforderung jeweils eine neue Verbindung aufbaut.

Webservice (*web service*) Ein Webservice ist ein von einem Informationssystem bereitgestellter, von anderen Systemen über das Internet nutzbarer Dienst auf der Basis von Internet-Standards. Das andere System kann über den Webservice Daten abrufen oder Funktionen aufrufen. Webservices können somit genutzt werden, um verschiedene Systeme zu integrieren. Webservices nutzen insbesondere die drei folgenden Standards: SOAP zum Nachrichtenaustausch, *Web Services Description Language* (WSDL) zur Beschreibung der durch einen Webservice angebotenen Dienste und UDDI (*Universal Description, Discovery and Integration*) zum Aufbau von Verzeichnisdiensten, über die Webservices gefunden werden können.

Weiterentwickelbarkeit (*evolvability*) Fähigkeit eines Softwaresystems, auch gravierende Änderungen der Anforderungen und des Umfelds ohne Verletzung der Architekturintegrität zu überstehen. (Syn.: Nachhaltigkeit)

WSDL (*WSDL; Web Services Description Language*) Standard, der festlegt, wie die von einem Webservice angebotenen Funktionen mit Hilfe von XML beschrieben werden können. Mit Hilfe der WSDL-Beschreibung eines Webservice kann ein anderes System ermitteln, wie es auf diesen Service zugreifen kann.

XML (*eXtensible Markup Language*) 1 Universell einsetzbare Sprache zum Austausch strukturierter Informatio-

nen. Basiert – wie die *Standard Generalized Markup Language* (SGML) – auf der Trennung von Inhalt und Struktur.

2 Eine Sprache (oder Meta-Sprache) zur Beschreibung der inhaltlichen Struktur von Dokumenten. XML ist ein W3C-Standard und in der Industrie weit verbreitet.

XML (*Extensible Markup Language*) Sprache zur Beschreibung der inhaltlichen Struktur von Dokumenten, die sowohl von Menschen als auch von Maschinen gelesen werden kann. XML ist ein W3C-Standard und in der Industrie weit verbreitet.

XML-RPC (*XML-RPC*) Die aufzurufende Methode und ihre Parameter werden bei XML-RPC in ein abstraktes XML-Dokument verpackt und über das HTTP-Protokoll an den Empfänger auf einem anderen Computersystem verschickt. Der Empfänger extrahiert die Daten, ruft die entsprechende Methode auf und gibt das Ergebnis wieder in Form eines XML-Dokuments über HTTP an den Sender zurück.

Zuverlässigkeit (*reliability*) Die Zuverlässigkeit eines Software-Produktes ist die Fähigkeit der Software, ihr Leistungs niveau unter festgelegten Bedingungen über einen festen Zeitraum zu bewahren.

Zwei-Phasen-Commit-Protokoll (*two phase commit protocol*) Bezeichnet das zweistufige Verfahren bei Ausführung verteilter Transaktionen. Vor dem dauerhaften Ablegen der bearbeiteten Datenbestände werden zunächst alle beteiligten Datenquellen über das 2PC-Protokoll zu ihrer Bereitschaft zur Speicherung (*prepared*) befragt. Erst nach Bestätigung aller Datenquellen werden die Schreibvorgänge dauerhaft ausgeführt. (Abk.: 2PC)

Literatur

[ACM03]

Alur, Deepak; Crupi, John; Malks, Dan; *core J2EE Patterns – Best Practices and Design Strategies*, Upper Saddle River, Prentice Hall PTR, 2003.

[Alla02]

Allaire, Jeremy; *Macromedia Flash MX – A next-generation rich client*, 2002, <http://ebookbrowse.com/macromedia-rich-client-allaire-2002-pdf-d31002764>.

Abgerufen am 7.1.2011.

[Balz08]

Balzert, Helmut; *Lehrbuch der Softwaretechnik – Softwaremanagement*, 2. Auflage, Heidelberg, Spektrum Akademischer Verlag, 2008.

[Balz09a]

Balzert, Helmut; *Lehrbuch der Softwaretechnik – Basiskonzepte und Requirements Engineering*, 3. Auflage, Heidelberg, Spektrum Akademischer Verlag, 2009.

[Balz09b]

Balzert, Heide; *UML 2 in 5 Tagen – Der schnelle Einstieg in die Objektorientierung*, 2. Auflage, Herdecke, W3L-Verlag, 2009.

[Balz11]

Balzert, Helmut; *Java: Objektorientiert programmieren – Vom objektorientierten Analysemodell bis zum objektorientierten Programm*, 2. Auflage, Herdecke, W3L-Verlag, 2011.

[BBR09]

Brcina, R.; Bode, S.; Riebisch, M.; *Optimisation Process for Maintaining Evolvability during Software Evolution*, in: Engineering of Computer Based Systems, 2009. ECBS 2009. 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2009, S. 196–205.

[Berg04]

Berg, Klaus; *Authentifizierung und Autorisierung mit JAAS*, in: JavaSPEKTRUM, 1/2004, S. 10–15.

[BFB+07]

Bauler, Pascal; Feltz, Fernand; Biri, Nicolas; Pinheiro, Philippe; *Entwurf von serviceorientierten Architekturen auf Basis von Open-Source-Software*, in: Praxis der Wirtschaftsinformatik, Februar 2007, S. 57- 65.

[BHS07]

Buschmann, F.; Henney, K.; Schmidt, D.; *Pattern-Oriented Software Architecture Vol. 4: A Pattern Language for Distributed Computing*, John Wiley & Sons, 2007.

[BHS07b]

Buschmann, F.; Henney, K.; Schmidt, D.; *Past, Present, and Future Trends in Software Patterns*, in: IEEE Software, July / August 2007, S. 31–37.

[Biro97]

Birolini, A.; *Zuverlässigkeit von Geräten und Systemen*, Berlin, Heidelberg, New York, Springer, 1997.

[BKM07]

Bianco, Phil; Kotermanski, Rick; Merson, Paulo; *Evaluating a Service-Oriented Architecture*, 2007, <http://www.sei.cmu.edu/reports/07tr015.pdf>.

Literatur

- [BMR+96] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael; *Pattern-Oriented Software Architecture – A System of Patterns*, New York, Wiley, 1996.
- [BuHe10a] Buschmann, Frank; Henney, Kevlin; *Five Considerations for Software Architecture, Part I*, in: IEEE Software, May/June 2010, S. 63–65.
- [BuHe10b] Buschmann, Frank; Henney, Kevlin; *Five Considerations for Software Architecture, Part II*, in: IEEE Software, July/August 2010, S. 12–14.
- [Cold04] Coldewey, Jens; *Stabile Veränderungen – Refactoring im Projekteinsatz*, in: OBJEKTSPEKTRUM, 1/2004, S. 71–76.
- [DDN09] Demeyer, Serge; Ducasse, Stéphane; Nierstrasz, Oscar; *Object-Oriented Reengineering Patterns*, Square Bracket Associates, 2009.
- [DePe02] Denninger, Stefan; Peters, Ingo; *Enterprise JavaBeans 2.0*, Addison-Wesley, 2002.
- [DiGi09] DiPippo, L.; Gill, C.D.; *Design Patterns for Distributed Real-Time Embedded Systems*, Springer, 2009.
- [Drac05] von Drachenfels, Heiko; *Komponentenorientierte Programmierung im Kleinen*, in: Informatik-Spektrum, 22. April 2005, S. 136–143.
- [Duck95] Duckett, Mike; *The Two-Phase Commit Protocol*, 1995, <http://ei.cs.vt.edu/~cs5204/sp99/distributedDBMS/duckett/tppc.html>.
Abgerufen am 21.8.2010.
- [EgGr04] Egyed, Alexander; Grünbacher, Paul; *Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help*, in: IEEE Software, November/December 2004, S. 50–58.
- [EiSt07] Eilebrecht, Karl; Starke, Gernot; *Patterns kompakt*, 2. Auflage, München, Spektrum Akademischer Verlag, 2007.
- [Enge02] Engels, Horst; *CAN-Bus – Technik einfach, anschaulich und praxisnah vorgestellt*, 2. Auflage, München, Franzis, 2002.
- [Fiel00] Fielding, Roy Thomas; *Architectural Styles and the Design of Network-based Software Architectures*, 2000, <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [Fow02] Fowler, Martin; *Patterns of Enterprise Application Architecture*, 1. Auflage, Addison-Wesley, 2002.
- [Fowl97] Fowler, Martin; *Analysis Patterns – Reusable Object Models*, Menlo Park, Addison Wesley, 1997.
- [Fowl05] Martin Fowler; *Refactoring – Improving the Design of Existing Code*, 17. Auflage, Boston, Addison Wesley, 2005.

Literatur

[Fowl06]

Fowler, Martin; *GUI Architectures*, 2006, <http://martinfowler.com/eaaDev/uiArchs.html>.

[GHJ+95]

Gamma, Erich; Richard Helm; Ralph Johnson; John Vlissides; *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston, Addison Wesley, 1995.

[GHJ+96]

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John; *Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software*, 1. Auflage, Addison Wesley, 1996.

[Glin07]

Glinz, Martin; *On Non-Functional Requirements*, in: 15th IEEE International Requirements Engineering Conference, DOI 10.1109/RE.2007.45, 2007, S. 21–26.

[Glin08]

Glinz, Martin; *A Risk-Based, Value-Oriented Approach to Quality Requirements*, in: IEEE Software, March/April, 2008, S. 34–41.

[Gonc09]

Goncalves, Antonio; *Beginning Java EE 6 Platform with GlassFish 3: From Novice to Professional*, Apress, 2009.

[HaOl07]

Haft, Martin; Ollek, Bernd; *Komponentenbasierte Client-Architektur*, in: Informatik-Spektrum, 30.3.2007, S. 143–158.

[HeZe03]

Hein, Manfred; Zeller, Henner; *Java WebServices. Entwicklung plattformübergreifender Dienste mit XML und SOAP*, Addison Wesley, 2003.

[HoLa10]

Hosseini, Reza; Lauer, Oliver; *Wiederverwendung von Frontend-Anwendungen in einer SOA*, in: JavaSPEKTRUM, 6/2010, S. 26–29.

[ISO 26262]

ISO 26262: ISO-Norm für sicherheitsrelevante elektrische/elektronische Systeme in Kraftfahrzeugen, <http://www.iso.org>.

[JTA10]

Oracle; *Java Transaction API (JTA)*, 2010, <http://www.oracle.com/technetwork/java/javase/tech/jta-138684.html>.

Abgerufen am 21.8.2010.

[Kras88]

Krasner, Glenn; Pope, Stephen; *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*, in: Journal Of Object Oriented Programming (JOOP), August/September 1988, 1988.

[Leis11]

Leisegang, Christoph; *Model View Presenter: Entwurfsmuster für Rich Clients – Schlüsselfigur*, in: iX, 1/2011, S. 128–133.

[LGK+99]

Lai, Charlie; Gong, Li; Koved, Larry; Nadalin, Anthony; Schemers, Roland; *User Authentication and Authorization in the Java Platform*, in: 15 th Annual Computer Security Applications Conference, 1999.

[LHR88]

Lieberherr, Karl J.; Holland, Ian M.; Riel, Arthur; *Objectoriented Programming: An objective sense of style*, in: Proceedings OOPSLA '88, ACM SIGPLAN Notices, Vol. 23, 1988, S. 323–334.

[Ligg09]

Liggesmeyer, Peter; *Software-Qualität: Testen, Analysieren und Verifizieren von Software*, 2. Auflage, Heidelberg, Spektrum Akademischer Verlag, 2009.

Literatur

- [Ling11] Lingstädt, Dirk; *NoSQL- Einsatzgebiete für die neue Datenbank-Generation*, in: JavaSPEKTRUM, 1/2011, S. 11–15.
- [MaBr01] Malan, Ruth; Bredemeyer, Dana; *Defining Non-Functional Requirements*, 2001, http://www.bredemeyer.com/pdf_files/NonFunctReq.PDF. Abgerufen am 24. Mai 2010.
- [Maie11] Maier, Peter; *Architektur-Bewertung mit Methode: Die "Architecture Tradeoff Analysis Method" (ATAM) im Überblick*, in: OBJEKTspektrum, 1/2011, S. 73–79.
- [MSDN01] *Developing Components*, [http://msdn.microsoft.com/en-us/library/51sc2s5c\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/51sc2s5c(VS.71).aspx).
- [MSDN02] *Simplifying Deployment and Solving DLL Hell with the .NET Framework*, <http://msdn.microsoft.com/en-us/library/ms973843.aspx>.
- [MSDN03] *Calling a .NET Component from a COM Component*, <http://msdn.microsoft.com/en-us/library/ms973802.aspx>.
- [Müll10] Müller, Frank; *Es geht auch einfach – Datenbanken ohne SQL und Relationen*, in: iX, 2/2010, S. 122–126.
- [MZn10] Mairiza, Dewi; Zowghi, Didar; Nurmuliani, Nurie; *An Investigation into the Notion of Non-Functional Requirements*, in: ACM – SIGAPP – SAC'10, Sierre, Switzerland, March 22–26, 2010, S. 311–317.
- [Nand11] Nandico, Oliver; *Integration für den Benutzer: Einheitliche Benutzungsoberfläche mit Interaktionsservices*, in: OBJEKTspektrum, 2/2011, S. 71–75.
- [Nati09] *Nationale Roadmap Embedded Systems*, 2009, <http://www.zvei.de/index.php?id=1829>.
- [NeSc10] Neubauer, André; Schmidt, Stephan; *Die Webarchitekturen SOFEA und SOUI*, in: JavaSPEKTRUM, 1/2010, S. 15–18.
- [NoWe00] Noble, J.; Weir, C.; *Small Memory Software: Patterns for Systems with Limited Memory*, Addison-Wesley, 2000.
- [OMG09] *Service oriented architecture Modeling Language (SoaML)*, OMG, 2009, <http://www.omg.org/cgi-bin/doc?ptc/09-04-01.pdf>.
- [OMG09a] Object Management Group; *OMG Unified Modeling Language (OMG UML), Superstructure*, V2.2, 2009, <http://www.omg.org/spec/UML/2.2/Superstructure>.
- [OMG96] *Common business objects and business object facility*, Hrsg. Object Management Group Common Facilities RFP-4, OMG document cf/96-01-04, 1996.
- [Opit10] Opitz, Matthias; *Die Auswirkungen nichtfunktionaler Anforderungen auf Softwarearchitekturen demonstriert anhand einer Fallstudie*, Master-Arbeit, Lehrstuhl für Softwaretechnik, Ruhr-Universität Bochum, 2010.

Literatur

[PBG07]

Posch, Torsten; Birken, Klaus; Gerdom, Michael; *Basiswissen Softwarearchitektur*, 2. Auflage, Heidelberg, dpunkt.verlag, 2007.

[PiRö07]

Pieper, Daniel; Röttgers, Carsten; *WCF, SCA oder JBI?*, in: JavaSPEKTRUM, 5/2007, S. 8–12.

[Pont01]

Pont, M.J.; *Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers*, Addison-Wesley, 2001.

[Powe99]

Powell Douglass, Bruce; *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Boston, Addison-Wesley Professional, 1999.

[RaBe00]

Rajlich, Václav T.; Bennett, Keith H.; *A Staged Model for the Software Life Cycle*, in: Computer, July 2000, S. 66–71.

[ReHa09]

Reussner, Ralf; Hasselbring, Wilhelm; *Handbuch der Software-Architektur*, Heidelberg, dpunkt.verlag, 2009.

[Reif11]

Reif, Andreas; *Continuous Integration – Prinzipien und Tools*, in: JavaSPEKTRUM, 2/2011, S. 31–35.

[RiBo09]

Riebisch, Matthias; Bode, Stephan; *Software-Evolvability*, in: Informatik-Spektrum, 4/2009, S. 339–343.

[SaBh08]

Sarma, Vinod; Bhagavatula, Srinivas Rao; *Freeway patterns for SOA Systems*, in: Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP '08), ACM, 2008.

[SAE AS5506]

SAE AS5506 Standard: Sprache zur Beschreibung der Softwarearchitektur und der Ausführungsplattform von Performanz-kritischen, eingebetteten, Echtzeit-systemen, 2004, <http://standards.sae.org/as5506/>.

[SaSc75]

Saltzer, J.H.; Schroeder, M.D.; *The Protection of Information in Computer Systems*, in: Proceedings of the IEEE, Sept. 1975, S. 1278–1308.

[SFH+06]

Schumacher, Markus; Fernandez-Buglioni, Eduardo; Hybertson, Duane; Buschmann, Frank; Sommerlad, Peter; *Security Patterns: Integrating Security and Systems Engineering*, John Wiley & Sons, 2006.

[SGS+10]

Soares, Gustavo; Gheyi, Rohit; Serey, Dalton; Massoni, Tiago; *Making Program Refactoring Safer*, in: IEEE Software, July/August 2010, S. 52–57.

[ShGa96]

Shaw, M.; Garlan, D.; *Software Architecture: Perspektives on an Emerging Discipline*, Upper Saddle River, Prentice Hall, 1996.

[Snee83]

Sneed, Harry; *Software-Wartungsorganisation*, in: Tagungsband des Struktur-Kongresses 83, CDI, Frankfurt, Nov. 1983.

[SNL06]

Steel, Christopher; Nagappan, Ramesh; Lai, Ray; *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, Upper Saddle River, Prentice Hall International, 2006.

Literatur

- [SoAr]
SoftwareArchitectures.com; *Quality Attributes*, 2010, <http://www.softwarearchitectures.com/go/Discipline/DesigningArchitecture/QualityAttributes/tabid/64/Default.aspx>.
Abgerufen am 24. Mai 2010.
- [Stal06]
Stal, Michael; *Using Architectural Pattern and Blueprints for Service-Oriented Architecture*, in: IEEE Software, March/April 2006, S. 4-61.
- [Stal09]
Stal, Michael; *Das beste Rezept*, in: iX, 4/2009, S. 140-144.
- [Star05]
Starke, Gernot; *Effektive Software-Architekturen – ein praktischer Leitfaden*, 2. Auflage, München, Hanser-Verlag, 2005.
- [Stec10]
Steck, Werner; *SOA-Frontends: Serviceorientierte Ansätze helfen bei der Konsolidierung von Client-Landschaften*, in: Wirtschaftsinformatik und Management, 4/2010, S. 24-28.
- [Stee06]
Steel, Christopher; *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, Prentice Hall, 2006.
- [Sun02]
Core J2EE Patterns, 2002, http://java.sun.com/blueprints/core_j2eepatterns/Patterns, 2001-2002.
- [Sun09]
The Java EE 6 Tutorial, Volume 1, Basic Concepts, Santa Clara, USA, Sun microsystems, 2009.
- [TLS10]
Thomas, Oliver; Leyking, Katrina; Scheid, Michael; *Serviceorientierte Vorgehensmodelle: Überblick, Klassifikation und Vergleich*, in: Informatik-Spektrum, 4/2010, S. 363-378.
- [TVS10]
Telea, Alexandru C.; Voinea, Lucian; Sassenburg, Hans; *Visual Tools for Software Architecture Understanding: A Stakeholder Perspective*, in: IEEE Software, Nov./Dec. 2010, S. 46-53.
- [Uta05]
Utas, G.; *Robust Communications Software: Extreme Availability, Reliability, and Scalability for Carrier-Grade Systems*, John Wiley & Sons, 2005.
- [Vine10]
Vine, Andrea; *I18n in Software Design, Architecture and Implementation*, 2010, <http://developers.sun.com/dev/gadc/technicalpublications/articles/archi18n.html>.
Abgerufen am 25.7.2010.
- [VKZ02]
Voelter, M.; Kircher, M.; Zdun, U.; *Remoting Patterns – Patterns for Enterprise, Realtime and Internet Middleware*, John Wiley & Sons, 2004.
- [W3C04]
Haas, Hugo; Brown, Allen; *Web Services Glossary – W3C Working Group Note 11 February 2004*, 2004, <http://www.w3.org/TR/ws-gloss/>.
- [W3Ce04]
Web Services Description Language (WSDL), 2004, <http://www.w3.org/TR/2004/WD-wsdl10-primer-20041221/>.

Literatur

[Weße06]

Weßendorf, Matthias; *Web Services & Mobile Clients*, Herdecke, W3L-Verlag, 2006.

[WHS11]

Welke, Richard; Hirschheim, Rudy; Schwarz, Andrew; *Service-Oriented Architecture Maturity*, in: Computer, February 2011, S. 61–67.

[Wieg03]

Wiegers, Karl E.; *Software Requirements*, 2. Auflage, Microsoft Press, 2003.

[Wißm09]

Wißmann, Dieter; *JavaServer Pages – Dynamische Websites mit JSP erstellen*, 2. Auflage, Herdecke, W3L-Verlag, 2009.

[WoEm10]

Woods, Eoin; Emery, David; *Is UML Sufficient for Describing Architectures?*, in: IEEE Software, Nov./Dec. 2010, S. 54, 56.

[ZiAr10]

Ziesche, Peter; Arinir, Doga; *Java: Nebenläufige & verteilte Programmierung – Konzepte, UML 2-Modellierung, Realisierung mit Java*, 2. Auflage, Herdecke, W3L-Verlag, 2010.

Sachindex

A

Ablösung 1
Abnahme 527, **527**
Abnahmeprotokoll 527, 530
Abnahmetest 527
Abschaltung 1
Abstrakte Fabrik **462**, 505
Abstrakte Schnittstelle 505
Abstraktionsebene 26
ACID-Prinzipien 178
ActiveX **198**
Adobe Flash **198**
ADO.NET 444
AJAX **198**, 364, 473
Aktualisierung
 automatische 525
ALMA 491
Altsystem 119, **532**
Analysierbarkeit 110, 116, 119
Änderbarkeit 110, 116, 119
Änderung 539
Änderungsmanagement 536
Anforderung
 funktional 109
 nichtfunktional 109
Angemessenheit 110
Annotation **434**
Anpassbarkeit 111, 132
Anpassung 539
Antwortzeit 128
Anwendungsart 135
Anwendungssystem
 Kontext 137
AppFabric 307
architecture pattern 37
Architektur
 Einflussfaktoren 135
 logische 16, 26
 Peer-to-Peer 192
 physische 27, 191
 QS 489
 QS-Verfahren 489
 Sichten 23
 SOA 201
 Statische Sicht 16
 verteilt 191
Architekturanalyse 489
Architekturmuster 37

Architekturprinzip
 Ökonomie 32
 Konzeptionelle Integrität 30
 Selbstorganisation 34
 Sichtbarkeit 34
 Symmetrie 33
 Trennung von
 Zuständigkeiten 31
Artefakt **9**
artifact 9
ASP.NET 364, **469**
ASPX 470
Assoziation abbilden 430
ATAM 491
Attraktivität 110, 130
Ausführungsgeschwindigkeit
 129, 131
Auslastung 129
Austauschbarkeit 111, 133
Authentifizierung 122, **154**,
 162
Authentisierung **154**
authorization 154
Automatische Aktualisierung
 525
Autorisierung **154**, 165

B

Batch-Anwendung 136
Bausteinsicht 23
Bedienbarkeit 110, 130
Bedienungskomfort 131
Bedingungen
 in Polymorphie 563
Befehls-Muster 75
Beispiel
 Anzeigenverwaltung 506
 Börsenkurse 275
 Buchung WPF 476
 DemoCallback 45
 DemoSocketTCP 209
 DemoSocketUDP 214
 Fernbedienung 76
 Flugbuchung 187
 Flugbuchung ASP.NET 470
 HelloGlobal 146, 147
 HelloWorldEJB 329
 HelloWorldRMI 223

Immobilienverwaltung 509
MitteilungGlobal 148, 149
MWST-Funktion 43
Personaldaten 85
Prämie 502
Routenberechnung 104
Sortieren 97
Sparkasse 90
Staumelder 55, 56, 269
Staumelder-REST 290
Unterhaltungsgeräte 69, 72
Veranstaltung 514
Verein 515
Wetterstation 84
Belastungstest 527
Benutzbarkeit 110, 130
Benutzerdokumentation 530
Benutzerproduktivität 131
Benutzerreaktionszeit 131
Benutzerverwaltung 153
Benutzungsfreundlichkeit 131
Benutzungsoberfläche
 Binnenarchitektur 451
Beobachter-Muster 54
Betatest 529
Betrieb 2
Betriebsphase 2, 129, 131
Betriebssicherheit **121**, 122
BPEL 379
BPMM 379
Brauchbarkeit 131
Business Delegate 348

C

Callback-Mechanismus 42
Chat **192**
Client 230
Client-Server-Anwendung 524
Client-Server-Architektur 136,
 193
Client-Server-Beziehung 230
Code
 duplicierter 556
Command and Controller
 Strategy-Muster 416,
 command pattern 75
controller 63
CORBA **242**

Sachindex

- Architektur 243
Dienste 246
IDL 245
IIOP 245
IR 245
Java 247
ORB 243
CRUD 425
C/S-Architektur 193
CSS **198**
- D**
- DAO-Muster **425**
Data Transfer Object 420
Daten-zentrische Bausteinsicht 404
Daten-zentrische Kontextsicht 402
Datenverlust 129
DCOM 303
Deployment **525**
deployment diagram 11
Derby 438
design 6, 483
design pattern 37
Desktop-Anwendung 135
Dienstanbieter 230
Dienstgüte 109
Dienstleistung 311
Dienstnehmer 230
direkte Umstellung 529
Direktzugriff 424
Dokumentation 484
Drei-Schichten-Architektur **47**
Duplizierter Code 556
Durchsatz 129
- E**
- EAST-ADL 401
EDM 446
efficiency 128
Effizienz 110, **128**, 131
Einführung 528
Einführungsprotokoll 528, 530
Einfachheit 32
Einflussfaktoren 135
Wechselwirkungen 138
Eingabeaufforderung 229
Eingebettete Systeme 399, 400, 402
Anforderungen 400
Anforderungsdiagramm 401
Architekturmuster 408
Daten-zentrische Bausteinsicht 404
- Daten-zentrische Kontextsicht 402
EAST-ADL 401
Erkennungsmuster 408
Homogeneous Redundancy-Muster 411
PSC-Muster 409
Qualitätsanforderungen 401
SAE-AADL 401
Vermeidungsmuster 408
Verteilungssicht 406
Wiederherstellungsmuster 408
- Einheitlichkeit 30
Einprägsamkeit 131
Einrichtung 528
Einzelplatz-Anwendung 21, 135, 524
Emergenz 35
Enterprise JavaBeans 224
Entwerfen 483
Entwerfen im Großen 6
Entwerfen im Kleinen 6
Entwurf 6
Reihenfolge der Entscheidungen 485
Entwurf-durch-Innovation 37
Entwurf-durch-Routine 37
Entwurfsüberarbeitung 542
Entwurfsmuster **37**, 308
Business Delegate 348
Command and Controller Strategy 417
DAO 425
Fassade 313
Front Controller 417
J2EE 346
Java EE 346
MVP 459
Service to Worker 421
Session Facade 346
Sitzungsfassade 346
Transfer Object 420
Verteilte Anwendungen 308
View Helper 418
Wert-Objekt 311
- Entwurfsprozess 483
Entwurfssymmetrie 30
Ergonomie 131
Erkennungsmuster 408
Erlernbarkeit 110, 130
Erwartungskonformität 131
Erweiterbarkeit 119
Erweiterung 539
ESB **202**, **378**
- eSQL 447
evolvability 119
- F**
- Fabrik 505
Fabrik-Dienst 308, **309**
Fabrikmethode 426
Fabrikmethoden-Muster 89
facade pattern 69
Facelets 355
Fachkonzept 26
factory method pattern 89
Fallstudie
KV – Überblick 15
KV – CORBA 247
KV – Einzelplatz 21
KV – Globalisiert 151
KV – GUI 460
KV – JAAS 167
KV – JPA 438
KV – JTA 180
KV – REST 298
KV – RMI 237
KV – SOAP 284
KV – Sockets 216
KV – XML-RPC 257
KV mit Java EE als CS-Anwendung 335
KV mit Java EE als Web-Anwendung 353
KV mit Java EE und Webservices 379
Fassaden-Klasse **313**
Fassaden-Muster 69, 151
Fat-Client 194, 363
Fehlerraten 129
Fehlertoleranz 110, 125
Feinentwurf 6, 18
Forward Engineering **540**
Framework 25, **25**, 63
Fremdschlüssel 428
Front-Controller 417
Front-Controller-Muster 416
FTP 263
functional requirements 109
Funktionale Anforderungen 109
Funktionalität 110
Funktionssicherheit **123**
- G**
- g11n 143
Gebrauchstauglichkeit 110, **130**
Geheimnisprinzip 29

- Genauigkeit 110
Gesamtdokumentation 530
Gesamtprodukt 530
GlassFish 327
Globalisierung 143
 Java 145
 globalization 143
- H**
- Hüllklasse 314
high-level-design 6
Homogeneous
 Redundancy-Muster 411
HTTP **254**, 263, 265, 327
HTTPS 263, 327
- I**
- i18n 143
IDL 243, 245, 266
IEC 61508 10 123
implementieren 494
Implementierung 18, 494
 Prinzipien 497
Implementierungsprozess 494
Inbetriebnahme 528, 529
Individual-Software 522
Infrastruktur 17, 26, 137
Inkrementelle Installation 525
Installation 18, **524**
 auf dem Server 525
 inkrementell 525
Installationsprogramm 523
Installierbarkeit 111, 133
Instant Messaging **192**
Integrität 122
Integrität der Architektur 119
Internationalisierung **143**
internationalization 143
Interoperabilität 110, 363, 366
ISO/IEC 9126 110
ISO/IEC 9126-1 116, 121, 124,
 128, 130, 132
- J**
- JAAS 161, 326
JACC 326
JAF 325
JAR-Archiv 9
Java 366
 EJB 328
 JAAS 161, 326
 JACC 326
 JAF 325
 JavaMail 325
- JAX-RS 327
JAX-WS 327
JAXB 268
JAXP 327
JCA 325
JMS 325
JMX 325
JNDI 325, 330
JPA 325
JTA 180
Managed Bean 357
RMI 222, 237
SAAJ 268, 327
Sockets 208, 216
StAX 327
UDP 213
- Java-Applet **198**
Java Collection Framework **225**
Java EE 323, 362, 433
 Architektur 323
 EJB 328
 JNDI 330
 Java EE
 Session Bean 332
 SLSB 341
J2EE 323
JavaMail 325
JavaScript **198**
Java SE 323, 362, 433, 439
 JPA 439
JAX-RS 290, 327
JAX-WS 268, 327
JAXB 268
JAXP 327
JBI 378
JBoss 327
JCA 325
JDBC **424**
JMS 263, 325
JMX 325
JNDI 325, 330
 Lookup 331
JPA 325, **426**, **433**, 438
 EntityManager 435
 EntityManagerFactory 435
 JPQL 436
 Provider 434
JPQL **426**, **436**
JSF **198**, **354**
JSP **198**, 355, **419**
JTA 181, 325
JUDDI 386
JVM **11**, 231
- JXTA 192
- K**
- Kapazität 129
Klasse
 Persistence 435
Klasse abbilden 429
Knoten 11
Koexistenz 111, 133
Kommando-Muster 75
Kommunikationsarten
 Architekturstil 317
 Charakteristika 316
 Sicherheit 318
 Vergleich 317
Komplexität 33
Komponente **25**
Komposition 508
Konfigurationsmanagement 536
Konsolenfenster 229
Kontextsicht 23
Konzeptionelle Integrität 30
Korrektur 535
Kunde 230
- L**
- L10N 143
Laufzeitsicht 23
layers pattern 46
Lean Client 194
Lebenszyklus **1**, **533**
legacy system 119
Leistung 128
Leistungsverbesserung 536
LightSwitch 377
LINQ 447
listeners pattern 54
localization 143
Logische Architektur 26
Logische Subsysteme 191
Lokalisierung **143**
- M**
- maintainability* 116
Maintenance **533**
Managed Bean 357
manifest 9
marker interface 223
Markierungs-Schnittstelle 223
Markteinführung 528
Mehrbenutzerfähigkeit 122,
 153, 321
Mehrplatz-Anwendung 135

Sachindex

- Metadaten 266
Migration 322
 Alt-System 554
 Daten 555
Migrationsstrategie 18
Mikro-Architektur 37
Minimalismus 32
model 63
model view controller pattern 63
Modi 129
Muster 37
 Adapter 89
 Architektur- 37
 Befehls- 75
 Beobachter- 54
 Beziehungen zwischen 41
 Checkpoint- 157
 Dekorator 89
 Entwurfs- 37
 Fabrikmethoden- 89
 Fassaden- 69
 Klassifizierung 38
 Kommando- 75
 MVC 63, 197
 Produzenten-Konsumenten 75
 Proxy 83
 Referenzmonitor- 160
 rollenbasiertes
 Zugangssteuerungs- 159
 Schichten- 46
 Sprachen 41
 Stellvertreter 83
 Strategie 96
 zentrales Zugangs- 156
zur Authentifizierung 156
zur Autorisierung 156
MVC-Muster **62**, 63, 197, **459**
MVP 459
MVVM-Muster 479
- N**
- Nachhaltigkeit 119
Namensdienst 206, **225**, 309, 330
Nebenläufige
 Transaktionsverarbeitung 129
.NET 362
 ADO.NET 444
 Anwendungarten 364
 Application Server 307
 ASP 469
- Ausführungsumgebungen 364
Bibliotheken 366
DCOM 303
EDM 446
Eigenschaften 363
eSQL 447
LINQ 447
MVVM-Muster 479
Netzkommunikation 302
ORM 446
Programmiersprachen 366
Silverlight 478
Technische Merkmale 363
Transaktionen 186
WCF 303
Windows Forms 474
Windows Server AppFabric 307
WPF 474
WSDL 304
XAML 475
XML 444
.NET Framework 362
NetWeaver 326
Netzkommunikation 205
NFR 109
Nichtfunktionale
 Anforderungen 109
 Abhängigkeiten 114
 Klassifizierung 111
 Zielkonflikte 114
node 11
nonfunctional requirements 109
NoSQL-Datenbanken 423
- O**
- Objekt
 POJO 434
Objektidentität **429**
Objektorientiertes
 Datenbanksystem **423**
Objektverwaltung 309
observer pattern 54
ODBC **424**
OID-Attribut **429**
Ökonomie 32
OMG **242**
OOA 21
OOA-Modell 16, 415
OOB 364
OOD-Modell 16, 415
Optimierung 536
ORB 242
- Kern 243
Orchestrierung **201**
ORM 376, 428, 446
Ortstransparenz 205
- P**
- Paket 25, **25**
Parallelauf 529
Parameter 311
pattern language 41
Peer-to-Peer-Architektur 137, 192
performance 128
Performance 308
Persistenz **423**
Pflege **539**
Physische Architektur 27, 191
Pilotinstallation 529
Pilotkunde 529
POJO-Klasse 427
Port 208
Portabilität 111, 119, **132**
portability 132
P2P-Architektur 192
Primärschlüssel 428, 434
Prinzip **29**
 defensives Programmieren 501
 integrierte Dokumentation 500
 problemadäquate Datentypen 499
 Verbalisierung 497
Prinzip der Abstraktion 29
Prinzip der Aufteilung von Privilegien 155
Prinzip der Bindung und Kopplung 29
Prinzip der Hierarchisierung 29
Prinzip der kleinstmöglichen Überraschung 30
Prinzip der Lokalität 29
Prinzip der Modularisierung 29
Prinzip der psychologischen Akzeptanz 155
Prinzip der sicheren Voreinstellungen 155
Prinzip der Strukturierung 29
Prinzip der Verbalisierung 29
Prinzip der vollständigen Zugriffsüberprüfung 155
Prinzip des kleinsten allgemeinen Mechanismus 155

- Prinzip des kleinsten Privilegs 155
Prinzip des offenen Entwurfs 155
Produktfamilie 136
Produktlinie 136
Programm
 Benutzer (JPA) 436
Programmüberarbeitung 542
Programmierung 494
 proxy 232
 Proxy-Muster 83
 proxy pattern 83
Prozess-Modell 483
PSC-Muster 409
publisher subscriber pattern 54
- Q**
QoS 109, 243
QS
 Architektur 489
Qualitätscharakteristika 110
Qualitätsmerkmale 116, 121, 123, 124, 128, 130, 132
Quality of Service 109
- R**
Rückgabewert 311
Rückruf-Mechanismus 42
Rechenzeit 129
Redo 75
Redokumentieren 541
Reengineering 540, **542**, 553, 559, 563
Refactoring 559, 563
registry 201
Regressionstest **516**
Reife 110, 124
Rekonstruieren 540
Relationales Datenbanksystem **423**
Respezifizieren 541
REST 287
 Java 290
 JAX-RS 290
 Konzepte 288
Restfehler 535
Restrukturieren 541, 559, 563
Restrukturierung **513**
 Katalog 517
 Klassen 513
 Methoden 513
Reverse Engineering **540**, **545**
RIA 363, 478
RIA-Web-Architektur 198
- Rich Client* 194
RMI 222
 Begriffe 230
 Hello World 222
 Stummel-Objekt 232
RMI-Compiler **228**
RMI-IIOP 327
Rolle **154**, 165
Rollout **525**
RPC 263
Rundreise **307**, 310
- S**
SAAJ 268, 327
SAAM 490
SAE-AADL 401
 safety 123
Sanierung 543
SCA 378
Schichten-Muster 46
Schlüsselattribut 434
Schnittstelle 505
 EntityManager 435
 EntityManagerFactory 435
Schulung 528
 security 121
Selbstorganisation 34
separation of concerns 31
Server 230
Serverinstallation 525
Service to Worker 421
Service to Worker-Muster 416
Serviceorientierte Architektur 136, 201
Servlet **198**
Session Bean 315, 332
Session Facade 346
Shell 229
Sicherheit 110
Sichtbarkeit 34
Silverlight 364, 478
Single Sign-On **153**
Singleton-Muster **74**
Skalierbarkeit 322
Slim Client 194
SLSB 341, 343
Smart Client 194
SMTP/POP3 263
SOA 136, **201**, **377**
SOA-Service **201**
SOAP **263**, 264, 283, **378**
 Attachments 265
 Webservice 265
 WSDL 266
Sockets 207
- TCP 208
UDP 213
Softwarearchitektur **23**
Speicher-Inanspruchnahme 129
Speicheradresse 232
Speicherplatz 128
Spezifikationsüberarbeitung 542
SQL **436**, 444
SQLCLR 364
SSO 153
Stabilisierung 535
Stabilität 110, 116
Stakeholder **400**
Stand-alone-Anwendung 136
Standard-Software 522
Stapel 232
Statische Sicht 23
StAX 327
Stellvertreter 232
Stellvertreter-Muster 83
Stereotyp **9**
Strategie-Muster 96
strategy pattern 96
Stresstest 527
Struts 63
stub 232
Stummel-Objekt **232**
Subsystem 15, 24
 Applikation 15, 415
 Benutzungsoberfläche 15, 451
 logisch 191
 Persistenz 15, 423
Symmetrie 33
SysML 401, 405
 Blockdefinitionsdiagramm 405
 internes Blockdiagramm 405
- T**
Tabellen 428
TCP-Sockets 208
Technische Anforderungen 109
Testbarkeit 110, 116, 119
Thick Client 194
Thin Client 194
transaction 177
Transaktion **177**, **178**
Transaktionssystem 178
Transfer-Object 420, 426
Transfer Object-Muster 416
Transport-System 307
Trennung von Belangen 32

Sachindex

- Trennung von Zuständigkeiten 31
Tupel 428
- U**
- Übergabe 527
Übergangsverzögerung 129
281, 283, **378**, 380
UDP-Sockets 213
Ultra-Thin Client 195
UML **6**
Umstellung
 Datenbestände 528
 direkte 529
 Parallellauf 529
 Versuchslauf 529
Undo 75
Unternehmensanwendungen 321, 323
Unternehmenslösung 136, 321
Updates 525
URI 288
URL 288
usability 130
Use Case 490
- V**
- Value Object* 420
Verbalisierung **497**
Verbrauchsverhalten 110, 128, 129
Vererbung 508
Vererbung abbilden 432
Verfügbarkeit 122, 126, 322
Verleger-Abonenten-Muster 54
Vermeidungsmuster 409
Verständlichkeit 110, 130
Versuchslauf 529
- Verteilsoftware 523
Verteilte Anwendungen
 Fassade 313
Verteilte Anwendungen
 Entwurfskonzepte 307
 Fabrik-Dienst 308
 Wert-Objekte 310
Verteilung 523
Verteilungsart 136
Verteilungsdiagramm 11
Verteilungsmuster 191
Verteilungssicht 23
Verteilungssicht eingebetteter Systeme 406
Vertraulichkeit 122
Verzögerung 129
Verzeichnisdienst 202
view 63
View Helper 418
View Helper-Muster 416
Voice over IP **192**
- W**
- Wartbarkeit 110, **116**, 535
Wartezeit 129
Wartung **535**
Wartung & Pflege 527
Wartungsfehler 536
Wartungsorganisation 536
WCF 303, 378
Web-Anwendung 524
Web-Architektur 136, **196**
 MVC 197
 RIA 198
Web-Client 197
Web-Server 197
WebLogic 328
Webservice **204**, 265, 364, **377**
- JAX-WS 268
Nutzung 275
UDDI 281
WebSphere 327
Weiterentwickelbarkeit **119**, 539
Wert-Objekt-Entwurfsmuster 311
Wert-Objekte 310, 313, 315
Wiederherstellbarkeit 110, 125
Wiederherstellungsmuster 409
Windows Forms 364, 375, 474
Windows Workflow Foundation 373
WMI 364
WPF 364, 375, 474
WSDL 266, 281, 283, 304, **378**
- X**
- XA 181
XAML 475
XML 254, 263, 423, 444
XML-RPC **254**
XPDL 379
- Z**
- Zeitplanung 528
Zeitverhalten 110, 128
Zero Client 195
Zugriffskontrolle 166
Zugriffssteuerung 122
Zuständigkeiten
 Neuverteilung von 559
Zuverlässigkeit 110, 124, **124**
Zuverlässigkeit i.e.S. 126
Zwei-Phasen-Commit-Protokoll **179, 182**