



Joachim Goll

# Architektur- und Entwurfsmuster der Softwaretechnik

Mit lauffähigen Beispielen in Java

*2. Auflage*



Springer Vieweg

---

# Architektur- und Entwurfsmuster der Softwaretechnik

---

Joachim Goll

# Architektur- und Entwurfsmuster der Softwaretechnik

Mit lauffähigen Beispielen in Java

2., aktualisierte Auflage

Joachim Goll  
Fakultät Informationstechnik  
Hochschule Esslingen  
Deutschland

ISBN 978-3-658-05531-8  
DOI 10.1007/978-3-658-05532-5

ISBN 978-3-658-05532-5 (eBook)

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

Springer Vieweg

© Springer Fachmedien Wiesbaden 2013, 2014

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk 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.

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist eine Marke von Springer DE. Springer DE ist Teil der Fachverlagsgruppe Springer Science+Business Media.  
[www.springer-vieweg.de](http://www.springer-vieweg.de)

# Vorwort


## Zum Inhalt


Architekturen von Softwaresystemen sollen einfach erweiterbar und weitestgehend standardisiert sein, damit die Entwickler sich leicht über Architekturen austauschen können. Für den objektorientierten Entwurf haben sich zahlreiche wertvolle Architektur- und Entwurfsmuster herausgebildet. Diese Muster basieren auf objektorientierten Prinzipien für den Entwurf wie dem Prinzip der Dependency Inversion. Daher werden in diesem Buch zuerst die wichtigsten dieser Prinzipien sorgfältig erklärt. Anschließend wird gezeigt, wie diese objektorientierten Prinzipien in den verschiedenen Architektur- und Entwurfsmustern umgesetzt werden. Alle vorgestellten Muster werden durch lauffähige Beispiele in Java illustriert.

Nach der Vorstellung der wichtigsten Prinzipien für den Entwurf untersucht das vorliegende Buch eine ganze Reihe bewährter Architektur- und Entwurfsmuster, damit der Leser prüfen kann, ob er diese Muster für sein Projekt verwenden kann. Lauffähige Beispiele für die aufgeführten Entwurfsmuster und zum Teil auch für die behandelten Architekturmuster sollen die Prüfung dieser Muster erleichtern. Nicht jedes Beispiel des Buchs ist vollständig abgedruckt. Alle Beispiele sind aber auf dem begleitenden Webauftritt vollständig enthalten.

Das Buch wendet sich an Studierende der Informatik und der ingenieurwissenschaftlichen Disziplinen, berufliche Umsteiger und Entwickler in der Praxis und an interessierte Lehrer und Schüler, welche das Interesse haben, sich zum einen auf bewährte Prinzipien für den Entwurf abzustützen und zum anderen bewährte Muster für den Entwurf größerer Softwaresysteme bewerten zu können und ihre Einsatzmöglichkeiten zu beurteilen. Dabei hat dieses Buch das ehrgeizige Ziel, dem Leser die Architektur- und Entwurfsmuster auf der Basis der Prinzipien so präzise wie möglich und dennoch in leicht verständlicher Weise vorzustellen.

Die Beispiele dieses Buches sind in der Programmiersprache Java formuliert. In der Beschreibung der Muster werden die statischen Beziehungen zwischen den beteiligten Klassen sowie das dynamische Verhalten bei der Zusammenarbeit zwischen den beteiligten Objekten mit Hilfe von Diagrammen der Unified Modeling Language (UML) beschrieben. Grundkenntnisse in beiden Bereichen – Java und UML – sollte ein Leser besitzen, damit das Buch für ihn besonders nützlich ist.

"Lernkästchen", auf die grafisch durch eine kleine Glühlampe (  ) aufmerksam gemacht wird, stellen eine Zusammenfassung des behandelten Inhalts in kurzer Form dar. Sie erlauben eine rasche Wiederholung des Stoffes. Ein fortgeschrittener Leser kann mit ihrer Hilfe gezielt bis zu derjenigen Stelle vorstoßen, an der für ihn ein detaillierter Einstieg erforderlich wird.

"Warnkästchen", die durch ein Vorsicht-Symbol (  ) gekennzeichnet sind, zeigen Fallen und typische, gern begangene Fehler an, die in der Praxis oft zu einer langwierigen Fehlersuche führen, oder noch schlimmer, erst im Endprodukt beim Kunden erkannt werden.

Jedes Kapitel enthält einfache Übungsaufgaben. Die Lösungen wurden jedoch bewusst nicht in das Buch aufgenommen, damit sie nicht zum vorschnellen Nachschlagen verleiten. Sie finden die Lösungen auf dem begleitenden Webauftritt.

## Zur Entstehung

Das Buch "Methoden und Architekturen der Softwaretechnik" [Gol11] enthielt in 2 Kapiteln einen Überblick über gängige Entwurfs- und Architekturmuster. Bei einer Neuauflage dieses Buches wurde entschieden, diese beiden Themen wegen ihrer Bedeutung in ein separates Buch auszulagern. Diese Themen sollten aber nicht nur sorgfältig dargestellt, sondern sollten auf Basis der objektorientierten Prinzipien für den Entwurf, die den Mustern zugrunde liegen und die für ein vertieftes Verständnis der Muster notwendig sind, von Grund auf hergeleitet werden. Nach der Aufnahme der hinter diesen Mustern stehenden Entwurfsprinzipien und einer gründlichen Überarbeitung der Muster entstand daraus das nun vorliegende Buch.

## Schreibweisen

In diesem Buch ist der Quellcode sowie die Ein- und Ausgabe der Beispielprogramme in der Schriftart `Courier New` geschrieben. Dasselbe gilt für Programmteile wie Klassennamen, Namen von Operationen und Variablen etc., die im normalen Text erwähnt werden.

Wichtige Begriffe im normalen Text sind **fett** gedruckt, um sie hervorzuheben.

## Wichtige Hinweise zum begleitenden Webauftritt

Das vorliegende Buch wird durch einen Webauftritt begleitet. Unter der Adresse <http://www.stz-softwaretechnik.de/aembuch> finden Sie die Übungsaufgaben, die zugehörigen Lösungen sowie die Beispiele des Buches. Ebenfalls dort finden Sie die Bilder der einzelnen Kapitel als Unterstützung für Lehrende, die selbst einen Kurs auf der Basis dieses Buches gestalten wollen.

## Danksagung

Beim Entstehen dieses Buches haben viele mitgewirkt. Die einzelnen Beiträge reichen von der Ausarbeitung aussagefähiger Beispiele bis hin zum kritischen Überarbeiten der Dokumentation einzelner Muster. Hier sind Herr Benjamin Adolphi, Herr Sebastian Bickel, Herr Manuel Gotin, Herr Konstantin Holl, Herr Dominic Kadynski, Herr Micha Koller, Herr Paul Krohmer und Herr Philipp Stehle zu nennen.

Das Versions- und Konfigurationsmanagement für das Buch wurde über eine lange Zeit hinweg in sorgfältiger Weise von Herrn Steffen Wahl, Herrn Christian Tolk, Herrn Fabian Wirsum und Frau Jennifer Rauscher durchgeführt.

Dafür herzlichen Dank!

Esslingen, im März 2014

J. Goll

# Inhaltsverzeichnis

Begriffsverzeichnis.....	IX
Abkürzungsverzeichnis.....	XXIII
Wegweiser durch das Buch .....	XXV
1 Prinzipien für den objektorientierten Entwurf .....	1
1.1 Weiterentwickelbarkeit, Korrektheit und Verständlichkeit .....	4
1.2 Kapselung, Abstraktion und Information Hiding.....	6
1.3 Separation of Concerns und das Single Responsibility-Prinzip .....	8
1.4 Interface Segregation-Prinzip .....	10
1.5 Loose Coupling.....	11
1.6 Liskovsches Substitutionsprinzip .....	12
1.7 Design by Contract.....	14
1.8 Open-Closed-Prinzip .....	20
1.9 Das Dependency Inversion-Prinzip und Inversion of Control.....	26
1.10 Verringerung der Abhängigkeiten bei der Erzeugung von Objekten.....	31
1.11 Zusammenfassung .....	40
1.12 Aufgaben .....	42
2 Softwarearchitekturen.....	43
2.1 Der Begriff einer Softwarearchitektur.....	46
2.2 Qualitäten einer Softwarearchitektur .....	47
2.3 Referenzarchitekturen, Architektur- und Entwurfsmuster .....	49
2.4 Aufgaben, Sichten und Prototypen bei der Konzeption einer Softwarearchitektur.....	50
2.5 Die Bedeutung eines Softwarearchitekten für ein Projekt.....	56
2.6 Zusammenfassung .....	59
2.7 Aufgaben .....	62
3 Muster beim Softwareentwurf.....	63
3.1 Einsatz von Mustern .....	65
3.2 Eigenschaften von Mustern und ihre Konstruktion .....	67
3.3 Abgrenzung zwischen Architekturmustern, Entwurfsmustern und Idiomen ..	68
3.4 Schema für die Beschreibung von Entwurfs- und Architekturmustern.....	70
3.5 Zusammenfassung .....	71
3.6 Aufgaben .....	72
4 Objektorientierte Entwurfsmuster .....	73
4.1 Klassifikation von Entwurfsmustern .....	74
4.2 Übersicht über die vorgestellten Entwurfsmuster .....	76
4.3 Das Strukturmuster Adapter .....	79
4.4 Das Strukturmuster Brücke.....	88
4.5 Das Strukturmuster Dekorierer .....	98
4.6 Das Strukturmuster Fassade .....	116
4.7 Das Strukturmuster Kompositum.....	124
4.8 Das Strukturmuster Proxy.....	137
4.9 Das Verhaltensmuster Schablonenmethode .....	145
4.10 Das Verhaltensmuster Befehl.....	153
4.11 Das Verhaltensmuster Beobachter .....	163

4.12	Das Verhaltensmuster Strategie .....	174
4.13	Das Verhaltensmuster Vermittler .....	181
4.14	Das Verhaltensmuster Zustand .....	192
4.15	Das Verhaltensmuster Rolle .....	202
4.16	Das Verhaltensmuster Besucher .....	214
4.17	Das Verhaltensmuster Iterator .....	231
4.18	Das Erzeugungsmuster Fabrikmethode .....	243
4.19	Das Erzeugungsmuster Abstrakte Fabrik .....	251
4.20	Das Erzeugungsmuster Singleton .....	262
4.21	Das Erzeugungsmuster Objektpool .....	272
4.22	Zusammenfassung .....	281
4.23	Aufgaben .....	284
5	Architekturmuster .....	287
5.1	Das Architekturmuster Layers .....	290
5.2	Das Architekturmuster Pipes and Filters .....	304
5.3	Das Architekturmuster Plug-in .....	313
5.4	Das Architekturmuster Broker .....	326
5.5	Das Architekturmuster Service-Oriented Architecture .....	351
5.6	Das Architekturmuster Model-View-Controller .....	377
5.7	Zusammenfassung .....	397
5.8	Aufgaben .....	398
	Literaturverzeichnis .....	399
	Index .....	405



# Begriffsverzeichnis

- **Abgeleitete Klasse**

Eine abgeleitete Klasse (Unterklasse, Subklasse) wird von einer anderen Klasse, der sogenannten Basisklasse (Oberklasse, Superklasse), abgeleitet. Eine abgeleitete Klasse erbt die Struktur (Attribute mit Namen und Typ) und das Verhalten (Methoden) ihrer Basisklasse in einer eigenständigen Kopie.

- **Abhängigkeit**

Eine Abhängigkeit ist eine spezielle Beziehung zwischen zwei Modellelementen, die zum Ausdruck bringt, dass sich eine Änderung des unabhängigen Elementes auf das abhängige Element auswirken kann.

- **Abstract Window Toolkit (AWT)**

Die Klassenbibliothek AWT ist die Vorgängerin der Klassenbibliothek Swing. AWT-GUI-Komponenten werden aufgrund ihrer Abhängigkeit vom Betriebssystem als "schwergewichtig" bezeichnet. "Leichtgewichtige" Swing-GUI-Komponenten werden hingegen mit Hilfe der Java 2D-Klassenbibliothek durch die Java-Virtuelle Maschine selbst auf den Bildschirm gezeichnet und sind damit in Aussehen und Verhalten unabhängig vom Betriebssystem.

- **Abstrakte Klasse**

Von einer abstrakten Klasse können keine Instanzen gebildet werden.

Das liegt technisch gesehen meist daran, dass eine abstrakte Klasse eine oder mehrere Methoden ohne Methodenrumpf, d. h. nur die entsprechenden Methodenköpfe, hat. Eine Methode ohne Methodenrumpf kann in einer abgeleiteten Klasse implementiert werden.

Eine abstrakte Klasse kann aber auch vollständig sein und keine abstrakte(n) Methode(n) besitzen. Da sie aber nur eine Abstraktion darstellt, gibt es von ihr keine Instanzen. Ein "Gerät" oder ein "Getränk" beispielsweise kann nicht instanziiert werden.

- **Abstraktion**

Eine Abstraktion ist das Weglassen des jeweils Unwesentlichen und die Konzentration auf das Wesentliche.

- **Aggregation**

Eine Aggregation ist eine spezielle Assoziation, die eine Beziehung zwischen einem Ganzen und einem Teil ausdrückt. Bei einer Aggregation ist – im Gegensatz zu einer Komposition – die Lebensdauer eines Teils von der Lebensdauer des besitzenden Objekts entkoppelt.

- **Akteur**

Ein Akteur ist eine Rolle, ein Fremdsystem oder ein Gerät in der Umgebung eines Systems. Ein Akteur steht mit einem System in Wechselwirkung.

- **Aktor**

Siehe Akteur

- **Anwendungsfall** (engl. **use case**)

Ein Anwendungsfall ist eine Beschreibung einer Leistung, die ein System als Service zur Verfügung stellt, einschließlich verschiedener Ausprägungen des Ablaufs der Erbringung dieser Leistung. Die Leistung eines Anwendungsfalls wird oft durch eine Kollaboration, d. h. eine Interaktion von Objekten, bereitgestellt.

- **Architektur**

Unter der Architektur eines Systems versteht man:

- eine Zerlegung des Systems in seine physischen Komponenten,
- eine Beschreibung, wie durch das Zusammenwirken der Komponenten die verlangten Funktionen erbracht werden, sowie
- eine Beschreibung der Strategie für die Architektur, d. h. für die Zerlegung (Statik) und für das Verhalten (Dynamik)

mit dem Ziel, die verlangte Funktionalität des Systems an den Systemgrenzen zur Verfügung zu stellen.

- **Assoziation**

Eine Beschreibung eines Satzes von Verknüpfungen (Links). Dabei verbindet eine Verknüpfung bzw. ein Link zwei oder mehr Objekte als Peers (Gleichberechtigte). Eine Assoziation ist somit prinzipiell eine symmetrische Strukturbeziehung zwischen Klassen. Man kann aber die Navigation auf eine einzige Richtung einschränken.

- **Attribut**

Den Begriff eines Attributs gibt es bei Klassen/Objekten/Assoziationen und bei Datenbanken. Die Objektorientierung hat diesen Begriff von dem datenorientierten Paradigma übernommen. Er bedeutet:

1. Ein Attribut ist eine Eigenschaft einer Klasse oder eines Objekts.
2. Eine Assoziationsklasse charakterisiert eine Assoziation.
3. Eine Spalte innerhalb einer Relation (Tabelle) einer Datenbank wird auch als Attribut bezeichnet. Hierbei handelt es sich jedoch nicht um den Inhalt der Spalte selber, sondern um die Spaltenüberschrift. Ein Attributwert ist der konkrete Inhalt eines Spaltenelements in einer Zeile.

- **Basisklasse**

Eine Basisklasse (Superklasse, Oberklasse, übergeordnete Klasse) steht in einer Vererbungshierarchie über einer aktuell betrachteten Klasse.

- **Beziehung**

Eine Beziehung zwischen zwei oder mehr Elementen beschreibt, dass diese Elemente zueinander Bezug haben. Hierbei gibt es statische und dynamische Beziehungen.

- **Classifier**

Classifier ist ein Begriff aus dem Metamodell von UML. Ganz allgemein ist jedes Modellelement von UML, von dem eine Instanz gebildet werden kann, ein Classifier. Ein Classifier hat in der Regel eine Struktur und ein Verhalten. Ein Classifier ist beispielsweise die Abstraktion einer Klasse. Schnittstellen – die auch Classifier sind – haben in der Regel als einzige Ausnahme keine Attribute, d. h. keine Struktur.

- **Delegation**

Mechanismus, bei dem ein Objekt eine Nachricht nicht komplett selbst interpretiert, sondern diese auch weiterleitet.

- **Dependency Inversion**

Ist ein Objekt von einem anderen abhängig, so wird mit Dependency Inversion die Richtung der Abhängigkeit herumgedreht, d. h. das seither abhängige Objekt wird zum unabhängigen und umgekehrt.

- **Dependency Inversion-Prinzip**

Das Dependency Inversion-Prinzip dient zur Vermeidung bzw. Reduzierung von Abhängigkeiten unter Modulen, die in einer hierarchischen Beziehung zueinander stehen. Es fordert, dass eine Klasse einer höheren Ebene nicht von einer Klasse einer tieferen Ebene abhängen darf. Stattdessen soll eine Klasse einer höheren Ebene beispielsweise ein Interface oder eine abstrakte Klasse als Abstraktion aggregieren. Diese Abstraktion wird von der höheren Klasse vorgegeben. Die untergeordneten Klassen sollen nur von der entsprechenden Abstraktion abhängen.

- **Design**

Wird hier im Sinne von Entwurf verwendet.

- **Design Pattern**

Siehe Entwurfsmuster

- **Diagramm**

Ein Diagramm stellt eine Ansicht eines Systems aus einer bestimmten Perspektive dar.

In UML enthält ein Diagramm meist eine Menge von Knoten, die über Kanten in Beziehungen stehen. Ein Beispiel einer anderen grafischen Darstellung ist das Zeitdiagramm, welches analog zu einem Pulsdiagramm der Elektrotechnik ist.

- **Einschränkung**

Siehe Randbedingung

- **Entität**

Eine Entität hat im Rahmen des betrachteten Problems eine definierte Bedeutung. Sie kann einen Gegenstand, ein Wesen oder ein Konzept darstellen.

- **Entity-Objekt**

Ein Entity-Objekt ist eine Abstraktion einer Entität d. h. eines Gegenstandes, Konzepts oder Wesens der realen Welt.

- **Entwurfsmuster** (engl. **design pattern**)

Klassen oder Objekte in Rollen, die in einem bewährten Lösungsansatz zusammenarbeiten, um gemeinsam die Lösung eines wiederkehrenden Problems zu erbringen.

- **Ereignis**

Ein Ereignis ist ein Steuerfluss oder eine Kombination von Steuerflüssen, auf die ein System reagiert.

- **Framework**

Ein Framework offeriert dem nutzenden System Klassen, von welchen es ableiten und somit deren Funktionslogik erben kann. Ein Framework bestimmt die Architektur der Anwendung, also die Struktur im Groben. Es definiert weiter die Unterteilung in Klassen und Objekte, die jeweiligen zentralen Zuständigkeiten, die Zusammenarbeit der Klassen und Objekte sowie deren Kontrollfluss [Gam95].

- **Geheimnisprinzip**

Das Geheimnisprinzip sorgt dafür, dass die internen, privaten Strukturen eines Objekts einer Klasse nach außen unzugänglich sind. Nur der Implementierer einer Klasse kennt normalerweise die internen Strukturen und Methodenrümpfe eines Objekts. Implementierung und Schnittstellen werden getrennt. Die Daten eines Objekts sind nur über die Methodenköpfe der Schnittstelle erreichbar.

- **Generalisierung**

Eine Generalisierung ist die Umkehrung der Spezialisierung. Wenn man generalisieren möchte, ordnet man oben in der Vererbungshierarchie die allgemeineren Eigenschaften ein und nach unten die spezielleren, da man durch die Vererbung die generalisierten Eigenschaften wieder erbt. In der Vererbungshierarchie geht also die Generalisierung nach oben und die Spezialisierung nach unten.

- **Geschäftsprozess**

Ein Geschäftsprozess ist ein Prozess der Arbeitswelt mit fachlichem Bezug. Er stellt eine Zusammenfassung verwandter Einzelaktivitäten, um ein geschäftliches Ziel zu erreichen, dar.

- **Gruppierung**

Eine Gruppierung ist eine Zusammenfassung von Elementen. Hierzu existiert in der UML das Modellelement eines Pakets. Ein Paket ist kein Systembestandteil, sondern ein konzeptionelles Ordnungsschema, um Elemente übersichtlich in Gruppen zusammenzufassen.

- **Identität**

Jedes Objekt unterscheidet sich von einem anderen und hat damit eine eigene Identität, selbst wenn die Werte der Attribute verschiedener Objekte gleich sind.

- **Idiom**

Ein Idiom ist ein Muster in einer bestimmten Programmiersprache und damit auf einem tieferen Abstraktionsniveau als ein Entwurfs- oder Architekturmuster. Dieser Begriff kann für die Implementierung eines Entwurfsmusters in einer Programmiersprache angewandt werden, für die Lösung bestimmter technischer Probleme, die nicht den Charakter eines Entwurfsmusters haben, aber auch im Sinne einer Programmierrichtlinie.

- **Information Hiding**

Siehe Geheimnisprinzip

- **Instanz**

Eine Instanz ist eine konkrete Ausprägung des Typs eines Modellelements in UML.

- **Instanziierung**

Das Erzeugen einer Instanz eines Typs.

- **Interaktion**

Eine Interaktion ist ein dynamisches Verhalten, welches durch den Versand einer einzelnen Nachricht zwischen zwei Objekten oder durch die Wechselwirkung eines Satzes von Objekten charakterisiert ist.

- **Interprozesskommunikation**

Interprozesskommunikation (IPC) ist die Kommunikation zwischen Prozessen als parallelen Einheiten. Damit umfasst der Begriff der Interprozesskommunikation die Betriebssystem-Prozess-zu-Betriebssystem-Prozess-Kommunikation und auch die Thread-to-Thread-Kommunikation. Kommunizieren Prozesse innerhalb eines Rechners, so kann der Austausch von Informationen mit Hilfe des lokalen Betriebssystems erfolgen. Liegen die parallelen Prozesse auf verschiedenen Rechnern, muss zwischen den Rechnern auf eine Rechner-zu-Rechner-Kommunikation zugegriffen werden.

- **Inversion of Control**

Bei der Umkehrung des Kontrollflusses gibt das ursprüngliche Programm die Steuerung des Kontrollflusses an ein anderes – meist wiederverwendbares – Modul ab. Damit kommt man vom Pollen zur ereignisgetriebenen Programmierung. Oft ruft dann ein mehrfach verwendbares Modul wie etwa ein Framework ein spezielles Modul wie beispielsweise ein Anwendungsprogramm auf.

- **Kanal**

Ein Kanal bezeichnet eine Punkt-zu-Punkt-Verbindung. Er wird in der Regel durch eine Linie symbolisiert. Er arbeitet nach dem First-In-First-Out-Prinzip. Was als Erstes an den Kanal übergeben wurde, verlässt ihn auch als Erstes.

- **Kapselung**

Das Konzept der Kapselung ist eines der wichtigsten Konzepte der objektorientierten Programmierung. Darunter versteht man die sinnvolle Zusammenfassung von Daten und zugehörigen Funktionen in der gemeinsamen Kapsel, der Klasse. Daten und zugehörige Funktionen werden nicht getrennt.

- **Kardinalität**

Den Begriff der Kardinalität gibt es bei Datenbanken und bei UML:

1. Auf dem Gebiet der Datenbanken legt die Kardinalität in der Systemanalyse fest, wie viele Entitäten mit wie vielen anderen Entitäten in einer Beziehung stehen bzw. wie viele Datensätze einer Tabelle beim Systementwurf mit wie vielen Datensätzen einer anderen Tabelle in Beziehung stehen. Die Kardinalität kann durch das Verhältnis zweier Kardinalzahlen wie 1-zu-1, 1-zu-n oder n-zu-m beschrieben werden.
2. Eine Kardinalität bezeichnet in UML eine Kardinalzahl (Anzahl der betrachteten UML-Elemente). Mit Hilfe von Kardinalitätszahlen kann eine Multiplizität gebildet werden, wie beispielsweise 1..m.

- **Klasse**

Eine Klasse stellt im Paradigma der Objektorientierung einen Datentyp dar, von dem Objekte erzeugt werden können. Eine Klasse hat eine Struktur und ein Verhalten. Die Struktur umfasst die Attribute. Die Methoden und ggf. der Zustandsautomat der Klasse bestimmen das Verhalten der Objekte. Jedes Objekt einer Klasse hat seine eigene Identität.

- **Klassendiagramm**

Ein Klassendiagramm zeigt insbesondere Klassen und ihre wechselseitigen statischen Beziehungen (Assoziationen, Generalisierungen, Realisierungen, Abhängigkeiten).

- **Kollaboration**

Eine Kollaboration besteht aus einer Reihe von Objekten, die zusammenarbeiten, um beispielsweise gemeinsam die Leistung eines Anwendungsfalls zu erbringen. Eine Kollaboration von Objekten erbringt eine Funktionalität.

- **Kommunikationsdiagramm**

Ein Kommunikationsdiagramm zeigt in einer zweidimensionalen Anordnung die betrachteten Objekte und ihre Verknüpfungen sowie die Nachrichten, die entlang der Verknüpfungen zwischen den Objekten ausgetauscht werden. Durch die Gruppierung von Objekten, die zusammenarbeiten, und durch das Zeichnen der Verknüpfungen zwischen den Objekten ist trotz der dynamischen Sicht auch die strukturelle Organisation der Objekte ersichtlich.

- **Komponente**

Den Begriff Komponente gibt es bei der Zerlegung von Systemen und in der Komponententechnologie:

1. Eine Komponente<sup>1</sup> ist ein Systembestandteil, d. h. ein Zerlegungsprodukt.
2. Eine Komponente im Sinne der Komponententechnologie ist ein modularer Teil eines Systems, der die Implementierung seiner Funktionalität hinter einem Satz modulexterner Schnittstellen verbirgt. Im Gegensatz zu einer Klasse müssen bei einer Komponente alle Methodenaufrufe über Schnittstellen erfolgen. Komponenten können auf Knoten, die Rechner darstellen, dauerhaft installiert sein. Es gibt aber auch Komponenten, die als Agenten von Knoten zu Knoten wandern können. Eine Komponente im Sinne der Komponententechnologie beinhaltet in der Praxis meist eine oder mehrere Klassen, Schnittstellen oder kleinere Komponenten.

- **Komponentenmodell**

Ein Komponentenmodell beschreibt in der Softwaretechnik einen Rahmen zur Ausführung von Systemen, welche sich aus einzelnen Komponenten zusammensetzen. Dabei definiert das Komponentenmodell neben einem Container, in dem die einzelnen Komponenten ausgeführt werden (Laufzeitumgebung), auch die Schnittstellen der Komponenten zum Container sowie deren Schnittstelle zu anderen Komponenten. Zudem werden zahlreiche Mechanismen wie etwa zur Persistierung, zur Sicherheit oder zur Versionsverwaltung von Komponenten definiert.

- **Komposition**

Eine Komposition ist ein Spezialfall einer Assoziation und beschreibt eine Beziehung zwischen einem Ganzen und seinen Teilen, bei der die Existenz eines Teils mit der Existenz des Ganzen verknüpft ist. Ein Teil kann dabei nur einem einzigen Ganzen zugeordnet sein.

- **Konkrete Klasse**

Eine konkrete Klasse kann im Gegensatz zu einer abstrakten Klasse instanziiert werden, d. h. es können Objekte von dieser Klasse gebildet werden.

- **Konstruktor**

Ein Konstruktor ist eine spezielle Methode. Ein Konstruktor trägt den Namen der Klasse, wird beim Erzeugen eines Objekts aufgerufen und dient zu dessen Initialisierung. Ein Konstruktor hat keinen Rückgabotyp und kann nicht vererbt werden.

- **Kontrollobjekt**

Ein Kontrollobjekt entspricht keiner Entität der realen Welt. Ein Kontrollobjekt entspricht einer Ablaufsteuerung.

- **Lebenslinie**

Lebenslinien gibt es in UML in Sequenz- und Kommunikationsdiagrammen. Ein Objekt als Interaktionspartner in einem Kommunikationsdiagramm wird in UML auch als Lebenslinie bezeichnet. Damit finden sowohl in Kommunikationsdiagrammen als auch in Sequenzdiagrammen nach UML Interaktionen zwischen Lebenslinien statt.

---

<sup>1</sup> Eine solche Komponente sollte auch Schnittstellen haben, damit diese Komponenten über Schnittstellen lose gekoppelt sind. Das Innere einer solchen Komponente sollte verborgen sein.

- **Link**  
Siehe Verknüpfung
- **Marshalling**  
Siehe Serialisierung
- **Methode**  
Eine Methode implementiert eine Operation.
- **Middleware**  
Eine Middleware ist eine Programmschicht, welche sich über mehrere Rechner erstreckt und vor allem eine Interprozesskommunikation für verteilte Anwendungen zur Verfügung stellt. Weitere Funktionen einer Middleware sind beispielsweise Persistenzdienste, Funktionalitäten der Informationssicherheit oder die Verwaltung von Namen.
- **Multiplizität**  
Eine Multiplizität bezeichnet in UML einen Bereich zulässiger Kardinalitäten.
- **Nachricht**  
Objekte können über Nachrichten (Botschaften) kommunizieren. Die Interpretation einer Nachricht (Botschaft) obliegt dem Empfänger.
- **Navigation**  
Bei der Navigation werden die Zugriffsmöglichkeiten auf Objekte, deren Klassen untereinander Beziehungen haben, betrachtet. Die Navigation ist eine Angabe, ob man von einem Objekt an einem Ende einer Verknüpfung zu einem Objekt am anderen Ende dieser Verknüpfung kommen kann. Bei der direkten Navigierbarkeit kann der Zugriff ohne Umwege erfolgen.
- **Nebenläufigkeit**  
Zwei Vorgänge A und B heißen nebenläufig, wenn sie voneinander unabhängig bearbeitet werden können, wenn es also gleichgültig ist, ob zuerst A und dann B kommt oder zuerst B und dann A.
- **Notiz**  
Eine Notiz ist in UML ein Kommentar zu einem oder mehreren Modellelementen.
- **Oberklasse**  
Siehe Basisklasse
- **Objekt**  
Siehe Klasse



- **Objektdiagramm**

Ein Objektdiagramm ist eine Momentaufnahme der Objekte mit ihren Verknüpfungen zu einem bestimmten Zeitpunkt.

- **Operation**

Eine Operation stellt in UML die Abstraktion einer Methode dar. Sie umfasst den Namen, die Übergabeparameter, den Rückgabetyt, die Vor- und Nachbedingungen und die Spezifikation der Operation. Eine Operation wird in einer Klasse durch eine Methode implementiert. Der Kopf einer Operation mit dem Namen der Operation, Übergabeparametern und Rückgabetyt entspricht dem Kopf der zugehörigen Methode.

Die Spezifikation einer Operation kann so allgemein sein, dass dieselbe Operation in verschiedenen Klassen implementiert werden kann. Eine Operation erhält durch die Spezifikation eine bestimmte Bedeutung (Semantik), die für alle Klassen, die diese Operation in Form einer Methode implementieren, dieselbe ist. So kann eine Operation `drucke()` spezifiziert werden durch "Gib den Namen und Wert aller Attribute aus". Eine solche Operation `drucke()` kann mit derselben Spezifikation in Klassen mit einer voneinander verschiedenen Anzahl von Attributen implementiert werden.

- **Operationelle Sicht**

Hier betrachtet man nur die erforderliche Funktionalität der Anwender und nicht für den Systemadministrator.

- **Paket**

Ein Paket in UML dient zur Gruppierung von Modellelementen wie Klassen, Schnittstellen, Anwendungsfällen etc. und von Unterpaketen. Ein Paket stellt einen Namensraum dar, ist eine Einheit für den Zugriffsschutz und erleichtert die Übersicht.

- **Panel**

Ein Panel stellt einen unsichtbaren Objektbehälter dar, mit dessen Hilfe es möglich ist, grafische Komponenten zu Gruppen zusammenzufassen.

- **Paradigma**

Ein Paradigma ist ein Denkkonzept.

- **Persistenz**

Persistenz von Objekten bedeutet, dass ihre Lebensdauer über eine Sitzung hinausgeht, da sie auf nicht-flüchtigen Speichermedien wie z. B. Platten abgespeichert werden.

- **Polymorphie**

Polymorphie bedeutet Vielgestaltigkeit. So kann beispielsweise ein Objekt eines Subtyps auch in Gestalt der entsprechenden Basisklasse auftreten.

- **Protokoll**

Ein Satz von Regeln für die Kommunikation zwischen zwei Kommunikationspartnern.

- **Randbedingung**

Randbedingungen – oft auch Einschränkungen genannt – sind Bedingungen, die gelten müssen. Sie können formal oder informell beschrieben sein. Sie können z. B. von benachbarten Systemen vorgegeben werden.

- **Realisierung**

Eine Realisierung ist eine statische Beziehung zwischen zwei Elementen (Classifiern nach UML), in der das eine Element einen Vertrag spezifiziert und das andere Element sich verpflichtet, diesen Vertrag bei der Realisierung einzuhalten. Realisierungsbeziehungen gibt es beispielsweise bei Classifiern wie Klassen, die Schnittstellen implementieren, und bei Kollaborationen, die Anwendungsfälle implementieren.

- **Rolle**

Der Begriff einer Rolle ist mehrdeutig:

1. Rolle als Stellvertreter<sup>2</sup>
2. Rolle realisiert eine Schnittstelle.

In Assoziationen und im Rollen-Entwurfsmuster kann jedes Objekt eine Rolle einnehmen, indem es die Schnittstelle, welche die Rolle definiert, implementiert und nach außen in seinem Verhalten vertritt.

- **Schnittstelle**

Eine Zusammenstellung von Operationen, die dazu dienen, einen Service eines Classifiers – insbesondere einer Klasse oder einer Komponente – zu spezifizieren. Eine bestimmte Schnittstelle kann alle Operationen oder aber auch nur einen Teil der Operationen eines Classifiers wie z. B. einer Klasse oder Komponente repräsentieren. Eine Schnittstelle spezifiziert einen Vertrag, den der die Schnittstelle realisierende Classifier erfüllen muss.

- **Selbstdelegation**

Ein Objekt ruft aus einer Methode heraus eine andere eigene Methode auf.

- **Sequenzdiagramm**

Ein Sequenzdiagramm beschreibt den Austausch von Nachrichten oder Methoden zwischen Objekten bzw. Objekten und Akteuren in einer zeitlich geordneten Form.

---

<sup>2</sup> Von UML 1.x zu UML 2 hat sich der Begriff der Instanz geändert. Beim Modellieren mit Repräsentanten eines Typs – wie beispielsweise beim Erstellen eines Sequenzdiagramms – werden die Repräsentanten eines Typs nicht mehr als Instanzen oder Objekte bezeichnet, sondern als Rollen. Dies ist der Fall, wenn sie generisch sind und viele konkrete Instanzen oder Objekte an ihre Stelle treten können (siehe UML Superstructure Specification [OMGUSS, S. 14], Kapitel "Semantic Levels and Naming"). In UML 1.x wurden beispielsweise die Repräsentanten einer Klasse in einem Sequenzdiagramm noch als Instanzen bezeichnet.

- **Serialisierung**

Als Serialisierung (oder Marshalling) wird das Wandeln eines Methodenaufrufs in eine Nachricht bezeichnet.

- **Session**

Siehe Sitzung

- **Signatur in der Programmierung und in UML**

Der Begriff einer Signatur kann in UML und in Programmiersprachen jeweils anders verwendet werden:

1. In UML: Exportschnittstelle mit Methodennamen, Liste der Parametertypen und Rückgabotyp.
2. In Java: Exportschnittstelle mit Methodennamen und Liste der Parametertypen.

- **Sitzung** (engl. **session**)

Eine Sitzung bezeichnet eine bestehende Verbindung zum Austausch von Daten zwischen zwei adressierbaren Einheiten eines Netzwerkes.

- **Spezialisierung**

Siehe Generalisierung

- **Stakeholder**

Ein Stakeholder ist eine natürliche bzw. juristische Person oder Gruppe von Personen, die Interesse am entsprechenden System hat. Dieses Interesse wirkt sich in Einflüssen auf Hardware- oder Softwareanforderungen für ein System aus.

- **Struktur**

Die Struktur eines Systems ist sein statischer Aufbau.

- **Subklasse**

Siehe abgeleitete Klasse

- **Subsystem**

Eine Komponente, die einen größeren Teil eines Systems darstellt.

- **Superklasse**

Siehe Basisklasse

- **Tabelle**

Eine Tabelle ist eine Zusammenfassung analog strukturierter Daten. Bei relationalen Datenbanken spricht man auch von Relationen.

- **Typ**

Ein Typ hat Attribute und Operationen und für seine Attribute einen Wertebereich.

- **Unterklasse**

Siehe abgeleitete Klasse

- **Use Case**

Siehe Anwendungsfall

- **Vererbung**

In der Objektorientierung kann eine Klasse von einer anderen Klasse erben bzw. von ihr abgeleitet werden. Durch die Vererbung besitzt sie automatisch die Attribute und Methoden der Klasse, von der sie ableitet, d. h. sie erbt die Struktur und das Verhalten ihrer Basisklasse.

- **Vererbungshierarchie**

Durch die Vererbungsbeziehung zwischen Klassen entsteht eine Hierarchie: abgeleitete Klassen werden ihren Basisklassen untergeordnet bzw. Basisklassen sind ihren abgeleiteten Klassen übergeordnet.

Da in Java alle Klassen direkt oder indirekt von der Klasse `Object` abgeleitet sind, stellt diese Klasse die Wurzel einer Vererbungshierarchie in Java dar.

- **Verhalten**

Im Verhalten eines Systems kommt seine Funktionalität zum Ausdruck.

- **Verknüpfung**

Eine Verknüpfung ist eine Instanz einer Assoziation. Sie wird auch Link genannt.

- **Verwendungsbeziehung**

Will man zum Ausdruck bringen, dass ein Element ein anderes nutzt, kann man dafür in UML eine Abhängigkeit mit «use» auszeichnen, d. h. eine «use»-Beziehung – auf Deutsch eine Verwendungsbeziehung – verwenden. Damit bringt man zum Ausdruck, dass die Bedeutung (Semantik) eines Elements von der Bedeutung eines anderen Elements abhängt.

- **Zustandsautomat**

Siehe Zustandsübergangsdiagramm

- **Zustandsdiagramm**

Ein Zustandsdiagramm von UML ist eine Variante eines Zustandsübergangsdiagramms nach Harel bzw. nach Hatley/Pirbhai. Ein Zustandsdiagramm besteht aus Zuständen (eventuell mit Regionen<sup>3</sup>) des näher zu beschreibenden Classifiers und aus Transitionen. Zustände beschreiben Verzögerungen, Eintritts-, Austritts- und Zustandsverhalten. Transitionen spezifizieren Ereignisse für Zustandsübergänge, deren Bedingungen und das Verhalten des Classifiers beim Übergang. Ein Zustandsdiagramm wird zur Modellierung von reaktiven Systemen benötigt.

---

<sup>3</sup> Regionen dienen zur Beschreibung von Parallelitäten.

- **Zustandsübergangsdiagramm**

In Zustandsübergangsdiagrammen werden Zustände von Systemen nach David Harel oder Hatley/Pirbhai als Graph modelliert. Der Graph beschreibt einen Automaten mittels Zuständen und Transitionen (Zustandsübergängen). Transitionen enthalten die Spezifikation über das auslösende Ereignis, die Bedingung für einen Zustandswechsel und die beim Zustandsübergang ausgelöste(n) Aktion(en).

# Abkürzungsverzeichnis

<b>API</b>	Application Programming Interface
<b>ASN.1/BER</b>	Abstract Syntax Notation 1/Basic Encoding Rules
<b>AWT</b>	Abstract Window Toolkit
<b>CORBA</b>	Common Object Request Broker Architecture
<b>CSS</b>	Cascading Style Sheets
<b>CSV</b>	Comma-Separated Values
<b>DbC</b>	Design by Contract
<b>DBMS</b>	Database Management System
<b>DIP</b>	Dependency Inversion Principle
<b>DV</b>	Datenverarbeitung
<b>EDV</b>	Elektronische Datenverarbeitung
<b>EJB</b>	Enterprise JavaBeans
<b>FIFO</b>	First-In-First-Out
<b>GIOP</b>	General Inter-ORB Protocol
<b>GoF</b>	"Gang of Four"
<b>GUI</b>	Graphical User Interface
<b>HTML</b>	Hypertext Markup Language
<b>HTTP</b>	Hypertext Transfer Protocol
<b>IDL</b>	Interface Definition Language
<b>IEEE</b>	Institute of Electrical and Electronics Engineers
<b>I/O</b>	Input/Output
<b>IIOP</b>	Internet Inter-ORB Protocol
<b>IoC</b>	Inversion of Control
<b>IPC</b>	Interprozesskommunikation
<b>ISO</b>	International Organization for Standardization
<b>ISP</b>	Interface Segregation Principle
<b>JAX-RS</b>	Java API for RESTful Web Services
<b>JAX-WS</b>	Java API for XML Web Services
<b>JRE</b>	Java Runtime Environment
<b>JSP</b>	JavaServer Pages
<b>JVM</b>	Java-Virtuelle Maschine
<b>LSP</b>	Liskov Substitution Principle
<b>MEP</b>	Message Exchange Pattern
<b>MHS</b>	Message Handling System
<b>MIB</b>	Management Information Base
<b>MMI</b>	Man-Machine Interface
<b>MVC</b>	Model-View-Controller
<b>OCP</b>	Open-Closed Principle
<b>OMG</b>	Object Management Group
<b>ORB</b>	Object Request Broker
<b>OSGi</b>	Früher: Open Services Gateway Initiative
<b>OSI</b>	Open Systems Interconnection
<b>REST</b>	Representational State Transfer
<b>SEU</b>	Software-Entwicklungsumgebung

<b>SNA</b>	Systems Network Architecture
<b>SOAP</b>	SOA-Protokoll, früher: Simple Object Access Protocol
<b>SRP</b>	Single Responsibility Principle
<b>STL</b>	Standard Template Library
<b>SW</b>	Software
<b>TCP/IP</b>	Transmission Control Protocol/Internet Protocol
<b>UDDI</b>	Universal Discovery, Description, Integration
<b>UML</b>	Unified Modeling Language
<b>URI</b>	Uniform Resource Identifier
<b>W3C</b>	World Wide Web Consortium
<b>WSDL</b>	Web Services Description Language
<b>WSIL</b>	Web Services Inspection Language
<b>XML</b>	Extensible Markup Language

# Wegweiser durch das Buch

Die nachfolgenden kurzen Inhaltsbeschreibungen der einzelnen Kapitel sollen dem Leser den Weg durch das Buch erleichtern. Hier eine kurze Übersicht über die Kapitel:

**Kapitel 1** diskutiert grundlegende Prinzipien für den objektorientierten Entwurf. Dabei werden wichtige Prinzipien für den Entwurf einzelner Klassen und für die gegenseitige Zusammenarbeit mehrerer Klassen besprochen. Bei den Prinzipien für den Entwurf einzelner Klassen werden die Prinzipien der Kapselung, der Abstraktion und des Information Hiding, dann das Prinzip Separation of Concerns und das Single Responsibility-Prinzip sowie das Interface Segregation-Prinzip erörtert. Die Prinzipien für die Zusammenarbeit mehrerer Klassen behandeln das Prinzip Loose Coupling, das Liskovsche Substitutionsprinzip, das Prinzip Design by Contract, das Open-Closed-Prinzip und das Dependency Inversion-Prinzip. Es folgt zusätzlich noch die Analyse von Inversion of Control und der Verringerung der Abhängigkeiten bei der Erzeugung von Objekten, auch wenn diese Techniken nicht unter dem Begriff "Prinzip" laufen.

**Kapitel 2** betrachtet die Definition des Begriffs Softwarearchitektur und erstrebenswerte nicht funktionale Qualitäten einer Softwarearchitektur. Referenzarchitekturen und Muster beim Softwareentwurf werden einander gegenübergestellt. Nach den analytischen und konstruktiven Aufgaben beim Bau eines Systems, den verschiedenen Sichten auf eine Softwarearchitektur und der Erstellung von Prototypen wird die Bedeutung eines Softwarearchitekten für ein Projekt diskutiert.

**Kapitel 3** erörtert die jeweiligen Eigenschaften von Architekturmustern, Entwurfsmustern und Idiomen. Ein Schema für die Beschreibung von Entwurfs- und Architekturmustern schließt dieses Kapitel ab.

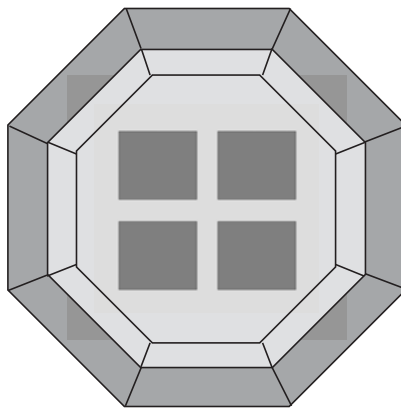
**Kapitel 4** untersucht objektorientierte Entwurfsmuster. Objektorientierte Entwurfsmuster sind bewährte Ansätze für den Entwurf bzw. Lösungswege, mit denen man bestimmte Probleme – meist auf der Ebene von Teilsystemen – in der Softwareentwicklung objektorientiert lösen kann. Ein Entwurfsmuster besteht aus Klassen in Rollen, die zusammenarbeiten, um ein bestimmtes Problem zu lösen. Jedes dieser Muster ist für eine Gattung von Problemen anwendbar. Besprochen werden Strukturmuster, Verhaltensmuster und Erzeugungsmuster. Bei den Strukturmustern werden die Muster Adapter, Brücke, Dekorierer, Fassade, Kompositum und Proxy behandelt. Bei den Verhaltensmustern werden die Muster Schablonenmethode, Befehl, Beobachter, Strategie, Vermittler, Zustand, Rolle, Besucher und Iterator untersucht. Die dargestellten Erzeugungsmuster umfassen die Muster Fabrikmethode, Abstrakte Fabrik, Singleton und Objektpool.

**Kapitel 5** befasst sich mit Architekturmustern. Mit Architekturmustern können Systeme in Systemkomponenten zerlegt werden. Im Gegensatz zu Entwurfsmustern sind Architekturmuster grobkörniger. Ein Architekturmuster kann – muss aber nicht – mehrere verschiedene Entwurfsmuster beinhalten. Beispielsweise kommt das Muster Layers ohne den Einsatz von Entwurfsmustern aus. Behandelt werden in Kapitel 5 die Architekturmuster Layers, Pipes and Filters, Plug-in, Broker, Service-Oriented Architecture und Model-View-Controller.



# Kapitel 1

## Prinzipien für den objektorientierten Entwurf



- 1.1 Weiterentwickelbarkeit, Korrektheit und Verständlichkeit von Programmen
- 1.2 Abstraktion, Kapselung und Information Hiding
- 1.3 Separation of Concerns und das Single-Responsibility-Prinzip
- 1.4 Interface Segregation-Prinzip
- 1.5 Loose Coupling
- 1.6 Liskovsches Substitutionsprinzip
- 1.7 Design by Contract
- 1.8 Open-Closed-Prinzip
- 1.9 Das Dependency Inversion-Prinzip und Inversion of Control
- 1.10 Verringerung der Abhängigkeiten bei der Erzeugung von Objekten
- 1.11 Zusammenfassung
- 1.12 Aufgaben

# 1 Prinzipien für den objektorientierten Entwurf

Fast alles auf der Welt unterliegt einem gewissen Alterungsprozess. Erstaunlicherweise gilt dies auch für Software – erstaunlicherweise deshalb, weil Software eigentlich ein künstlich geschaffenes Artefakt ist und sich nicht abnutzt. Trotzdem altert Software, weil sich ihre Umgebung ändert wie z. B. die Anforderungen oder das Betriebssystem, und zeigt dabei nach Martin [Mar03] auch Eigenschaften<sup>4,5</sup> wie Starrheit und Unbeweglichkeit, die typisch für alternde Lebewesen sind:

- **Starrheit** (engl. **rigidity**)

Ist das Design zu stark auf das gerade vorliegende Problem angepasst, so funktioniert dieses Design nur solange, bis die Software erweitert werden muss oder sich das zu lösende Problem verändert. Die Implementierung einer erforderlichen Erweiterung kann weitere Änderungen in abhängigen Modulen, die bereits als stabil erachtet wurden, nach sich ziehen, so dass der Aufwand für Änderungen nicht kalkulierbar ist.

- **Zerbrechlichkeit** (engl. **fragility**)

Ein starr entwickeltes Design weist häufig eine hohe Zerbrechlichkeit auf. Dies bedeutet, dass wenn man eine Veränderung an einer bestimmten Stelle durchführt, plötzlich andere Teile des Systems unerwarteterweise nicht mehr korrekt laufen können. Oft entstehen diese Probleme in Teilen einer Software, die keinen konzeptionellen Bezug zu der geänderten Stelle aufweisen. Das liegt an starken Abhängigkeiten zwischen den verschiedenen Klassen, welche dazu führen können, dass eine Änderung in der einen Klasse zu einem Fehler in einer von ihr abhängigen Klasse führt.

- **Unbeweglichkeit** (engl. **immobility**)

Die Unbeweglichkeit von Software beschreibt die fehlende Möglichkeit zur Wiederverwendung einzelner Module einer bereits bestehenden Anwendung. Wenn Teile aus der vorhandenen Software nur mit großem Aufwand ausgeschält werden können, da sie wechselseitig stark abhängig sind, sind die entsprechenden Teile schwer wiederzuverwenden. Ein solches Design wird meist nicht wiederverwendet, weil die Kosten einer Heraustrennung von Anteilen höher sind als die Kosten für eine Neuentwicklung des Systems.

- **Zähigkeit** (engl. **viscosity**)

Es gibt sowohl eine Viskosität des Designs, als auch der Umgebung. Die **Zähigkeit eines Designs** beschreibt, wie gut sich ein System erweitern lässt. Wenn es bei Änderungen also aufwendiger ist, das bestehende Design beizubehalten, als es anzupassen, so weist das System ein hohes Maß an Zähigkeit auf. Die **Zähigkeit der Umgebung** betrifft die Arbeitsumgebung wie beispielsweise die verwendeten Bibliotheken. Die Ingenieure können dazu verleitet werden, im Sinne des Designs nicht-optimale Änderungen durchzuführen, nur um Performance-Einschränkungen der Entwicklungsumgebung auszuweichen.

---

<sup>4</sup> Robert C. Martin fand drastischere Begriffe (siehe beispielsweise [Mar03]): Software verrottet bzw. modert (engl. to rot) und dabei entstehen Gerüche (engl. smells). Als "Gerüche" bezeichnet er neben anderen die oben genannten negativen Eigenschaften von Software.

<sup>5</sup> Bitte beachten Sie, dass diese Eigenschaften nicht orthogonal, sondern miteinander gekoppelt sind.

Diese negativen Eigenschaften, die durchaus auch in Kombination auftreten können, sind meist die Folge eines schlechten Systementwurfs. Wenn Sie diese Schwachstellen analysieren, so sehen Sie, dass die Abhängigkeiten der verschiedenen Softwareteile untereinander, aber auch zu Bibliotheken oder infolge der verwendeten Programmiersprache etc. stets die Hauptursache für die aufgeführten Schwächen sind.

Ein System mit vielen **Abhängigkeiten** seiner Teile ist **nicht weiterentwickelbar** (engl. **evolvable**).



Existieren starke Abhängigkeiten, so kann weder das System eine andere Architektur erhalten, noch können seine Komponenten in einer anderen Architektur wiederverwendet werden. Starke Abhängigkeiten verhindern eine kostengünstige Weiterentwicklung. Es müssten viele Teile geändert oder neu geschrieben werden. Bei einem System ohne Weiterentwickelbarkeit ist das Investment in das System verloren, wenn es zu einer gravierenden Änderung der Architektur kommt.

Der Begriff **evolvability** stammt aus der Evolutionsbiologie. Der Begriff *evolvability* wird auch bei Software verwendet. *Evolvability* (Evolvierbarkeit) wird hier mit „Weiterentwickelbarkeit“ übersetzt. Dies bedeutet nicht nur die Erweiterbarkeit im Rahmen einer bestimmten Architektur, sondern auch die Fähigkeit, dass die Architektur des Systems umgestellt werden kann.

Um Schwächen eines Systems zu vermeiden, muss es das Ziel beim Entwurf eines Systems sein,

- die Abhängigkeiten der verschiedenen Systemteile zu minimieren,
- die Korrektheit des Systems zu gewährleisten und
- ferner das System verständlich aufzubauen und zu dokumentieren.

Ein System muss

- **weiterentwickelbar**,
- **korrekt** und
- **verständlich**

gebaut werden.



Das System muss also auf Änderungen vorbereitet sein und muss leicht geändert werden können, d. h. weiterentwickelbar sein. Es darf einfach nicht passieren, dass das System bei einer Änderung der Architektur weggeworfen werden muss. Korrektheit bedeutet, dass das System seine Aufgaben korrekt erfüllen muss. Zudem muss sich das System dem Menschen in jeglicher Hinsicht erschließen, in anderen Worten verständlich sein.

Prinzipien<sup>6</sup> für den objektorientierten Entwurf können dabei helfen, dieses Ziel zu erreichen und damit die Wartbarkeit der Software zu verbessern sowie ihre Lebensdauer zu erhöhen.

Die Prinzipien für den objektorientierten Entwurf lassen sich dahingehend unterscheiden, ob sie sich mit dem Entwurf einer einzelnen Klasse oder mit dem Entwurf für die Zusammenarbeit zwischen verschiedenen Klassen befassen.



Prinzipien für den objektorientierten **Entwurf einzelner Klassen** sind:

- Kapselung, Abstraktion und Information Hiding,
- Separation of Concerns und das Single Responsibility-Prinzip sowie
- das Interface Segregation-Prinzip.

Prinzipien für den objektorientierten **Entwurf miteinander kooperierender Klassen** sind<sup>7</sup>:

- Loose Coupling,
- das liskovsche Substitutionsprinzip,
- Design by Contract,
- das Open-Closed-Prinzip und
- das Dependency Inversion-Prinzip.

Kapitel 1.1 untersucht den Beitrag der behandelten Prinzipien zur Weiterentwickelbarkeit, Korrektheit und Verständlichkeit. Kapitel 1.2 bis Kapitel 1.4 befassen sich mit den Prinzipien für den Entwurf einzelner Klassen. Kapitel 1.5 bis Kapitel 1.9 diskutieren die Prinzipien für den Entwurf miteinander kooperierender Klassen. Kapitel 1.10 geht auf die Verringerung der Abhängigkeiten bei der Erzeugung von Objekten z. B. durch Dependency Injection ein.

## 1.1 Weiterentwickelbarkeit, Korrektheit und Verständlichkeit

Prinzipien, die die **Weiterentwickelbarkeit**, **Korrektheit** und **Verständlichkeit** fördern, werden bei Architektur- und Entwurfsmustern in besonderem Maße eingesetzt. Im Folgenden werden die erwähnten Prinzipien für den objektorientierten Entwurf in Verbindung mit diesen Qualitätsmerkmalen gebracht.

<sup>6</sup> Von Robert C. Martin gibt es den Begriff SOLID für eine Gruppe dieser Prinzipien. Diese Gruppe umfasst das **S**ingle Responsibility-Prinzip, das **O**pen-Closed-Prinzip, das **L**iskovsche Substitutionsprinzip, das **I**nterface Segregation-Prinzip und das **D**ependency Inversion-Prinzip. Der Begriff SOLID setzt sich aus den Anfangsbuchstaben dieser Prinzipien zusammen.

<sup>7</sup> Es gibt noch weitere Prinzipien. Als Beispiele sollen hier nur noch KISS ("Keep it simple, stupid") oder DRY ("Don't Repeat Yourself") – vermeide beispielsweise duplizierten Code – genannt werden.

### 1.1.1 Weiterentwickelbarkeit

Weiterentwickelbarkeit bedeutet nicht nur die Erweiterbarkeit im Rahmen einer bestimmten Architektur, sondern auch die Wiederverwendbarkeit vorhandener Komponenten im Rahmen einer anderen Architektur. Die **Weiterentwickelbarkeit** wird gefördert durch die Einhaltung

- des Interface Segregation-Prinzips,
- von Loose Coupling,
- des liskovschen Substitutionsprinzips,
- des Open-Closed-Prinzips und
- des Dependency Inversion-Prinzips.

Des Weiteren dienen Inversion of Control und verschiedene Techniken bei der Erzeugung von Objekten wie z. B. Dependency Injection der Reduktion der schädlichen Abhängigkeiten in erheblichem Maße, auch wenn sie nicht den Status eines Prinzips haben.

Das **Interface Segregation-Prinzip** will unnötige Abhängigkeiten eines Moduls wie einer Klasse oder Komponente von nicht benötigten Schnittstellen vermeiden.

Kapselung und Abstraktion ermöglichen nicht nur eine schmale Schnittstelle, sondern auch das Verstecken der Daten und der Methodenrumpfe eines Objekts im nicht zugänglichen Inneren einer Kapsel (**Information Hiding**). Bei Einhalten des Information Hiding sind die verschiedenen Objekte über schmale Schnittstellen als **Abstraktion** nur schwach gekoppelt (**Loose Coupling**).

Bei Einhaltung des **liskovschen Substitutionsprinzips** können Programme, die Referenzen auf Objekte von Basisklassen verwenden, auch für Referenzen auf Objekte von abgeleiteten Klassen verwendet werden. Dies stellt eine Erweiterung und damit auch eine Weiterentwicklung dar.

Das **Open-Closed-Prinzip** fordert, dass vorhandene lauffähige Software nicht verändert werden darf, die vorhandene Software aber erweiterbar sein soll.

Das **Dependency Inversion-Prinzip** dient zur Vermeidung bzw. Reduzierung von Abhängigkeiten unter Modulen, die in einer hierarchischen Beziehung zueinander stehen.

### 1.1.2 Korrektheit

Das **liskovsche Substitutionsprinzip** postuliert, dass eine Referenz auf ein Objekt einer Basisklasse jederzeit auch auf ein Objekt einer abgeleiteten Klasse zeigen kann. Damit eine solche Ersetzung korrekt funktioniert, ist die Einhaltung des Prinzips **Design by Contract** absolut erforderlich. Beide Prinzipien tragen zur **Korrektheit polymorpher Programme**<sup>8</sup> wesentlich bei.

---

<sup>8</sup> Korrektheit von Software umfasst viele Aspekte, siehe etwa [Gol12]. An dieser Stelle hier ist nur ausgeführt, welche der behandelten objektorientierten Entwurfsprinzipien für den Entwurf einen Beitrag zur Korrektheit liefern können.

Die Einhaltung der beiden Prinzipien ermöglicht die polymorphe Verwendung von ganzen Programmsystemen, die Referenzen auf Objekte von Basisklassen enthalten. Diese Programme können dann auch korrekt für Referenzen auf Objekte von abgeleiteten Klassen verwendet werden, ohne, dass bei jedem aktuellen Objekt eine Typprüfung stattfinden muss. Die Wiederverwendung ganzer Programmsysteme ist mehr als nur die Wiederverwendung einzelner Klassen im Falle der Vererbung oder Aggregation.

### 1.1.3 Verständlichkeit

Zur **Verständlichkeit einer Klasse** tragen wesentlich bei:

- Kapselung, Abstraktion und Information Hiding sowie
- Separation of Concerns und das Single Responsibility-Prinzip.

Das Prinzip der **Kapselung** hält die Daten und Methoden eines Objektes in der Kapsel einer Klasse. Es ermöglicht eine schmale Schnittstelle eines Objekts nach außen als **Abstraktion** seines Verhaltens. Die nicht in der schmalen Schnittstelle sichtbaren Anteile des Verhaltens (Methodenrümpfe, Servicemethoden) und die Daten werden verborgen (**Information Hiding**).

Das **Single Responsibility-Prinzip** und das Prinzip der **Separation of Concerns** beabsichtigen beide vom Kern her dasselbe. Sie fordern beide, dass eine Klasse nur einer einzigen Aufgabe nachgeht. Damit ist eine Klasse von anderen Klassen entkoppelt, ist in sich selbst kompakt und hat einen starken Zusammenhalt (engl. **strong cohesion**). Strong cohesion wurde bereits von Tom DeMarco [DeM79] gefordert:

"The more valid a module's reason for existing as a module, the more cohesive it is."<sup>9</sup>

Da verschiedene Klassen verschiedene Aufgaben erfüllen, sind verschiedene Klassen voneinander zu trennen, was die Übersichtlichkeit wesentlich erleichtert.

## 1.2 Kapselung, Abstraktion und Information Hiding

Die Begriffe Kapselung, Abstraktion und Information Hiding sind miteinander eng verwandt. Im Folgenden werden die Begriffe und ihre Zusammenhänge erläutert:

- **Kapselung**

Der Begriff der Kapselung ist eines der wichtigsten Konzepte der objektorientierten Programmierung. Es besteht in diesem Fall keine Trennung zwischen Daten und Funktionen wie in der funktionsorientierten Programmierung, z. B. in C.

Methoden und Daten verschmelzen bei der Kapselung zu einem Objekt. Daten und Methoden befinden sich also zusammen in einer Kapsel vom Typ einer Klasse.



<sup>9</sup> Frei übersetzt bedeutet das: Je besser begründet werden kann, warum ein Modul als Modul überhaupt existiert, umso besser ist sein Zusammenhalt.

Der Begriff der Kapselung umfasst sowohl die Realisierung von Information Hiding zum Verbergen der inneren Details, d. h. der Daten und der Implementierung des Verhaltens, als auch die Realisierung von Abstraktion zur Sichtbarmachung und zum Ansprechen des Verhaltens der Kapsel durch eine Schnittstelle nach außen.

- **Abstraktion**

Eine Schnittstelle einer Methode abstrahiert das Verhalten eines Objekts. Ein Objekt implementiert sein Verhalten in Schnittstellenmethoden, die außerhalb des Objekts nur als Abstraktion in Form der Methodenköpfe sichtbar sind.



Ein Objekt sollte nur über wohl definierte Schnittstellenmethoden mit seiner Umwelt in Kontakt treten. Die Methodenköpfe eines Objekts stellen die Schnittstellen eines Objekts zur Außenwelt dar<sup>10</sup>.

- **Information Hiding**

Die Daten einer Kapsel und die Rümpfe der Schnittstellenmethoden, d. h. die Implementierungsentscheidungen, sollen nach außen nicht direkt sichtbar sein. Die inneren Eigenschaften sollen vor der Außenwelt verborgen sein. Nur die Aufrufschnittstelle der Schnittstellenmethoden soll exportiert werden.



Man spricht daher auch von Information Hiding oder **Geheimnisprinzip**. Dieses Prinzip bedeutet, dass ein Teilsystem (hier ein Objekt) nichts von den Implementierungsentscheidungen eines anderen Teilsystems wissen darf. Damit wird vermieden, dass ein Teilsystem von der Implementierung eines anderen Teilsystems abhängt.

Eine Kapsel entsteht, indem man Abstraktion und Information Hiding anwendet. Durch Information Hiding werden immer Details wie etwa Daten verborgen. Durch Abstraktion entsteht die Schnittstelle nach außen.



Kapselung, Abstraktion und Information Hiding sind im folgenden Bild dargestellt:

---

<sup>10</sup> Darüber hinaus kann es auch noch Methoden geben, die nach außen nicht sichtbar sind. Sie dienen als Hilfsmethoden (**Service-Methoden**) und können nur durch eine Methode des gleichen Objekts aufgerufen werden.

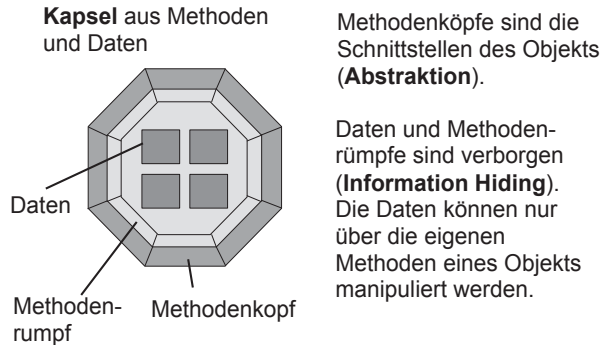


Bild 1-1 Daten und Methoden – die Bestandteile von Objekten

Die Operationen bzw. Methoden auf den Daten eines Objekts werden als eine "innere Fähigkeit" eines Objekts betrachtet. Wie die Operationen im Detail ablaufen, ist von außen nicht sichtbar. Von außen ist nur sichtbar, wie eine Methode aufgerufen werden kann.

### 1.3 Separation of Concerns und das Single Responsibility-Prinzip

Das Single Responsibility-Prinzip gilt nur für Klassen, Separation of Concerns gilt prinzipiell. Vor dem Hintergrund der Objektorientierung sind diese beiden Prinzipien eng miteinander verwandt.

**Separation of Concerns** ist ein altes Prinzip des Software Engineering und besagt, dass jede Funktion, die einer getrennten Aufgabe entspricht, in einem eigenen Modul realisiert wird.



In der Veröffentlichung "On the role of scientific thought" postulierte Dijkstra [Dij82]: "It is what I sometimes have called **the separation of concerns**, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by **focussing one's attention upon some aspect**: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant". Für die Modularisierung einer Verantwortlichkeit werden die anderen Verantwortlichkeiten also nicht betrachtet. Das sind jeweils eigene Module.

Der Begriff Single Responsibility-Prinzip stammt von Robert C. Martin [marsrp] und ist auch in [Mar03] ausführlich beschrieben:

"There should never be more than one reason for a class to change."



Übersetzt bedeutet das, dass es niemals mehr als einen Grund geben sollte, eine Klasse zu ändern.



Nach dem **Single Responsibility-Prinzip (SRP)** der Objektorientierung sollte also jede Klasse nur eine einzige Verantwortung haben. Alle Methoden sollen zur Erfüllung dieser Aufgabe beitragen. Eine Änderung der Verantwortlichkeit einer Klasse betrifft dann immer nur die jeweilige Klasse. Wenn in einer Klasse mehrere Verantwortlichkeiten wären, so könnte eine Änderung an einer Verantwortlichkeit die anderen Verantwortlichkeiten dieser Klasse beeinflussen.

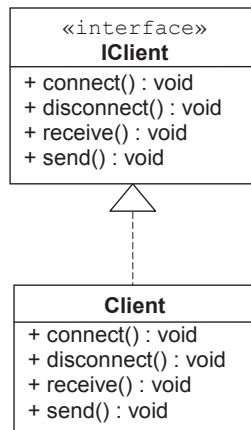
Das entspricht der Strategie der Separation of Concerns. Nach dem Prinzip der Separation of Concerns darf sogar jede Methode nur eine einzige Verantwortung haben. Damit soll eine Methode nur geändert werden, wenn sich die Anforderungen an die entsprechende Funktion geändert haben.

Man kann Separation of Concerns als einen prinzipiellen Prozess der Zerlegung betrachten und das Single Responsibility-Prinzip als ein objektorientiertes Design-Prinzip für Klassen.



**Beispiel**, angelehnt an [marsrp]:

Bild 1-2 zeigt ein einfaches Design, das mehrere Verantwortlichkeiten in einem Interface und damit auch in der Implementierung beinhaltet:



*Bild 1-2 Anwendung mit zwei Verantwortlichkeiten*

Das Interface `IClient` gibt einerseits Operationen zum Verbindungsauf- und -abbau und andererseits Operationen zur Übertragung von Daten vor. Die Implementierung dieser Operationen soll in einer Klasse `Client` erfolgen. Diese Lösung verstößt aber gegen das Single Responsibility-Prinzip, da ein Verbindungsauf- und -abbau und das Senden/Empfangen von Daten zwei grundsätzlich verschiedene Verantwortlichkeiten sind. Sie sollten deshalb getrennt voneinander behandelt werden. So können beispielsweise die Mechanismen zum Senden und Empfangen von Datenpaketen ausgetauscht werden, ohne dass der Verbindungsauf- und -abbau davon beeinflusst wird.

Bild 1-3 zeigt, dass die zwei Verantwortlichkeiten auf jeweils eine Klasse mit jeweils einer einzigen Verantwortlichkeit aufgeteilt werden können:

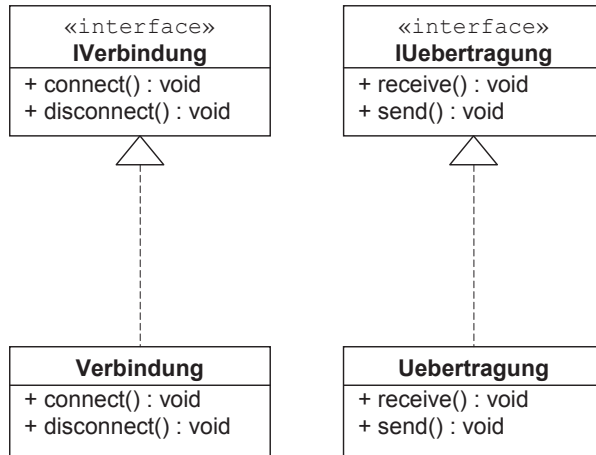


Bild 1-3 Anwendung mit getrennten Verantwortlichkeiten

Die verschiedenen Verantwortlichkeiten können nun über getrennte Interfaces und Implementierungen verwendet werden. Wird zum Beispiel das Interface `IVerbindung` zum Senden von Daten verändert, ist nur die dieses Interface implementierende Klasse `Verbindung` betroffen. Der Entwickler kann sich sicher sein, dass ein dort auftretender Fehler auch nur hier zu suchen ist. Die Klasse `Uebertragung` ist von dieser Änderung in keiner Weise betroffen. Es können somit auch keine Fehler durch eine Änderung des Interface `IUebertragung` in der Klasse `Verbindung` entstehen.

## 1.4 Interface Segregation-Prinzip

Das Interface Segregation-Prinzip (ISP) verlangt, dass große Interfaces in kleinere Interfaces aufgeteilt werden, so dass die kleineren Interfaces den Anforderungen der verschiedenen Clients genügen. Die Clients erhalten also nur Interfaces, die sie auch nutzen. Damit hängen sie nicht von nicht benötigten Schnittstellen ab. Dieses Prinzip stammt von Robert C. Martin [marisp] (siehe auch [Mar03]). Es lautet:

"Clients should not be forced to depend upon interfaces that they do not use."

Clients sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie gar nicht brauchen. Damit schlagen Änderungen an nicht benötigten Schnittstellen nicht auf die Client-Programme durch.



Die Anwendung dieses Prinzips erfordert die Analyse jeder einzelnen Methode in einem Interface. Es ist hierbei zu prüfen, ob eine betrachtete Methode von jedem Client der Klasse benötigt wird. Ist dies nicht der Fall, muss überlegt werden, welche Methoden in welcher Kombination von einem Client oder einer Gruppe von Clients benötigt

werden. Jedem Client oder jeder Gruppe von Clients wird daraufhin ein speziell zusammengestelltes Interface zur Verfügung gestellt, das von der betrachteten Klasse implementiert wird.

## 1.5 Loose Coupling

Entwirft man komplexe Systeme, so ist hierbei ein erster Schritt, diese Systeme in einfachere Teile, die Subsysteme bzw. Teilsysteme, zu zerlegen. Die Identifikation eines Teilsystems nach dem Prinzip "Teile und herrsche" ist dabei eine schwierige Aufgabe.

Als Qualitätsmaß für die Güte eines Entwurfs werden hierbei

- die funktionelle **Kopplung** (engl. **coupling**), d. h. die Stärke der Wechselwirkungen zwischen den Teilsystemen, und
- die **Kohäsion** (engl. **cohesion**), d. h. die Stärke der inneren Abhängigkeiten oder der Zusammenhalt innerhalb eines Teilsystems,

betrachtet.

Ein Entwurf gilt dann als gut, wenn

- innerhalb eines Teilsystems eine möglichst hohe Bindungsstärke oder **starke Kohäsion** (engl. **strong cohesion**) und
- zwischen den Teilsystemen eine **schwache Wechselwirkung** (engl. **loose coupling**)

besteht. Man strebt also beim Entwurf "loosely coupled" Teilsysteme mit einer starken Kohäsion der einzelnen Teilsysteme an.



Anstelle von loose coupling spricht man auch von low oder weak coupling und anstelle von einer strong cohesion auch von einer high oder tight cohesion.

Dabei müssen die Wechselwirkungen zwischen den Teilsystemen "festgezurt", in anderen Worten in Form von Schnittstellen definiert werden. Die Implementierung der Teilsysteme interessiert dabei nicht und wird verborgen (Information Hiding, siehe Kapitel 1.2).

Die Vorteile von "loosely coupled" Teilsystemen sind eine hohe Flexibilität des Verbunds mit einer Spezialisierung der Teilsysteme und eine vereinfachte Austauschbarkeit. Daraus resultieren weniger Abhängigkeitsprobleme.



## 1.6 Liskovsches Substitutionsprinzip

Polymorphie ist ein zentrales Konzept der objektorientierten Programmierung und betrifft sowohl Methoden als auch Objekte (siehe [Gol12]).

**Polymorphie (Vielgestaltigkeit) von Objekten** bedeutet, dass ein Objekt in einer Vererbungshierarchie von verschiedenem Typ sein kann. Es kann vom Typ einer Basisklasse sein, aber auch vom Typ einer Unterklasse.



Das ist genauso zu sehen wie im Alltag, wo beispielsweise ein Student einerseits als eine normale Person, andererseits aber auch als Student, d. h. als eine Instanz einer Unterklasse `Student` der Klasse `Person` auftreten kann.

Wichtig ist, dass im Falle des Überschreibens einer Methode der Basisklasse in einer Unterklasse die überschreibende Methode der Unterklasse gleichartig wie die überschriebene Methode der Basisklasse aufgerufen wird und auch gleichartig antwortet, wobei die Implementierung der Methode jedoch eine andere Ausprägung annimmt.



Barbara Liskov hat in [Lis87] Bedingungen formuliert, unter denen ein Programm die Verwendung polymorpher Objekte gefahrlos unterstützt. Die Bedingungen sind seitdem als **liskovsches Substitutionsprinzip (LSP)** bekannt. Das liskovsche Substitutionsprinzip fordert:

Will man Polymorphie von Objekten für ein Programm haben, so muss im Programm eine Referenz auf eine Instanz einer abgeleiteten Klasse in einem Programm an die Stelle einer Referenz auf eine Instanz einer Basisklasse treten können. Überschreibende Methoden in der abgeleiteten Klasse dürfen hierbei die Verträge einer Basisklasse mit Kundenklassen, d. h. die Vor- und Nachbedingungen der Methoden der Basisklasse und ihre Klasseninvarianten nicht brechen.



Solange bei der Ableitung von einer Basisklasse der Vertrag der entsprechenden Basisklasse in der abgeleiteten Unterklasse nicht gebrochen wird, ist es möglich, das für Basisklassen geschriebene Programm auch für Unterklassen zu verwenden, die eventuell erst später erfunden werden.

Findet ein Überschreiben – also die Neudefinition einer Operation in einer abgeleiteten Klasse – statt, so muss darauf geachtet werden, dass in der abgeleiteten Klasse die Verträge der Basisklasse eingehalten werden.



Wenn das liskovsche Substitutionsprinzip nicht gelten würde, müsste stets geprüft werden, von welchem Typ das aktuelle Objekt ist, auf welches die Referenz gerade verweist.



Der **Vertrag** einer Klasse umfasst die Vor- und Nachbedingungen der Methoden und die Invarianten einer Klasse. Dabei kann eine Klasse mit verschiedenen Kundenklassen verschiedene Verträge haben. Was es bedeutet, dass ein Vertrag durch eine abgeleitete Klasse nicht gebrochen werden darf, wird in Kapitel 1.7 erläutert.

Man muss das liskovsche Substitutionprinzip unter zwei Gesichtspunkten betrachten:

1. Erweitert eine Klasse eine andere Klasse nur und überschreibt nichts, sondern fügt nur Attribute und Methoden hinzu, dann ist ganz klar, dass eine Instanz einer abgeleiteten Klasse an die Stelle einer Instanz ihrer Basisklasse treten kann. Dabei wird nämlich einfach durch einen Cast der erweiternde Anteil wegprojiziert. Zusätzliche Methoden der abgeleiteten Klasse werden dabei nicht angesprochen. Bei einer reinen Erweiterung wird bei der Ableitung keine Methode überschrieben.
2. Dieses Casten erfolgt natürlich auch, wenn die abgeleitete Klasse Methoden der Basisklasse überschreibt. Im Fall einer späten Bindung von Instanzmethoden wie in Java wird dann die überschreibende Methode an Stelle der überschriebenen Methode aufgerufen. Das darf aber keinen schädlichen Einfluss auf den Ablauf des Programms haben! Dies erzwingt, dass der Programmierer beim Überschreiben dafür sorgen muss, dass die überschreibende Instanzmethode den Vertrag der entsprechenden überschriebenen Instanzmethode einhält.

Werden Instanzmethoden in einer abgeleiteten Klasse überschrieben, so tritt in der abgeleiteten Klasse die überschreibende Instanzmethode an die Stelle der ursprünglichen, überschriebenen Methode. Wird ein Objekt der abgeleiteten Klasse als Objekt der Basisklasse betrachtet, wird trotzdem die überschriebene Methode aufgerufen, also für jedes Objekt die Methode seines Typs.



### Beispiel:

Bild 1-4 zeigt eine Klassifikation von Vögeln. Die abstrakte Klasse `Vogel` deklariert die abstrakten Operationen `essen()` und `fliegen()`.

Ein Pinguin ist ein Vogel, er ist jedoch flugunfähig. Bei Einsatz der Vererbung muss aber die Methode, die das Fliegen abbildet, in einer abgeleiteten Klasse vorhanden sein. Daher ist der im Folgenden gezeigte Entwurf einer Vererbungshierarchie falsch:

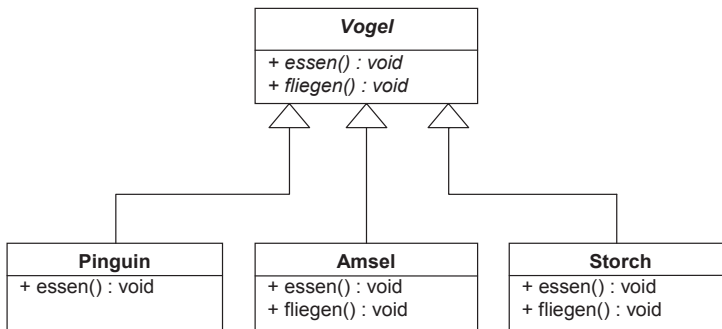


Bild 1-4 Jede abgeleitete Klasse besitzt sämtliche Methoden der abstrakten Klasse

Wird die Methode `fliegen()` in der Klasse **Pinguin** implementiert, wobei eine Exception geworfen wird oder einfach nichts passiert (leerer Methodenrumpf), da ein Pinguin ja nicht fliegen kann, so bricht dieses Vorgehen den Vertrag, den die Basis-klass **Vogel** mit einer Kundenklasse hat. Die abgeleitete Klasse **Pinguin** verstößt somit gegen das liskovsche Substitutionsprinzip. Wird dies fälschlicherweise billigend in Kauf genommen, so kann dies zu einem unerwarteten Verhalten führen, beispielsweise dann, wenn man über eine Liste von Vögeln iteriert und für jedes Element dieser Liste die Methode `fliegen()` aufruft.

Durch Einteilen der Vögel in flugfähige und flugunfähige Tiere kann – wie in folgendem Bild gezeigt – das Problem gelöst werden:

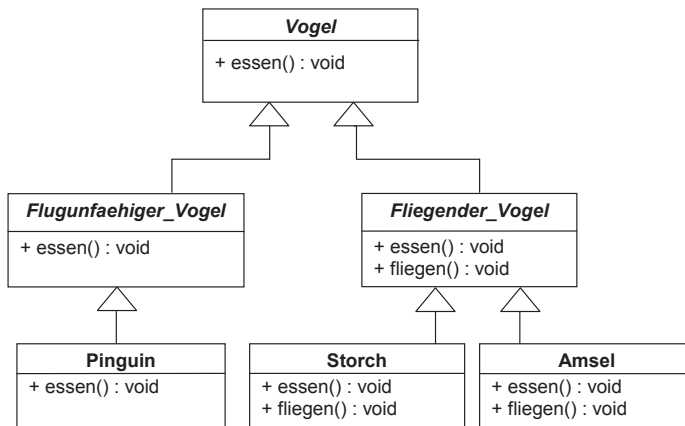


Bild 1-5 Einteilung der Vögel in unterschiedliche Gruppen

## 1.7 Design by Contract

**Entwurf durch Verträge** (engl. **Design by Contract**, abgekürzt: **DbC**) wurde von Bertrand Meyer [Mey97], dem Entwickler der Programmiersprache Eiffel, als Entwurfstechnik eingeführt. Diese Technik wurde in Eiffel in eine Programmiersprache umgesetzt, stellt aber ein allgemeingültiges Prinzip dar, das beim objektorientierten Ent-

wurf unabhängig von der jeweiligen objektorientierten Programmiersprache eingesetzt werden kann.

Eine Klasse besteht nicht nur aus Methoden und Attributen – eine Klasse wird von anderen Klassen – hier Kunden genannt – benutzt und hat damit Beziehungen zu all ihren Kunden.

Das Prinzip "Entwurf durch Verträge" fordert für diese Beziehungen eine formale Übereinkunft zwischen den beteiligten Partnern, in der präzise definiert wird, unter welchen Umständen ein korrekter Ablauf des Programms erfolgt.



Um was es hier vor allem geht, ist, dass beim Aufruf einer Methode Aufrufer und aufgerufene Methode sich gegenseitig aufeinander verlassen können. Die Beziehung zwischen Aufrufer und aufgerufener Methode kann man formal als einen **Vertrag** bezeichnen, der nicht gebrochen werden darf, da ansonsten eine Fehlersituation entsteht. Bei einem Vertrag haben in der Regel beide Seiten Rechte und Pflichten.

In der Technik "Entwurf durch Verträge" von Bertrand Meyer werden Verträge mit Hilfe von **Zusicherungen** spezifiziert. Eine Zusicherung ist ein boolescher Ausdruck an einer bestimmten Programmstelle, der niemals falsch werden darf.

### 1.7.1 Zusicherungen

Ein Vertrag enthält drei verschiedene Arten von Zusicherungen:

- **Vorbedingungen**,
- **Nachbedingungen** und
- **Invarianten**.

So wie im Alltag ein Vertrag die Beziehungen zwischen Parteien (Personen, Organisationen) regelt, beschreibt ein Vorbedingungen-Nachbedingungen-Paar den Vertrag einer Methode mit ihrem Kunden, dem Aufrufer.

Der **Vertrag einer Methode** umfasst die **Vor- und Nachbedingungen einer Methode**.



Invarianten beziehen sich nicht auf eine einzelne Methode. **Invarianten** beziehen sich immer auf eine **Klasse**. Eine Invariante muss also für jedes einzelne Objekt einer Klasse erfüllt sein, damit ein System korrekt arbeitet oder in einem korrekten Zustand ist.

Da die Invarianten von allen Methoden einer Klasse, die von einem Kunden aufgerufen werden können, eingehalten werden müssen, um die Korrektheit zu gewährleisten, spricht man auch von **Klasseninvarianten**.

Der **Vertrag einer Klasse** umfasst die **Verträge der Methoden** sowie die **Invarianten der Klasse**. Verschiedenen Kunden einer Klasse können verschiedene Verträge zur Verfügung gestellt werden.



Eine **Vorbedingung** (engl. **precondition**) stellt die Einschränkungen dar, unter denen eine Methode korrekt aufgerufen wird. Eine Vorbedingung stellt eine Verpflichtung für einen Aufrufer dar, sei es, dass der Aufruf innerhalb der eigenen Klasse erfolgt oder von einer Kundenklasse kommt. Ein korrekt arbeitendes System führt nie einen Aufruf durch, der nicht die Vorbedingung der gerufenen Methode erfüllen kann. Eine Vorbedingung bindet also einen Aufrufer. Eine Vorbedingung definiert die Bedingung, unter der ein Aufruf zulässig ist. Sie stellt eine **Verpflichtung für den Aufrufer** dar und einen **Nutzen für den Aufgerufenen**.

Der Aufrufer hat die Pflicht, die **Vorbedingungen** des Aufzurufenen zu erfüllen. Er muss prüfen, ob eine Vorbedingung erfüllt ist. Den Nutzen hat der Aufgerufene.



Eine **Nachbedingung** (engl. **postcondition**) stellt den Zustand nach dem Aufruf einer Methode dar. Eine Nachbedingung bindet eine Methode einer Klasse. Sie stellt die Bedingung dar, die von der Methode eingehalten werden muss. Eine Nachbedingung ist eine **Verpflichtung für den Aufgerufenen** und ein **Nutzen für den Aufrufer**. Mit der Nachbedingung wird garantiert, dass der Aufrufer nach Ausführung einer Methode einen Zustand mit gewissen Eigenschaften vorfindet, natürlich immer unter der Voraussetzung, dass beim Aufruf der Methode die Vorbedingung erfüllt war.

Bei einer **Nachbedingung** hat der Aufgerufene die Pflicht, den Nutzen hat der Aufrufer. Der Aufgerufene muss also die Nachbedingung überprüfen.



Wie bei einem guten Vertrag im täglichen Leben haben also Aufrufer und Aufgerufenen Pflichten und Vorteile. Der Aufgerufenen hat den Vorteil, dass er unter den gewünschten Bedingungen ablaufen kann. Der Aufrufer hat den Vorteil, dass gewisse Bedingungen nach dem Aufruf einer Methode erfüllt sind.



Eine **Invariante** (engl. **invariant**) ist eine Zusicherung bezüglich einer Klasse. Sie muss also von allen Objekten dieser Klasse eingehalten werden. Die Eigenschaften der Invariante gelten für alle Operationen einer Klasse und nicht individuell nur für eine Methode. Sie sind damit Klasseneigenschaften im Gegensatz zu Vor- und Nachbedingungen von Methoden, die den Aufruf und das Ergebnis einzelner Methoden charakterisieren.



Invarianten müssen von allen exportierten Methoden einer Klasse vor und nach dem Methodenaufruf eingehalten werden. Die Invarianten gelten während der gesamten Lebensdauer der Objekte einer solchen Klasse.

Eine Invariante kann während der Ausführung einer exportierten Methode oder beim Aufruf von Service-Methoden, die nicht außerhalb der entsprechenden Klasse sichtbar sind – also nicht exportiert werden –, temporär verletzt werden. Dies stellt kein Problem dar, da die Invariante erst nach Ausführung einer exportierten Methode dem Aufrufer wieder zur Verfügung steht.

Eine Klasseninvariante muss vor und nach dem Aufruf einer nach außen sichtbaren Methode eingehalten sein. Beim Aufruf von internen – beispielsweise privaten – Methoden einer Klasse müssen die Invarianten einer Klasse nicht unbedingt beachtet werden und können auch mal nicht erfüllt sein.



Invarianten gelten bei exportierten Methoden vor und nach dem Aufruf, bei dem Aufruf von Konstruktoren nach dem Aufruf.

### Beispiel:

Betrachtet wird die Klasse `Stack`, die einen Speicher für Objekte realisiert, der nach dem Stack-Prinzip organisiert ist und nur eine begrenzte Zahl von Objekten (gegeben durch die Stackgröße) aufnehmen kann. Die Klasse `Stack` kann in Java beispielsweise die folgende Schnittstelle anbieten:

```
abstract class Stack
{
    void push (Object o);
    Object pop();
}
```

Die Klasse `Stack` hat die **Invariante**, dass die Zahl der Elemente auf dem Stack bei allen Operationen größer gleich Null und kleiner gleich der gegebenen maximalen Stackgröße sein muss.

Die Methode `push()` hat die **Vorbedingung**, dass die Zahl der Elemente auf dem Stack kleiner gleich als die maximale Stackgröße sein muss. Die **Nachbedingung** von `push()` ist, dass die Zahl der Elemente auf dem Stack um 1 größer als vor dem Aufruf ist.

Die **Vorbedingung** von `pop()` ist, dass die Zahl der Elemente auf dem Stack größer gleich 1 sein muss. Die **Nachbedingung** von `pop()` ist, dass die Zahl der Elemente auf dem Stack um 1 kleiner als vor dem Aufruf ist.

Nach Abarbeitung einer Methode bzw. einem Konstruktor müssen die Invarianten wiederhergestellt sein.

### 1.7.2 Einhalten der Verträge beim Überschreiben

Um das Einhalten der Verträge beim Überschreiben zu betrachten, wird die folgende Situation zugrunde gelegt:

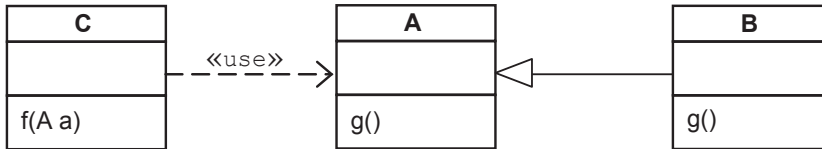


Bild 1-6 Überschreiben einer Methode

Die Klasse *B* sei von der Klasse *A* abgeleitet und soll die Methode *g()* aus *A* überschreiben. Aufrufer von *g()* sei eine Methode *f()* einer Kundenklasse *C*. Die Methode *f()* soll die folgende Aufrufschnittstelle besitzen: *f(A a)*. Mit anderen Worten: An *f()* kann beispielsweise eine Referenz auf ein Objekt der Klasse *A* oder eine Referenz auf ein Objekt der von der Klasse *A* abgeleiteten Klasse *B* übergeben werden. Der Kunde *f()* kann zur Laufzeit nicht wissen, ob ihm eine Referenz auf ein Objekt der Klasse *A* oder der Klasse *B* übergeben wird (liskovsches Substitutionsprinzip). Dem Kunden *f()* ist auf jeden Fall nur die Klasse *A* bekannt und daran richtet er sich aus! Also kann *f()* beim Aufruf von *g()* nur den Vertrag der Methode *g()* aus *A* beim Aufruf beachten. *f()* stellt also die Vorbedingungen für *g()* aus *A* sicher und erwartet im Gegenzug, dass *g()* seine Nachbedingungen erfüllt. Das folgende Bild visualisiert, dass es für *f()* kein Problem darstellt, eine schwächere Vorbedingung beim Aufruf von *g()* zu erfüllen:

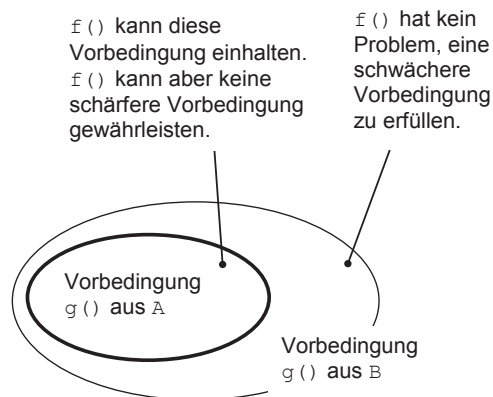


Bild 1-7 Aufweichen einer Vorbedingung in einer abgeleiteten Klasse

Wie im täglichen Leben auch, darf ein Vertrag übererfüllt werden, er darf aber nicht verletzt werden! Dies hat zur Konsequenz, dass *g()* aus *B* die **Vorbedingungen nicht verschärfen** kann, denn darauf wäre der Kunde *f()* überhaupt nicht eingerichtet. *g()* aus *B* darf **aber die Vorbedingungen aufweichen** (siehe Bild 1-7). Dies stellt für *f()* überhaupt kein Problem dar, denn aufgeweichte Vorbedingungen kann *f()* sowieso mühelos einhalten. In entsprechender Weise liegt es auf der Hand, dass *g()* aus *B* die Nachbedingungen nicht aufweichen darf, denn der Kunde *f()* erwartet die

Ergebnisse in einem bestimmten Bereich (siehe Bild 1-8). Das folgende Bild zeigt das Verschärfen einer Nachbedingung in einer abgeleiteten Klasse:

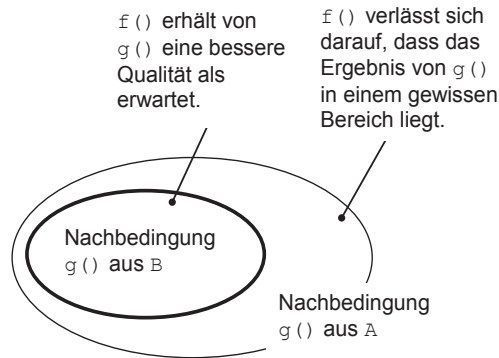


Bild 1-8 Verschärfen einer Nachbedingung in einer abgeleiteten Klasse

Da ein Kunde sich auf die Basisklasse eingestellt hat, muss sichergestellt werden, dass der überschreibende Anteil sich korrekt verhält.

Eine Nachbedingung kann nur verschärft werden (kleinerer Wertebereich), denn auf eine aufgeweichte Nachbedingung (größerer Wertebereich) wäre der Kunde gar nicht vorbereitet, da er sich an der Basisklasse orientiert. Genausowenig kann der Kunde einen engeren Wertebereich bei der Vorbedingung akzeptieren, da er auf den Wertebereich der Basisklasse vorbereitet ist. Eine Vorbedingung kann also nur aufgeweicht werden, denn eine aufgeweichte Vorbedingung kann der Kunde problemlos erfüllen.



### Beispiel:

Eine überschreibende Methode einer abgeleiteten Klasse darf:

- eine Nachbedingung der überschriebenen Methode nicht aufweichen, d. h., wenn eine Methode z. B. einen Rückgabewert vom Typ `int` hat und garantiert, dass sie nur Werte zwischen 1 und 10 liefert, so darf die überschreibende Methode keine Werte außerhalb dieses Bereichs liefern,
- eine Vorbedingung der überschriebenen Methode nicht verschärfen, d. h., wenn eine Methode z. B. einen formalen Parameter vom Typ `int` spezifiziert und einen gültigen Wertebereich zwischen 1 und 10 hat, so darf die überschreibende Methode diesen Wertebereich nicht einschränken.

Wird etwa eine Invariante einer Basisklasse durch eine überschreibende Methode in einer abgeleiteten Klasse aufgeweicht, so kann das zu einem Fehlverhalten der abgeleiteten Klasse führen, denn darauf ist eine Kundenklasse nicht vorbereitet.



Eine Verschärfung einer Invariante führt zu keinen Problemen, da die Invariante nach wie vor im erwarteten Bereich der Basisklasse liegt.

Abgeleitete Klassen dürfen Invarianten nur verschärfen.



## 1.8 Open-Closed-Prinzip

Das Open-Closed-Prinzip (OCP) stammt von Bertrand Meyer [Mey97] und lautet:

"Module sollten offen sein und geschlossen".



Dieser scheinbare Widerspruch lässt sich folgendermaßen erklären:

- Ein Modul sollte **offen** sein in dem Sinne, dass es erweiterbar ist. Zum Beispiel sollten neue Methoden oder Datenfelder hinzugefügt werden können. Eine Klasse sollte also angepasst und dabei abgeändert werden können, ohne den Code und die Schnittstelle der ursprünglichen Klasse zu ändern. Erweiterungen können z. B.
  - durch statische Vererbung oder
  - dynamisch über die Verwendung einer aggregierten Abstraktion<sup>11</sup> (siehe beispielsweise das Brückenmuster oder Bild 1-12)

erfolgen.

- Ein Modul ist **geschlossen**, wenn es zur Benutzung in unveränderter Form durch andere Module im Rahmen einer Bibliothek zur Verfügung steht. Dies bedeutet, dass ein Modul stabil ist und wiederverwendet werden kann.

Auf der Implementierungsebene bedeutet Geschlossenheit, dass der Code geschlossen für Veränderungen ist. Der bereits vorhandene Code eines Moduls kann übersetzt und getestet werden, er ist lauffähig und kann in eine Bibliothek aufgenommen und benutzt werden.

Durch die **Geschlossenheit des Quellcodes bzw. des lauffähigen Codes** gegenüber Veränderungen erreicht man Robustheit und Wiederverwendbarkeit der vorhandenen Programme in einer Bibliothek. Durch die **Offenheit des Quellcodes bzw. des lauffähigen Codes** gewinnt man eine Wiederverwendbarkeit des vorhandenen Quellcodes als Teil von neuen Modulen und damit eine Erweiterbarkeit des Vorhandenen.



<sup>11</sup> Eine aggregierte Abstraktion kann eine aggregierte Schnittstelle bzw. eine aggregierte abstrakte Klasse sein.

Die statische Vererbung ist ein Beispiel dafür, dass der Quellcode einer Klasse zwar geschlossen gegenüber Veränderungen aber dennoch offen für Erweiterungen ist. Bei der Anwendung des liskovschen Substitutionsprinzips wird der lauffähige Code einer Klasse nicht verändert, trotzdem ist die Klasse erweiterbar, da die Referenzen auf Objekte von Klassen zeigen können, die es zum Zeitpunkt der Erstellung des Quellcodes noch gar nicht gab.

Im Falle eines Entwurfs bzw. einer Spezifikation eines Moduls bedeutet die Geschlossenheit, dass das Modul offiziell abgenommen ist und selbst als wiederverwendbarer Baustein zur Verfügung steht.

Durch die **Geschlossenheit** von **Spezifikationen** gegenüber Veränderungen wird ihr Inhalt stabil und selbst wieder verwendbar. Durch die **Offenheit** von **Spezifikationen** gewinnt man ihre Wiederverwendbarkeit als Teil eines neuen Moduls.



### Beispiel:

Im folgenden Beispiel symbolisiert die abstrakte Basisklasse `GrafischesElement` ein Framework, das nicht verändert, aber erweitert werden kann. Geschlossen (closed) ist der Quellcode der abstrakten Basisklasse `GrafischesElement`. Eine Erweiterung dieser Basisklasse erfolgt in den abgeleiteten Klassen `Dreieck` und `Viereck`, die anschließend von einem grafischen Editor verwendet werden sollen.

Jedes grafische Element weist bestimmte Eigenschaften und Methoden auf, die in ihrer Funktionsweise unabhängig von der Ausprägung des Elementes sind. Dazu zählt z. B. die Schwerpunktskoordinate eines Elements und die zugehörigen Verschiebungsmethoden für den Schwerpunkt. Im Gegensatz dazu gibt es Methoden, die von den abgeleiteten Klassen selbst implementiert werden müssen – beispielsweise die Darstellungsroutine.

Die für die abgeleiteten Klassen invarianten Anteile werden in der Klasse `GrafischesElement` abgebildet. Diese Basisklasse kann sowohl die von allen Elementen in gleicher Weise genutzten Methoden und Eigenschaften kapseln, als auch Methodenköpfe als Vorgabe für die Implementierung der abgeleiteten Klassen vorweisen.

Aufgrund der Abstraktheit der Methodenköpfe ist diese Basisklasse abstrakt. Daher kann keine Instanz von ihr gebildet werden. Außerdem müssen die von allen Elementen genutzten Methoden vor Veränderungen – beispielsweise durch Überschreiben – geschützt werden. Diese Methoden werden in UML `leaf`-Methoden genannt und werden in Java durch das Schlüsselwort `final` deklariert.

In Bild 1-9 ist ein Beispiel für die Modellierung der Vererbungshierarchie dargestellt:

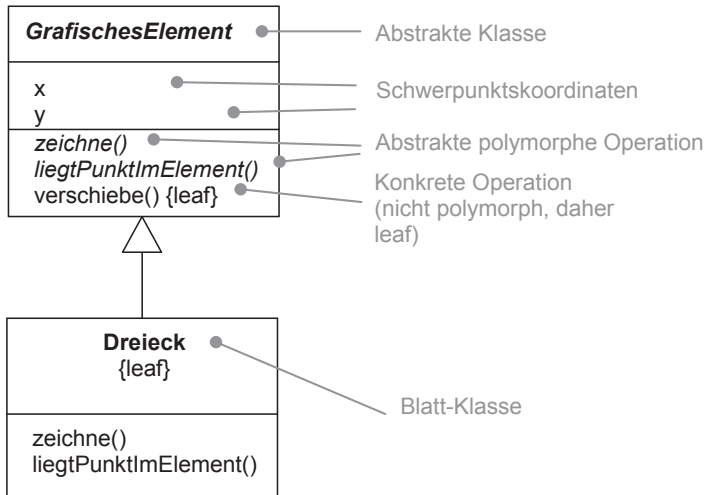


Bild 1-9 Die abstrakte Klasse *GrafischesElement* als symbolisches Framework

Das folgende Beispiel ist hier im Buch nur verkürzt wiedergegeben. Ausgelassene Stellen sind mit ... markiert. Der vollständige Quellcode des Beispiels ist auf dem begleitenden Webauftritt zu finden.

Die abstrakte Klasse *GrafischesElement* könnte wie folgt aussehen:

```
// Datei: GrafischesElement.java
import java.awt.Graphics;

public abstract class GrafischesElement
{
    // Schwerpunktskoordinaten
    protected int x, y;

    // hier koennen weitere Eigenschaften, wie bspw. Fuellfarbe
    // definiert werden.

    // Konstruktor mit Schwerpunktskoordinaten
    public GrafischesElement(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // Methode zur Verschiebung des Schwerpunktes; durch final als
    // leaf-Methode deklariert.
    final public void verschiebe(int x, int y)
    {
        this.x += x;
        this.y += y;
    }
}
```

```

    final public int getX()
    {
        return x;
    }

    final public int getY()
    {
        return y;
    }

    // Methodenkopf der Darstellungsroutine; Graphics ist eine
    // Klasse, welche das Auftragen von Grafiken auf Komponenten
    // erlaubt. In der zeichne()-Methode wird anschliessend das
    // darzustellende Element auf das Graphics-Objekt aufgetragen.
    public abstract void zeichne(Graphics g);

    // Methode zum Ueberpruefen, ob Punkt im Element liegt.
    public abstract boolean liegtPunktImElement (int x, int y);
}

```

Die Anwendung soll Dreiecke und Vierecke als grafische Elemente enthalten, die die abstrakte Basisklasse `GrafischesElement` nicht abändern, aber erweitern. Zum Zeichnen eines Vierecks wird in der folgenden Klasse `Viereck` die in AWT<sup>12</sup> vorhandene Klasse `java.awt.Rectangle` benutzt:

```

// Datei: Viereck.java
import java.awt.Graphics;
import java.awt.Rectangle;

public class Viereck extends GrafischesElement
{
    int laenge, hoehe;
    Rectangle viereck;

    public Viereck(int x, int y, int laenge, int hoehe)
    {
        super(x, y);
        this.laenge = laenge;
        this.hoehe = hoehe;
    }

    // Implementierte zeichne()-Methode der Basisklasse
    public void zeichne(Graphics g)
    {
        // Viereck erstellen
        viereck = new Rectangle(this.x - (int) (0.5 * laenge),
                                this.y - (int) (0.5 * hoehe), laenge, hoehe);
    }
}

```

<sup>12</sup> Die Klassenbibliothek AWT kann als Vorgänger der Klassenbibliothek Swing angesehen werden. AWT-GUI-Komponenten werden aufgrund ihrer Abhängigkeit vom Betriebssystem als "schwerge-wichtig" bezeichnet. "Leichtgewichtige" Swing-GUI-Komponenten werden hingegen mit Hilfe der Java 2D-Klassenbibliothek durch die JVM selbst auf dem Bildschirm gezeichnet und sind damit in Aussehen und Verhalten unabhängig vom Betriebssystem.

```

        // Viereck zeichnen
        g.fillRect(viereck.x, viereck.y, viereck.width,
                   viereck.height);
    }

    public boolean liegtPunktImElement(int x, int y)
    {
        return viereck.contains(x,y);
    }
}

```

Da in AWT keine eigene Klasse zum Zeichnen von Dreiecken vorhanden ist, wird ein Dreieck mit Hilfe der Klasse `java.awt.Polygon` als Polygonzug gezeichnet. Dies ist in der Methode `getDreieck()` der Klasse `Dreieck` zu sehen:

```

// Datei: Dreieck.java
import java.awt.Graphics;
import java.awt.Polygon;

final public class Dreieck extends GrafischesElement
{
    int seitenlaenge;
    Polygon dreieck;

    public Dreieck(int x, int y, int seitenlaenge)
    {
        super(x, y);
        this.seitenlaenge = seitenlaenge;
    }

    public Polygon getDreieck(int seitenlaenge)
    {
        // Ein Dreieck wird als Polygon betrachtet:
        // Zuerst werden die Koordinaten der Eckpunkte berechnet und
        // dann ein Polygon erzeugt. Das Polygon wird als Ergebnis
        // zurueckgegeben.
        int xArr[]={getX()-(seitenlaenge/2),
                    getX()+(seitenlaenge/2),getX()};
        int yArr[]={getY()+(seitenlaenge/2),getY()+(seitenlaenge/2),
                    getY()-(seitenlaenge/2)};
        // Neues Polygon zurueckgeben
        return new Polygon(xArr, yArr, 3);
    }

    public void zeichne(Graphics g)
    {
        dreieck = getDreieck(seitenlaenge);
        g.fillPolygon(dreieck);
    }
}

```



```

    public boolean liegtPunktImElement (int x, int y)
    {
        return dreieck.contains(x,y);
    }
}

```

Die Klassen `Dreieck` und `Viereck` werden dann anschließend von einem grafischen Editor genutzt. Eine einfache Editor-Implementierung könnte dann wie folgt aussehen:

```

// Datei: Editor.java
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.ArrayList;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Editor extends JPanel
{
    static private Editor editor = null;

    // Hoehe und Weite des Editors. Außerdem die Hoehe der
    // Auswahlleiste.
    final static int hoehe = 500, weite = 500;
    final static int auswahlLeisteHoehe = 75;

    // Liste fuer Auswahlelemente
    final static ArrayList<GrafischesElement> auswahlElement =
        new ArrayList<GrafischesElement>();
    // Liste fuer die auf der Editorflaeche platzierten Elemente
    ArrayList<GrafischesElement> gezeichneteElemente =
        new ArrayList<GrafischesElement>();

    // Referenz auf das naechste zu zeichnende Objekt
    GrafischesElement naechstesElement = null;

    static public void main(String[] args)
    {
        // Fenster anfordern
        JFrame frame = new JFrame();
        // Groesse setzen
        frame.setSize(weite, hoehe);
        . . .
    }
    . . .
}

```

Dieser Editor kann Drei- oder Vierecke erstellen und verschieben. Das folgende Bild zeigt einen Screenshot des grafischen Editors:

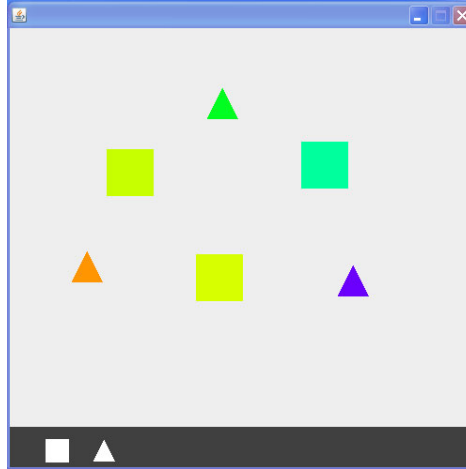


Bild 1-10 Grafischer Editor

## 1.9 Das Dependency Inversion-Prinzip und Inversion of Control

Bei dem Dependency Inversion-Prinzip und bei Inversion of Control findet eine Umkehr der Abhängigkeiten (**Dependency Inversion**) statt. Bei dem Dependency Inversion-Prinzip erfolgt die Umkehr der Abhängigkeiten in **hierarchischen Systemen**. Bei Inversion of Control erfolgt die Umkehr der Abhängigkeiten **zwischen gleichberechtigten Partnern**.

### 1.9.1 Das Dependency Inversion-Prinzip

Das Dependency Inversion-Prinzip (DIP) betrachtet die Abhängigkeit von Klassen in Hierarchien. Dieses Prinzip stammt von Robert C. Martin [mardip] (siehe auch [Mar03]) und lautet:

"High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions."



Klassen werden oft in Ebenen eingeordnet. Die oberste Ebene beschreibt nach Booch [Boo95] die Politik der Geschäftsprozesse, die nächste Ebene die Ebene der Mechanismen und wiederum die nächste Ebene die Ebene der Hilfsdienste. Eine jede dieser Ebenen bietet bestimmte Dienste an. In der klassischen Technik – wie man es von der prozeduralen Technik gewöhnt ist, aber auch objektorientiert praktizieren kann –, ruft eine Klasse einer höher stehenden Ebene die Dienste einer Klasse einer tiefer stehenden Ebene auf. Ändert sich jedoch eine Klasse einer tiefer stehenden Ebene, so kann es sein, dass die nutzenden Klassen der höher stehenden Ebene auch geändert werden müssen. Und damit ist bei dieser Vorgehensweise eine höher stehende Ebene von einer tiefer stehenden Ebene abhängig, wie das folgende Bild zeigt:

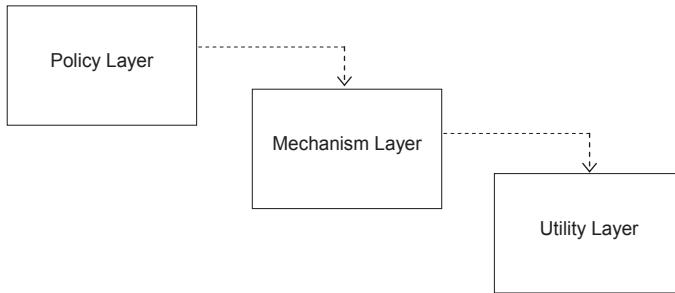


Bild 1-11 Einfache Ebenenstruktur

In einem klassischen Entwurf würde eine Klasse einer höheren Ebene direkt eine Methode einer Klasse der untergeordneten Ebene aufrufen und würde dadurch von ihr abhängig.



Der **Lösungsansatz des Dependency Inversion-Prinzips** ist, dass eine Klasse einer höheren Ebene beispielsweise ein Interface oder eine abstrakte Klasse als **Abstraktion** aggregiert. Diese Abstraktion wird von der höheren Klasse vorgegeben.



Die untergeordneten Klassen sollen nur von der entsprechenden Abstraktion abhängen. Damit erklärt sich auch der Begriff Dependency Inversion: Durch die Anwendung von Dependency Inversion werden die **Abhängigkeitsbeziehungen**, die normalerweise aus einem klassischen Entwurf resultieren und die, wie in Bild 1-11 zu sehen ist, von oben nach unten verlaufen, **invertiert** und verlaufen stattdessen von unten nach oben.

Das Dependency Inversion-Prinzip bedeutet:

- Eine Klasse einer höheren Ebene soll nicht von einer Klasse einer tieferen Ebene abhängig sein.
- Als Abstraktion soll die Klasse der höheren Ebene ein Interface oder eine abstrakte Klasse aggregieren. Diese Abstraktion soll wiederum von der Klasse der tieferen Ebene implementiert werden.
- Die Abstraktion soll von der höheren Ebene vorgegeben werden.
- Eine Abstraktion soll also nicht von einer Klasse einer tieferen Ebene abhängen.
- Hingegen soll eine Klasse einer tieferen Ebene von einer Abstraktion abhängen.



Das folgende Bild zeigt ein Beispiel dafür, dass eine Klasse einer höheren Ebene ein Interface als Abstraktion aggregiert und eine Klasse der tieferen Ebene diese Abstraktion implementiert:

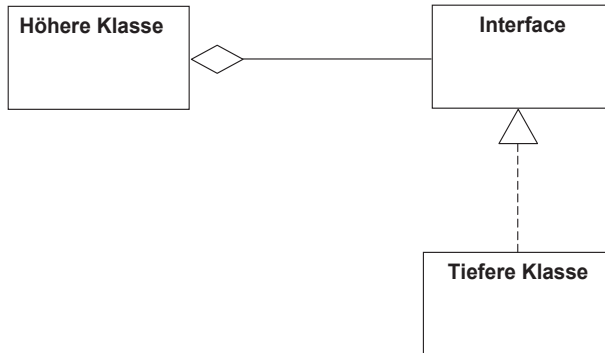


Bild 1-12 Architektur zur Erzeugung von Dependency Inversion in einer Hierarchie

Klassen einer höheren Ebene sollen also nicht durch die direkte Verwendung von Klassen einer tieferen Ebene von diesen abhängen. Dies ist im Bild 1-12 noch nicht ganz offensichtlich: Über die Aggregation ist die höhere Klasse zunächst vom Interface abhängig.

Nur wenn die höhere Klasse das Interface (die Abstraktion) selbst vorgibt, ist sie davon unabhängig, da sie quasi Eigentümerin des Interface ist. Dadurch wird das Interface auf die höhere Ebene gehoben und die verbleibenden Abhängigkeiten sind alle von unten nach oben gerichtet.



Durch diese **Umkehrung der Abhängigkeiten** ist nun auch **eine einfache Wiederverwendung von Elementen einer höheren Ebene** möglich, da sie nicht mehr von Elementen tieferer Ebenen abhängen. Außerdem können auch Klassen höherer Ebenen einfach unter Verwendung von Mock-Objekten<sup>13</sup> getestet werden, da die untere Schicht komplett ausgetauscht werden kann, ohne die höhere Schicht zu beeinflussen.

### 1.9.2 Inversion of Control

Bei Inversion of Control wird die **Kontrolle umgedreht**. Dabei wird ebenfalls eine Umkehrung der Abhängigkeiten erreicht. Anders als beim Dependency Inversion-Prinzip kann durch Inversion of Control eine **Umkehrung der Abhängigkeiten** auch zwischen zwei Objekten der gleichen Ebene erreicht werden.

<sup>13</sup> Mock-Objekte bilden echte Objekte beim Testen nach. Während ein Stub-Objekt allgemein nur zeigt, dass die Aufruf-Schnittstelle erfüllt ist und beim Testen irgendwelche Dummy-Daten liefert, erzeugt ein Mock-Objekt zu einem Testfall passende Daten, die aber keine echten Daten sind.

**Inversion of Control** kann zwischen Objekten derselben Hierarchiestufe eingesetzt werden, um Abhängigkeitsbeziehungen zu invertieren.



Bei der Umkehrung des Kontrollflusses gibt das ursprüngliche Programm die Steuerung des Kontrollflusses an ein anderes – meist wiederverwendbares – Modul ab. Damit kommt man vom Pollen zur ereignisgetriebenen Programmierung. Oft ruft dann ein mehrfach verwendbares Modul wie etwa ein Framework ein spezielles Modul wie beispielsweise ein Anwendungsprogramm auf.

Die Inversion des Kontrollflusses führt vom prozeduralen Programm mit Pollen zur ereignisgetriebenen Programmierung.



Ein Beispiel zur Anwendung von Inversion of Control ist beim **Beobachter-Muster** (siehe Kapitel 4.11) zu sehen. Wenn ohne den Einsatz dieses Musters ein Beobachter reagieren will, wenn sich der Zustand eines beobachtbaren Objektes ändert, dann muss er, wenn er die Kontrolle hat, d. h. ohne den Einsatz des Beobachter-Musters arbeitet, dieses Objekt per Polling beobachten. Durch Inversion of Control geht jedoch die Kontrolle an das zu beobachtende (beobachtbare) Objekt. Dieses weiß ja, wann sich sein Zustand ändert. Tritt dieser Fall ein, so kann das beobachtbare Objekt seine Beobachter benachrichtigen. Damit das beobachtbare Objekt vom Beobachter unabhängig ist, muss jeder Beobachter ein **Callback-Interface**, welches durch das zu beobachtende Objekt definiert wird, implementieren.

Die Architektur zur Erreichung der Inversion of Control ist genau dieselbe wie bei der **Dependency Inversion** in Hierarchien:

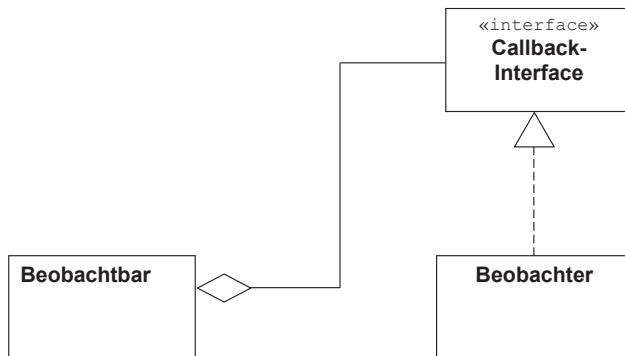


Bild 1-13 Architektur zur Verwirklichung von Inversion of Control

Allerdings stehen die Klassen `Beobachtbar` und `Beobachter` nicht in einer hierarchischen Beziehung zueinander sondern befinden sich **auf derselben Ebene**.

Wenn der Beobachtbare einen Beobachter direkt aufrufen würde, wäre er von ihm abhängig. Wenn der Beobachtbare aber das Callback-Interface selbst vorgibt und aggregiert, wird der Beobachtbare aber vom Beobachter unabhängig (**Dependency Inver-**

sion). Das Callback-Interface wird in der folgenden Beschreibung des Beobachter-Musters `IBeobachter` genannt.

Berücksichtigt man, dass ein Beobachter auf den Beobachtbaren über das von ihm selbst vorgegebene Interface `IBeobachtbar` zugreift, so kommt man zum Klassendiagramm des Beobachtermusters. Dies ist im Folgenden dargestellt:

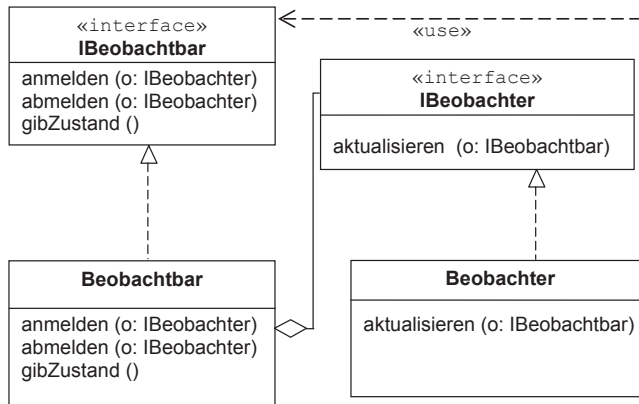


Bild 1-14 Klassendiagramm des Beobachter-Entwurfsmusters

Durch das Callback-Interface `IBeobachter` mit dem Methodenkopf `aktualisieren(o: IBeobachtbar)` wird der Beobachtbare vom Beobachter unabhängig. Hält der Beobachter das Callback-Interface nicht ein, ist er kein Beobachter mehr. Durch die Einführung des Callback-Interface wird ein Beobachtbarer mehrfach verwendbar und ist nicht vom speziellen Beobachter abhängig. Er ruft nun die speziellen Beobachter über das Callback-Interface auf.

Auf ähnliche Weise funktionieren Views als Listener und deren Controller. Die Kontrolle ist im Controller. Eine View muss nicht dauernd einen Controller nach Ereignissen pollen. Ein Controller schreibt einer View eine Callback-Schnittstelle für Ereignisse des Controllers vor. Mehrere Views können sich als Listener bei einem Controller anmelden, der dann Ereignisse generiert. Ein solcher Controller entspricht hierbei einem Beobachtbaren. Er hat die Kontrolle, generiert Ereignisse und ist wiederverwendbar.

Das Observer- bzw. Listener-Muster ist beispielsweise im Event Handling-Mechanismus von Swing umgesetzt.

**Inversion of Control** bedeutet in der Praxis meist, dass ein spezielles Modul wie eine Anwendung die Steuerung des Kontrollflusses an ein wiederverwendbares Modul abgibt. Damit das wiederverwendbare Modul das spezielle Modul nutzen kann, definiert das wiederverwendbare Modul ein **Callback-Interface**, das von dem speziellen Modul implementiert werden muss.



Inversion of Control oder Umkehrung des Kontrollflusses ist ein Paradigma, das auch von **Frameworks** umgesetzt wird, um eine Funktion einer Anwendung aufzurufen.

Dies ist ein Unterschied zwischen Frameworks und klassischen Funktionsbibliotheken: Statt dass die Anwendung den **Kontrollfluss** steuert und lediglich Funktionen einer Bibliothek aufruft, wird die Steuerung der Ausführung bestimmter **Programmenteile** an das Framework abgegeben. Dieses übernimmt dann die Kontrolle. Bei Vorliegen eines Ereignisses wird die entsprechende Funktion der Anwendung über eine vom Framework vorgegebene Callback-Schnittstelle vom Framework aufgerufen. Hierzu wird ein Objekt bei einem Framework registriert. Das Framework ruft dann eine Callback-Methode auf, die vom Framework mittels eines Interface vorgegeben ist. Das beim Framework registrierte Objekt implementiert dieses Interface und kann somit durch die implementierte Callback-Methode vom Framework benachrichtigt werden.

## 1.10 Verringerung der Abhängigkeiten bei der Erzeugung von Objekten

Auch wenn die Techniken zur Erzeugung von Objekten, die im Folgenden beschrieben werden wie z. B. Dependency Injection, in der Literatur häufig mit dem Begriff Inversion of Control verbunden werden, folgen wir diesem Gedanken nicht. Wir folgen hier Martin Fowler:

"Inversion of Control is too generic a term, and thus people find it confusing. As a result with a lot of discussion with various [Inversion of Control] advocates we settled on the name Dependency Injection." [fowioc]

**Der Begriff Inversion of Control wird in diesem Buch also nur im Zusammenhang mit der Umkehrung des Kontrollflusses verwendet.**

Das **Dependency Inversion-Prinzip** und **Inversion of Control** stellen letztendlich nur einen **ersten Schritt** dar, um Abhängigkeiten zu verringern. Würde ein Objekt einer Klasse ein Objekt einer anderen Klasse selbst erzeugen, dann würde eine weitere Abhängigkeit – nämlich eine sogenannte **«Create»-Abhängigkeit** – entstehen, wie das folgende Bild zeigt:

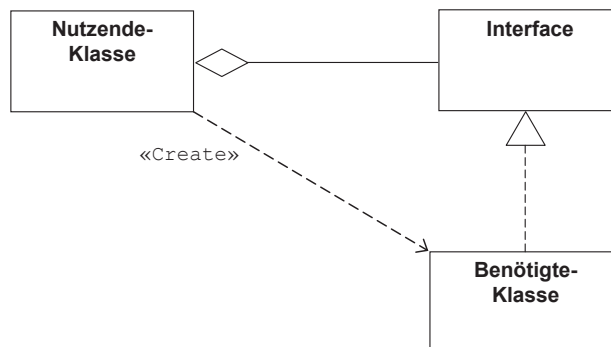


Bild 1-15 Neue Abhängigkeit durch die Erzeugung eines Objekts

Die Techniken, um solche **«Create»-Abhängigkeiten** zu vermeiden, werden im Folgenden beschrieben.

Zunächst wird die Ausgangssituation vor der Verringerung der Abhängigkeiten betrachtet: Wenn ein Objekt von einem anderen Objekt abhängig ist, besitzt es in der Regel eine Referenz auf das andere Objekt, um es für seine Zwecke bei Bedarf nutzen zu können. Allgemein gesprochen sind diese beiden Objekte miteinander verknüpft.

Um ein Objekt mit einem anderen Objekt zu verknüpfen und damit eine Abhängigkeitsbeziehung herzustellen, gibt es mehrere Möglichkeiten. Die offensichtlichste ist, dass – wie in Bild 1-15 dargestellt – ein Objekt dasjenige Objekt, das es benötigt, **selbst** erzeugt. Damit es dies kann, muss das erzeugende Objekt Informationen über das andere Objekt statisch im Quellcode halten. Dadurch wird es bereits von dem anderen Objekt abhängig.

Durch Verwendung einer **Fabrikmethode** (siehe Kapitel 4.18) kann bei der Erzeugung eines benötigten Objektes zwar etwas Flexibilität gewonnen werden. Aber auch in einer Fabrikmethode wird ein Objekt erzeugt und die Klasse des zu erzeugenden Objekts statisch im Quellcode der entsprechenden Unterklasse festgelegt.

Die **Strategie für die Verringerung der Abhängigkeit** eines nutzenden Objekts von einem zu erzeugenden Objekt ist:

Bei der Erzeugung von Objekten wird die Kontrolle darüber, welches Objekt wann erzeugt wird, von der nutzenden Klasse an eine dafür eigens vorgesehene Instanz abgegeben.



Hierfür werden die beiden folgenden Techniken verwendet:

- **Dependency Look-Up**

Bei einem Dependency Look-Up sucht ein Objekt, das ein anderes Objekt braucht, nach diesem anderen Objekt z. B. in einem Register, um die Verknüpfung mit diesem herzustellen. Damit braucht es nur noch den Namen des anderen Objekts zu kennen und ist von dem anderen Objekt weitgehend entkoppelt. Aber natürlich hängt es dann vom Register ab.

- **Dependency Injection**<sup>14</sup>

Hierbei wird die Erzeugung von Objekten und die Zuordnung von Abhängigkeiten zwischen Objekten an eine dafür vorgesehene Instanz delegiert. Damit sind die Objekte untereinander selbst nicht abhängig, aber die erzeugende Instanz ist von allen beteiligten Objekten abhängig. Da die erzeugende Instanz alle Objekte und Verbindungen erzeugt, ist ein Objekt vom Injektor unabhängig.

### 1.10.1 Dependency Look-Up

Bei einem Dependency Look-Up sucht ein Objekt, das ein anderes Objekt braucht, nach diesem anderen Objekt über dessen Namen, um die Verknüpfung mit diesem herzustellen. Die einzige Abhängigkeit zwischen den beiden Objekten ist, dass das

---

<sup>14</sup> Der Begriff Dependency Injection wurde zum ersten Mal von M. Fowler in [fowioc] benutzt, um diese Technik von der Inversion of Control abzugrenzen.



suchende Objekt den Namen des gesuchten Objekts kennen muss, allerdings aber nicht mehr die Details über das gesuchte Objekt wie beispielsweise dessen Klasse.

Dependency Look-Up bedeutet, dass ein Objekt seine **Verknüpfung** mit einem anderen Objekt **zur Laufzeit** herstellt, indem es nach diesem anderen Objekt über den Objektnamen zur Laufzeit sucht.



Das gesuchte Objekt kann selber wieder verknüpft sein. Ob diese weiteren Verknüpfungen bereits hergestellt sind oder nicht, hängt von der Anwendung ab. Prinzipiell kann die Suche rekursiv fortgesetzt werden, bis alle Verknüpfungen aufgelöst sind. Dadurch können die Verknüpfungen sehr flexibel hergestellt werden, was beispielsweise bei Objekten von Vorteil ist, die von Frameworks erzeugt werden, aber noch Verknüpfungen zu den Objekten der Anwendung benötigen.

Zur Unterstützung eines Dependency Look-Up gibt es mehrere Möglichkeiten:

- Die einfachste Möglichkeit hierzu ist, die benötigten Objekte in einer **zentralen Klasse** zu halten, auf die alle anfragenden Objekte Zugriff haben. Wird ein Objekt oder eine Beziehung über den Objektnamen in dieser zentralen Klasse gesucht, erhält das suchende Objekt alles, was es braucht. Es braucht also de facto nur den Namen eines anderen Objekts zu kennen.

Eine weitere Möglichkeit besteht im Einsatz einer **Registratur** (engl. **registry**). Die erzeugten Objekte werden registriert, also in eine Registratur eingetragen. Wird ein Objekt für eine Verknüpfung benötigt, wird bei der Registratur über den Objektnamen nach einem entsprechenden Objekt gesucht. Die Registratur liefert eine Referenz auf das gesuchte Objekt zurück, womit dann eine Verknüpfung hergestellt werden kann.

- Eine häufig eingesetzte Lösung ist es, alle in einer Anwendung benötigten Objekte in einem **Container**-Objekt als Behälter zu halten und nur die Referenz auf dieses Container-Objekt allen anfragenden Objekten zur Verfügung zu stellen. Im Programmcode können dann über dieses Container-Objekt die benötigten Objekte abgefragt werden<sup>15</sup>. Je nach Implementierung des Containers kann entweder ein Programmierer dafür zuständig sein, das Container-Objekt mit Objekten zu füllen und ggf. Verknüpfungen zwischen ihnen herzustellen, oder das Container-Objekt ist – beispielsweise mit Hilfe von Properties-Dateien (siehe Kapitel 1.10.2) – selbst in der Lage, die benötigten Objekte und ihre Verknüpfungen zu erzeugen. Die Objekte erhalten dabei ihre Verknüpfungen zu den zugehörigen Objekten von dem Container-Objekt als Dienstleistung. Damit dieser Dienst genutzt werden kann, muss er den Objekten bekannt sein und damit sind die **Objekte von diesem Dienst abhängig**. Allerdings sind sie untereinander unabhängig, was ja das Ziel dieses Ansatzes ist.

<sup>15</sup> Dies kann mehr oder weniger als eine Verallgemeinerung des Musters **Objektpool** (siehe Kapitel 4.21) gesehen werden, da die in einem Container-Objekt gespeicherten Objekte gänzlich unterschiedliche Typen haben können.

### Beispiel für ein Dependency Look-Up mit Hilfe einer Registry

Im Folgenden wird die Technik Dependency Look-Up anhand eines Beispiels verdeutlicht. Als Beispiel wird ein Plotter betrachtet, dessen Aufgabe es ist, eine Reihe von Daten, die von einer Datenquelle erzeugt werden, aufzubereiten und grafisch darzustellen. Die Klasse `Plotter` aggregiert das Interface `IDatenquelle`, welches auf verschiedene Arten realisiert werden kann. In diesem Beispiel wird das Interface `IDatenquelle` von der Klasse `DatenquelleImpl` realisiert. Alle vorhandenen Realisierungen einer Datenquelle werden in einer Registratur verwaltet. Die Klasse `Plotter` verwendet die Registratur, um eine konkrete Datenquelle auszuwählen, damit die darin enthaltene Datenreihe dann dargestellt werden kann. Hier das Interface `IDatenquelle`:

```
// Datei: IDatenquelle.java
import java.awt.Point;
import java.util.ArrayList;
public interface IDatenquelle
{
    public ArrayList<Point> getDatareihe();
}
```

In der Klasse `DatenquelleImpl` wird eine konkrete Datenreihe aus Zufallszahlen generiert:

```
// Datei: DatenquelleImpl.java
import java.awt.Point;
import java.util.ArrayList;

public class DatenquelleImpl implements IDatenquelle
{
    public ArrayList<Point> getDatareihe()
    {
        ArrayList<Point> datenreihe = new ArrayList<Point>();
        int anzPunkte = (int) (Math.random()*30)+2;
        int startPunkt = -1*(int) (Math.random()*30);

        for (int x = startPunkt; x < anzPunkte; x++)
        {
            datenreihe.add(new Point(x, (int) (Math.random()*20)-
                (int) (Math.random()*20)));
        }
        return datenreihe;
    }
}
```

Die Klasse `Registratur` dient in diesem Beispiel der Verwaltung aller existierenden Datenquellen. Bei der Klasse `Registratur` wurde das **Singleton-Muster** (siehe Kapitel 4.20) eingesetzt, da von dieser Klasse nur ein einziges Objekt existieren darf. Die Klasse `Registratur` kann nicht nur Datenquellen verwalten, sondern es können bei ihr beliebige Objekte registriert werden. Nachfolgend der Quelltext der Klasse `Registratur`:

```
// Datei: Registratur.java
// Die Klasse Registratur kann beliebig viele Objekte verwalten.
// Jedes Objekt wird durch einen String-Schlüssel registriert und
// kann durch diesen angefordert werden.

import java.util.HashMap;

public class Registratur
{
    private Registratur({});

    private HashMap<String, Object> objekt =
        new HashMap<String, Object>();
    private static Registratur registratur = null;
    public static Registratur getRegistratur()
    {
        if(registratur == null)
            registratur = new Registratur();
        return registratur;
    }

    public Object getObjekt(String objektBezeichnung)
    {
        return objekt.get(objektBezeichnung);
    }

    public void registriereObjekt(String bezeichnung, Object objekt)
    {
        this.objekt.put(bezeichnung, objekt);
    }
}
```

In der folgenden Klasse `Plotter` wird die grafische Ausgabe einer Datenreihe generiert. Der Programmcode zur Erzeugung der grafischen Ausgabe wird hier im Buch aus Gründen der Übersichtlichkeit nicht abgedruckt. Ausgelassene Stellen sind mit ... markiert. Auf dem begleitenden Webauftritt ist das Beispiel komplett enthalten. Hier soll im Wesentlichen gezeigt werden, wie ein Objekt der Klasse `Plotter` mit Hilfe einer `Registratur` eine Datenquelle finden und dann die von dieser Datenquelle erzeugte Datenreihe ausgeben kann. Das ist in der Methode `plot()` der Klasse `Plotter` zu sehen:

```
// Datei: Plotter.java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Point;
import java.util.ArrayList;

import javax.swing.JFrame;
import javax.swing.JPanel;
```

```

public class Plotter extends JPanel
{
    // Die Klasse Plotter besitzt eine Referenz auf eine Datenquelle
    private IDatenquelle datenquelle;

    // Referenz auf die zu plottenden Punkte einer Datenreihe der
    // Datenquelle
    ArrayList<Point> datenreihe = null;
    . . .
    public void plot()
    {
        // Datenquelle von der Registratur anfordern
        Registratur registratur = Registratur.getRegistratur();
        datenquelle = (IDatenquelle)
            registratur.getObjekt("Datenquelle");

        // Wenn keine Datenquelle in der Registratur vorhanden ist,
        // beende den Plotvorgang
        if (datenquelle == null) return;

        // Fordere die Datenreihe des Plotters an
        datenreihe = datenquelle.getDatenreihe();

        // Wenn weniger als zwei Punkte in Datenreihe, verlasse Funktion
        if (datenreihe == null || datenreihe.size() <= 1)
            return;

        // Durch den Aufruf der Methode setVisible() wird veranlasst,
        // dass die Methode paintComponent() aufgerufen wird und
        // damit das Fenster gezeichnet wird.
        frame.setVisible(true);
        . . .
    }

    protected void paintComponent(Graphics g)
    {
        . . .
    }
}

```

Die `main()`-Methode der Klasse `TestPlotter` erzeugt eine Datenquelle und registriert das erzeugte Objekt unter dem Namen "Datenquelle" in der Registratur. Der Plotter, der anschließend gestartet wird, kann über die Registratur mit Hilfe dieser Bezeichnung auf die Datenquelle zugreifen und dieses Objekt benutzen. Hier der Quellcode der Klasse `TestPlotter`:

```

// Datei: TestPlotter.java
public class TestPlotter
{
    static public void main(String[] args)
    {
        // Bekomme Referenz auf die Registratur
        Registratur registry = Registratur.getRegistratur();
    }
}

```

```
// Registriere eine Datenquelle als Objekt
registry.registriereObjekt(
    "Datenquelle", new DatenquelleImpl());

// plotten ...
Plotter plotter = new Plotter();
plotter.plot();
}
```

Das folgende Bild zeigt eine vom Plotter gezeichnete Datenreihe:

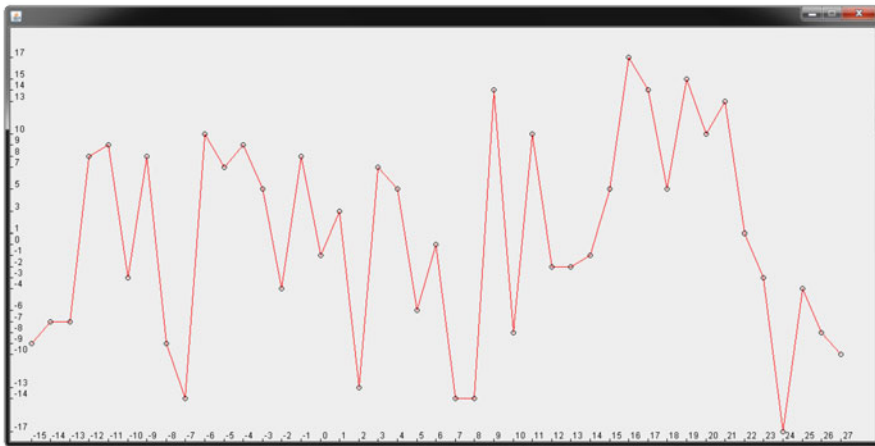


Bild 1-16 Geplottete Datenreihe

### 1.10.2 Dependency Injection

Der Ansatzpunkt von Dependency Injection ist der gleiche wie beim Dependency Look-Up: Direkte Abhängigkeitsbeziehungen zwischen miteinander verknüpften Objekten bzw. Klassen sollen vermieden werden.

Bei Dependency Injection werden Abhängigkeiten **von außen** übergeben (injiziert). Ein Objekt sucht nicht aktiv wie beim Dependency Look-Up in einer Registry, sondern bleibt passiv, was die Verknüpfung betrifft.



Die Objekte erhalten also Referenzen auf Objekte, die sie benötigen, von außen – von einer eigenen Instanz, dem Injektor. Ein Injektor erzeugt alle Objekte und Verbindungen. Der Programmcode eines Objekts bzw. seiner Klasse ist daher von den anderen Klassen entkoppelt. Es bestehen zur Kompilierzeit keine Abhängigkeiten von den anderen Klassen. Erst zur Laufzeit werden die Verbindungen zwischen den einzelnen Objekten durch den Injektor hergestellt. Auf die Realisierungsformen von Dependency Injection sowie auf die Implementierungsmöglichkeiten des Injektors wird im Verlauf des Kapitels noch detailliert eingegangen.

Dependency Injection bedeutet, dass eine **Verknüpfung** zwischen Objekten **zur Laufzeit** von einer eigenen Instanz (**Injektor**) hergestellt wird und nicht zur Kompilierzeit.



### Vorteile von Dependency Injection

Durch Dependency Injection wird eine Klasse unabhängig von anderen Klassen und braucht auch kein Wissen darüber, wie Objekte solcher Klassen zu erzeugen sind. Wenn Klassen keine Abhängigkeiten zu anderen Klassen haben, sind sie als isolierte Elemente auch am einfachsten zu testen. Man kann eine Klasse also isolieren, wenn ihre Objekte keine Objekte von anderen Klassen zu erzeugen brauchen, sondern ihnen Referenzen auf die benötigten Objekte zur Laufzeit mittels Dependency Injection übergeben werden.

### Realisierungsformen der Dependency Injection

Üblicherweise erzeugt in objektorientierten Systemen ein Objekt **die von ihm benötigten** Objekte selbst. Dadurch besitzt es auch eine Referenz auf das jeweilige erzeugte Objekt. Bei Dependency Injection überträgt man die Aufgabe für das Erzeugen und für das Verknüpfen von Objekten an eine eigene Instanz wie etwa ein Framework.

Für Dependency Injection gibt es nach [fowioc] folgende drei Möglichkeiten:

- Es können Abhängigkeiten zu anderen Objekten direkt bei der Erzeugung über einen Konstruktor mit Parametern erzeugt werden (**Constructor Injection**).
- Es ist aber auch denkbar, dass diese erst später vom Injektor über set-Methoden festgelegt werden (**Setter Injection**).
- Die dritte Technik ist, Interfaces zu definieren und für das Injizieren zu verwenden (**Interface Injection**).



Während die ersten beiden Möglichkeiten – also Constructor Injection und Setter Injection – offensichtlich sind, bedarf die Vorgehensweise mittels Interface Injection einer ausführlichen Betrachtung.

Bei Interface Injection wird zusätzlich zu der Abstraktion für die zu injizierenden Objekte (**IAbstraktion**) noch ein weiteres Interface (**IInjectAbstraktion**) für die Injektion dieser Objekte vorgegeben, siehe Bild 1-17. Die Klasse eines Objekts, in das eine Verbindung injiziert werden soll, muss das Interface **IInjectAbstraktion** implementieren. Das Interface **IInjectAbstraktion** definiert eine Methode **inject()**, die von einem Injektor zur Laufzeit aufgerufen werden kann, um die Verbindung von einem Objekt der implementierenden Klasse – also der Klasse **Abhängige-Klasse** – zu einem Objekt einer konkreten zu injizierenden Klasse herzustellen. Das folgende Bild zeigt die bei Interface Injection beteiligten Klassen und Interfaces, allerdings der Einfachheit halber ohne einen Injektor und nur mit einer einzigen konkreten zu injizierenden Klasse:

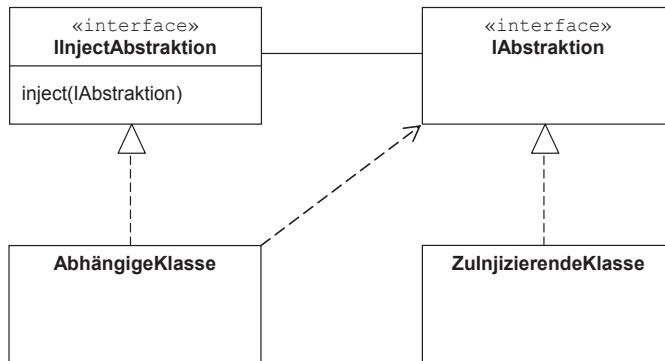


Bild 1-17 Klassendiagramm für Interface Injection

Die Assoziation zwischen den beiden Interfaces im Bild 1-17 deutet an, dass die beiden Interfaces logisch zusammengehören. M. Fowler empfiehlt in [fowioc], dass beide auch zusammen entworfen werden.

Ein Vorteil von Interface Injection ist, dass eine Klasse durch die Implementierung des entsprechenden Interface explizit kund tut, dass sie zur Injektion bereit ist. Ein weiterer Vorteil ist, dass ein solches Interface von mehreren Klassen implementiert werden kann. Damit wird der Injektor nicht von jeder einzelnen Klasse abhängig, sondern der Injektor ist nur vom Interface abhängig. Der Nachteil ist, dass **jede Klasse**, die das Interface implementiert, **von dem Interface abhängig** wird.

### Implementierungsmöglichkeiten für den Injektor

Häufig übernimmt ein Framework die Rolle des Injektors, dann muss sich eine Anwendung nicht weiter um die Realisierung des Injektors kümmern. Ein Injektor kann aber auch selbst in einer Anwendung implementiert werden. Das Wissen, welche Verknüpfungen wie zu setzen sind, kann dann im Programmcode des Injektors niedergeschrieben sein. Der Injektor selbst hat dann viele Abhängigkeiten, denn er muss ja verschiedene Objekte erzeugen und Abhängigkeiten zu diesen Objekten in anderen Objekten injizieren.

Abhängigkeiten werden daher oft in einer **Konfigurationsdatei** festgehalten. Eine Konfigurationsdatei beschreibt die durchzuführenden Verknüpfungen textuell über Namen und Abbildungen. Konfigurationsdateien sind heute meist XML-basiert. Die Realisierung von Dependency Injection mittels Konfigurationsdateien bietet mehr Flexibilität, da die Datei leicht geändert werden kann, ohne dass am Programmcode Folgeänderungen nötig werden.

Bei Verwendung einer Konfigurationsdatei werden die Namen von Klassen bzw. Objekten und Beziehungen dem Framework erst über den Inhalt der Konfigurationsdatei mitgeteilt. Die Realisierung von Dependency Injection über Annotationen ist ein anderer gängiger Weg.



Im Programmtext sind keine Abhängigkeiten mehr zu sehen, wenn die Abhängigkeiten in die Konfigurationsdatei ausgelagert wurden. Die Verständlichkeit des Programmtextes nimmt dadurch aber ab, da der Inhalt einer weiteren Datei betrachtet werden muss.

Im Folgenden soll kurz skizziert werden, wie ein auf einer Konfigurationsdatei basierender Injektor in Java auch selbst programmiert werden kann.

Dependency Injection lässt sich in Java beispielsweise mit Hilfe von sogenannten **Properties-Dateien**<sup>16</sup> realisieren. Darin wird die Zuordnung zwischen Referenzen und Objekten (also die Verknüpfung) festgelegt. Eine Klasse, deren Aufgabe es ist, die Abhängigkeiten zu injizieren, kann eine Properties-Datei lesen und auf Grund der darin enthaltenen Informationen mit Hilfe der **Reflection-API**<sup>17</sup> von Java die Objekte erzeugen und die Referenzen entsprechend – beispielsweise über set-Methoden – setzen. Ein vollständiges Beispiel hierfür ist auf dem begleitenden Webaufttritt zu finden.

Die Abhängigkeit zwischen Objekten wird somit aus dem Programmcode in die Properties-Datei verlagert. Dadurch brauchen die Abhängigkeiten erst beim Starten eines Programmes vorzuliegen bzw. können sogar noch zur Laufzeit verändert werden.

## 1.11 Zusammenfassung

Objektorientierte Prinzipien für den Entwurf führen dazu, einen Entwurf

- verständlich,
- korrekt und
- erweiterbar

zu machen.

Dabei betreffen diese Prinzipien sowohl die Konstruktion einzelner Klassen, als auch ihre Zusammenarbeit (siehe Kapitel 1.1).

Kapselung, Abstraktion und Information Hiding (siehe Kapitel 1.2) führen zu schmalen Schnittstellen zwischen den einzelnen Objekten.

Separation of Concerns im engeren Sinne und das Single Responsibility-Prinzip führen dazu, dass jede Klasse eine einzige Verantwortung trägt (siehe Kapitel 1.3).

Das Interface Segregation-Prinzip (siehe Kapitel 1.4) fordert, dass Klassen nicht von Schnittstellen abhängen sollen, die sie nicht brauchen. Änderungen an nicht benötigten Schnittstellen schlagen somit nicht auf eine Klasse durch.

---

<sup>16</sup> Eine Properties-Datei in Java ist eine Textdatei, die zur Konfiguration einer Anwendung eingesetzt werden kann. Eine Eigenschaft (engl. property) wird darin in Form eines Textes, der unter einem bestimmten Namen abgelegt ist, beschrieben.

<sup>17</sup> Die Reflection-API von Java ermöglicht es, zur Laufzeit auf Informationen über Klassen und Objekte – sogenannte Metadaten – zuzugreifen und diese auch zu modifizieren.



Ein guter Entwurf hat innerhalb eines Teilsystems eine starke Kohäsion und zwischen den Teilsystemen eine schwache Kopplung (loose coupling – siehe Kapitel 1.5).

Nach dem liskovschen Substitutionsprinzip (siehe Kapitel 1.6) kann in einem polymorphen Programm stets eine Referenz auf ein Objekt einer abgeleiteten Klasse an die Stelle einer Referenz auf ein Objekt der entsprechenden Basisklasse treten, wenn die abgeleitete Klasse die Verträge (siehe Kapitel 1.7) der Basisklasse einhält.

Nach dem Open-Closed-Prinzip von Bertrand Meyer (siehe Kapitel 1.8) müssen Module offen und geschlossen sein.

Durch die Geschlossenheit des Quellcodes bzw. des lauffähigen Codes gegenüber Veränderungen erreicht man Robustheit und Wiederverwendbarkeit der vorhandenen Programme in einer Bibliothek. Durch die Offenheit des Quellcodes bzw. des lauffähigen Codes gewinnt man eine Wiederverwendbarkeit des vorhandenen Quellcodes als Teil von neuen Modulen und damit eine Erweiterbarkeit des Vorhandenen.

Durch die Geschlossenheit von Spezifikationen gegenüber Veränderungen wird ihr Inhalt stabil und selbst wieder verwendbar. Durch die Offenheit von Spezifikationen gewinnt man ihre Wiederverwendbarkeit als Teil eines neuen Moduls.

Das Dependency Inversion-Prinzip (siehe Kapitel 1.9.1) hat zum Ziel, Abhängigkeiten zu invertieren und betrachtet dafür die Schnittstellenproblematik in einer Hierarchie. Durch die Einführung von Abstraktionen (Interfaces oder abstrakte Klassen) an geeigneter Stelle können die Abhängigkeiten in einer Hierarchie umgekehrt werden.

Inversion of Control (siehe Kapitel 1.9.2) erzeugt wie das Dependency Inversion-Prinzip eine Umkehrung der Abhängigkeit, aber nicht in einer Hierarchie, sondern auf derselben Ebene.

Dependency Look-Up und Dependency Injection sind Techniken, um Abhängigkeitsbeziehungen zwischen Objekten zur Laufzeit mit Hilfe einer speziell dafür vorgesehenen Instanz herzustellen. Durch den Einsatz von Dependency Look-Up (siehe Kapitel 1.10.1) sucht ein Objekt andere Objekte, die es benötigt, beispielsweise in einer Registratur. Bei Dependency Injection wird die Erzeugung von Objekten und die Zuordnung von Abhängigkeiten an eine dritte, neutrale Partei, den sogenannten Injektor, delegiert (siehe Kapitel 1.10.2).

## 1.12 Aufgaben

### Aufgaben 1.12.1: Objektorientierter Entwurf einzelner Klassen

- 1.12.1.1 Von wem stammen jeweils die Prinzipien Separation of Concerns und das Single Responsibility-Prinzip?
- 1.12.1.2 Sind Separation of Concerns und das Single Responsibility-Prinzip im Hinblick auf Klassen äquivalent?
- 1.12.1.3 Was bedeutet das Interface Segregation-Prinzip?

### Aufgaben 1.12.2: Objektorientierter Entwurf miteinander kooperierender Klassen

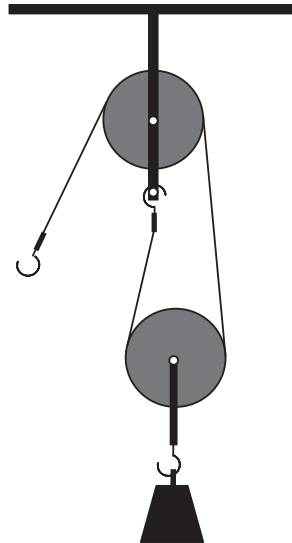
- 1.12.2.1 Erklären Sie strong cohesion und loose coupling.
- 1.12.2.2 Erklären Sie das liskovsche Substitutionsprinzip.
- 1.12.2.3 Erklären Sie das Open-Closed-Prinzip.
- 1.12.2.4 Erklären Sie das Prinzip der Dependency Inversion in einer Hierarchie.
- 1.12.2.5 Erklären Sie die Inversion der Kontrolle des Flusses.
- 1.12.2.6 Was ist Dependency Look-up? Was ist Dependency Injection?
- 1.12.2.7 Was ist Constructor Injection? Was ist Setter Injection? Was ist Interface Injection?

### Aufgaben 1.12.3: Zusicherungen

- 1.12.3.1 Was ist eine Zusicherung?
- 1.12.3.2 Betreffen Vor- und Nachbedingungen eine Methode oder eine Klasse?
- 1.12.3.3 Wer hat die Pflicht bei einer Vorbedingung, wer hat den Nutzen?
- 1.12.3.4 Wer hat die Pflicht bei einer Nachbedingung, wer hat den Nutzen?
- 1.12.3.5 Betreffen Invarianten eine Methode oder eine Klasse?
- 1.12.3.6 Kann man bei einer Ableitung Vorbedingungen verschärfen?
- 1.12.3.7 Kann man bei einer Ableitung Nachbedingungen aufweichen?

# *Kapitel 2*

## Softwarearchitekturen



- 2.1 Der Begriff einer Softwarearchitektur
- 2.2 Qualitäten einer Softwarearchitektur
- 2.3 Referenzarchitekturen, Architektur- und Entwurfsmuster
- 2.4 Aufgaben und Sichten bei der Konzeption einer Softwarearchitektur
- 2.5 Die Bedeutung eines Softwarearchitekten für ein Projekt
- 2.6 Zusammenfassung
- 2.7 Aufgaben

## 2 Softwarearchitekturen

Entwurfs- und Architekturmuster betreffen Softwarearchitekturen. Das Finden der passenden Muster ist ein wichtiger Schritt bei der Konzeption der Architektur eines Systems. Muster kommen beim Entwurf immer dann ins Spiel, wenn es standardisierte Kombinationen von Komponenten eines Systems geben soll, die durch ihre Zusammenarbeit eine bestimmte Problemstellung in

- verständlicher,
- einfacher und in
- ausbaufähiger

Weise lösen sollen.

Eine Architektur modelliert Funktionen in Struktur und Verhalten und ist wesentlich verantwortlich für die nicht funktionalen Qualitäten eines Systems, die nicht auf einzelne Funktionen abgebildet werden können.



Architektur und Funktionen sind zweierlei, auch wenn die Funktionen im Korsett einer Architektur "laufen".



Genau dieselbe Funktionalität kann man auch mit unterschiedlichen Architekturen realisieren. Jede Architektur hat aber andere Eigenschaften.

Muster betreffen zwar einen wichtigen Aspekt einer Systemarchitektur, die Wahl der passenden Muster betrifft aber nur eine einzige Facette der bei der Konzeption einer Architektur durchzuführenden Aktivitäten.

Daher soll in Kapitel 2.4 eine Einführung gegeben werden, die die Breite der Aufgaben beim Finden der passenden Softwarearchitektur für ein System widerspiegelt.<sup>18</sup>

Die Softwarearchitektur eines Systems wird als vorläufiges Endergebnis in dem Entwicklungsschritt Systementwurf konzipiert. Das Ergebnis des Systementwurfs oder des Designs<sup>19</sup> soll die Architektur des Systems sein. Mit dem Ausdenken der Architektur wird allerdings fast immer schon früher begonnen.



Der Entwurf mit seinem Ziel, die Architektur festzulegen, schlägt die Brücke zwischen der Modellierung des Fachkonzepts im Rahmen der Systemanalyse und der Implementierung der Programme.

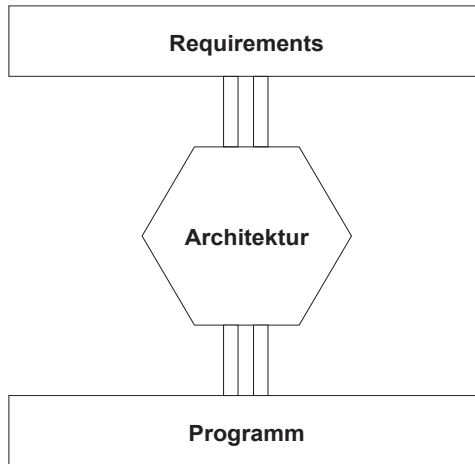
<sup>18</sup> Ausführliche Betrachtungen dieses Themas sind beispielsweise in [Sta02] und [Sta11] zu finden.

<sup>19</sup> In diesem Buch werden die Begriffe Entwurf und Design äquivalent verwendet.

Letztendlich verknüpft eine Architektur die Anforderungen an das zu realisierende System mit der technischen Lösung.



Dies soll das folgende Bild symbolisieren:



*Bild 2-1 Architektur als Mittler zwischen Requirements und Programm*

Eine Architektur stellt sicher, dass

- funktionale und
- nicht funktionale Anforderungen, d. h.
  - Anforderungen an Qualitäten, die nicht auf einzelne Funktionen abgebildet werden können, sowie
  - Einschränkungen des Lösungsraums

erfüllt werden können. Ein Beispiel für eine Anforderung an Qualitäten, die nicht auf einzelne Funktionen abgebildet werden können, ist die Performance. Ein Beispiel für eine Einschränkung des Lösungsraums ist das Vorschreiben des zu verwendenden Datenbankmanagementsystems.

Eine **Architektur** stellt eine "Blaupause" dar, die eine Abstraktion der Implementierung ist. Sie dient damit zur **Reduktion der Komplexität** und ist eine **Vorgabe für die Struktur und die Abläufe der Implementierung**.



Da die Architektur eine Abstraktion der Programme darstellt, gewinnt man ein Verständnis für eine Implementierung am schnellsten dann, wenn man zuerst die Architektur und dann erst die Programme betrachtet.

Die Architektur eines zu entwickelnden Systems beeinflusst wesentlich Implementierung, Test, Integration und Wartung. Während der Realisierung ist die Architektur die Basis für die Überwachung des Projektfortschritts.

Eine Architektur kann durch Prototypen verifiziert werden (siehe Kapitel 2.4.4).

Einerseits ist eine Softwarearchitektur eine direkte Vorgabe auf höherer Ebene für die zu implementierenden Programme, andererseits können aber auch die bei der Implementierung gemachten Erfahrungen wiederum zu einer Verbesserung der Architektur führen. Damit ist ersichtlich, dass das **Finden der passenden Architektur oft iterativ** abläuft.



Kapitel 2.1 befasst sich mit dem Begriff einer Softwarearchitektur. Kapitel 2.2 beleuchtet die Qualitätseigenschaften einer Softwarearchitektur und Kapitel 2.3 stellt Referenzarchitekturen, Architektur- und Entwurfsmuster einander gegenüber. Kapitel 2.4 befasst sich mit den durchzuführenden Aufgaben und Sichten bei der Konzeption einer Softwarearchitektur und der Erstellung von Prototypen. Zum Schluss erläutert Kapitel 2.5 die Bedeutung eines Softwarearchitekten für ein Projekt.

## 2.1 Der Begriff einer Softwarearchitektur

Was eine Softwarearchitektur ist, wird in verschiedenen Definitionen nicht einheitlich wiedergegeben. Die folgende Definition erscheint den Autoren am klarsten:

Die **Architektur** eines Systems umfasst:

- die statische **Zerlegung** des Systems in seine physischen Bestandteile (Komponenten) und bei verteilten Systemen die Verteilung dieser Komponenten auf die einzelnen Rechner (Deployment),
- die **Beschreibung des dynamischen Zusammenwirkens** aller Komponenten und
- die Beschreibung der **Strategie für die Architektur**, d. h. wie die Architektur in Statik und Dynamik funktionieren soll<sup>20</sup>,

mit dem Ziel, alle nach außen geforderten Leistungen des Systems erzeugen zu können.



Das Wort Komponenten bedeutet hier nicht Komponenten im Sinne einer bestimmten Komponententechnologie, sondern Komponenten im Sinne der Zerlegung eines Systems in kleinere physische Einheiten und ihre Wechselwirkungen.

<sup>20</sup> Mit dieser Beschreibung können auch Projektneulinge die Prinzipien der Architektur verstehen.

Die Zerlegung umfasst die Struktur bzw. Statik, das Zusammenwirken hingegen das Verhalten bzw. die Dynamik.



Das Verhalten eines Systems baut auf seiner Struktur auf und lässt sich von dieser nicht trennen.

Diese Definition ist ähnlich der von der IEEE in der "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems" festgelegten Definition:

"Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution" [IEEE 1471].

Beim Entwurf muss also geklärt werden, welche Komponenten benötigt werden und wie sie mit anderen Komponenten und der Außenwelt über Schnittstellen zusammenarbeiten. Zu jeder Schnittstelle einer Komponente gehört ein Vertrag mit ihrem Kunden.



Die Komponenten einer Architektur sollen eine **große innere Kohäsion** haben, aber untereinander **wechselseitig möglichst schwach gekoppelt** sein, damit bei einer Änderung nicht das ganze System geändert werden muss. Dabei ist die innere Struktur der Komponenten zu verbergen.



Kollaborationen der Komponenten erlauben es, ein kollektives Verhalten zu erzeugen, das über das Verhalten einer einzelnen Komponente hinausgeht.

## 2.2 Qualitäten einer Softwarearchitektur

Eine Architektur bestimmt die angebotene Nutzerfunktionalität.

Weiterhin muss eine Architektur bestimmten Anforderungen an die Qualität der Architektur genügen.

Zu den nicht funktionalen **Qualitäten** einer Architektur gehören:

- **Administrierbarkeit**

Bei verteilten Systemen sollte der Administrator eine Single System View<sup>21</sup> haben.

<sup>21</sup> Dies bedeutet, dass der Systemverwalter zum einen von seinem Rechner aus das ganze System betrachten kann, als wäre es ein einziger Rechner. Überdies muss der Systemadministrator in der Lage sein, von seinem Arbeitsplatz aus das ganze System zu managen.

- **Änderbarkeit, Ausbaufähigkeit, Erweiterbarkeit**

Mögliche Erweiterungen eines Systems werden auf Basis der Architektur des vorliegenden Systems bewertet und durchgeführt. Die Flexibilität eines Systems kann durch den Einsatz flexibler Muster gefördert werden. Performance und Flexibilität sind aber prinzipiell zwei verschiedene Zielrichtungen, da die Flexibilität durch das Einfügen von Zwischenschichten erhöht, die Performance aber durch den höheren Rechenaufwand verschlechtert wird.

- **Bedienbarkeit**

Sowohl der ungeübte als auch der geübte Nutzer sollte gleichermaßen mit dem System interagieren können.

- **Einfachheit**

Das ursprüngliche System muss einfach sein. Bei Erweiterungen kann dann die Architektur von alleine noch komplexer werden.

- **Migrationsfähigkeit**

Die Software sollte auf andere Rechnertypen portiert werden können.

- **Performance**

Performance-kritische Anteile kann man nicht verteilen, sondern muss sie zentral auf einem einzigen Rechner halten, wenn diese Anteile über Shared Memory gekoppelt sind. Software-Cluster<sup>22</sup> aus Betriebssystem-Prozessen wird man zur Performance-Steigerung auf verschiedene Rechner verteilen.

- **Skalierbarkeit**

Ein System muss es von den Antwortzeiten her verkraften, dass die Zahl seiner Nutzer stark ansteigt. Es muss skalierbar sein, d. h. es darf trotz steigender Benutzerzahlen keine wesentliche Verschlechterung seiner Performance erfahren.

- **Stabilität**

Softwarearchitekturen sollen einerseits stabil sein. Andererseits sollen sie flexibel sein und Änderungen der Kundenwünsche in einem gewissen Rahmen erlauben.

- **Testbarkeit**

Eine Architektur, die schwer zu testen ist, ist zu verwerfen (**Design to Test**).

- **Unabhängigkeit der Komponenten**

Eine weitestmögliche Unabhängigkeit der Komponenten erleichtert Änderungen des Systems.

- **Verfügbarkeit**

Eine hohe Verfügbarkeit kann den Einsatz fehlertoleranter Verfahren erfordern.

- **Verständlichkeit**

Das Entwicklungsteam muss in der Lage sein, die Charakteristika einer Architektur schnell zu erfassen und zu benennen.

- **Wartbarkeit**

Wartbarkeit erfordert Verständlichkeit, Einfachheit, Korrektheit und Erweiterbarkeit der Architektur.

---

<sup>22</sup> Software-Cluster haben ein Cluster-internes Shared Memory zwischen mehreren Betriebssystem-Prozessen, kommunizieren zwischen den Clustern aber über Kanäle.



- **Wiederverwendbarkeit**

Die Wiederverwendbarkeit ist eng mit der Flexibilität bzw. Änderbarkeit der Architektur gekoppelt. Zu diesem Zweck kann die Architektur erweiterungsfähige Muster enthalten.

Die genannten nicht funktionalen Qualitäten einer Architektur können nicht auf einzelne Funktionen abgebildet werden und sind auch nicht unabhängig voneinander. Daneben gibt es aber auch Qualitäten, die an das Vorhandensein bestimmter Funktionen geknüpft werden. Dies betrifft:

- die Security,
- die Safety und
- die Parallelität bzw. Nebenläufigkeit.

## 2.3 Referenzarchitekturen, Architektur- und Entwurfsmuster

Eine **Referenzarchitektur** wie auch ein **Architekturmuster** beschreibt nicht nur ein einzelnes System, sondern eine ganze Klasse von Systemen für eine bestimmte Problemstellung. Allerdings ist der Begriff einer Referenzarchitektur schillernd und nicht einheitlich definiert.

In diesem Buch wird der Begriff einer Referenzarchitektur in dem Sinne verwendet, dass eine Referenzarchitektur wiederverwendbare Teile und individuelle Teile einer Softwarearchitektur für einen bestimmten Geschäftszweck (Domain) enthält.



Eine Referenzarchitektur ebenso wie ein Architekturmuster gibt die Zerlegung in bestimmte Teile eines Systems vor, die miteinander nach einem typischen Schema interagieren. Werden diese Vorgaben beim Systementwurf beachtet, so entstehen standardisierte Architekturen. Erweiterbare Teile sind leicht wiederverwendbar.

Architekturmuster und Referenzarchitekturen stellen **Schablonen** für viele konkrete Softwarearchitekturen dar. Allerdings enthalten Architekturmuster im Gegensatz zu Referenzarchitekturen keine individuellen Teile<sup>23</sup>, die sich auf das fachliche Problem beziehen.



Letztendlich enthalten auch **Referenzarchitekturen** ebenso wie Architektur- und Entwurfsmuster Komponenten in Rollen. Eine Referenzarchitektur kann hierbei verschiedene Architekturmuster enthalten, beispielsweise MVC für die Oberfläche und Layers für die Kommunikationsprotokolle, und – wie bereits erwähnt – auch **individuelle Komponenten**.

<sup>23</sup> Dasselbe gilt auch für Entwurfsmuster.

**Vorteile** von Referenzarchitekturen sind:

- Systeme, die auf einer Referenzarchitektur beruhen, müssen nicht vollständig selbst entworfen werden.
- Komponenten können ggf. wiederverwendet werden.
- Referenzarchitekturen haben Eigenschaften wie eine bestimmte Erweiterbarkeit.

Ein **Nachteil** von Referenzarchitekturen ist:

- Referenzarchitekturen passen nicht immer zu 100% und können ggf. nicht optimal an die Bedürfnisse angepasst werden. Durch die Verwendung einer Referenzarchitektur muss man mit gewissen Vorgaben als Einschränkungen leben.

**Architekturmuster** haben

- die Zerlegung des betrachteten Systems<sup>24</sup> in erweiterbare Subsysteme nach einer bestimmten Strategie und
- die Zusammenarbeit der erweiterbaren Subsysteme

zum Ziel.

Sogenannte **Entwurfsmuster** stellen erweiterbare Klassen eines Systems in bestimmten Rollen dar, die durch ihr Zusammenwirken eine bestimmte Problemstellung lösen. Architekturmuster sind größere Einheiten als Entwurfsmuster. **Architekturmuster** können – aber müssen nicht – mehrere verschiedene Entwurfsmuster beinhalten.<sup>25</sup> **Referenzarchitekturen** sind wiederum größere Muster als Architekturmuster, da sie verschiedene Architekturmuster enthalten können und überdies noch individuell zu entwickelnde Teile beinhalten.

Während eine **konkrete Architektur** neben der Erfüllung nicht funktionaler Eigenschaften auch **konkreten funktionalen Anforderungen** genügt, stehen bei einer Referenzarchitektur und bei einem **Architekturmuster** bzw. **Entwurfsmuster** die nicht funktionalen Anforderungen wie **Standardisierung**, **Verständlichkeit**, **Einfachheit** und **Ausbaufähigkeit** im Vordergrund.



## 2.4 Aufgaben, Sichten und Prototypen bei der Konzeption einer Softwarearchitektur

Es gibt verschiedene Typen von Systemen. Der richtige Systemtyp muss zuallererst identifiziert werden. Einige Beispiele für solche Typen sind:

<sup>24</sup> Dies muss nicht unbedingt das zu realisierende Gesamtsystem sein.

<sup>25</sup> Ein Beispiel hierfür ist das Architekturmuster MVC, das fast immer die Entwurfsmuster Beobachter, Strategie und Kompositum enthält.

- **Systeme mit harter Echtzeit<sup>26</sup>** wie:
  - Steuergeräte (Embedded Systems) eines Autos. Bei Embedded Systems bringt die Software ein Gerät zum Laufen.
  - Steuerung einer Walzstraße in einem Stahlwerk.
- **Systeme mit weicher Echtzeit** wie
  - Multimedia-Geräte (Videoplayer, MP3-Player etc.).
  - Videokonferenzsysteme.
  - Das Embedded System eines Handy.
- **Systeme ohne Echtzeit** wie
  - Buchungssysteme mit Transaktionen und weltweit verteilten Arbeitsplätzen.

Die Konzeption einer Systemarchitektur erfordert zum einen Teil die Durchführung analytischer Aufgaben, zum anderen Teil die Durchführung konstruktiver Aufgaben.

Kapitel 2.4.1 behandelt die analytischen Aufgaben und Kapitel 2.4.2 die konstruktiven Aufgaben bei der Konzeption eines Systems.

## 2.4.1 Analytische Aufgaben bei der Konzeption eines Systems

Analysiert werden müssen:

- **der Einsatzzweck des Systems und seine Hauptaufgaben.**  
Hierzu müssen die Anforderungen mit dem Kunden abgestimmt sein.
- **die Einbettung der Abläufe der Anwendungsfälle des Systems in die Aufbau- und Ablauforganisation des Kunden.**  
Hierbei ist zu beachten, dass sich bei der Einführung eines neuen Systems durchaus die Aufbau- und Ablauforganisation des Nutzers ändern kann.
- **die organisatorische Sicht der Schnittstellen des Systems zu Fremdsystemen und Nutzern.**  
Bei den Schnittstellen interessiert die Art der Schnittstelle wie "Drehstuhlschnittstelle"<sup>27</sup>, Datenschnittstelle, Prozeduraufruf. Bei den Nutzern interessiert die Art der Arbeitsplätze, die verschiedenen Rollen wie beispielsweise Verkäufer, Buchhaltung oder Systemadministrator und die vorgesehenen Interaktionsmöglichkeiten.
- **die technische Sicht der Schnittstellen eines Systems zu seiner Umgebung.**  
Die Analyse der Schnittstellen des Systems stellt für das Innere des Systems selbst eine **Blackbox-Sicht** dar.
- **die Struktur und die Abläufe des Systems aus logischer Sicht.**  
Hierbei werden in der Analyse in der Regel nur die Verarbeitungsfunktionen betrachtet.

<sup>26</sup> Harte Echtzeit ist praktisch deterministisch. Wenn das nicht erfüllt ist, tritt ein Schaden ein. Bei weicher Echtzeit kann es einzelne Ausreißer in einem gewissen Maße geben.

<sup>27</sup> Bei einer Drehstuhlschnittstelle werden Daten zwischen Systemen ausgetauscht, da die Systeme nicht miteinander reden können oder dürfen.

## 2.4.2 Konstruktive Aufgaben bei der Konzeption eines Systems

Beim Bau eines Systems sind neben **analytischen** auch **konstruktive** Fähigkeiten gefragt. Bei der Analyse wird geklärt, was zu tun ist. Bei der Konstruktion wird der Bau eines Systems konzipiert und umgesetzt. Hierzu gehören der Aufbau eines Systems aus Komponenten und die Zusammenarbeit dieser Komponenten. Dabei muss darauf geachtet werden, dass die Komponenten untereinander schwach gekoppelt sind, damit das System leicht änderbar ist. Als Komponenten können dabei selbst zu entwickelnde Komponenten und Produkte eingesetzt werden.

### 2.4.2.1 Zerlegung eines verteilten Systems

Bei der Konstruktion eines Systems wird auf der obersten Ebene zuerst das **Gesamtsystem** aus operationeller Sicht<sup>28</sup> – d. h. aus Sicht der Anwender und nicht der Systemadministration – betrachtet. Hierbei ist zu untersuchen:

- der Hardwareaufbau der Rechner (**Hardwarearchitektur des Gesamtsystems**),
- die **Verteilung der Artefakte** der verschiedenen Funktionalitäten auf die verschiedenen Rechner des Systems und in der Regel die **Schichtenstruktur der Software jedes einzelnen Rechners** sowie die **Strategie für die Dynamik der Architektur** des Gesamtsystems, d. h. eine Beschreibung, wie das Gesamtsystem ablaufen soll,
- wie getestet werden soll<sup>29</sup> und
- was ausgeliefert werden soll (**Bereitstellungssicht**)<sup>30</sup>.

Die Struktur und das Verhalten des operationellen Systems müssen zunächst in groben Zügen festgelegt werden.



Der wichtigste Punkt bei der Konzeption der konkreten Architektur eines Systems ist die Bereitstellung der gewünschten Funktionalität der Geschäftsprozesse unter Beachtung der geforderten Qualitäten.



Die Abbildung der Geschäftsprozesse auf die Anwendungsfälle des Systems inklusive der dafür erforderlichen technischen Funktionen wie der Datenhaltung mit den gewünschten Qualitätseigenschaften und damit die Konstruktion des Systems stellt die eigentliche Kernfunktionalität des Systems für einen Anwender dar.

<sup>28</sup> Hier betrachtet man nur die erforderliche Funktionalität der Anwender und nicht für den Systemadministrator.

<sup>29</sup> Eine Architektur, die nur schwer getestet werden kann, ist zu verwerfen (**Design to Test**)

<sup>30</sup> Dies umfasst u. a. Rechner-Hardware, Software, Generier-Prozeduren und Dokumentation.

Die im Folgenden aufgeführten technischen Konzepte:

- Sicherheitskonzept,
- Prozesskonzept (Parallelität und Interprozesskommunikation/Kommunikation bzw. Middleware),
- Datenhaltungskonzept,
- Testkonzept,
- Ein- und Ausgabekonzept,
- Konzept für das Management des Systems insbesondere für den **Start-up** und **Shut-down** des Gesamtsystems,
- Konzept für die einzusetzenden Architekturmuster wie z. B. Model-View-Controller (siehe Kapitel 5.6) und
- Konzept der Ausnahmebehandlung

haben einen wesentlichen Einfluss auf die Zerlegung des Systems, genauso wie die Verwendung von Produkten.

### Sicherheitskonzept

Zum Sicherheitskonzept gehören:

- die erforderlichen **Security-Maßnahmen** und
- die erforderlichen **Safety-Maßnahmen**.  
Zu den Safety-Maßnahmen gehören auch die Bestimmung und Gewährleistung der erforderlichen Fehlertoleranz, um die gewünschte Verfügbarkeit zu erreichen, und das Konzept für einen Wiederanlauf im Fehlerfall wie z. B. eine Rekonfiguration im Falle hoch verfügbarer Systeme.

### Prozesskonzept

Zum Prozesskonzept gehört die Organisation paralleler Prozesse als Betriebssystem-Prozesse bzw. Threads und die Organisation der Kommunikation/Interprozesskommunikation sowie die Betrachtung einer eventuell erforderlichen Synchronisation zwischen verschiedenen Prozessen.

### Datenhaltungskonzept

Das Datenhaltungskonzept befasst sich mit Themen wie

- Persistenzhaltung von Daten des (verteilten) Systems, die im Rahmen der Anwendungsfälle der Software eines Rechners verarbeitet werden,
- Caching,
- Transaktionen und
- Redundanz der Datenhaltung.

## Testkonzept

Das Testkonzept sollte

- die Testumgebung,
- den Testprozess,
- die Teststrategie,
- die einzusetzenden Testmethoden und
- die Testfälle/Testdaten

enthalten.

## Ein- und Ausgabekonzept

Die Ein- und Ausgabe (Man-Machine-Interface, abgekürzt MMI) umfasst Fragen wie

- Aufbau der Dialoge,
- Dialogsteuerung,
- Ergonomie,
- Gestaltung der Oberfläche und
- Syntax- und Semantikprüfung der Eingabe.

## Konzept für das Management des Systems insbesondere für den Start-up und Shut-down des Gesamtsystems

Das Konzept für das Management des Systems befasst sich mit der Konzeption der Funktionen des Systemverwalters. Auch bei einem verteilten System müssen diese in einer **Single System View** durchführbar sein, das heißt, der Systemverwalter muss das System zentral beobachten und steuern können. Es muss insbesondere definiert werden, wie der Start-up bzw. Shut-down des (verteilten) Systems erfolgen soll.

## Einzusetzende Architekturmuster

Einzusetzende Architekturmuster wie z. B. Model-View-Controller beeinflussen eine Architektur wesentlich.

## Konzept der Ausnahmebehandlung

Die Strategie, wie Ausnahmen behandelt werden sollen, ist systemweit festzulegen. Überdies muss der Einsatz von Produkten und ihre Integration in das System betrachtet werden. Dies betrifft beispielsweise Produkte für

- grafische Oberflächen,
- Persistenz oder das
- Workflow Management.

### 2.4.2.2 Software eines einzelnen Rechners

Nach der Konzeption des Gesamtsystems geht man für jeden einzelnen Rechner in eine detaillierte Sicht der Konstruktion seiner Anwendungssoftware. Diese umfasst:

- die **externen Schnittstellen** des Rechners,
- die Zerlegung des Systems in Komponenten, also die **statische Struktur** der Software jedes Rechners, wie er aus Komponenten aufgebaut ist (die Software-Komponenten auf einem Rechner, ihre internen Schnittstellen und die statische Beziehungen zwischen den Komponenten) und
- die Modellierung und Umsetzung der Abläufe, d. h. das **Verhalten** des Systems eines Rechners. Hierbei werden die Funktionalität der Komponenten und die Abläufe in Kollaborationen, d. h. die dynamischen Beziehungen zwischen den Komponenten des operationellen Systems, betrachtet.

Für jeden Rechner des Systems müssen seine externen Schnittstellen, die Struktur und das Verhalten der auf ihm installierten Software bekannt sein.



### 2.4.3 Sichten auf eine Softwarearchitektur

Eine Architektur kann von einem Entwickler aus verschiedenen Sichten betrachtet werden. Jede dieser Sichten zeigt andere Aspekte der Architektur besonders gut. Diese Sichten sind beispielsweise:

- die **Stakeholdersicht** (Analyse der Widersprüche und Konsolidierung der Wünsche der verschiedenen Stakeholder),
- die **Analysesicht für das System in seiner Umgebung** (Blackbox-Sicht),
- die **Analysesicht für ein logisches Modell des Systems** (Whitebox-Sicht),
- die **Sicht des Entwicklungsprozesses für das System**<sup>31</sup>,
- die **Konstruktionssicht auf das operationelle System und die Programme des Systemverwalters**<sup>32</sup> sowie
- die **Sicht der Entscheidungsdokumentation** (Voraussetzungen von Entscheidungen und deren Begründung).

<sup>31</sup> Beispielsweise müssen die Software-Entwicklungsumgebung (SEU), der Aufbau der eigenen Projektbibliotheken, die Generierung des Systems und die Testverfahren festgelegt werden.

<sup>32</sup> Die Konstruktionssicht auf die Programme des Systemverwalters umfasst alle Programme und Befugnisse des Systemverwalters. Zu diesen Programmen des Systemverwalters gehören u. a. das Hoch- und Abfahren des Systems oder die Fehlerbehandlung zur Laufzeit. Dabei werden Start-Up, Rekonfiguration und Shut-Down des (verteilten) operationellen Systems betrachtet. Hierbei versteht man hier unter Rekonfiguration ein Verfahren für die Fehlerbehandlung bei Ausfällen von Ressourcen eines fehlertoleranten Systems.

### 2.4.4 Prototypen einer Softwarearchitektur

Es gibt verschiedene Prototypen zur Verifikation einer Architektur:

- horizontale Prototypen,
- vertikale Prototypen sowie
- Realisierbarkeitsprototypen ("Angsthasen"-Prototypen).

Horizontale Prototypen demonstrieren die Tauglichkeit einer Schicht wie z. B. des MMI. Vertikale Prototypen sind ein Durchstich durch alle Schichten eines Systems. Realisierbarkeitsprototypen (sogenannte "Angsthasen"-Prototypen) untersuchen experimentell, ob bestimmte Systemteile realisierbar sind. Was man nicht theoretisch zeigen kann, muss man eben experimentell beweisen.

## 2.5 Die Bedeutung eines Softwarearchitekten für ein Projekt

Für den Bau eines Softwaresystems ist ein Softwarearchitekt genauso wichtig wie ein Architekt beim Bau eines Hauses. Wie ein Architekt beim Bau von Häusern die Zusammenarbeit zwischen Auftraggeber, Behörden und Handwerkern koordiniert, muss auch ein Softwarearchitekt zwischen Auftraggeber, Management und dem Entwicklungsteam vermitteln und die Entwickler des Entwicklungsteams fachlich führen können, auch wenn für die disziplinarische Führung des Entwicklungsteams ein Projektleiter zuständig ist.

Ob die Rolle eines Softwarearchitekten neben anderen Rollen auf eine einzige Person projiziert wird oder ob es im Projekt sogar mehrere kooperierende Architekten gibt, hängt von der Größe eines Projektes ab.



Als "Drehscheibe" eines neuen Systems hat ein Softwarearchitekt Umgang mit Stakeholdern<sup>33</sup> aller Art. Dabei muss eine Architektur mit dem Entwicklungsteam, dem Management und dem Kunden abgestimmt sein und sollte von allen bejaht werden.



Ein Architekt sollte durch Erfahrungen bei vergleichbaren Systemen beeindruckend sein. Seine Meinung zu Projektrisiken sollte gefragt sein. Er sollte die Aufwände für die Realisierung der Lösungsvorschläge und ihrer Alternativen abschätzen können. Er sollte Management, Kunden und Entwicklungsteam konzeptionell überzeugen können und außerdem ein Organisationstalent sein.

<sup>33</sup> Ein Stakeholder ist eine natürliche bzw. juristische Person oder eine Gruppe von Personen, die Interesse am entsprechenden System hat. Dieses Interesse wirkt sich in Einflüssen auf Hardware- und/oder Softwareanforderungen für ein System aus.



Die Entwicklung einer Architektur erfordert:

- **Fähigkeiten zur Analyse** der fachlichen und technischen Aufgaben eines Systems und seines Umfelds,
- **konstruktive Fähigkeiten** für den Bau eines Systems,
- eine **ausgeprägte Kommunikationsfähigkeit** zum Umgang mit den übrigen Stakeholdern und Projektbeteiligten sowie
- die Fähigkeit, das **Risiko, den Aufwand und die Zukunftssicherheit** der Realisierung von Lösungsvorschlägen und Alternativen einigermaßen zuverlässig **abschätzen zu können**.

### 2.5.1 Technische Fähigkeiten eines Softwarearchitekten

Es ist wichtig, zu wissen, wie vergleichbare Systeme derselben Fachdomäne technisch umgesetzt werden, falls es solche Systeme überhaupt bereits gibt. Insbesondere müssen die Fachbegriffe einer Domäne verstanden werden. Ein Architekt sollte aber über die Fachdomäne hinaus auch über fundiertes technisches Wissen in der aktuell einzusetzenden Programmiersprache und über den Bau vergleichbarer technischer Systeme – auch von technischen Alternativen inklusive der aktuellen technischen Produkte – verfügen. Ein guter Architekt sollte bereits Erfahrungen im Bau vergleichbarer Systeme vorweisen können.

Softwarearchitekten stützen sich auf:

- ihre Erfahrungen in der Programmierung, insbesondere mit der aktuell zum Einsatz kommenden Programmiersprache,
- ihre Erfahrungen im Bau von Systemen,
- ihre Kenntnis über bzw. auf das Studium von Altsystemen und fachlich ähnlichen Systemen,
- ihre Fähigkeiten, Modelle methodisch und standardisiert zu erstellen,
- den Bau von Prototypen,
- bekannte Entwurfs- und Architekturmuster,
- den Stand der Technik und
- kommerzielle Produkte wie Frameworks, Betriebssysteme oder Compiler.

### 2.5.2 Kommunikative Fähigkeiten eines Softwarearchitekten

Für den Entwurf einer Architektur erhält ein Softwarearchitekt Vorgaben beispielsweise zu

- den Kosten,
- dem Risiko und
- der Zeit,

die für die Entwicklung eines Systems eingeplant sind. Überdies muss eine Architektur in die Systemlandschaft des Kunden passen.

Ein Architekt hat organisatorische Schnittstellen zu:

- **Kunde und Management**  
bei Plänen, die Kosten, Risiko, Zeit und Qualität betreffen.
- **Kunde, Management und Entwicklungsteam**<sup>34</sup>  
bei der Umsetzung des Systems und der Betrachtung der Konsequenzen von Anforderungen wie etwa dem Preis.
- **Projektleiter**  
bei der Zuweisung von Aufgaben an und zur Steuerung der Entwickler.
- **Entwickler**  
bei der Akzeptanz und Umsetzung der Architektur durch die Entwickler.<sup>35</sup>
- **Systemmanager/Hardwarelieferant**  
bei der Auswahl der für die Implementierung der beabsichtigten Architektur geeigneten Hardware.
- **Testteam**  
bei Fragen der Testbarkeit der Architektur und der Funktionalität des Systems.

Eine positive "Bedienung" all dieser Schnittstellen erfordert die Eigenschaften eines "kleinen" Diplomaten für einen hochkarätigen Techniker. Der Architekt darf aber nicht relativ unscheinbar im Hintergrund wirken, sondern muss auch aktiv Werbung für seine Architektur betreiben, da er letztendlich alle Stakeholder und Projektbeteiligten von seinem Konzept überzeugen muss, so dass ein jeder die vorgeschlagene Architektur für die beste Lösung hält.



### 2.5.3 Entscheidungen beim Bau einer Softwarearchitektur

Beim Entwurf einer Softwarearchitektur müssen – meist auf der Basis unsicheren Wissens – zahlreiche Entscheidungen gefällt werden, beispielsweise bei der Wahl der eingesetzten Technologien oder der Wahl der eingesetzten Muster.



Einflussfaktoren auf eine Softwarearchitektur können ihren Ursprung in der Technik, aber auch in der Organisation z. B. des Kunden oder des Lieferanten haben.

Da sich die Anforderungen an ein System während der Laufzeit der Entwicklung eines Systems ändern, muss man insbesondere darauf achten, ob eine Änderung der Anforderungen architekturrelevant ist und Änderungen der Architektur erzwingt oder nicht.

<sup>34</sup> inklusive Unterauftragnehmern

<sup>35</sup> Hierbei benötigt der Architekt die Rückkopplung der Entwickler. Fachlich gesehen ist der Architekt der wichtigste Ansprechpartner für einen Entwickler.

Entscheidungen in der Technik betreffen beispielsweise:

- **Struktur**  
Welche Komponenten des Systems mit welchen Verantwortlichkeiten gibt es?
- **Schnittstellen**  
Über welche Abstraktion einer Komponente kann man auf eine bestimmte Komponente zugreifen?
- **Abläufe**  
Wie sieht die Dynamik des Systems aus?
- **Muster**  
Welche Muster sollen verwendet werden?
- **Produkte**  
Welche kommerziellen Produkte wie Datenbankmanagementsystem, Frameworks etc. sollen eingesetzt werden?
- **Tools**  
Welche Werkzeuge sollen den Entwicklungsprozess unterstützen?
- **Testen**  
Wie und wie lange soll mit welchen Methoden getestet werden?

Bei der Verwendung von kommerziellen Produkten und Tools muss insbesondere beachtet werden, welchen Einfluss sie auf die Architektur haben. Bei der Realisierung eines neuen Teilsystems für ein vorhandenes System kann das Teilsystem nicht wie auf der "grünen" Wiese entwickelt werden, sondern es unterliegt gravierenden Einschränkungen durch das bereits bestehende System.

In allen Fällen müssen stets Alternativen gegeneinander abgewogen und bewertet werden.

In der Organisation sind ebenso Entscheidungen fällig wie zum Beispiel:

- **Einsatzplanung**  
Korrelation der Fähigkeiten der Mitarbeiter und der Personalressourcen mit den technischen Aufgaben des zu realisierenden Systems.
- **Kosteneffizienz**  
Nicht jede Architektur hat dieselben Kosten bei ihrer Umsetzung.

Alle relevanten Entscheidungen sind in einer **Entscheidungsdocumentation** mitsamt ihren Voraussetzungen festzuhalten. Ändern sich die Voraussetzungen im Verlauf des Projekts, können ggf. auch gewisse Entscheidungen revidiert werden. Oftmals muss ein Architekt innovative, risikoreiche Ansätze gegen eine eher konventionelle Vorgehensweise abwägen. Man sieht oft erst nach Jahren, ob die getroffenen Entscheidungen sich als zukunftsicher erweisen.

## 2.6 Zusammenfassung

Nach Kapitel 2.1 umfasst der Begriff einer Softwarearchitektur:

- die Zerlegung eines Systems in Komponenten inklusive dem Deployment (Struktur, Statik),

- die Zusammenarbeit der Komponenten (Verhalten, Dynamik) und
- die Beschreibung der Strategie für die Zerlegung und die Dynamik der Architektur

mit dem Ziel, nach außen die geforderte Leistung des Systems anbieten zu können.

Kapitel 2.2 befasst sich vor allem mit verschiedenen erstrebenswerten nicht funktionalen Qualitäten einer Architektur wie Bedienbarkeit, Performance, Testbarkeit oder Skalierbarkeit.

Kapitel 2.3 betrachtet die Unterschiede zwischen Referenzarchitekturen, Architekturmustern und Entwurfsmustern.

Kapitel 2.4 diskutiert analytische und konstruktive Aufgaben beim Bau eines Systems, verschiedene Sichten auf eine Softwarearchitektur und die Erstellung von Prototypen für eine Softwarearchitektur.

Die analytischen Aufgaben beim Bau eines Systems (siehe Kapitel 2.4.1) umfassen:

- den Einsatzzweck des Systems und seine Hauptaufgaben,
- die Einbettung der Abläufe der Anwendungsfälle des Systems in die Aufbau- und Ablauforganisation des Kunden,
- die organisatorische Sicht der Schnittstellen des Systems zu Fremdsystemen und Nutzern,
- die technische Sicht der Schnittstellen eines Systems zu seiner Umgebung und
- die Erstellung eines Modells des Systems aus logischer Sicht.

Die konstruktiven Aufgaben beim Bau eines Systems beinhalten (siehe Kapitel 2.4.2) für das verteilte Gesamtsystem:

- die **Hardwarearchitektur** des Gesamtsystems,
- die Verteilung der Artefakte der verschiedenen Funktionalitäten der Anwendungsfunktionen auf die verschiedenen Rechner des Systems und in der Regel die Schichtenstruktur der Software jedes einzelnen Rechners des verteilten Systems (**Statik**) sowie die **Dynamik** der Architektur des Gesamtsystems, d. h. eine Beschreibung, wie das Gesamtsystem ablaufen soll,
- wie getestet werden soll<sup>36</sup> und
- was ausgeliefert werden soll (**Bereitstellungssicht**)<sup>37</sup>.

Die konstruktiven Aufgaben beinhalten für jeden einzelnen Rechner die detaillierte Sicht der Konstruktion seiner Anwendungssoftware (siehe Kapitel 2.4.2.2). Diese umfasst:

- die **externen Schnittstellen** des Rechners,
- die Zerlegung des Systems in Komponenten, also die **statische Struktur** der Software jedes Rechners, wie er aus Komponenten aufgebaut ist, d. h. die Software-

<sup>36</sup> Eine Architektur, die nur schwer getestet werden kann, ist zu verwerfen (**Design to Test**).

<sup>37</sup> Dies umfasst u. a. Rechner-Hardware, Software, Generier-Prozeduren und Dokumentation.

Komponenten auf einem Rechner, ihre internen Schnittstellen und die statische Beziehungen zwischen den Komponenten und

- die Modellierung und Umsetzung der Abläufe, d. h. das **Verhalten** des Softwaresystems eines Rechners. Hierbei werden die Funktionalität der Komponenten und die Abläufe in Kollaborationen, d. h. die dynamischen Beziehungen zwischen den Komponenten des operationellen Systems, betrachtet.

Die Sichten auf eine Softwarearchitektur (siehe Kapitel 2.4.3) umfassen:

- die **Stakeholdersicht**,
- die Analysesicht für die **Einbettung des Systems** in seine Umgebung,
- die Analysesicht für ein **logisches Modell** des Systems,
- die **Sicht des Entwicklungsprozesses** für das System,
- die **Konstruktionssicht** für das operationelle System und die Programme des Systemverwalters,
- die **Sicht der Auslieferung (Bereitstellungssicht)** und
- die **Sicht der Entscheidungsdokumentation**.

Die Erstellung von Prototypen (siehe Kapitel 2.4.4) umfasst:

- horizontale Prototypen,
- vertikale Prototypen sowie
- Realisierbarkeitsprototypen.

Kapitel 2.5 stellt klar, dass ein Architekt außer ausgeprägten analytischen und konstruktiven Tätigkeiten auch sehr kommunikativ sein muss und überdies einigermaßen zuverlässig den Aufwand für die Realisierung von Lösungsvorschlägen schätzen können muss.

## 2.7 Aufgaben

### Aufgaben 2.7.1: Softwarearchitektur allgemein

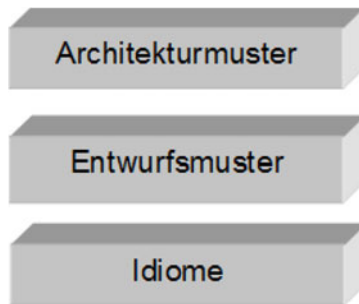
- 2.7.1.1 Was ist ein Stakeholder?
- 2.7.1.2 Wie lautet die Definition des Begriffs Architektur?
- 2.7.1.3 In welchem Entwicklungsschritt entsteht die Architektur einer Software?
- 2.7.1.4 Was ist das Ergebnis des Systementwurfs?
- 2.7.1.5 Welche Prototypen gibt es zur Verifikation einer Architektur und wozu werden die einzelnen Typen verwendet?
- 2.7.1.6 Beschreiben Sie das Verhältnis zwischen Programm, Anforderungen und Architektur.

### Aufgaben 2.7.2: Referenzarchitektur, Architektur- und Entwurfsmuster

- 2.7.2.1 Was ist ein Architekturmuster?
- 2.7.2.2 Was ist ein Entwurfsmuster?
- 2.7.2.3 Ist ein Architekturmuster gröber als eine Referenzarchitektur?
- 2.7.2.4 Ist ein Architekturmuster feiner als ein Entwurfsmuster?

# *Kapitel 3*

## Muster beim Softwareentwurf



- 3.1 Einsatz von Mustern
- 3.2 Eigenschaften von Mustern und ihre Konstruktion
- 3.3 Abgrenzung zwischen Architekturmustern, Entwurfsmustern und Idiomen
- 3.4 Schema für die Beschreibung von Entwurfs- und Architekturmustern
- 3.5 Zusammenfassung
- 3.6 Aufgaben

### 3 Muster beim Softwareentwurf

In der Softwareentwicklung gibt es praktisch für alle Entwicklungsschritte Muster: Muster für die Analyse, für den Entwurf, für die Implementierung und für das Testen. Aber auch für speziellere Aufgaben existieren Muster wie etwa für den Entwurf grafischer Oberflächen oder für den Umgang mit Datenbanken.

Dieses Buch konzentriert sich auf Muster beim Softwareentwurf. Hierbei unterscheidet man – je nach Granularität – zwischen:

- **Architekturmustern** (engl. **architectural patterns**) und
- **Entwurfsmustern** (engl. **design patterns**).

Eine Abgrenzung von Architektur- und Entwurfsmustern erfolgt in Kapitel 3.3.

Muster für den Entwurf sind bewährte Lösungsvorschläge für bestimmte Problemstellungen, die beim Entwurf von Systemen beachtet werden sollten, da sie sich bereits in mehreren Systemen bewährt haben.



Der Ursprung der Muster beim Entwurf geht auf den Architekten Christopher Alexander<sup>38</sup> zurück. Er hatte in den 70er Jahren eine Sammlung von Mustern für den Städtebau zusammengestellt. Christopher Alexander erkannte, dass Gebäude oder auch ganze Straßenzüge zwar dieselben Elemente enthalten können, dennoch aber nach einem ganz anderen Muster aufgebaut sein können. Mit anderen Worten, er identifizierte Muster durch Elemente und ihre typischen Beziehungen:

"... beyond its elements, each building is defined by certain patterns of relationships among the elements ..." [Ale77]

In der städtebaulichen Architektur ist diese bahnbrechende Idee der Muster allerdings bis heute bei weitem nicht so verbreitet und anerkannt, wie sie es in der Softwareentwicklung ist.

Muster beim Entwurf stellen einen wesentlichen Beitrag dar, die Softwareentwicklung auf ihrem Weg zur ausgereiften Ingenieurwissenschaft ein gutes Stück voranzubringen. Muster sind grundsätzlich plattformunabhängig und nicht auf eine bestimmte Programmiersprache beschränkt. Die Namen der Muster erweitern die Fachsprache und erlauben es geschulten Entwicklern, sich auf einem hohen Abstraktionsniveau über ein Problem und seine Lösung verständigen zu können.

---

<sup>38</sup> Christopher Alexander ist Architekt und Städteplaner. Er erhielt 1963 eine Professur für Architektur an der University of California in Berkley und wurde Direktor des Center for Environment Structure.



Ein Muster arbeitet oft mit Abstraktionen, sei es in der Notation einer abstrakten Klasse oder einer Schnittstelle. Die echten Klassen sind dann abgeleitete Klassen. Sie sind dem Muster nicht bekannt. Sie werden vom Nutzer des Musters definiert. Sie treten dann entweder zur Kompilierzeit bei konkreten klassenbasierten Mustern oder zur Laufzeit bei objektorientierten Mustern an die Stelle der Abstraktionen des Musters.



Um Muster anderen Entwicklern zugänglich zu machen, müssen diese dokumentiert werden. Muster werden nicht erfunden, sondern in Anwendungen als Lösungen für typische Probleme und Sachverhalte "entdeckt", auf Konferenzen vorgetragen und dann von der Fachwelt akzeptiert oder auch nicht. Sie können für viele Anwendungen eingesetzt werden.

Kapitel 3.1 befasst sich mit dem Einsatz von Mustern beim Softwareentwurf. Kapitel 3.2 beschreibt die Eigenschaften von Mustern beim Softwareentwurf und ihre Konstruktion. Kapitel 3.3 grenzt Architektur-, Entwurfsmuster und Idiome voneinander ab. In Kapitel 3.4 wird das Schema vorgestellt, das in diesem Buch verwendet wird, um die Muster zu dokumentieren.

## 3.1 Einsatz von Mustern

Das Hauptziel von Mustern für den Softwareentwurf ist es, einmal gewonnene Erkenntnisse wiederverwendbar zu machen und durch ihre Anwendung die Flexibilität einer Architektur zu erhöhen.



Die meisten Systementwürfe enthalten zahlreiche Muster, wobei diese Tatsache allein noch kein Merkmal eines guten Entwurfs ist. Auch hier hat Christopher Alexander [Ale77] die passenden Worte gefunden: "Es ist möglich, Gebäude durch das lose Aneinanderreihen von Mustern zu bauen. Ein so konstruiertes Gebäude stellt eine Ansammlung von Mustern dar. Es besitzt keinen inneren Zusammenhalt. Es hat keine wirkliche Substanz. Es ist aber auch möglich, Muster so zusammenzufügen, dass sich viele Muster innerhalb desselben Raums überlagern. Das Gebäude besitzt einen inneren Zusammenhalt; es besitzt viele Bedeutungen, auf kleinem Raum zusammengefasst. Durch diesen Zusammenhalt gewinnt es an Substanz." Ein guter Entwurf entsteht also oft erst dann, wenn mehrere Muster untereinander und mit der Anwendung so geschickt verknüpft werden, dass Synergieeffekte entstehen.

Ein Muster sollte nur dann angewendet werden, wenn es tatsächlich sinnvoll ist und auch Vorteile bringt. Bevor ein Muster eingesetzt wird, sollte man sich also sehr genau überlegen, ob dieses Muster überhaupt zum Problem passt und welche Konsequenzen sich aus der Anwendung des Musters ergeben.



Aus Entwicklersicht bieten Muster eine gewisse Sicherheit. Diese Sicherheit beruht auf der Tatsache, dass die Lösung, die man verwenden will, sich in der Vergangenheit bereits bewährt hat. Dies führt im Allgemeinen jedoch zu dem Trugschluss, dass das Anwenden von Mustern in jedem Fall die richtige Entscheidung ist. Diese Annahme ist jedoch falsch. Das reine Anwenden von Mustern garantiert einem Entwickler nicht, dass das Muster für sein konkretes Entwurfsproblem sinnvoll ist.

Bei objektbasierten Mustern spielt das **Delegationsprinzip** oft eine wichtige Rolle. Hierbei delegiert das Objekt, das eine Nachricht empfängt, diese im Idealfall weiter. Die Weiterleitung erfolgt

- an ein vom empfangenden Objekt aggregiertes Teil-Objekt oder
- aber an ein Objekt, das zur Laufzeit die konkrete Implementierung einer Abstraktion (Schnittstelle bzw. abstrakten Klasse) ist, wobei die Abstraktion von der Klasse des empfangenden Objekts implementiert wird.

Es ist vollkommen klar, dass jegliche Form der Delegation und Abstraktion Performance kostet. Zusätzliche Schichten sind immer mit einer Verringerung der Performance eines Systems verbunden.



Wenn man Erweiterbarkeit anstrebt, lohnen sich Muster. Man muss sich aber den Einsatz von Mustern äußerst gründlich überlegen, wenn eine hohe Performance im Vordergrund steht und die Änderbarkeit keine Rolle spielt.

Obwohl der Einsatz von Mustern die Erweiterbarkeit einer Architektur erhöht, steigt meist auch gleichzeitig die Komplexität der Architektur.

Zur Gewährleistung der Erweiterbarkeit können neue Klassen und Operationen entstehen und damit auch neue Beziehungen zwischen den Klassen. Diese Komplexität führt zu einer Leistungsminderung des Systems, die unter Umständen nicht durch die gewonnene Flexibilität aufgewogen wird.



Dies ist vor allem der Fall, wenn diese Flexibilität spekulativ ist, d. h., wenn es unklar ist, ob sie später überhaupt jemals gebraucht wird oder nicht. Es gilt der Grundsatz: "So viel Flexibilität wie nötig, so wenig wie möglich!"

In Büchern über Muster ist die Lösung oft der Kern der Darstellung. Das Problem selbst jedoch, d. h., wann die Muster geeignet sind, wird in solchen Fällen zwar aufgeführt, oft jedoch leicht unterschätzt. Unter Umständen führt sogar keines der bekannten Muster zum gewünschten Ziel.

Ist man auf der Suche nach einer Lösung für ein konkretes Entwurfsproblem, kann es schwierig sein, das richtige Muster zu finden. Man muss das zu lösende Problem studieren und dann die Leistungen der eventuell in Frage kommenden Muster vergleichen.

## 3.2 Eigenschaften von Mustern und ihre Konstruktion

Muster haben das Ziel, die Eigenschaften von Architekturen zu verbessern. Hierzu gehören beispielsweise die folgenden Eigenschaften:

- Verständlichkeit und
- Erweiterbarkeit.

Verständlichkeit wird erreicht durch Einfachheit und indem jedes Muster einfach und in strukturierter Weise umfassend dokumentiert wird. Erweiterungen können beispielsweise

- durch statische Vererbung oder
- über die Verwendung einer aggregierten Schnittstelle<sup>39</sup> bzw. einer abstrakten Klasse als Abstraktion

erfolgen.

Objektorientierte Muster genügen den folgenden Konstruktionsprinzipien:

- **loosely coupled system** (siehe Kapitel 1.5) – lose Kopplung von Komponenten,
- **Abstraktion** (siehe Kapitel 1.2) – durch Generalisierung und durch Ausweisen von Schnittstellen oder abstrakten Klassen,
- **Information Hiding** (siehe Kapitel 1.2) durch Verbergen der Implementierung,
- **klare Verantwortlichkeiten** (siehe Kapitel 1.3) durch eine Ausweisung von Rollen und
- **dem Dependency Inversion-Prinzip**, wobei eine Klasse einer höheren Ebene nicht von einer Klasse einer tieferen Ebene abhängt.



Es soll an dieser Stelle betont werden, dass es möglich ist, Muster für nicht objektorientierte Systeme zu formulieren. Diese Muster kommen also insbesondere ohne Vererbung und Polymorphie aus [Bus98, S. 24]. Muster müssen nicht grundsätzlich objektorientiert sein. Die in diesem Buch vorgestellten **Entwurfsmuster** werden alle in einer objektorientierten Fassung vorgestellt.

Wer objektorientiert denkt, versucht stets durch **Generalisierung** die höchstmögliche Abstraktion – in anderen Worten die Basisklasse – zu finden, um daraus durch **Spezialisierung** die verschiedenen Varianten zu gewinnen.

<sup>39</sup> Siehe beispielsweise das Brücke-Muster.

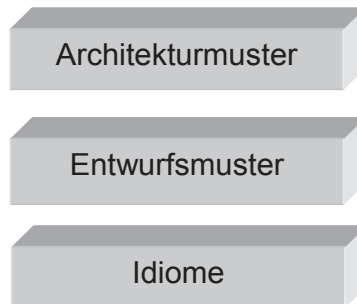
Einige Beispiele der Muster sind aus diesem Grund mit abstrakten Basisklassen entworfen, andere jedoch mit Schnittstellen. Vom Grundsatz her gilt, dass Schnittstellen meist vorzuziehen sind. Das hat zwei Gründe:

1. Bei Schnittstellen ist in einigen Programmiersprachen wie z. B. bei Java eine Mehrfachvererbung erlaubt im Gegensatz zu abstrakten Basisklassen.
2. Während abstrakte Basisklassen neben abstrakten Operationen auch fertige Operationen enthalten können, sind Schnittstellen übersichtlicher. Sie enthalten nur abstrakte Operationen.

Dass eine Schnittstelle einer abstrakten Klasse vorzuziehen ist, gilt jedoch nicht immer. Es wäre kontraproduktiv, wenn man beispielsweise eine Aggregation oder aber eine Default-Implementierung in jeder Implementierungsklasse realisieren müsste. Hier ist eine abstrakte Klasse vorzuziehen.

### 3.3 Abgrenzung zwischen Architekturmustern, Entwurfsmustern und Idiomen

Beim Entwurf eines Systems werden Architekturmuster, Entwurfsmuster und Idiome unterschieden:



*Bild 3-1 Muster bei der SW-Entwicklung*

Die einzelnen Muster unterscheiden sich vor allem im Abstraktionsgrad. Bei **Architekturmustern** betrachtet man die Architektur eines Systems, bei **Entwurfsmustern** in der Regel ein bestimmtes Problem innerhalb eines Subsystems. Ein Entwurfsmuster beeinflusst in der Regel nicht die grundsätzliche Architektur des Systems. Ein **Idiom** ist in diesem Zusammenhang ein Muster in einer bestimmten Programmiersprache wie z. B. ein ausprogrammiertes Entwurfsmuster<sup>40</sup>.

Die Architektur eines Systems kann mehrere **Architekturmuster** enthalten, z. B. eine Schichtenarchitektur (engl. Layers) für die Anwendungssoftware verschiedener Rechnerarten wie Client- und Server-Rechner oder das Architekturmuster Model-View-Controller (MVC) zur Trennung der Darstellung und der interaktiven Eingabe der Mensch-Maschine-Schnittstelle von dem Daten haltenden und verarbeitenden Model. Ein Architekturmuster wiederum kann – muss es aber nicht – zahlreiche Entwurfs-

<sup>40</sup> Auf die Möglichkeit weiterer Ausprägungen von Idiomen soll an dieser Stelle nicht eingegangen werden.

muster enthalten. So enthält beispielsweise das Architekturmuster einer Schichtenarchitektur (Layers) keine weiteren Entwurfsmuster. Das Architekturmuster MVC hingegen kann beispielsweise das Entwurfsmuster Beobachter, das Kompositum-Muster und das Strategie-Muster enthalten.

**Entwurfsmuster** werden als bewährte Konstruktionsprinzipien im Kleinen eingesetzt. Man spricht auch von **Mikroarchitekturen**.

Die Strukturen und Mechanismen der Entwurfsmuster bestimmen die Zerlegung eines Subsystems in Teile und deren Zusammenarbeit und greifen damit tief in ein Teilsystem ein.

In der objektorientierten Softwareentwicklung sind **Entwurfsmuster Klassen in Rollen**, die zusammenarbeiten, um **gemeinsam eine bestimmte Aufgabe zu lösen**.



Einige Autoren unterscheiden nicht zwischen Entwurfsmustern und Architekturmustern. Eigentlich wird ja auch die Architektur eines Systems beim Entwurf festgelegt. Insofern ist es also durchaus berechtigt, auch die Architekturmuster als Entwurfsmuster zu bezeichnen. Beispielsweise wird in [Eil07] das MVC-Muster zu den Entwurfsmustern gezählt. In dem vorliegenden Buch wird zwischen Architektur- und Entwurfsmustern unterschieden. Kapitel 4 enthält Entwurfsmuster und Kapitel 5 Architekturmuster.

**Entwurfsmuster** stellen **feinkörnige Muster** dar, während **Architekturmuster grobkörnige Muster** sind.



Die Idee der **Verwendung von Entwurfsmustern** in der Softwareentwicklung wurde erstmals 1987 von Kent Beck und Ward Cunningham für die Erstellung von grafischen Benutzerschnittstellen in Smalltalk aufgegriffen und angewandt. Es gibt mittlerweile eine ganze Reihe unterschiedlichster Entwurfsmuster, die in zahlreichen Büchern katalogisiert sind. Eine generelle Übertragung der Entwurfsmuster auf die Softwareentwicklung erfolgte durch die Promotion von Erich Gamma. Das Buch "Design Patterns – Elements of Reusable Object-Oriented Software" [Gam95] der sogenannten "**Gang of Four**"<sup>41</sup> (GoF) führte zum flächendeckenden Einsatz der Entwurfsmuster in der Softwareentwicklung. Dieses Buch enthält einen Katalog von 23 Entwurfsmustern, die nach Gamma [Gam95] in die drei Kategorien Strukturmuster, Verhaltensmuster und Erzeugungsmuster eingeteilt werden. Nachfolgend zum GoF-Buch entstanden zahlreiche weitere Bücher, die auf diesem Werk aufbauen und dabei neue Entwurfsmuster behandeln. Beispielhaft soll hier das Buch "Pattern-orientierte Software-Architektur: Ein Pattern-System" von Frank Buschmann et al. [Bus98] genannt werden.

Einen erschöpfenden Katalog von Entwurfsmustern wird es wahrscheinlich nie geben, denn es entstehen ständig neue Muster. Auch sind viele bereits unbewusst angewandte Muster noch unbenannt.

<sup>41</sup> Die "Gang of Four" sind Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. Diese Bezeichnung wurde den vier Autoren scherzhaft zugewiesen und ist ohne großen Widerspruch von ihnen angenommen worden.

### 3.4 Schema für die Beschreibung von Entwurfs- und Architekturmustern

In den folgenden Kapiteln werden Entwurfs- und Architekturmuster nach dem hier für Entwurfsmuster aufgeführten Schema vorgestellt:<sup>42</sup>

#### 1. Name/Alternative Namen

Der Name eines Entwurfsmusters hat eine wichtige Funktion. Er versucht, das Problem, die Lösung und die Konsequenzen in einem oder in zwei Wörtern zusammenzufassen. Von den meisten Entwurfsmuster-Autoren wird die Namensfindung als der schwierigste Teil eines Entwurfsmusters genannt. Der Name sollte mit größter Sorgfalt und Weitblick gewählt werden. Der Name geht in das Vokabular von Softwareentwicklern über und hilft ihnen, einen Entwurf auf einem höheren Abstraktionsniveau zu beschreiben.

#### 2. Problem

Hier wird beschrieben, welches Problem durch die Anwendung des Musters gelöst werden soll.

#### 3. Lösung

Die Beschreibung der Lösung enthält im Falle von Entwurfsmustern die an der Lösung beteiligten Klassen, Schnittstellen und Objekte sowie deren Rollen, Beziehungen, Zuständigkeiten und Interaktionen. Der Kern der Lösung des abstrakten Entwurfsproblems wird gezeigt. Durch diese abstrakte Sicht bilden Muster eine ideale Kommunikationsbasis für Entwickler. Zur Lösung gehören folgende Abschnitte:

- **Teilnehmer** – Rollenbeschreibung der Klassen,
- **Klassendiagramm** – Klassendiagramm mit Beschreibung der wechselseitigen statischen Beziehungen (Statik),
- **Dynamisches Verhalten** – Sequenzdiagramm mit Beschreibung der einzelnen Schritte (Dynamik),
- **Programmbeispiel**<sup>43</sup>

Entwurfsmuster versteht man tatsächlich oft am besten durch Code-Beispiele, auch wenn die Struktur und das Verhalten der Objekte durch ein Klassendiagramm und ein Sequenzdiagramm visualisiert werden.

#### 4. Bewertung

Wird ein Entwurfsmuster eingesetzt, so ergeben sich zwangsweise mehr oder weniger offensichtliche Konsequenzen für die Anwendung. Eine genaue Beschreibung der Vor- und Nachteile eines Entwurfsmusters ist dabei von größter Wichtigkeit, um Lösungsalternativen sorgfältig abwägen zu können. Vor- und Nachteile werden in den Abschnitten

- **Vorteile** und
- **Nachteile**

<sup>42</sup> Bei Architekturmustern kann jedoch noch hinzukommen, welche Entwurfsmuster vom Architekturmuster verwendet werden und für welchen Zweck.

<sup>43</sup> Für Architekturmuster gibt es in diesem Buch aus Platzgründen nicht immer sinnvolle Beispiele, da sinnvolle Beispiele den Rahmen des Buches sprengen würden. Manche Beispiele werden im Buch nur verkürzt präsentiert und sind dann vollständig auf dem begleitenden Webaufttritt zu finden.

dargestellt. Es werden dabei häufig die Auswirkungen eines Entwurfsmusters auf Speicherverbrauch und Performance betrachtet sowie Folgen für die Komplexität und die Erweiterbarkeit des Entwurfs.

### 5. Einsatzgebiete

Bei den Einsatzgebieten können **Anwendungsbeispiele** optional in einem eigenen Abschnitt aufgeführt werden.

### 6. Ähnliche Muster

Hier werden Gemeinsamkeiten mit ähnlichen Mustern und Unterschiede zu ähnlichen Mustern beschrieben.

## 3.5 Zusammenfassung

Entwurfs- und Architekturmuster sind bewährte Muster, die beim Entwurf von Systemen zur Kenntnis genommen werden sollten, da sie Lösungsvorschläge für bestimmte Problemstellungen sind, die sich bereits in mehreren Systemen bewährt haben. Sie stehen als "Blaupausen" zur Verfügung und werden von Hand ausprogrammiert, es sei denn, sie werden bereits durch komfortable Entwicklungswerkzeuge mit den erforderlichen Methodenköpfen zur Verfügung gestellt.

Muster sind also nicht etwa wiederverwendbarer Quellcode, der in einer Anwendung einfach übernommen werden kann, sondern Muster sind vielmehr weitergereichte Erfahrungen im Entwurf. Muster sind erprobt und dokumentiert und haben sich in mehreren Systemen bei einer vorgegebenen Problemstellung als Bauplan beim Entwurf einer Architektur bewährt. Die konkrete Implementierung von Mustern bleibt jedoch den Entwicklern selbst überlassen.

Kapitel 3.1 befasst sich mit der Fragestellung, wie man Muster beim Entwurf einsetzen soll.

Kapitel 3.2 betrachtet wesentliche Eigenschaften von Mustern, nämlich die Verständlichkeit und Erweiterbarkeit, und ihre Konstruktion.

Kapitel 3.3 grenzt Architekturmuster, Entwurfsmuster und Idiome gegenseitig ab.

Kapitel 3.4 stellt das in diesem Buch verwendete Schema zur Beschreibung von Mustern vor.

## 3.6 Aufgaben

### Aufgaben 3.6.1: Allgemeine Aufgaben

- 3.6.1.1 Erklären Sie den Zusammenhang zwischen Architekturmustern, Entwurfsmustern und Idiomen.
- 3.6.1.2 Muss ein Architektur- oder Entwurfsmuster objektorientiert sein?

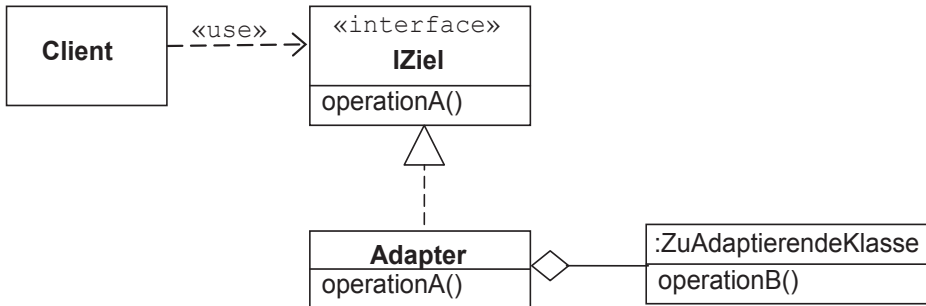
### Aufgaben 3.6.2: Entwurfsmuster

- 3.6.2.1 Was ist das Ziel der Entwurfsmuster?
- 3.6.2.2 Wie entsteht ein Entwurfsmuster?
- 3.6.2.3 Was sind Mikroarchitekturen?
- 3.6.2.4 Kann man sich sicher sein, dass man eine gute Architektur erstellt hat, wenn sie viele Entwurfsmuster enthält?



# Kapitel 4

## Objektorientierte Entwurfsmuster



- 4.1 Klassifikation von Entwurfsmustern
- 4.2 Übersicht über die vorgestellten Entwurfsmuster
- 4.3 Das Strukturmuster Adapter
- 4.4 Das Strukturmuster Brücke
- 4.5 Das Strukturmuster Dekorierer
- 4.6 Das Strukturmuster Fassade
- 4.7 Das Strukturmuster Kompositum
- 4.8 Das Strukturmuster Proxy
- 4.9 Das Verhaltensmuster Schablonenmethode
- 4.10 Das Verhaltensmuster Befehl
- 4.11 Das Verhaltensmuster Beobachter
- 4.12 Das Verhaltensmuster Strategie
- 4.13 Das Verhaltensmuster Vermittler
- 4.14 Das Verhaltensmuster Zustand
- 4.15 Das Verhaltensmuster Rolle
- 4.16 Das Verhaltensmuster Besucher
- 4.17 Das Verhaltensmuster Iterator
- 4.18 Das Erzeugungsmuster Fabrikmethode
- 4.19 Das Erzeugungsmuster Abstrakte Fabrik
- 4.20 Das Erzeugungsmuster Singleton
- 4.21 Das Erzeugungsmuster Objektpool
- 4.22 Zusammenfassung
- 4.23 Aufgaben

## 4 Objektorientierte Entwurfsmuster

In diesem Kapitel werden Entwurfsmuster objektorientiert unter Verwendung objektorientierter Techniken, insbesondere der Vererbung und der Polymorphie, dargestellt. Aus der Vielfalt existierender Entwurfsmuster wurden Entwurfsmuster aus den Kategorien Strukturmuster, Verhaltensmuster und Erzeugungsmuster ausgewählt. Diese Kategorien werden in Kapitel 4.1 erläutert. Ebenso wird der Unterschied zwischen klassenbasierten und objektbasierten Mustern aufgezeigt. Dieses Kriterium erlaubt ebenfalls eine Kategorisierung der vorgestellten Muster.

Kapitel 4.2 gibt jeweils eine kurze Übersicht über die ausgewählten Entwurfsmuster und enthält einen Wegweiser durch die Entwurfsmuster, der einem Neuling eine Lese-strategie für die in diesem Buch aufgeführten Entwurfsmuster vorschlägt. Danach werden zunächst Strukturmuster in den Kapiteln 4.3 bis Kapitel 4.8 besprochen. Nach der Diskussion der Verhaltensmuster in Kapitel 4.9 bis 4.17 werden Erzeugungsmuster in Kapitel 4.18 bis 4.21 behandelt.

Bei den Strukturmustern werden aufgeführt: Adapter (Kapitel 4.3), Brücke (Kapitel 4.4), Dekorierer (Kapitel 4.5), Fassade (Kapitel 4.6), Kompositum (Kapitel 4.7) und Proxy (Kapitel 4.8).

Die dargestellten Verhaltensmuster umfassen: Schablonenmethode (Kapitel 4.9), Befehl (Kapitel 4.10), Beobachter (Kapitel 4.11), Strategie (Kapitel 4.12), Vermittler (Kapitel 4.13), Zustand (Kapitel 4.14), Rolle (Kapitel 4.15), Besucher (Kapitel 4.16) und Iterator (Kapitel 4.17).

Bei den Erzeugungsmustern werden behandelt: Fabrikmethode (Kapitel 4.18), Abstrakte Fabrik (Kapitel 4.19), Singleton (Kapitel 4.20) und Objektpool (Kapitel 4.21).

Die hier aufgeführten Muster sind für das vorliegende Buch ausgewählt worden, weil sie den Autoren in der Praxis besonders häufig begegnet sind. Bei der Fülle der existierenden Muster muss diese Auswahl naturgemäß unvollständig sein.

### 4.1 Klassifikation von Entwurfsmustern

Entwurfsmuster können in bestimmte Kategorien eingeteilt werden. Die verschiedenen Muster-Kataloge verwenden dabei unterschiedliche Kategorisierungssysteme. In Gamma et al. [Gam95] werden die Entwurfsmuster in einer ersten Dimension in drei Kategorien eingeteilt:

- **Strukturmuster**,
- **Verhaltensmuster** und
- **Erzeugungsmuster**.

Hierbei erfassen Strukturmuster hauptsächlich die statische Struktur von Klassen bzw. Objekten, Verhaltensmuster beschreiben, wie Objekte durch ihr Zusammenwirken ein bestimmtes Verhalten erzeugen, und Erzeugungsmuster befassen sich mit der Erzeugung von Objekten.

In einer zweiten Dimension differenziert man nach Gamma [Gam95] zwischen klassenbasierten und objektbasierten Mustern:

- Bei **klassenbasierten Entwurfsmustern** werden die Beziehungen zwischen den Objekten unter Einsatz der statischen **Vererbung** für die Klassen der Objekte **zur Übersetzungszeit** festgelegt. Klassenbasierte Entwurfsmuster in diesem Buch sind die Schablonenmethode, die Fabrikmethode und die klassenbasierte Variante des Adapters. Die anderen in diesem Buch dargestellten Entwurfsmuster sind alle objektbasiert.
- Bei **objektbasierten Entwurfsmustern** wird oftmals auch die Vererbung eingesetzt. Der Typ eines Objektes kann in diesem Fall aber dynamisch **zur Laufzeit** geändert werden. Dies erreicht man beispielsweise, indem man gegen Abstraktionen wie Schnittstellen oder abstrakte Basisklassen programmiert. Bei Einhaltung der Verträge einer Methode kann dann zur Laufzeit an die Stelle einer Referenz vom Typ einer abstrakten Basisklasse bzw. eines Interface nach Liskov stets eine Referenz vom Typ einer abgeleiteten Klasse treten.

Im Folgenden wird auf die Kategorisierung "objektbasiert" oder "klassenbasiert" bei Strukturmustern, Verhaltensmustern und Erzeugungsmustern eingegangen:

- **Strukturmuster** werden in zwei weitere Kategorien aufgeteilt. In der ersten Unterkategorie befinden sich die **klassenbasierten** Strukturmuster, die über die Eigenschaft der Vererbung statisch die Klassen der betrachteten Objekte einer Struktur gewinnen. Das einzige Beispiel in diesem Buch für ein klassenbasiertes Strukturmuster ist der klassenbasierte Adapter. Die zweite Unterkategorie bilden die **objektbasierten** Strukturmuster. Das Proxy-Muster ist ein Beispiel für ein objektbasiertes Strukturmuster. Ein Proxy dient hierbei als Stellvertreter für ein anderes Objekt.

**Strukturmuster** befassen sich mit der Zusammensetzung und der Granularität von Klassen und Objekten.



- Auch bei **Verhaltensmustern** wird zwischen den **klassenbasierten** und den **objektbasierten** Verhaltensmustern unterschieden. Klassenbasierte Verhaltensmuster verwenden Vererbung, um statisch das gewünschte Verhalten zu gewinnen. Das einzige Beispiel in diesem Buch für klassenbasierte Verhaltensmuster ist die Schablonenmethode. Objektbasierte Verhaltensmuster dagegen können beispielsweise das liskovsche Substitutionsprinzip verwenden, um durch Austausch von Objekten zur Laufzeit das Verhalten abzuändern. Ein Beispiel für den dynamischen Austausch des Verhaltens ist das Strategie-Muster.

**Verhaltensmuster** befassen sich mit den Zuständigkeiten und der Zusammenarbeit zwischen Klassen bzw. Objekten. Sie beschreiben Interaktionen zwischen Objekten.



- **Erzeugungsmuster** verbergen die Erzeugung von Objekten in einer Operation bzw. einer Klasse. **Klassenbasierte** Muster wie die Fabrikmethode verwenden Operationen, deren Klasse durch Vererbung statisch gewonnen wurde. **Objektbasierte** Er-

zeugungsmuster wie die Abstrakte Fabrik verwenden zum dynamischen Austausch der zu erzeugenden Objekte das liskovsche Substitutionsprinzip. Hingegen kommt das Singleton-Muster komplett ohne Vererbung aus.

**Erzeugungsmuster** machen ein System unabhängig davon, wie seine Objekte erzeugt werden.



Neben den oben erwähnten Kategorien lassen sich noch weitere Kategorien finden. Als Beispiele werden genannt:

- **Muster für Nebenläufigkeit** (engl. **Concurrency Patterns**)  
Sie beschäftigen sich mit den spezifischen Entwurfsproblemen von nebenläufigen Programmen wie z. B. das Producer-Consumer-Pattern [Gr102] oder Rendezvous-Pattern (siehe [Dou02, 227ff.]).
- **Muster für die Persistenz** (engl. **Persistency Patterns**)  
Sie befassen sich mit dem persistenten Speichern von Objekten in Datenbanken wie z. B. das Identity Map-Pattern (siehe [Fow03]).

Mit der Anzahl der Entwurfsmuster steigt auch die Anzahl der Kategorien: wenn neue Muster veröffentlicht werden, gehören sie häufig zu neuen Spezialgebieten beim Entwurf, die bisher noch nicht durch Muster abgedeckt waren. Eine kurze Übersicht über die in diesem Buch ausgewählten Muster gibt nun das folgende Kapitel.

## 4.2 Übersicht über die vorgestellten Entwurfsmuster

In diesem Kapitel werden die in diesem Buch insgesamt vorgestellten Entwurfsmuster kurz beschrieben. Damit soll sich der Leser einen Überblick verschaffen können, um bei Bedarf gezielt nach einem Muster suchen zu können.

### 4.2.1 Strukturmuster

Die folgenden Strukturmuster wurden in dieses Buch aufgenommen:

- Das **Adapter-Muster** passt eine vorhandene "falsche" Schnittstelle an die gewünschte Form an. Der klassenbasierte Adapter leitet die zu adaptierende Klasse ab, um an die Schnittstelle der zu adaptierenden Klasse zu gelangen. Dadurch erbt der Adapter aber auch zusätzlichen "Ballast". Der objektbasierte Adapter aggregiert die zu adaptierende Klasse. Er ist wie im Falle des klassenbasierten Adapters speziell für die zu adaptierende Klasse geschrieben, erbt aber keinen "Ballast".
- Das **Brücke-Muster** trennt eine Implementierung und ihre Schnittstelle zum Client (Abstraktion) weitestgehend voneinander. Die Klassenhierarchie einer Abstraktion und einer Implementierung sind durch die sogenannte "Brücke" verbunden. Dadurch wird erreicht, dass beide Teile getrennt verändert und erweitert werden können.
- Das **Dekorierer-Muster** ermöglicht es, zur Laufzeit eine zusätzliche Funktionalität zu einem Objekt bzw. seinen Kindern in dynamischer Weise hinzuzufügen.

- Das **Fassade-Muster** stellt eine meist vereinfachte abstrakte Schnittstelle zum Zugriff auf die Klassen eines Subsystems bereit. Ein Fassadenobjekt delegiert Methodenaufrufe an Objekte der Klassen dieses Subsystems.
- Das **Kompositum-Muster** erlaubt es, dass bei der Verarbeitung von Elementen in einer Baumstruktur einfache und zusammengesetzte Objekte gleich behandelt werden.
- Das **Proxy-Muster** verbirgt die Existenz eines Objekts hinter einem Stellvertreterobjekt (Proxy) mit derselben Schnittstelle. Das Stellvertreterobjekt kapselt die Kommunikation zum echten Objekt. Es kann Funktionen an das echte Objekt weiterdelegieren und dabei auch Zusatzfunktionalität hinzufügen.

#### 4.2.2 Verhaltensmuster

Die folgenden Verhaltensmuster werden in späteren Kapiteln ausführlich beschrieben:

- Das Muster **Schablonenmethode** legt bereits in einer Basisklasse die Struktur eines Algorithmus fest. Realisiert werden variante Teile des Algorithmus in statisch definierten Unterklassen.
- Beim **Befehlsmuster** wird ein Befehl als Objekt gekapselt. Damit können die Erzeugung und die Ausführung eines Befehls getrennt werden. Befehle können erzeugt werden und zu einer späteren Zeit ausgeführt werden. Werden Befehle in einer Liste geführt, so können leicht Logging- oder Undo-Funktionen implementiert werden.
- Das **Beobachter-Muster** erlaubt es, dass ein Objekt von ihm abhängige Objekte von einer Änderung seines Zustands informiert, so dass ein abhängiges Objekt auf das entsprechende Ereignis reagieren und die geänderten Daten anfordern kann. Damit muss ein abhängiges Objekt nicht pollen (Inversion of Control).
- Mit Hilfe des **Strategie-Musters** kann ein ganzer Algorithmus (Strategie) gegen einen anderen funktional gleichwertigen Algorithmus dynamisch zur Laufzeit ausgetauscht werden. Das Muster zieht die alternativen Strategien heraus, abstrahiert sie durch ein Interface und kapselt sie in einem Objekt.
- Das **Vermittler-Muster** soll es erlauben, dass Objekte miteinander über einen Vermittler reden, der alle von einer bei ihm eingehenden Nachricht betroffenen Kollegen informiert. Dadurch bleiben die einzelnen Objekte unabhängig voneinander und sind austauschbar. Das Vermittler-Muster stellt die Koordination einer ganzen Gruppe von Objekten durch den zentralen Vermittler in den Vordergrund.
- Im **Zustandsmuster** werden die einzelnen Zustände durch eigene Klassen gekapselt, die von einer gemeinsamen abstrakten Basisklasse ableiten bzw. eine gemeinsame Schnittstelle implementieren. Ein zustandsabhängiges Objekt referenziert den aktuellen Zustand und führt Zustandsänderungen durch.
- Das **Muster Rolle** erlaubt es, dass ein betrachtetes Objekt in unterschiedlichen Kontexten zu unterschiedlichen Zeiten unterschiedliche Rollen annehmen kann. Auch können mehrere Objekte dieselbe Rolle spielen. Die Rollen werden als eigenständige Objekte realisiert.
- Das **Besucher-Muster** ermöglicht es, eine neue Operation zu einer bestehenden Datenstruktur hinzuzufügen, die auf den Objekten der Datenstruktur arbeitet, um ihre Funktion zu erbringen. Eine solche Operation wird in einem Besucherobjekt gekapselt. Beim Besucher-Muster müssen die zu besuchenden Objekte auf den Be-

such "vorbereitet" sein, dadurch dass sie eine entsprechende Methode dem Besucher zur Verfügung stellen.

- Mit Hilfe des **Iterator-Musters** kann eine aus Objekten zusammengesetzte Datenstruktur in verschiedenen Durchlaufstrategien durchlaufen werden, ohne dass der Client den Aufbau der Datenstruktur kennen muss.

### 4.2.3 Erzeugungsmuster

Die folgenden kurzen Zusammenfassungen beschreiben die einzelnen Erzeugungsmuster, die in diesem Buch vorgestellt werden:

- Das **Fabrikmethode-Muster** erlaubt es, dass die Erzeugung einer konkreten Instanz in der Methode einer Unterklasse gekapselt wird. Unterklassen legen bei diesem Muster die Struktur und das Verhalten der zu erzeugenden Objekte statisch fest.
- Beim Muster **Abstrakte Fabrik** kann durch Wahl der entsprechenden konkreten Fabrik eine Produktfamilie zur Laufzeit ausgewählt werden, aus der Objekte erzeugt werden sollen.
- Das **Singleton-Muster** stellt sicher, dass von einer Klasse nur ein einziges Objekt erzeugt werden kann. Dieses Muster arbeitet komplett ohne Vererbung.
- Das Muster **Objektpool** ermöglicht es, dass Objekte, deren Erzeugung oder Vernichtung aufwendig ist, nicht immer neu erzeugt und vernichtet werden müssen, sondern wiederverwendet werden können. Diese Objekte werden in einem Pool verwaltet und auf Anfrage den Anwendungen zur Verfügung gestellt. Damit sollen Ressourcen geschont werden.

### 4.2.4 Wegweiser durch die Entwurfsmuster

Für Neulinge im Umgang mit Entwurfsmustern sei erwähnt, dass die folgenden Muster besonders leicht zu verstehen sind und daher einen guten Einstieg in Entwurfsmuster bieten:

- Adapter,
- Fassade,
- Dekorierer,
- Kompositum,
- Schablonenmethode,
- Beobachter,
- Strategie,
- Singleton und
- Fabrikmethode.

Nach dem Erwerb der ersten Erfahrungen sollten dann die komplexeren Entwurfsmuster zur geeigneten Zeit durchgearbeitet werden.

## 4.3 Das Strukturmuster Adapter

### 4.3.1 Name/Alternative Namen

Adapter (engl. adapter).

### 4.3.2 Problem

Eine Klasse soll wiederverwendet werden. Die wiederzuverwendende Klasse bietet zwar die richtigen Daten an, hat aber eine unpassende Schnittstelle<sup>44</sup> für den Zugriff eines Clients auf diese Daten.

Das **Adapter-Muster** hat zum Ziel, eine vorhandene "falsche" Schnittstelle einer bereits vorhandenen Klasse an die vom Client gewünschte Form anzupassen.



Das Adapter-Muster wird in diesem Buch auf der Ebene von Klassen behandelt. Es kann aber auch auf der Ebene von Komponenten bzw. Subsystemen angewendet werden.

### 4.3.3 Lösung

Liegt bereits eine Klasse lauffähig vor, die den Anforderungen an die gesuchten Daten entspricht, aber allerdings eine inkompatible Schnittstelle für ihre Nutzer besitzt, wird man versuchen, auf dieser Klasse aufzusetzen und sie nicht neu zu entwickeln.

Das Adapter-Muster ist ein Entwurfsmuster, das dieses Anpassungsproblem behandelt. Der Adapter setzt auf der vorhandenen Klasse auf, verwendet sie weiter, ergänzt sie aber um ein "Übersetzungsprogramm" für die Methoden. Der Adapter bietet die gesuchten Methoden also nach außen an und ruft bei der vorhandenen Klasse aber deren Methoden auf. Das Adapter-Muster wird oft angewandt, wenn Klassen von Dritten, die nicht geändert werden sollen bzw. können – wie die Klassen einer Klassenbibliothek –, an eine Schnittstelle angepasst werden sollen.

Somit kann die vorhandene Klasse ohne eine Änderung weiter genutzt werden. Der Adapter wird einfach als Zusatzschicht eingezogen. Die Funktionalität des Adapters kann sich dabei von der Anpassung der Methodennamen bis hin zu komplexen Konvertierungen und Vereinfachungen erstrecken.

#### 4.3.3.1 Klassendiagramm

Im Fall des Adapter-Musters gibt es zwei Möglichkeiten, um einen Adapter aufzubauen:

<sup>44</sup> Als Beispiel aus dem täglichen Leben sei ein Adapter genannt, mit dem man einen deutschen Rasierapparat an eine amerikanische Steckdose anpasst. Um den Rasierapparat auch bei einer andersartigen Steckdose wiederverwenden zu können, bedarf es eines Adapters.

- einen klassenbasierten Adapter (im Folgenden kurz Klassen-Adapter genannt) und
- einen objektbasierten Adapter (kurz Objekt-Adapter genannt).

Der Klassen-Adapter ist zur Client-Klasse hin identisch aufgebaut wie der Objekt-Adapter. Beide Varianten werden nachfolgend vorgestellt:

### Variante 1: Klassen-Adapter (Adapter mit Vererbung)

Der Client benutzt die im Interface `IZiel` aufgelisteten Köpfe der Operationen. Die Klasse `Adapter` implementiert das Interface und leitet zugleich von der zu adaptierenden Klasse ab, um an die Schnittstelle der zu adaptierenden Klasse zu gelangen.

Die Klasse `Adapter` verwendet für den Aufruf der "alten" Operationen einfach eine Selbstdelegation an den ererbten Anteil. Die zu adaptierende Klasse stellt die anzupassende Komponente dar.



Die Klasse `Adapter` setzt beispielsweise den Aufruf einer Operation `operationA()` durch den Client in den Aufruf der von der zu adaptierenden Klasse geerbten Operation `operationB()` um. Der Client ruft in diesem Fall also indirekt die Operation `operationB()` auf, wenn er direkt beim Adapter die Operation `operationA()` aufruft. Innerhalb der Operation `operationA()` können Konvertierungen der Parameter oder des Rückgabewerts, aber auch komplexere Anpassungen vorgenommen werden.

Das folgende Bild 4-1 zeigt das Klassendiagramm des klassenbasierten Adapter-Musters:

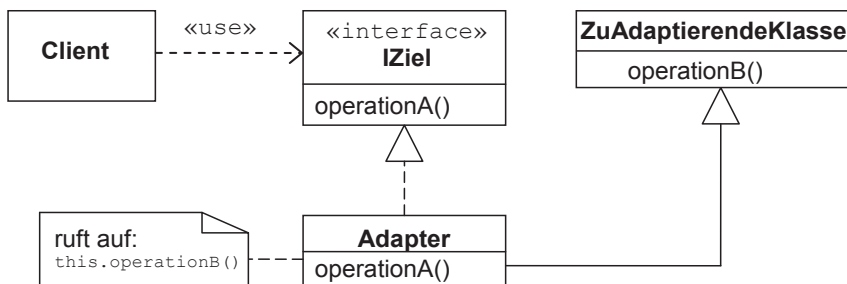


Bild 4-1 Klassendiagramm Klassen-Adapter

Zu bemerken ist, dass in dieser Lösung der Adapter alle Methoden – auch die der zu adaptierenden Klasse – anbietet, falls die verwendete Programmiersprache wie im Falle von Java keine private Vererbung zur Verfügung stellt. Im strengen Sinn handelt es sich wegen der übrigen geerbten Methoden der anzupassenden Klasse um keinen echten Adapter.





Die Funktionalität eines Adapters wird aber in den zusätzlichen Methoden zur Verfügung gestellt, die das Interface `IZiel` realisieren, hier in der Methode `operationA()`.

### Variante 2: Objekt-Adapter (Adapter mit Delegation)

Auch in dieser Variante stellt das Interface vom Typ `IZiel` die von der Client-Anwendung vorgegebene Aufrufschnittstelle dar. Die zu adaptierende Klasse wird mittels Aggregation dem Adapter bekannt gemacht.

Die Adapter-Klasse implementiert die Aufrufschnittstelle `IZiel` und delegiert den Aufruf an das aggregierte Objekt der zu adaptierenden Klasse weiter.



Wird die Operation `operationA()` beim Adapter (siehe Bild 4-2) aufgerufen, dann ruft diese wiederum in diesem Beispiel bei dem aggregierten Objekt der zu adaptierenden Klasse über die Referenz `ref` die Operation `operationB()` auf. Das Klassendiagramm des objektbasierten Adapter-Musters in Bild 4-2 veranschaulicht, wie der Aufruf einer Operation durch den Client an eine Operation mit anderem Namen weitergereicht wird:

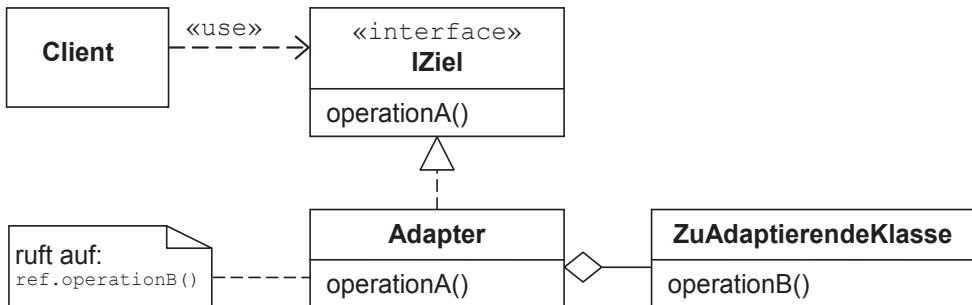


Bild 4-2 Klassendiagramm Objekt-Adapter

Die Adapter-Klasse in dieser Variante ist "schlanker" als die Adapter-Klasse in der Variante mit Vererbung, da sie nur die geforderte Schnittstelle implementiert und keinen geerbten "Ballast" in sich trägt.

Die weitere Beschreibung des Adapter-Musters basiert auf der in Bild 4-2 gezeigten Variante 2.

#### 4.3.3.2 Teilnehmer

Im Folgenden werden die Teilnehmer am Adapter-Muster aufgeführt:

### ZuAdaptierendeKlasse

Die Klasse `ZuAdaptierendeKlasse` bietet ihre Operationen mit Schnittstellen an, die nicht kompatibel sind mit den Schnittstellen, die der Client erwartet, um die Klasse benutzen zu können.

### Client

Der Client möchte die bereits vorhandene (zu adaptierende Klasse) nutzen, hat aber auf Grund seiner Anforderungen eine für die Nutzung der zu adaptierenden Klasse inkompatible Schnittstelle.

### IZiel

Das Interface `IZiel` entspricht den Anforderungen des Clients. Dieses Interface muss implementiert werden. Da das Interface `IZiel` nicht identisch mit der Schnittstelle der Klasse `ZuAdaptierendeKlasse` ist, wird ein Adapter benötigt.

### Adapter

Die Klasse `Adapter` übernimmt die Anpassung an die vom Client benötigte Schnittstelle, indem sie die Schnittstelle des Clients auf die Schnittstelle der zu adaptierenden Klasse abbildet. Ein Objekt der Klasse `Adapter` ermöglicht die Kommunikation zwischen dem Client und einem Objekt der zu adaptierenden Klasse.

#### 4.3.3.3 Dynamisches Verhalten

Das Client-Objekt ruft eine Operation des Adapter-Objekts auf. Dieses leitet den Aufruf an das Objekt der zu adaptierenden Klasse weiter, indem es eine Operation der zu adaptierenden Klasse aufruft, die das Richtige tut, aber eine falsche Schnittstelle hat. Das Ergebnis wird von dem Objekt der zu adaptierenden Klasse über das Adapter-Objekt an das Client-Objekt zurückgegeben. Dies zeigt das folgende Sequenzdiagramm:

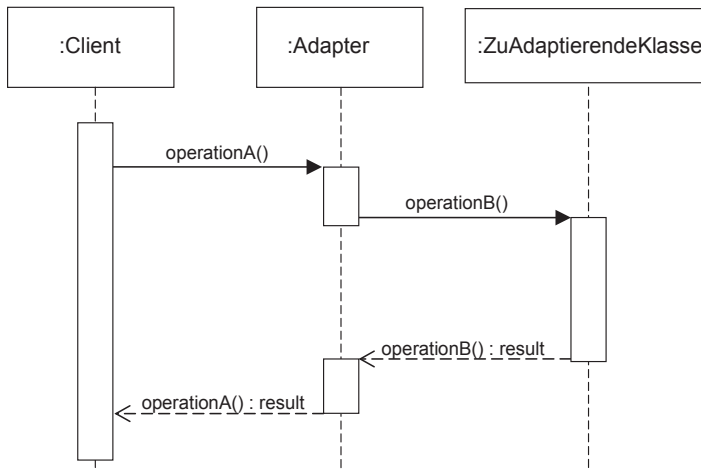


Bild 4-3 Sequenzdiagramm des Adapter-Musters

#### 4.3.3.4 Programmbeispiel

Als Beispiel soll eine kleine Anwendung dienen, die Personendaten aus einer CSV-Datei<sup>45</sup> ausliest. Das Einlesen von CSV-Dateien aus einer Datei in einen Zwischenpuffer sei in diesem Beispiel schon von einer Klasse implementiert worden, die vor längerer Zeit geschrieben wurde und deshalb nicht auf die Architektur der Anwendung, d. h. auf die Aufrufchnittstelle des Clients, abgestimmt ist. Um die Klasse zur Anwendung kompatibel zu machen, wird eine Adapter-Klasse eingesetzt. Hierbei stellt die Klasse `CSVLeser` die zu adaptierende Klasse dar, die Klasse `CSVLeserAdapter` den Adapter und die Klasse `TestAdapter` den Client. Die Klasse `CSVLeserAdapter` muss das Interface `IPersonenLeser` implementieren.

Zur Speicherung von Vor- und Nachname dient die Klasse `Person`. Die Klasse `Person` ist sozusagen eine Hilfsklasse und spielt im Adapter-Muster selbst keine Rolle. Hier die Klasse `Person`:

```
// Datei: Person.java
public class Person
{
    private String nachname;
    private String vorname;

    public Person (String nachname, String vorname)
    {
        this.nachname = nachname;
        this.vorname = vorname;
    }

    public void print()
    {
        System.out.println (vorname + " " + nachname);
    }
}
```

Die Klasse `CSVLeser` stellt eine vor längerer Zeit implementierte Klasse dar, also die zu adaptierende Klasse, die eine CSV-Datei mit Personendaten aus einer Datei in einen Zwischenpuffer einliest. Im Gegensatz zum Interface `IPersonenLeser` hat die Methode zum Einlesen der Personen einen anderen Namen und gibt kein Array vom Typ `Vector` mit Personen zurück, sondern ein Array vom Typ `Vector`, der ein `String`-Array mit den Vor- und Nachnamen enthält. Da diese Implementierung nicht kompatibel zum Interface `IPersonenLeser` ist, muss sie adaptiert werden. Hier die Klasse `CSVLeser`:

```
// Datei: CSVLeser.java
import java.io.*;
import java.util.Vector;
```

---

<sup>45</sup> CSV steht für Comma-Separated Values, d. h. Trennung von Spaltenwerten durch Kommata. In der Praxis werden alle Dateien, in denen die Spalten durch ein Trennzeichen separiert gespeichert sind, als CSV-Dateien bezeichnet. Das Trennzeichen muss also nicht unbedingt ein Komma sein.

```

public class CSVLeser
{
    // hat als Rückgabewert einen Vektor vom Typ eines String-Arrays
    public Vector<String []> lesePersonenDatei (String file)
    {
        Vector<String []> personen = new Vector<String []>();
        try
        {
            BufferedReader input =
                new BufferedReader (new FileReader (file));
            String strLine;

            while ((strLine = input.readLine()) != null)
            {
                String[] splitted = strLine.split ("," );
                if (splitted.length >= 2)
                    personen.add (new String []
                                {splitted[0], splitted[1]});
            }
            input.close();
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
        return personen;
    }
}

```

Das Interface `IPersonenLeser` dient als Schnittstelle für alle Klassen, die Personen-daten einlesen können. Es definiert die Methode `lesePersonen()`, die eine Liste von Personen zurückgibt. Hier das Interface `IPersonenLeser`:

```

// Datei: IPersonenLeser.java
import java.util.Vector;
public interface IPersonenLeser
{
    // hat als Rückgabewert einen Vektor von Personen
    public Vector<Person> lesePersonen();
}

```

Die Klasse `CSVLeserAdapter` dient als Adapter für die Klasse `CSVLeser`. Sie implementiert das Interface `IPersonenLeser` und delegiert das Einlesen der Personen aus der CSV-Datei an ein aggregiertes Objekt der Klasse `CSVLeser`. Die erhaltenen Daten werden anschließend so aufbereitet, dass sie der Schnittstelle genügen. Hier die Klasse `CSVLeserAdapter`:

```

// Datei: CSVLeserAdapter.java
import java.util.Vector;

public class CSVLeserAdapter implements IPersonenLeser
{
    private String file;

```

```
public CSVLeserAdapter (String file)
{
    this.file = file;
}

public Vector<Person> lesePersonen()
{
    CSVLeser leser = new CSVLeser();
    Vector<String []> gelesenePersonen =
        leser.lesePersonenDatei (file);
    Vector<Person> personenVector = new Vector<Person>();

    for (String [] person : gelesenePersonen)
        personenVector.add (new Person (person [0], person [1]));
    return personenVector;
}
}
```

Die Klasse `TestAdapter` stellt in diesem Beispiel den Client dar. Sie liest eine CSV-Datei mit Personendaten über den Adapter ein und gibt anschließend die eingelesenen Personen am Bildschirm aus. Im Folgenden die Klasse `TestAdapter`:

// Datei: `TestAdapter.java`

```
import java.util.Vector;

public class TestAdapter
{
    public static void main (String[] args)
    {
        IPersonenLeser leser = new CSVLeserAdapter ("Personen.csv");
        Vector<Person> personen = leser.lesePersonen();

        for (Person person : personen)
            person.print();
    }
}
```

Für den Test des Adapters dient die Datei `Personen.csv` (auf dem begleitenden Webauftritt verfügbar). Sie enthält die Personen-Datensätze durch Kommata getrennt.



Hier das Protokoll des Programmablaufs:

Heinz Mueller  
Volker Schmied  
Hannah Schneider

#### 4.3.4 Bewertung

Der Klassen-Adapter basiert auf Vererbung. Er kann somit nur diejenigen Klasse adaptieren, von der er abgeleitet ist. Ein Objekt-Adapter kann auf Grund des liskov-schen Substitutionsprinzips auch Unterklassen anpassen, da er eine Referenz auf ein

Objekt der zu adaptierenden (Basis-) Klasse besitzt. Allerdings kann er dabei keine erweiternden Methoden einer Unterklasse nutzen.

#### 4.3.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Das Adapter-Muster ermöglicht die Kommunikation zwischen zwei unabhängigen Softwarekomponenten, nämlich zwischen dem Client und einem Objekt der zu adaptierenden Klasse.
- Adapter können um beliebig viele Funktionen wie z. B. Filter erweitert werden.
- Adapter sind individuell an die jeweilige Lösung angepasst und können daher optimiert werden.
- Klassen können leicht ausgetauscht werden. Gegebenenfalls muss nur ein neuer Adapter zur Verfügung gestellt werden.
- Ein Objekt-Adapter kann auch auf ein Objekt einer Unterklasse der zu adaptierenden Klasse angewandt werden. Erweiternde Methoden der Unterklasse kann er dabei allerdings nicht nutzen.

#### 4.3.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Durch das Adapter-Muster wird beim Aufruf einer Operation ein zusätzlicher Zwischenschritt eingeführt, im Falle des Objekt-Adapters eine Delegation. Dies kann bei komplexen Adaptern zu zeitlichen Verzögerungen führen.
- Durch die individuelle Anpassung der Adapter an die jeweilige Lösung weisen die Adapter eine schlechte Wiederverwendbarkeit auf.

#### 4.3.5 Einsatzgebiete

Das Adapter-Muster ermöglicht die Zusammenarbeit von Klassen mit inkompatiblen Schnittstellen. Es wird in der Regel dazu verwendet, um unabhängig implementierte Klassen nachträglich zusammenarbeiten zu lassen. Die Anfragen werden dann nur noch über die Adapter-Klasse an die Klasse mit der inkompatiblen Schnittstelle delegiert.

Das Adapter-Muster ist einzusetzen, wenn:

- bereits bestehende Klassen mit unterschiedlichen Schnittstellen zusammenarbeiten sollen, ohne die Klassen zu überarbeiten, oder
- eine zu entwickelnde Klasse wiederverwendbar sein soll und zum Zeitpunkt der Entwicklung nicht klar ist, mit welchen weiteren Klassen sie zusammenarbeiten soll.

Das Adapter-Muster ist praktisch in jeder API verborgen. Ein Beispiel für das Entwurfsmuster Adapter sind Anwendungen, die für eine bestimmte grafische Oberfläche implementiert wurden und nun auf eine neue Plattform portiert werden sollen. Dabei stim-

men die Schnittstellen der bereits implementierten Anwendung und die Schnittstellen der "neuen" Plattform nicht überein, so dass ein Adapter dazwischen benötigt wird.

#### 4.3.6 Ähnliche Entwurfsmuster

**Brücke** und **Adapter** sind sich ähnlich. Bei beiden versteckt eine Schnittstelle die konkrete Implementierung. Die Brücke trennt gezielt beim Entwurf eine Schnittstelle (Abstraktion) und ihre Implementierung. Die Referenz der Abstraktion auf die Implementierung kann zur Laufzeit geändert werden. Beim Adapter hingegen soll eine existierende Schnittstelle an eine neue Schnittstelle angepasst werden. Der Adapter verbindet unabhängige, oft bereits fertig implementierte Klassen. Ein Adapter wird also meist erst nach der Entwicklung eines Systems eingesetzt, die Brücke muss beim Entwurf eines Systems bereits eingeplant werden.

Beim **Fassade-Muster** und beim **Adapter-Muster** werden vorhandene Programme gekapselt. Fassade und Adapter sind Wrapper. Im Gegensatz zur Fassadenschnittstelle, die von einem Subsystem definiert wird, wird die Adapterschnittstelle vom Client vorgegeben. Hinter der Fassade verbergen sich Subsystemklassen, hinter dem Adapter verbirgt sich nur eine einzige Klasse. Für die Fassadenschnittstelle gibt es keine Vorgaben von außerhalb des Subsystems. Sie kann vom Subsystem frei festgelegt werden. Generell gesprochen muss die Schnittstelle durch die Verwendung einer Fassade einheitlich werden. Sie wird aber meist vereinfacht. Der Adapter dient zur Anpassung einer Schnittstelle an die für den Client erforderliche Form.

Ein **Proxy** verwendet dieselbe Schnittstelle wie ein Objekt einer echten Klasse, das er repräsentiert. Die Schnittstelle darf hier also nicht verändert werden. Für den Adapter gilt diese Einschränkung nicht. Im Gegenteil, der Adapter passt eine vorhandene Schnittstelle an die gewünschte Form der Schnittstelle an.

## 4.4 Das Strukturmuster Brücke

### 4.4.1 Name/Alternative Namen

Brücke (engl. bridge), teilweise auch Verwendung des Namens Handle/Body.

### 4.4.2 Problem

Üblicherweise ist eine Abstraktion mit ihrer Implementierung statisch über eine Vererbung verknüpft: Man definiert in einer abstrakten Basisklasse die Schnittstelle von Methoden mittels Methodenköpfen und implementiert die Methoden in Subklassen, die von der abstrakten Basisklasse abgeleitet werden. Die Implementierung der abgeleiteten Klassen ist damit auf die Realisierung der Schnittstelle der abstrakten Basisklasse ausgerichtet. Dies gilt auch, wenn als Abstraktion ein Interface verwendet wird und die Implementierung über eine Realisierungsbeziehung erfolgt.

Eine solche statische Verknüpfung über eine Vererbungs- oder Realisierungsbeziehung ist aber oft nicht flexibel genug. Eine unabhängige Weiterentwicklung von Abstraktion und Implementierung ist dabei gar nicht möglich. Abstraktion und Implementierung sollen deshalb weitestgehend voneinander getrennt werden, damit sich beide Seiten unabhängig voneinander entwickeln können, ohne jeweils die andere Seite zu beeinflussen. Eine spezielle Abstraktion soll also nicht von der konkreten Implementierung abhängen.

Das Brücke-Muster hat zum Ziel, dass sich eine Abstraktion und ihre Implementierung unabhängig voneinander entwickeln können.



### 4.4.3 Lösung

Abstraktion und Implementierung werden weitestgehend voneinander getrennt und befinden sich jeweils in einer eigenen Klassenhierarchie.

Alle Operationen einer Abstraktionsklasse werden auf der Basis der abstrakten Operationen des Interface `IImplementierer` realisiert [Gam95]. Dieses Interface wird von der Abstraktion vorgegeben. Damit sind die Abstraktionen nicht von der konkreten Implementierung abhängig (**Dependency Inversion**).

Würde eine Abstraktion die Implementierung direkt ohne Vorgabe eines Interface aufrufen, würden Änderungen der Implementierung voll auf die Abstraktion durchschlagen. Die Abstraktion wäre von der Implementierung abhängig. Durch die Einführung eines Interface, welches die Abstraktion vorgibt, ist eine Abstraktion von ihrer konkreten Implementierung unabhängig (**Dependency Inversion**).



Beim Brücke-Muster können sich die Abstraktion und die Implementierung unter Einhaltung der von der Abstraktion vorgegebenen Schnittstelle für den Implementierer unabhängig voneinander verändern.



Die Verwendung des Brücke-Musters ermöglicht es, die Implementierung vor dem Client zu verstecken. Aus der Sicht des Clients ist das Interface `IImplementierer` vollständig verborgen, lediglich die Klasse `Abstraktion` bzw. `SpezAbstraktion` ist dem Client bekannt. Die Methoden einer Abstraktion sind unabhängig von einer konkreten Implementierung. Bleibt beispielsweise die Abstraktionsschnittstelle dieselbe und wird nur die Implementierung grundlegend verändert, so sind keine Änderungen am Client erforderlich.

Ein konkreter Implementierer ist vollständig vor dem Aufrufer (Client) versteckt. Die Abstraktion ist nur von der Schnittstelle des Implementierers, die die Abstraktion selbst vorgibt, aber nicht von der konkreten Implementierung abhängig.



#### 4.4.3.1 Klassendiagramm

Der Implementierer wurde in dem folgenden Beispiel als Interface realisiert. Der Implementierer kann aber auch als abstrakte Klasse oder als normale Klasse realisiert werden. Das Muster gibt hierzu keine Vorgabe. Ebenso kann anstelle der Klasse `Abstraktion` eine abstrakte Klasse verwendet werden. Das folgende Bild zeigt das Klassendiagramm des Brücke-Musters:

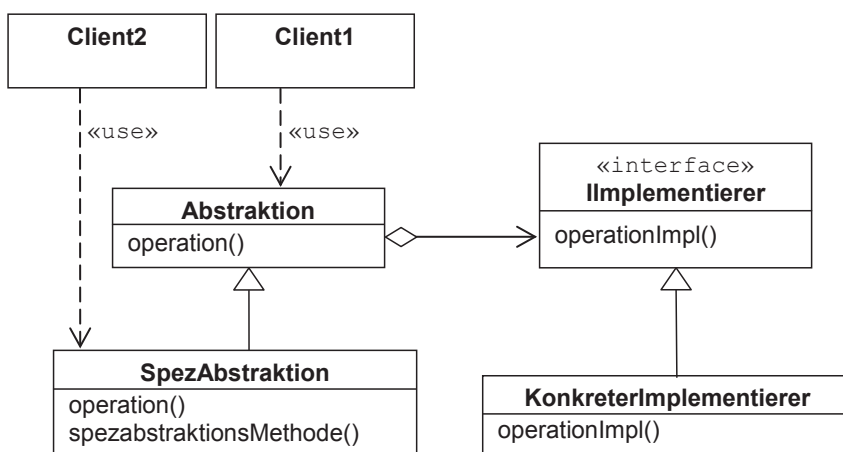


Bild 4-4 Klassendiagramm der Brücke mit dem Implementierer als Interface

Die Beziehung zwischen Abstraktion und Implementierung wird **Brücke** genannt, weil sie eine Brücke zwischen Abstraktion und Implementierung bildet, wobei beide sich in einer eigenen Spezialisierungshierarchie entwickeln können.



Die Brücke verbindet beide Seiten, d. h. beide Klassenhierarchien, miteinander. Die Methoden einer Abstraktion werden unter Verwendung der Methoden des Interface `IImplementierer` (siehe Bild 4-4) realisiert. Das Brücke-Muster legt nicht fest, wie die Implementierungsobjekte erzeugt werden. Beispielsweise kann eine abstrakte Fabrik (siehe Kapitel 4.19) eingesetzt werden, um zueinander passende Implementierungs- und Abstraktionsobjekte zu erzeugen.

Über die Brücke – also über die Aggregation – kann eine Abstraktion mit ganz verschiedenen Implementierungen arbeiten. Die Abstraktion kann ebenfalls weiterentwickelt werden.



Der konkrete Implementierer in Bild 4-4 erbt nicht von der Abstraktion, da dies eine starke Kopplung bedeuten würde. Für den Client muss ein Objekt der Klasse `Abstraktion` sichtbar sein. Auch eine möglicherweise weiterentwickelte Abstraktionsklasse (`SpezAbstraktion`) muss für einen Client sichtbar sein, wenn er diese direkt aufrufen will.

Das Interface `IImplementierer` und die konkreten Implementierer sollten für den Client nicht zu erreichen sein. Ansonsten bestünde die Gefahr, dass sie direkt vom Client angesprochen werden könnten. Eine solche direkte Verwendung würde alle Vorteile der Trennung wieder zunichtemachen.

#### 4.4.3.2 Teilnehmer

##### Client

Ein Client ruft die Klasse `Abstraktion` bzw. die Klasse `SpezAbstraktion` auf.

##### Abstraktion

Die Klasse `Abstraktion` definiert die Schnittstelle, über die ein Client auf die Funktionalität der Klasse `Abstraktion` zugreifen kann. Zusätzlich enthält die Klasse `Abstraktion` eine Referenz auf das von ihr selbst vorgegebene Interface `IImplementierer`.

##### SpezAbstraktion

Diese Klasse leitet von der Klasse `Abstraktion` ab und kann deren Schnittstelle erweitern. Auch ein Objekt der Klasse `SpezAbstraktion` kann von einem Client aufgerufen werden. Tritt ein Objekt der Klasse `SpezAbstraktion` bei einem Client aber nach dem liskovschen Substitutionsprinzip an die Stelle eines Objekts der Klasse `Abstraktion`, so kann der Client nur die Methoden der Klasse `Abstraktion` benutzen.

### Implementierer

Das Interface `IImplementierer` definiert die Schnittstelle einer konkreten Implementierung. Das Interface `IImplementierer` wird von der Klasse `Abstraktion` vorgegeben. Die Methoden des Interface `IImplementierer` müssen also die Erwartungen der Methoden der Klasse `Abstraktion` erfüllen. Das Interface `IImplementierer` bietet Operationen an, die zur Realisierung von Operationen der Klasse `Abstraktion` dienen.

### KonkreterImplementierer

Letztendlich muss das Interface `IImplementierer` für den jeweils vorliegenden konkreten Fall implementiert werden. Diese Aufgabe übernimmt die Klasse `KonkreterImplementierer`.

#### 4.4.3.3 Dynamisches Verhalten

Das folgende Sequenzdiagramm veranschaulicht den Mechanismus einer Brücke:

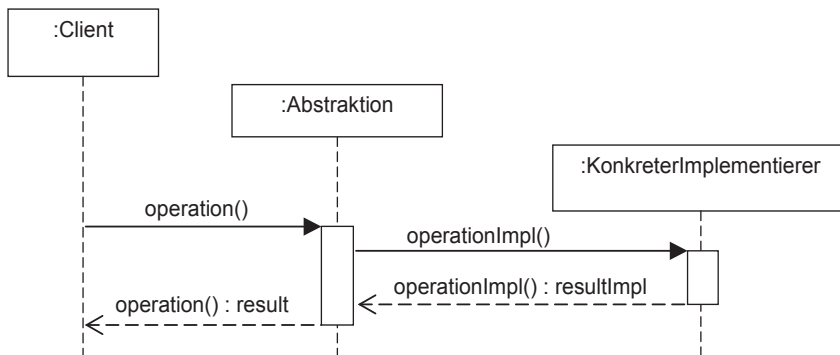


Bild 4-5 Sequenzdiagramm des Brücke-Musters

Alle Methodenaufrufe des Clients werden hier über die von der Klasse `Abstraktion` bereitgestellte Schnittstelle durchgeführt. Von dort werden sie an das Objekt der Klasse `KonkreterImplementierer` weitergeleitet. Der Client darf die konkrete Implementierung nicht kennen.

#### 4.4.3.4 Programmbeispiel

Das Muster soll am Beispiel einer Musikanlage ausprogrammiert werden. Dazu stelle man sich eine Musikanlage vor, die aus einem CD-Spieler und einem Kassettenspieler besteht. Die Benutzung der beiden Geräte wird in einer Klasse `Abspielgeraet` abstrahiert. Diese Klasse erlaubt es, ein Lied abzuspielen und die Anlage auszuschalten. Eine erweiterte Nutzung wird durch die Klasse `ListenAbspielgeraet` bereitgestellt. Sie bietet einen Modus zum Einschlafen, der es ermöglicht, eine Abfolge von Liedern zu spielen und die Anlage anschließend auszuschalten.

Bild 4-6 zeigt das Klassendiagramm dieses Beispiels, aus dem auch die Rollen der beteiligten Klassen und Interfaces im Rahmen des Brücke-Musters ersichtlich werden:

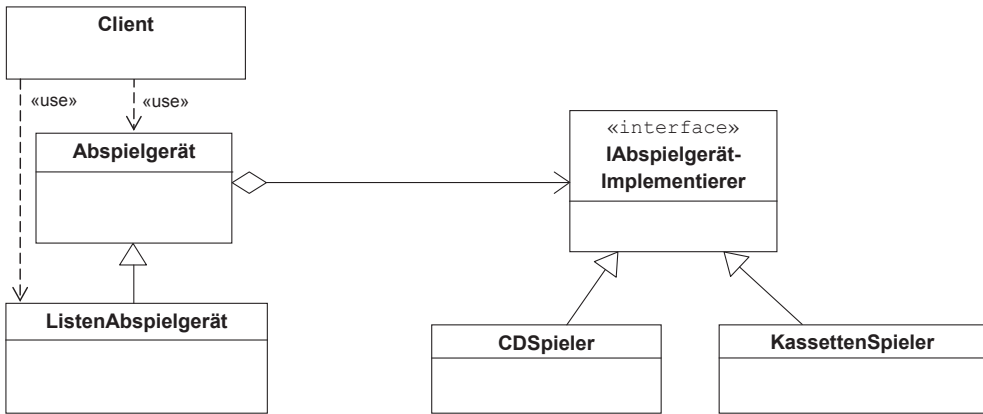


Bild 4-6 Klassendiagramm für das Beispiel

Die Klasse `AbspielDaten` ist eine Hilfsklasse, die die abzuspielenden Daten enthält. Sie gehört nicht zum Brücke-Muster und wird deswegen in Bild 4-6 nicht gezeigt. Hier die Klasse `AbspielDaten`:

```
// Datei: AbspielDaten.java
public class AbspielDaten
{
    private String daten;

    public AbspielDaten (String daten)
    {
        this.daten = daten;
    }

    public String toString()
    {
        return daten;
    }
}
```

Die Klasse `Abspielgeraet` dient zur Abstraktion gegenüber dem Client:

```
// Datei: Abspielgeraet.java
// Abstraktion
public class Abspielgeraet
{
    protected IAbspielgeraetImplementierer impl;

    public Abspielgeraet (IAbspielgeraetImplementierer impl)
    {
        this.impl = impl;
    }
}
```

```
public void spieleAb (int liedNummer)
{
    impl.springeZuTrack (liedNummer);
    AbspielDaten dat = impl leseDaten();
    // Daten ausgeben
    System.out.println (dat);
}

public void ausschalten()
{
    impl.ausschalten();
}
}
```

Um das Gerät auszuschalten, nachdem eine Liste von Liedern abgespielt wurde, dient die spezialisierte Klasse `ListenAbspielgeraet`:

```
// Datei: ListenAbspielgeraet.java
import java.util.ArrayList;

// Spez. Abstraktion
public class ListenAbspielgeraet extends Abspielgeraet
{
    private ArrayList<Integer> liste;

    public ListenAbspielgeraet (IAbspielgeraetImplementierer impl,
                                ArrayList<Integer> liste)
    {
        super (impl);
        this.liste = liste;
    }

    public void abspielenUndAusschalten() // neue Funktion
    {
        for (int i : liste)
        {
            impl.springeZuTrack (i);
            AbspielDaten daten = impl leseDaten();
            System.out.println (daten);
        }
        impl.ausschalten();
    }
}
```

Das Interface `IAbspielgeraetImplementierer` definiert Methoden zum Abspielen und Ausschalten:

```
// Datei: IAbspielgeraetImplementierer.java
// Implementierer als Interface
public interface IAbspielgeraetImplementierer
{
    // springt zu Track liedNummer
    public void springeZuTrack (int liedNummer);
}
```

```

    // liest die Daten zum Abspielen...
    public AbspielDaten leseDaten();

    // schaltet das Geraet aus...
    public void ausschalten();
}

```

Die Klasse `CDSpieler` stellt die Funktionalität für das Abspielen von CDs bereit:

```

// Datei: CDSpieler.java
// Konkreter Implementierer
public class CDSpieler implements IAbspielgeraetImplementierer
{
    public void springeZuTrack (int liedNummer)
    {
        // Inhaltsverzeichnis der CD durchsuchen...
        System.out.println ("Durchsuche Inhaltsverzeichnis");
        // Springe auf der CD an den Beginn des Lieds...
        System.out.println ("Springe zu Lied " + liedNummer +
                           " durch Positionierung des Lasers.");
    }

    public AbspielDaten leseDaten()
    {
        return new AbspielDaten ("CD-Daten");
    }

    public void ausschalten()
    {
        System.out.println ("CD-Spieler ausgeschaltet.");
    }
}

```

Für das Abspielen von Kassetten dient die Klasse `KassettenSpieler`. Da bei Kassetten nicht direkt an die Stelle gesprungen werden kann, an der sich ein Lied befindet, muss die Kassette vorgespult werden. Sobald das Gerät eine Pause erkennt, wird der Liedzähler `currentTrack` erhöht. Dies wird im Beispielprogramm der Einfachheit halber nur in Grundzügen realisiert. Hier die Klasse `KassettenSpieler`:

```

// Datei: KassettenSpieler.java
// KonkreterImplementierer
public class KassettenSpieler implements IAbspielgeraetImplementierer
{
    private int currentTrack;

    public KassettenSpieler()
    {
        System.out.println("Spule an Anfang zurueck");
        currentTrack = 1;
    }

    public void springeZuTrack (int liedNummer)
    {
        int diff = liedNummer - currentTrack;
    }
}

```

```

        if (diff > 0 )
        {
            System.out.println ("Spule um " + diff +
                                " Tracks vor.");
        }
        else
        {
            System.out.println ("Spule um " + (Math.abs (diff)+1) +
                                " Tracks zurueck.");
        }

        System.out.println ("Nun sind wir an der richtigen " +
                            "Stelle (Lied: " + liedNummer + ")");
        currentTrack = liedNummer;
    }

    public AbspielDaten leseDaten()
    {
        return new AbspielDaten ("Kassettdaten");
    }

    public void ausschalten()
    {
        System.out.println ("Kassettenspieler abgeschalten.");
    }
}

```

Der Client nutzt nur die Funktionen der Abstraktionsklassen `Abspielgeraet` und `ListenAbspielgeraet` zur Steuerung eines Abspielvorgangs. Er kennt insofern keine Implementierungsobjekte. Wie bereits in Kapitel 4.4.3.3 erwähnt wurde, sagt das Muster nichts über die Erzeugung der Implementierungsobjekte aus. Um das Beispiel übersichtlich zu halten, wurde ein einfacher Weg gewählt: Das Hauptprogramm in der Klasse `Client` erzeugt konkrete Implementierungsinstanzen vom Typ `CDSpieler` und `KassettenSpieler`. Diese werden dann den Objekten der Abstraktionsklassen `Abspielgeraet` und `ListenAbspielgeraet` übergeben. Hier der Programmcode des Clients:

**// Datei: Client.java**

```
import java.util.ArrayList;
```

```

public class Client
{
    public static void main (String[] args)
    {
        Abspielgeraet a = new Abspielgeraet (new CDSpieler());
        a.spieleAb (3);

        Abspielgeraet b = new Abspielgeraet (new KassettenSpieler());
        b.spieleAb (5);

        ArrayList<Integer> abspielliste = new ArrayList<Integer>();
        abspielliste.add (1);
        abspielliste.add (9);
        abspielliste.add (3);
    }
}

```

```

    ListenAbspielgeraet l = new ListenAbspielgeraet(new CDSpieler(),
                                                    abspielliste);
    // Dieser Client nutzt sowohl die Abstraktion als auch
    // die spezielle Abstraktion.
    l.abspielenUndAusschalten();
}
}

```

Das Beispielprogramm spielt im CD-Spieler Lied 3 ab, dann im Kassettendeck Lied 5. Abschließend werden Lied 1, 9 und 3 der eingelegten CD abgespielt und der CD-Spieler ausgeschaltet.



Hier das Protokoll des Programmlaufs:

```

Durchsuche Inhaltsverzeichnis
Springe zu Lied 3 durch Positionierung des Lasers.
CD-Daten
Spule an Anfang zurück
Spule um 4 Tracks vor.
Nun sind wir an der richtigen Stelle (Lied: 5)
Kassettendaten
Durchsuche Inhaltsverzeichnis
Springe zu Lied 1 durch Positionierung des Lasers.
CD-Daten
Durchsuche Inhaltsverzeichnis
Springe zu Lied 9 durch Positionierung des Lasers.
CD-Daten
Durchsuche Inhaltsverzeichnis
Springe zu Lied 3 durch Positionierung des Lasers.
CD-Daten
CD-Spieler ausgeschaltet.

```

#### 4.4.4 Bewertung

##### 4.4.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Eine Abstraktion und eine konkrete Implementierung sind nur über das von der Abstraktion vorgegebene Interface einer konkreten Implementierung gekoppelt. Damit ist eine Abstraktion nicht von der konkreten Implementierung abhängig (Dependency Inversion).
- Abstraktionen und Implementierungen können unabhängig voneinander in neuen Unterklassen weiterentwickelt werden. Neue Abstraktionen und neue Implementierungen können sehr einfach hinzugefügt werden.
- Eine konkrete Implementierung ist vor dem Client verborgen. Wenn die Abstraktion gleich bleibt und nicht verändert wird, dann kann die konkrete Implementierung sogar zur Laufzeit ausgetauscht werden. Dies hat den Vorteil, dass der Client nicht



angepasst werden muss. Die Abstraktionsklassen und der Client müssen daher nicht neu kompiliert werden.

- Es ist möglich, für komplett verschiedene Abstraktionen dieselben Implementierungsklassen zu verwenden.

#### 4.4.4.2 Nachteile

Der folgende Nachteil wird gesehen:

- Beim Brücke-Muster sind relativ viele Klassen an der Umsetzung beteiligt. Dies erschwert die Übersichtlichkeit. Bei der Planung und Entwicklung einer Softwareapplikation erhöht sich daher auch der zeitliche Aufwand, der für die Umsetzung benötigt wird.

#### 4.4.5 Einsatzgebiete

Die Brücke ist ein Muster, das eingesetzt wird, um eine Abstraktion unabhängig von der Implementierung zu machen. Dies ermöglicht es, eine Änderung an einer Implementierung durchzuführen, ohne anschließend die Abstraktionsklassen ändern zu müssen. Ebenso kann die Abstraktion weiterentwickelt werden, ohne die Implementierung zu ändern.

Das Brücke-Muster wird beispielsweise bei der Abstraktion verschiedener Datenbankschnittstellen wie etwa für Oracle, MySQL etc. angewandt oder bei der Erstellung plattformübergreifender grafischer Benutzeroberflächen.

#### 4.4.6 Ähnliche Entwurfsmuster

Brücke und **Adapter** sind sich ähnlich. Bei beiden versteckt ein Interface die konkrete Implementierung. Während das Brücke-Muster verwendet wird, um bereits beim Klassendesign die Implementierung von der Abstraktion zu separieren, wird der Adapter zur nachträglichen Anpassung der bereits bestehenden Schnittstelle einer zu adaptierenden Klasse an die vom Client geforderte Form verwendet. Der Adapter verbindet unabhängige, bereits fertig implementierte Klassen. Bei der Brücke sollen Abstraktion und Implementierung unabhängig weiter entwickelt werden können. Der Adapter wird in der Regel erst nach der Entwicklung eines Systems eingesetzt, die Brücke muss bei der Entwicklung eines Systems rechtzeitig eingeplant werden.

## 4.5 Das Strukturmuster Dekorierer

### 4.5.1 Name/Alternative Namen

Dekorierer (engl. decorator).

### 4.5.2 Problem

Es sollen Objekte einer Klassenhierarchie **dynamisch zur Laufzeit** zusätzliche Funktionalität erhalten.

Das **Dekorierer-Muster** soll es erlauben, zur Laufzeit eine zusätzliche Funktionalität zu einem vorhandenen Objekt in dynamischer Weise hinzuzufügen.



### 4.5.3 Lösung

Das Dekorierer-Muster ist ein objektbasiertes Entwurfsmuster. Die zu erweiternden Objekte gehören alle einer Klassenhierarchie an, an deren Spitze eine Basisklasse steht. Die Basisklasse wird im Folgenden auch als Komponente bezeichnet.

Der Lösungsansatz des Dekorierer-Musters ist es, in einem Dekorierer das zu verzierende, sprich zu erweiternde Objekt zu aggregieren und gleichzeitig einem Kunden, dem Client, dieselbe Schnittstelle wie die zu verzierende Komponente anzubieten.



Dazu leitet man die Klasse `Dekorierer` von der Klasse `Komponente` ab. Nach dem liskovschen Substitutionsprinzip kann dann im Programm ein Objekt der Klasse `Dekorierer` an die Stelle eines Objekts der Klasse `Komponente` treten, solange der Vertrag der Klasse `Komponente` mit dem Client nicht gebrochen wird. Aufrufe, die nicht verändert werden sollen, werden einfach vom Objekt der Klasse `Dekorierer` an die aggregierte Komponente weitergeleitet. Methodenaufrufe, die dekoriert werden sollen, werden überschrieben. Die überschreibende Methode enthält das Anbringen der Verzierung und den Aufruf der ursprünglichen Methode der aggregierten Komponente nach dem Delegationsprinzip, um die ursprüngliche Funktionalität sicherzustellen. Mit dem Verzierungsanteil darf die Klasse `Dekorierer` aber den Vertrag der überschriebenen Methode nicht brechen.

Das zu dekorierende Objekt merkt dabei gar nicht, dass es dekoriert wird. An ihm muss keine Veränderung vorgenommen werden.



#### 4.5.3.1 Klassendiagramm

Die Klasse `Komponente` ist die Basisklasse aller zu dekorierenden Objekte und legt deren Schnittstelle fest. Im folgenden Klassendiagramm sollen beispielsweise Objekte der Klassen `KonkreteKomponente1` oder `KonkreteKomponente2` dekoriert werden. Hier das Klassendiagramm:

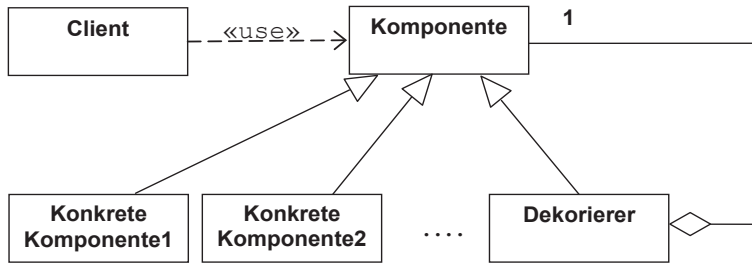


Bild 4-7 Klassendiagramm des Dekorierer-Musters

Die Klasse `Dekorierer` ist ebenfalls von der Klasse `Komponente` abgeleitet und aggregiert gleichzeitig genau ein Objekt vom Typ `Komponente`. Bei Einhaltung des liskovschen Substitutionsprinzips kann aber auch ein Objekt jeder beliebigen Subklasse (`KonkreteKomponente1`, `KonkreteKomponente2`, ...) der Klasse `Komponente` aggregiert und dekoriert werden.

Die von der Klasse `Komponente` geerbten Methoden werden alle vom `Dekorierer` überschrieben. Beim Aufruf einer überschreibenden Methode leitet der `Dekorierer` diesen Aufruf an das aggregierte Objekt weiter und kann dabei vor oder nach bzw. vor und nach der Delegation den Code für die Zusatzfunktionalität ausführen. Hält die Methode das liskovsche Substitutionsprinzip ein, kann der Aufruf der überschriebenen Methode ganz entfallen.

Da die Klasse `Dekorierer` auch von der Klasse `Komponente` abgeleitet ist, macht es für Clients bei Einhaltung des liskovschen Substitutionsprinzips keinen Unterschied, ob sie mit einem dekorierten Objekt arbeiten oder mit einer konkreten, undekorierten `Komponente`. Das bedeutet, dass die Referenz auf ein Objekt der Klasse `Komponente`, die ein Client-Programm benutzt, in einem Falle auf eine konkrete `Komponente`, in einem anderen Fall auf einen `Dekorierer` zeigen kann.

Die Tatsache, dass die Klasse `Dekorierer` sowohl von der Klasse `Komponente` abgeleitet ist als auch gleichzeitig ein Objekt vom Typ `Komponente` aggregiert, bedarf einer näheren Untersuchung. Im Folgenden werden zuerst die Methoden und dann die Daten der Klasse `Dekorierer` betrachtet.

#### Methoden des Dekorierers

Die Klasse `Dekorierer` erbt die Methoden der Klasse `Komponente`. Diese können vom `Dekorierer` überschrieben werden, aber der `Dekorierer` kann auch neue Methoden hinzufügen. Neu hinzugekommene – also erweiternde – Methoden können aber von einem Client, der nur die Schnittstelle der Klasse `Komponente` kennt und darüber auf

den Dekorierer zugreift, nicht genutzt werden. Sie werden einfach weggecastet. Somit verbleiben bei den Methoden für den Dekorierer zwei Möglichkeiten: er kann geerbte Methoden unverändert übernehmen oder geerbte Methoden überschreiben.

Soll der Dekorierer eine bestimmte Methode der zu dekorierenden Klasse um eine zusätzliche Funktionalität erweitern, so überschreibt er diese Methode unter Einhaltung des liskovschen Substitutionsprinzips. Um die bestehende Funktionalität einer konkreten Komponente zu nutzen, ruft der Dekorierer in der überschreibenden Methode die überschriebene Methode des aggregierten Objekts auf.



Ein Dekorierer muss das liskovsche Substitutionsprinzip einhalten.



Für Methoden der Klasse `Komponente`, die nicht vom Dekorierer erweitert werden, darf nicht einfach die Methode der Basisklasse `Komponente` aufgerufen werden. Da bei Einhalten des liskovschen Substitutionsprinzips ein Objekt einer von der Klasse `Komponente` abgeleiteten Klasse aggregiert werden kann und die aggregierte konkrete Komponente selbst bereits Methoden der Klasse `Komponente` überschrieben haben könnte, muss der Dekorierer die entsprechende von der Basisklasse `Komponente` geerbte Methode auf jeden Fall überschreiben. Die vom Dekorierer geerbte Methode der Klasse `Komponente` würde nicht die gewünschte Funktionalität liefern. Der Dekorierer muss also eine geerbte Methode überschreiben, auch wenn er sie nicht erweitern will. Er kann dabei einfach einen Methodenaufruf an das aggregierte Objekt weiterleiten. Es darf generell in der Klasse `Dekorierer` keine unveränderten geerbten Methoden geben.

Ein Dekorierer muss alle geerbten Methoden überschreiben.



In der Praxis wird daher häufig ein abstrakter Dekorierer eingeführt, der die Aggregation realisiert, alle geerbten Methoden überschreibt und die Aufrufe an das aggregierte Objekt delegiert. Von diesem abstrakten Dekorierer können dann konkrete Dekorierer ableiten. Der Entwickler eines konkreten Dekorierers muss nur noch die für ihn interessanten Methoden betrachten und überschreiben und braucht sich nicht um alle von der Klasse `Komponente` geerbten Methoden zu kümmern, da sie bereits vom abstrakten Dekorierer überschrieben werden. Diese Vorgehensweise hat insbesondere dann einen Vorteil, wenn mehrere konkrete Dekorierer zu entwickeln sind (siehe Kapitel 4.5.3.4).

In den folgenden Abschnitten wird von den zuvor beschriebenen Fällen nur noch der eigentlich interessante Fall herausgegriffen, nämlich dass der Dekorierer Methoden der Klasse `Komponente` überschreibt, um eine zusätzliche Funktionalität zu schaffen.

## Daten des Dekorierers

Die Konstruktion des Dekorierers mittels Vererbung und Aggregation führt dazu, dass ein Dekorierer drei verschiedene Datenanteile umfasst: einen geerbten, unveränderten Anteil, einen eigenen, erweiternden Anteil und einen Anteil, auf den der Dekorierer über die Aggregation zugreifen kann. Diese Anteile sind in Bild 4-8 dargestellt:

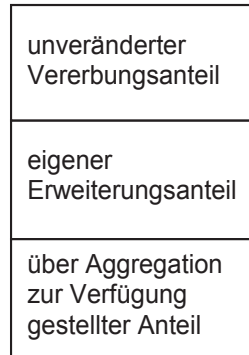


Bild 4-8 Aufbau eines Dekorierers

Bei den Daten hat der Dekorierer drei Anteile, den Vererbungsanteil, den Erweiterungsanteil und den aggregierten Anteil.



Unter dem Erweiterungsanteil sind Daten zu verstehen, welche zur Erfüllung einer weiteren Funktionalität den Dekorierer erweitern. Im Rahmen dieses Musters werden sie weggelassen. Erweiternde Daten müssen hier nicht weiter betrachtet werden. Sie werden im Rahmen des Musters automatisch weggecastet.

Die beiden anderen Anteile sind auf den ersten Blick gleichartige Datenanteile: Der von der Klasse `Komponente` geerbte und der über die Aggregation von einem Objekt vom Typ `Komponente` bereitgestellte Anteil. Der geerbte Anteil ist **statisch** und somit fix und kann immer nur die in der Klasse `Komponente` definierten Attribute umfassen. Der über die Aggregation bereitgestellte Anteil ist **dynamisch** und daher flexibel und kann nach Liskov auch Attribute der Objekte von Subklassen der Klasse `Komponente` umfassen.

Nur über den dynamischen Anteil ist der Dekorierer in der Lage, Objekte von beliebigen konkreten Komponenten zu dekorieren. Der geerbte Anteil darf daher nicht benutzt werden und ist im Dekorierer überflüssig.



Der Dekorierer darf die von der Klasse `Komponente` geerbten Daten nicht benutzen.



Die Aggregation zeigt erst einen Nutzen, wenn die Klasse `Komponente` mehrere Subklassen hat. Nur durch den Einsatz der Aggregation können alle Subklassen "gleichzeitig" erweitert werden.



Der Nutzen der Vererbung beim Dekorierer-Muster liegt darin, dass ein Dekorierer automatisch die gleiche Schnittstelle wie die Klasse `Komponente` anbietet. Damit kann bei Einhaltung der Verträge infolge des liskovschen Substitutionsprinzips ein Client ein Objekt der Klasse `Dekorierer` wie ein Objekt der Klasse `Komponente` oder ein Objekt einer von der Klasse `Komponente` abgeleiteten Klasse behandeln.

### Lösungsvariante mit einer Schnittstelle für die Klassenhierarchie

Steht an der Spitze der ursprünglichen Klassenhierarchie ein Interface statt einer Basisklasse, so kann das Dekorierer-Muster ebenfalls angewandt werden. In dieser Variante muss die Klasse `Dekorierer` das Interface `IKomponente` implementieren und ein Objekt vom Typ dieses Interface aggregieren, wie es das folgende Bild zeigt:

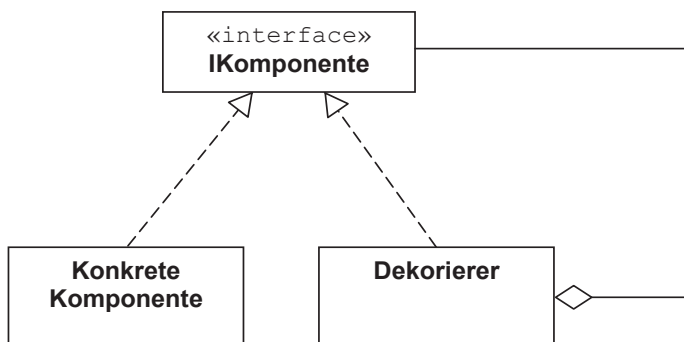


Bild 4-9 Dekorierer als Realisierung einer Schnittstelle

Auch in dieser Situation gilt bei Einhaltung der Verträge das liskovsche Substitutionsprinzip. Somit kann ein Client einen Dekorierer und konkrete Komponenten gleich behandeln.

Die Variante mit einem Interface hat aber einen entscheidenden Vorteil: Die Probleme mit den geerbten Anteilen existieren hier nicht.



Auf Grund dieses Vorteils wird im Programmbeispiel in Kapitel 4.5.3.5 mit dieser Lösungsvariante gearbeitet.

#### 4.5.3.2 Teilnehmer

##### Komponente

Mit `Komponente` ist die Basisklasse der Klassen derjenigen Objekte bezeichnet, welche um die Zusatzfunktionalität erweitert werden sollen. Die Komponente selbst wird nicht verändert.

##### Dekorierer

Die Klasse `Dekorierer` leitet von der Klasse `Komponente` ab und damit kann bei Einhaltung der Verträge nach dem liskovschen Substitutionsprinzip eine Referenz auf ein Objekt der Klasse `Dekorierer` an die Stelle einer Referenz auf die Klasse `Komponente` treten. Die Klasse `Dekorierer` aggregiert ein Objekt der Klasse `Komponente` und kann damit auf dieses Objekt bzw. auf ein Objekt einer von der Klasse `Komponente` abgeleiteten Klasse zugreifen. Der Dekorierer überschreibt alle geerbten Methoden und leitet die Aufrufe per Delegation an das aggregierte Objekt weiter. Die zusätzliche Funktionalität wird in den überschreibenden Methoden vor oder nach bzw. vor und nach dem Aufruf der jeweils überschriebenen Methode untergebracht.

##### Konkrete Komponenten

Diese Elemente sind Subklassen der Klasse `Komponente`. Wegen der Aggregationsbeziehung zwischen den Klassen `Dekorierer` und `Komponente` und infolge des liskovschen Substitutionsprinzips können alle Subklassen der Klasse `Komponente` "gleichzeitig" durch die Klasse `Dekorierer` erweitert werden.

#### 4.5.3.3 Dynamisches Verhalten

Der Client ruft eine Methode des Dekorierers auf. Der Aufruf wird zum einen an das aggregierte Objekt der Klasse `Komponente` weitergeleitet. Diese Methode wird dann von dem aggregierten Objekt der Klasse `Komponente` abgearbeitet. Das Objekt der Klasse `Komponente` gibt das Ergebnis über das Objekt der Klasse `Dekorierer` an das Client-Programm zurück. Enthält die überschreibende Methode zum anderen eine Zusatzfunktion des Dekorierers, so führt der Dekorierer vor oder nach bzw. vor und nach der Delegation des Methodenaufrufs diesen zusätzlichen Code selbst aus. Das folgende Bild visualisiert den Ablauf für die Dekoration einer Komponente:

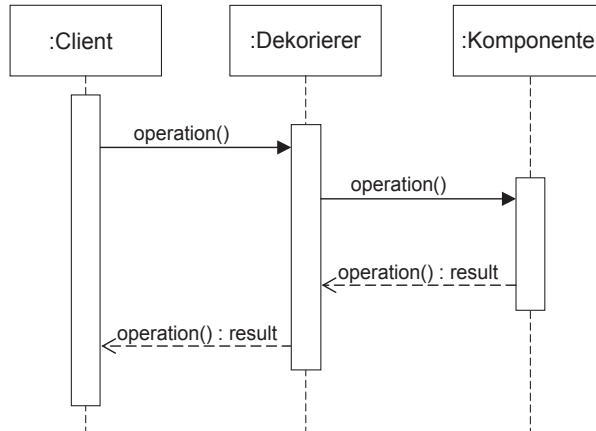


Bild 4-10 Sequenzdiagramm für die Dekoration einer Komponente

Wegen des liskovschen Substitutionsprinzips kann in Bild 4-10 auch eine konkrete Komponente bzw. Subklasse an die Stelle einer Komponente treten. Genau das ist in dem in Kapitel 4.5.3.5 folgenden Programmbeispiel der Fall.

#### 4.5.3.4 Mehrere Dekorierer

Sollen die Komponenten um mehrere verschiedene Funktionalitäten erweitert werden, können mehrere Dekorierer erstellt werden. Diese können durch die vorgeschlagene Konstruktion sogar alle "gleichzeitig" angewendet werden, indem ein Dekorierer einen anderen Dekorierer aggregiert.

In dieser Situation ist dann aber – besonders dann, wenn viele Methoden nur delegiert werden müssen – ein abstrakter Dekorierer sinnvoll. Im abstrakten Dekorierer können alle Methoden einfach nur delegiert werden. Konkrete Dekorierer leiten von dem abstrakten Dekorierer ab und müssen nur noch einige Methoden überschreiben, um die gewünschte Funktionalität zu erbringen. Das folgende Klassendiagramm zeigt einen abstrakten Dekorierer:

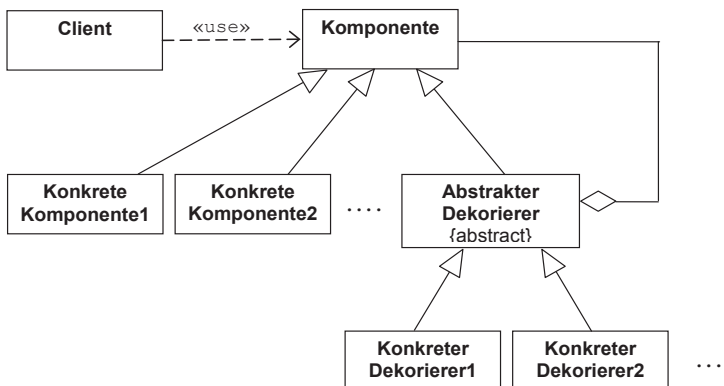


Bild 4-11 Klassendiagramm des Dekorierer-Musters mit mehreren konkreten Dekorierern



Bild 4-11 zeigt, dass mehrere konkrete Dekorierer (`KonkreterDekorierer1`, `KonkreterDekorierer2`) eingesetzt werden können und dass ein abstrakter Dekorierer generischen Code für alle ableitenden konkreten Dekorierer bereitstellen kann.

Dieses Schema eines abstrakten Dekorierers wird auch im anschließenden Programmbeispiel benutzt, da in diesem Beispiel mehrere konkrete Dekorierer vorkommen.

#### 4.5.3.5 Programmbeispiel

Das folgende Bild zeigt das Klassendiagramm dieses Beispiels:

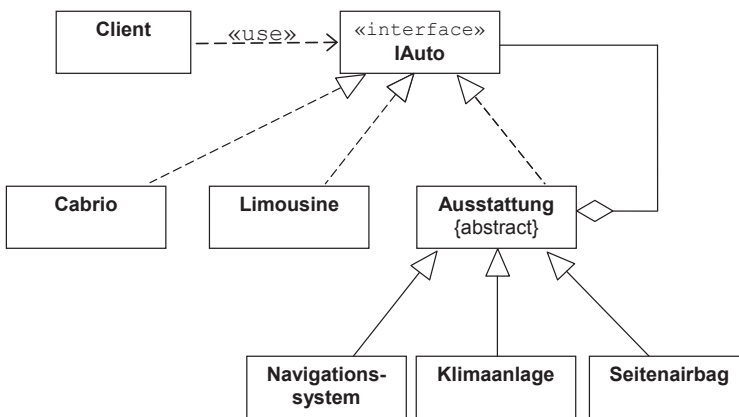


Bild 4-12 Klassendiagramm des Programmbeispiels zum Dekorierer

Hier wird das Dekorierer-Muster verwendet, um die konkreten Komponenten der Klasse `Limousine` und `Cabrio` zu dekorieren, welche beide die Schnittstelle `IAuto` implementieren. Wie bereits in Kapitel 4.5.3.1 erwähnt wurde, vermeidet diese Lösungsvariante mit einer Schnittstelle die Probleme mit geerbten Anteilen.

Die Klassen `Limousine` und `Cabrio` können nun mit Ausstattungen vom Typ `Klimaanlage`, `Navigationssystem` und `Seitenairbags` dekoriert werden, welche die konkreten Dekorierer darstellen und von der abstrakten Klasse `Ausstattung` erben. Dadurch kann z. B. die Klasse `Cabrio` mit Ausstattungen vom Typ `Klimaanlage` und `Seitenairbags` dekoriert werden, so dass es für diesen Fall keine eigene Klasse (z. B. `CabrioKlimaAirbags`) geben muss.

Die Methoden `gibKosten()` und `zeigeDetails()` dienen der letztendlichen Ausgabe in diesem Programmbeispiel. Sie zeigen auf, dass sich durch die Dekoration eine Schachtelung der Methodenaufrufe ergibt, durch die beispielsweise ein Aufsummieren der Kosten möglich ist.

Hier das Interface `IAuto`:

```
// Datei: IAuto.java
public interface IAuto
{
    public int gibKosten();
    public void zeigeDetails();
}
```

Die Klasse `Limousine` stellt eine zu dekorierende konkrete Komponente dar. Die Methoden geben die Art des Autos sowie dessen Grundkosten aus bzw. zurück:

```
// Datei: Limousine.java
class Limousine implements IAuto
{
    public void zeigeDetails()
    {
        System.out.print ("Limousine");
    }
    public int gibKosten()
    {
        return 35000;
    }
}
```

Die Klasse `Cabrio` stellt eine weitere zu dekorierende konkrete Komponente dar, die die gleichen Methoden mit jedoch anderen Werten als die entsprechenden Methoden der Klasse `Limousine` besitzt:

```
// Datei: Cabrio.java
class Cabrio implements IAuto
{
    public void zeigeDetails()
    {
        System.out.print ("Cabrio");
    }

    public int gibKosten()
    {
        return 50000;
    }
}
```

Die Klasse `Ausstattung` stellt einen **abstrakten Dekorierer** dar, der generischen Code für alle Ausstattungen enthalten kann. In diesem Beispiel enthält der abstrakte Dekorierer nur die Deklaration der Instanzvariablen `auto`, die auf die Schnittstelle `IAuto` zeigt, und den Konstruktor zum Setzen dieser Instanzvariablen. Auf ein Überschreiben der geerbten Methoden im abstrakten Dekorierer gemäß Kapitel 4.5.3.4 wurde verzichtet, da in diesem Beispiel die konkreten Dekorierer alle Methoden selbst überschreiben. Hier die Klasse `Ausstattung`:

**// Datei: Ausstattung.java**

```
public abstract class Ausstattung implements IAuto
{
    protected IAuto auto;

    public Ausstattung (IAuto pIAuto)
    {
        auto = pIAuto;
    }
}
```

Die **konkreten Dekorierer** vom Typ Klimaanlage, Navigationssystem und Seitenairbags sind von der abstrakten Klasse Ausstattung abgeleitet und besitzen ähnliche Methoden wie die Klassen Limousine und Cabrio. Diese Methoden dienen zur Aus- bzw. Rückgabe der Art der Ausstattung sowie deren Kosten. Es folgen die Klassen Klimaanlage, Navigationssystem und Seitenairbags:

**// Datei: Klimaanlage.java**

```
class Klimaanlage extends Ausstattung
{
    public Klimaanlage(IAuto pIAuto)
    {
        super(pIAuto);
    }

    public void zeigeDetails() // "dekoriert" die Details
    {
        auto.zeigeDetails();
        System.out.print ("", Klimaanlage);
    }

    public int gibKosten() // "dekoriert" die Kosten
    {
        return auto.gibKosten() + 1500;
    }
}
```

**// Datei: Navigationssystem.java**

```
class Navigationssystem extends Ausstattung
{
    public Navigationssystem (IAuto pIAuto)
    {
        super (pIAuto);
    }

    public void zeigeDetails() // "dekoriert" die Details
    {
        auto.zeigeDetails();
        System.out.print ("", Navigationssystem);
    }
}
```

```

    public int gibKosten() // "dekoriert" die Kosten
    {
        return auto.gibKosten() + 2500;
    }
}

// Datei: Seitenairbags.java
class Seitenairbags extends Ausstattung
{
    public Seitenairbags(IAuto pIAuto)
    {
        super (pIAuto);
    }

    public void zeigeDetails() // "dekoriert" die Details
    {
        auto.zeigeDetails();
        System.out.print(", Seitenairbags");
    }

    public int gibKosten() // "dekoriert" die Kosten
    {
        return auto.gibKosten() + 1000;
    }
}

```

Die Klasse `Client` zeigt die Dekoration von Komponenten mit verschiedenen Ausstattungen vom Typ `Klimaanlage`, `Navigationssystem` und `Seitenairbags`. In den im Folgenden gezeigten drei Fällen des Client-Beispiels werden jeweils nach der Objekterzeugung Details hinsichtlich der dekorierten Variante sowie die Gesamtkosten ausgegeben:

```

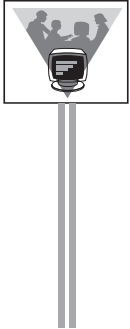
// Datei: Client.java
class Client
{
    public static void main(String[] args)
    {
        // Auto mit Klimaanlage
        IAuto auto = new Klimaanlage (new Limousine());
        auto.zeigeDetails();
        System.out.println ("\nfuer "+ auto.gibKosten() +
                             " Euro\n");

        // Dynamische Erweiterung der Limousine mit Ausstattungen
        auto = new Navigationssystem (new Seitenairbags(auto));
        auto.zeigeDetails();
        System.out.println ("\nfuer "+ auto.gibKosten() +
                             " Euro\n");

        // Cabrio Variante
        auto = new Navigationssystem (new Seitenairbags(new Cabrio()));
        auto.zeigeDetails();
    }
}

```

```
        System.out.println ("\nfuer "+ auto.gibKosten() +  
                             " Euro\n");  
    }  
}
```



Hier das Protokoll des Programmlaufs:

Limousine, Klimaanlage  
fuer 36500 Euro

Limousine, Klimaanlage, Seitenairbags,  
Navigationssystem  
fuer 40000 Euro

Cabrio, Seitenairbags, Navigationssystem  
fuer 53500 Euro

## 4.5.4 Bewertung

### 4.5.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Eine Komponente kennt ihren Dekorierer nicht.
- Vorteilhaft ist vor allem, dass man dynamisch erweitern kann und nicht die Klasse z. B. durch die Bildung von Unterklassen "statisch" erweitert. Zur Laufzeit des Programms kann die zusätzliche Funktionalität des Dekorierers dynamisch hinzugefügt und wieder entfernt werden. Wird hingegen eine Klasse statisch durch Bildung von Unterklassen erweitert, so wird statisch zum Zeitpunkt des Kompilierens festgelegt, von welcher Klasse das Objekt ist und somit auch, welche Funktionalität in der Klassenhierarchie verwendet wird.
- Es ist möglich, mehrere Klassen einer Vererbungshierarchie gleichzeitig mit einem Dekorierer zu erweitern. Man kann sogar Klassen erweitern, die es beim Schreiben des Dekorierers noch gar nicht gab, also neue Unterklassen der Klasse `Komponente`.
- Ein weiterer Vorteil des Dekorierers ist die Möglichkeit, mehrere unterschiedliche Dekorierer zu kombinieren. So kann man einen Dekorierer dekorieren, falls noch eine weitere Zusatzfunktionalität erforderlich ist. Hierbei erweitert ein Dekorierer die Funktionalität eines anderen Dekorierers und dieser wiederum dekoriert das eigentlich zu erweiternde Objekt. Dieses Prinzip lässt sich quasi beliebig oft wiederholen. Somit können beliebige Funktionalitäten von verschiedenen Dekorierern kombiniert werden.

#### 4.5.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Dekoriert ein Dekorierer eine Klasse mit sehr vielen Methoden, erweitert aber nur sehr wenige davon, besteht der Dekorierer zum größten Teil nur aus Delegationsmethoden. Eine Delegation führt aber zu einer Verzögerung.
- Fehler in kombinierten Dekorierern sind oft sehr schwer zu finden.
- Man hat einen geerbten Anteil, den man gar nicht nutzt. Dieser Nachteil entfällt, wenn an Stelle einer abstrakten Komponente eine Schnittstelle verwendet wird, die auch der Dekorierer realisiert (siehe Lösungsvariante in Kapitel 4.5.3.1).

#### 4.5.5 Einsatzgebiete

Die Möglichkeiten des Dekorierer-Musters sollen im Folgenden anhand von zwei Beispielen gezeigt werden. Das eine Beispiel (siehe Kapitel 4.5.5.1) ist in den Stream-Klassen von Java zu finden, beim anderen Beispiel (siehe Kapitel 4.5.5.2) handelt es sich um die Erweiterung von grafischen Elementen. Dabei sollen Elemente von grafischen Oberflächen, beispielsweise Textfelder oder Schaltflächen, über Dekorierer mit grafischen Zusätzen wie etwa mit einem Rahmen erweitert werden können.

##### 4.5.5.1 Anwendungsbeispiel 1: Streams

Das erste Beispiel zeigt, wie man den Stream-Klassen von Java eine Pufferung beim Schreiben als Zusatzfunktionalität hinzufügen kann. Um dieses Beispiel übersichtlich zu halten, werden nur die im Folgenden aufgeführten von der Klasse `OutputStream` abgeleiteten Klassen betrachtet. Für die Implementierung der Pufferung wird hier eine einfache Lösung gewählt, die so nicht in der entsprechenden Klasse einer Java-Plattform zu finden sein muss.

Die Klasse `PipedOutputStream` kann Daten in eine Pipe schreiben, während die Klasse `FileOutputStream` Daten in eine Datei schreiben kann. Diese Klassen sind im Folgenden in einer Vererbungshierarchie zusammen mit der abstrakten Klasse `OutputStream` dargestellt:

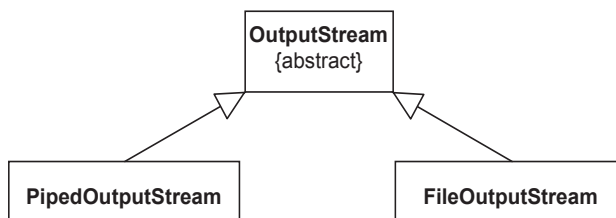


Bild 4-13 Vererbungshierarchie der `OutputStream`-Klassen

Möchte man nun das Klassensystem so erweitern, dass man in eine Pipe oder eine Datei auch gepuffert schreiben kann, so gibt es zwei Möglichkeiten – eine schlechte und eine gute. Die schlechte und aufwendige Lösung liegt auf der Hand. Man bildet

zwei Unterklassen und implementiert dort die Funktionalität, wie in folgendem Bild zu sehen ist:

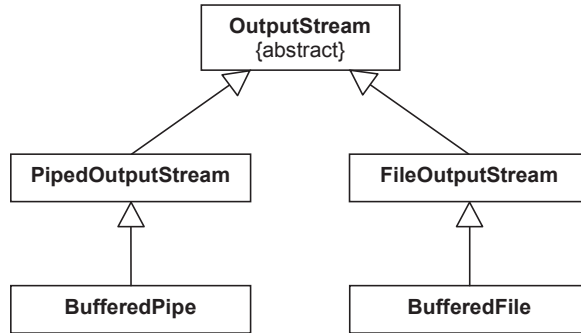


Bild 4-14 Erweiterung von Funktionalität durch Ableiten

Diese Lösung ist schon deshalb unbefriedigend, weil genau der gleiche Code in unterschiedlichen Klassen mehrmals auftaucht. Wie so oft, liegt die bessere Lösung nicht direkt auf der Hand. Aber es ist klar, dass die Fähigkeit der Pufferung ausgelagert werden muss, so dass sie für alle Klassen, die diese Funktionalität zusätzlich nutzen wollen, nur ein einziges Mal implementiert wird. Das folgende Bild zeigt die bessere Lösung:

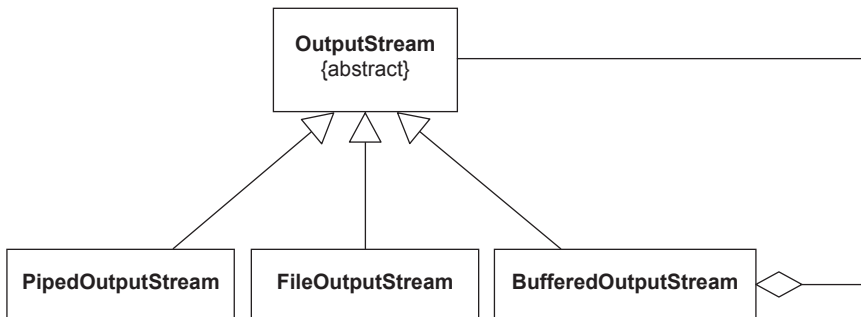


Bild 4-15 Beispiel für den Dekorierer

Die Klasse `BufferedOutputStream` wird als Dekorierer angelegt. Der Dekorierer implementiert genau dieselbe Schnittstelle wie die anderen `OutputStream`-Klassen, allerdings leitet er alle Aufrufe direkt an das aggregierte Objekt vom Typ `OutputStream` weiter, nachdem er die notwendigen Aktivitäten zur Pufferung durchgeführt hat. Zu beachten ist, dass für ein überschreibendes Objekt stets die überschreibende Methode aufgerufen wird. Eine einfache Implementierung eines Dekorierers vom Typ `BufferedOutputStream` sieht folgendermaßen aus:

```
import java.io.*;

public class BufferedOutputStream extends OutputStream
{
```

```
byte[] buffer = new byte[1024]; // Puffer für 1024 Bytes
int counter = 0;

OutputStream out; // dekoriertes Objekt

public BufferedOutputStream (OutputStream s)
{
    out = s;
}

public void close() throws IOException
{
    if (counter > 0) // falls Puffer nicht leer: leeren
    {
        out.write (buffer, 0, counter);
        counter = 0;
    }
    out.close(); //eigentliche Operation delegieren
}

public void flush() throws IOException
{
    if (counter > 0) // falls Puffer nicht leer: leeren
    {
        out.write (buffer, 0, counter);
        counter = 0;
    }
    out.flush(); //eigentliche Operation delegieren
}

public void write (int i) throws IOException
{
    buffer[counter] = (byte)i; // in Puffer legen
    counter ++;
    if (counter == 1024) // falls Puffer voll
    {
        out.write (buffer); // Puffer in Stream schreiben
        counter = 0;
    }
}

public void write (byte[] b) throws IOException
{
    int i;
    for (i = 0; i <= b.length; i++)
    {
        buffer[counter] = b[i]; // in Puffer legen
        counter ++;
        if (counter == 1024) // falls Puffer voll
        {
            out.write (buffer); // Puffer in Stream schreiben
            counter = 0;
        }
    }
}
```



```

public void write (byte[] b, int off, int len) throws IOException
{
    int i;
    for (i = off; i<= off+len; i++)
    {
        buffer[counter] = b[i]; // in Puffer legen
        counter ++;
        if (counter == 1024)    // falls Puffer voll
        {
            out.write (buffer); // Puffer in Stream schreiben
            counter = 0;
        }
    }
}
}

```

Der Dekorierer leitet also alle Methodenaufrufe an das aggregierte Objekt vom Typ `OutputStream` weiter, nachdem er die Pufferung durchgeführt hat.

Sowohl die Klasse `PipedOutputStream` als auch die Klasse `FileOutputStream` sind von der Klasse `OutputStream` abgeleitet. Ein Objekt einer abgeleiteten Klasse kann nach dem liskovschen Substitutionsprinzip an die Stelle eines vom Dekorierer aggregierten Objekts treten, wenn die Verträge bei der Ableitung eingehalten werden. Es wird damit dekoriert, in diesem Beispiel also gepuffert.

#### 4.5.5.2 Anwendungsbeispiel 2: grafische Komponenten

Im folgenden Beispiel dekoriert der Dekorierer im wahrsten Sinne des Wortes, nämlich er verziert grafische Komponenten mit einem Rahmen:

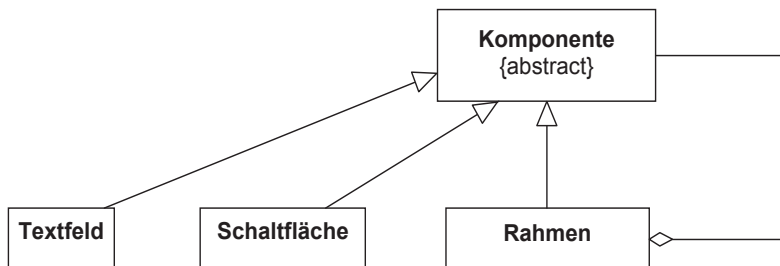


Bild 4-16 Weiteres Beispiel für den Dekorierer

Auf lauffähigen Code wird an dieser Stelle verzichtet.

Ein ganz ähnliches Beispiel folgt in Kapitel 4.7.5 beim Kompositum-Muster (vgl. Bild 4-24). Der Unterschied zwischen diesen beiden Beispielen liegt aber in der Funktionalität: Durch die Dekoriererklass `Rahmen` kann jede grafische Komponente mit einem Rahmen verziert werden. Diese Klasse ist aber nicht in der Lage, mehrere grafische Komponenten zu einer Gruppe zusammenzufassen und diese Gruppe dann zu umrahmen. Das Zusammenfassen leistet beispielsweise die Klasse `Fenster` in Bild 4-24 beim Kompositum-Muster. Da die Klasse `Fenster` wiederum eine grafische Kompo-

nente darstellt, könnte sie nun durch die Dekoriererklasse mit einem Rahmen verziert werden.

### 4.5.6 Ähnliche Entwurfsmuster

Vom Aufbau her ist das Entwurfsmuster **Kompositum** sehr ähnlich zu dem Dekorierer-Muster. Wenn man die beiden Klassendiagramme in Bild 4-7 und in Bild 4-11 vergleicht, so sieht man als einzigen Unterschied die Multiplizitätsangabe bei der Aggregation. Die Kardinalität ist beim Dekorierer-Muster eins, beim Kompositum-Muster aber beliebig. Betrachtet man alleine das Klassendiagramm, so kann ein Dekorierer als Spezialfall des Kompositum-Musters mit nur einer einzigen Komponente angesehen werden. Es zeigen sich jedoch erhebliche Unterschiede in der Verwendung des Musters. Während mit Hilfe des Dekorierer-Musters Funktionalität dynamisch gebildet werden kann, dient das Kompositum-Muster dazu, Objekte zu einer hierarchischen Struktur zusammenzusetzen.

Ein **Proxy** kann ebenso wie ein Dekorierer ein Objekt um eine zusätzliche Funktionalität erweitern. Sowohl ein Proxy-Objekt als auch ein Dekorierer-Objekt befinden sich zwischen einer Anwendung und dem eigentlichen Objekt. In beiden Fällen halten sie eine Referenz auf das eigentliche Objekt und können über diese Referenz Anfragen und Befehle an das eigentliche Objekt delegieren. Vor oder nach bzw. vor und nach einer Delegation kann eine Zusatzfunktionalität eingefügt werden. Beim Proxy-Muster ist das aber eher ein Nebeneffekt, der sich aus der Konstruktion ergibt. Beim Proxy-Muster steht eine andere Aufgabe – eine Art "Türsteher"-Funktion – im Vordergrund: ein Proxy-Objekt ermöglicht oder verhindert den Zugriff auf die Funktionalität des eigentlichen Objekts. Beim Proxy-Muster geht man in der Regel davon aus, dass das eigentliche Objekt schon die richtige Funktionalität besitzt, beim Dekorierer-Muster soll die Funktionalität des eigentlichen Objekts dynamisch durch ein Dekorierer-Objekt erweitert werden.

Das Entwurfsmuster **Strategie** kann das Verhalten eines sogenannten Kontextobjekts durch ein anderes Verhalten austauschen. Ein Kontextobjekt ist das Objekt, das die konkrete Strategie aufruft. Die Gesamtfunktionalität eines Kontextobjektes ändert sich durch den Austausch der Strategie nicht, weil alle Strategien die gleiche Funktionalität realisieren. Beim Dekorierer-Muster hingegen wird die Funktionalität eines Objektes geändert bzw. erweitert. Während alle Strategien die gleiche Funktionalität realisieren, kann hingegen jeder Dekorierer etwas anderes machen. Ein Dekorierer behält das Verhalten des zu dekorierenden Objekts bei und versieht dieses Objekt mit einer Zusatzfunktionalität. Beim Strategiemuster hingegen wird von der Umgebung dieses Musters zielgerichtet das Kontextobjekt mit einer anderen Strategie versehen (kompletter Austausch eines Algorithmus). Damit wird keine Zusatzfunktionalität erzeugt, sondern es findet ein Austausch eines gesamten Algorithmus für das Kontextobjekt statt.

Ebenso wie ein **Besucher** realisiert ein Dekorierer eine neue Funktionalität. Das Besucher-Muster erlaubt es, in einer Datenstruktur eine neue zentrale Funktion hinzuzufügen. Mit dem Dekorierer-Muster können jedoch nur einzelne Objekte erweitert werden. Ein weiterer Unterschied ist, dass beim Besucher-Muster die zu besuchenden Objekte auf den Besuch "vorbereitet" sein müssen, dadurch dass sie eine entsprechende Methode dem Besucher zur Verfügung stellen.

Beim Architekturmuster **Pipes and Filters** kann eine Zusatzfunktionalität erzeugt werden, indem ein weiterer Filter in die Filterkette eingeschoben wird. Filterketten sind sehr flexibel und können leicht umkonfiguriert werden. Filter können gut in anderen Filterketten wiederverwendet werden. Ein Dekorierer ist jedoch an die dekorierte Klasse gebunden.

## 4.6 Das Strukturmuster Fassade

### 4.6.1 Name/Alternative Namen

Fassade (engl. facade oder auch façade).

### 4.6.2 Problem

Häufig verwendete Funktionen eines Subsystems sollen über eine gemeinsame (einheitliche) und meist vereinfachte Schnittstelle angesprochen werden. Die Fassade soll die einzelnen Instanzen des betreffenden Subsystems verbergen und die Funktionalität der Fassade an die Klassen des Subsystems delegieren. Damit soll bei stabiler Fassade die interne Architektur oder Implementierung geändert werden können. Die Fassade soll also keine neue Funktionalität definieren, sondern nur wissen, welche Klassen hinter der Fassade für die Bearbeitung eines Aufrufs durch die Fassade zuständig sind und deren Methoden bei Bedarf aufrufen.

Das **Fassade-Muster** soll eine meist vereinfachte, einheitliche Schnittstelle zum Zugriff auf die Klassen eines Subsystems bereitstellen.



### 4.6.3 Lösung

Große Softwaresysteme bestehen meist aus mehreren Subsystemen, die miteinander interagieren. Eine Zerlegung eines Systems in Subsysteme dient nicht nur der Übersichtlichkeit und der besseren Wartbarkeit, sondern kann auch Dritten den Zugriff auf Funktionen des Systems erleichtern.

Damit sich die Nutzer eines Systems nicht mit den Interna der Subsysteme des Systems auseinandersetzen müssen, erfolgt der Zugang zu einem Subsystem beim Fassade-Muster über eine meist vereinfachte und einheitliche Schnittstelle.



Diese Schnittstelle wird in einer Fassadenklasse bereitgestellt und kapselt ein Subsystem gegenüber den aufrufenden Kunden (Clients).

Die Fassade wird frei entworfen. Sie verbirgt die Details der internen Struktur eines Subsystems. Dadurch kann die interne Architektur eines Subsystems geändert werden, ohne, dass die Kundenobjekte abgeändert werden müssen. Aufrufer und Internas des Subsystems können somit unabhängig voneinander ausgetauscht werden.



Die Zugriffe auf die Fassade werden von der Fassade an die Klassen des Subsystems weitergeleitet (delegiert).



Dabei kann eine Methode einer Fassadenklasse auf mehrere Klassen eines Subsystems zugreifen, wobei diese Klassen jeweils einen Teil der entsprechenden Aufgabe ausführen.

Das Fassade-Muster lässt offen, ob Direktzugriffe auf Subsystemklassen – also Zugriffe, die nicht über die Fassade gehen – möglich sind. Da es sich bei einer Fassade meist um eine Vereinfachung der von den Klassen angebotenen Schnittstellen handelt, sollte man aber davon ausgehen, dass es Klienten gibt, denen eine vereinfachte Schnittstelle nicht genügt. Diese Klienten müssen dann an der Fassade vorbeigehen und einen Direktaufruf bei den Subsystemklassen machen. Die sich daraus ergebenden Nachteile werden in Kapitel 4.6.4 ausführlich diskutiert.

Die Beschreibung des Musters benutzt den Begriff Subsystem, ohne ihn präziser zu definieren. Ein Subsystem kann auch ein System sein. Man spricht allgemein von Systemen n-ter Stufe. In Java kann man beispielsweise ein Paket (`package`) als Subsystem bezeichnen und auf dieser Ebene das Muster anwenden. Java erlaubt es, bei einem Paket über die Paketsichtbarkeit Direktzugriffe auf die Klassen eines Pakets zu unterbinden.

#### 4.6.3.1 Klassendiagramm

Die einheitliche und in der Regel vereinfachte Schnittstelle zwischen den Clients und den Klassen eines Subsystems wird durch eine eigene Klasse zur Verfügung gestellt, die sogenannte Fassadenklasse. Diese ist im folgenden Bild dargestellt:

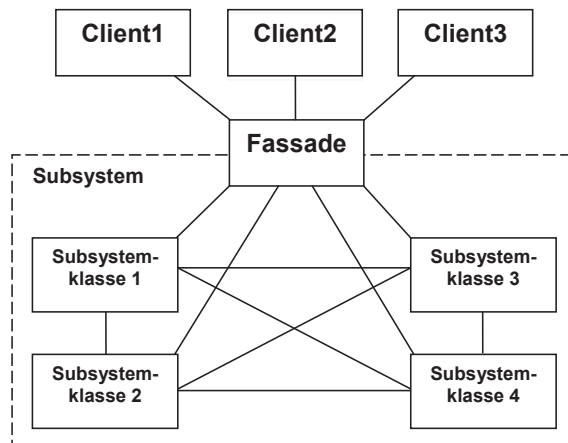


Bild 4-17 Beispiel für ein Klassendiagramm des Fassade-Musters

Dabei ist die Hüllkurve um das Subsystem frei und nicht UML-konform gezeichnet.

Das Fassade-Muster fasst die Funktionalität eines Subsystems zusammen und bietet diese über eine definierte und meist vereinfachte Schnittstelle an.



Damit bietet das Fassade-Muster statt vieler einzelner Schnittstellen eine **gemeinsame** Schnittstelle nach außen an. Diese Schnittstelle vereinfacht den Zugriff auf die Klassen des Subsystems.

#### 4.6.3.2 Teilnehmer

##### Fassade

Die Fassade bietet eine einheitliche und meist vereinfachte Schnittstelle zum Zugriff auf die Klassen eines Subsystems an.

##### SubsystemklasseX

Auf die Klasse eines Subsystems wird in der Regel über die zugeordnete Fassade zugegriffen.

#### 4.6.3.3 Dynamisches Verhalten

Ein Client ruft eine Methode eines Objekts einer Fassadenklasse auf. Das Objekt der Fassadenklasse leitet diesen Aufruf an ein Objekt einer Klasse des Subsystems weiter. Eventuelle Ergebnisse eines Methodenaufrufs bei einer Subsystemklasse werden auf dem umgekehrten Weg an den Client zurückgeliefert.

Der Einfachheit halber wird in Bild 4-18 nur die Weiterleitung des ursprünglichen Methodenaufrufs des Clients an ein Objekt einer einzigen Subsystemklasse dargestellt. Prinzipiell können viele Methoden aus unterschiedlichen Subsystemklassen von einer Fassade aufgerufen werden, um die Aufgaben der von der jeweiligen Fassadenklasse angebotenen Methoden zu erledigen.

Das folgende Sequenzdiagramm zeigt den Zugriff auf ein Objekt einer Subsystem-Klasse über eine Fassade:

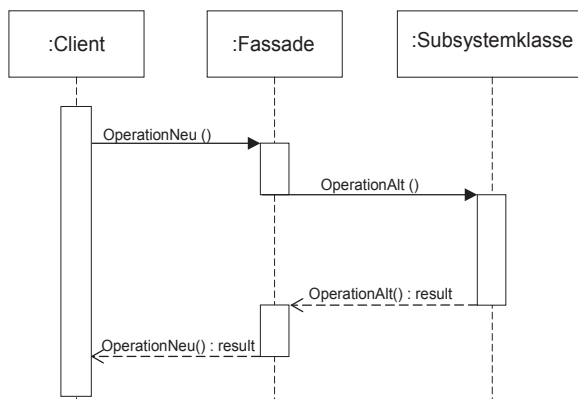


Bild 4-18 Sequenzdiagramm des Fassade-Musters

#### 4.6.3.4 Programmbeispiel

In dem folgenden Beispiel wird das Fassade-Muster dazu verwendet, die Lagerverwaltung dreier Lager zu vereinfachen. Diese Lager bilden drei Subsystemklassen. Es soll durch eine Fassade sichergestellt werden, dass aus jedem der drei Lager stets gleich viele Teile entnommen bzw. ihm hinzugefügt werden.

Die Klasse `FahrwerkLager` repräsentiert die erste Subsystemklasse:

[illegible]

Die Klasse `GetriebeLager` stellt die zweite Subsystemklasse dar:

[illegible]

Die Klasse `MotorLager` bildet die dritte Subsystemklasse:

**// Datei: MotorLager.java**

```
public class MotorLager
{
    int motorenAnzahl = 0;

    public void lagerFuellen (int anzahl)
    {
        motorenAnzahl = motorenAnzahl + anzahl;
        System.out.println ("Lager wurde um " + anzahl +
                             " Motoren aufgestockt.");
    }

    public void motorEntnehmen (int anzahl)
    {
        motorenAnzahl = motorenAnzahl - anzahl;
        System.out.println ("Fuer die Produktion wurden " + anzahl +
                             " Motoren entnommen.");
    }
}
```

Die Klasse `LagerFassade` stellt die Fassade dar. Sie ermöglicht die gemeinsame Aufstockung und Abbuchung von Lagerteilen aus allen drei Lagern mit den Methoden `alleLagerFuellen()` und `produktionsteileEntnehmen()`:

**// Datei: LagerFassade.java**

```
public class LagerFassade
{
    private FahrwerkLager fw;
    private GetriebeLager g;
    private MotorLager m;

    LagerFassade()
    {
        fw = new FahrwerkLager();
        g = new GetriebeLager();
        m = new MotorLager();
    }

    public void alleLagerFuellen (int anzahl)
    {
        System.out.println ("Bestaende aller Lager werden " +
                             "gefüllt.");
        fw.lagerFuellen (anzahl);
        g.lagerFuellen (anzahl);
        m.lagerFuellen (anzahl);
        System.out.println();
    }
}
```



```

public void produktionsteileEntnehmen (int anzahl)
{
    System.out.println ("Alle fuer die Produktion " +
                        "notwendigen Teile werden entnommen.");
    fw.fahrwerkEntnehmen (anzahl);
    g.getriebeEntnehmen (anzahl);
    m.motorEntnehmen (anzahl);
    System.out.println();
}
}

```

Die Klasse `TestFassade` erstellt ein Objekt der Klasse `LagerFassade` und kann somit alle drei Lager mit den von der Fassade bereitgestellten Methoden verwalten:

**// Datei: TestFassade.java**

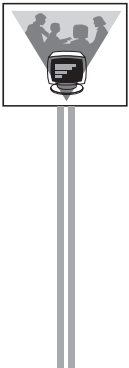
```

public class TestFassade
{
    public static void main (String[] args)
    {
        LagerFassade fassade = new LagerFassade();

        // Lager werden durch Lagerverwaltung gefuehlt.
        fassade.alleLagerFuellen (10);

        // Teile werden durch die Produktion entnommen.
        fassade.produktionsteileEntnehmen (5);
    }
}

```



Hier das Protokoll des Programmlaufs:

```

Bestaende aller Lager werden gefuehlt.
Lager wurde um 10 Fahrwerke aufgestockt.
Lager wurde um 10 Getriebe aufgestockt.
Lager wurde um 10 Motoren aufgestockt.

```

```

Alle fuer die Produktion notwendigen Teile werden
entnommen.
Fuer die Produktion wurden 5 Fahrwerke entnommen.
Fuer die Produktion wurden 5 Getriebe entnommen.
Fuer die Produktion wurden 5 Motoren entnommen.

```

## 4.6.4 Bewertung

### 4.6.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Subsystemklassen kennen ihre Fassade nicht.
- Die Verwendung von Subsystemfunktionen kann einfacher werden.

- Ein Subsystem und eine darauf zugreifende Client-Klasse sind lose gekoppelt. Dadurch wird eine Änderung oder ein Austausch eines Subsystems erleichtert. Man muss in diesem Fall nur die zugeordnete Fassade anpassen, nicht jedoch die Clients, die diese Fassade benutzen.
- Die Fassade dient oft zur objektorientierten Einhüllung von **Legacy-Systemen**<sup>46</sup>. Ein nicht objektorientiertes System kann dadurch plötzlich objektorientiert wirken. Die Fassade kapselt die Klassen/Objekte eines Subsystems bzw. eines Legacy-Systems. Ein Direktzugriff darf bei einem Legacy-System nicht erlaubt werden, wenn es durch die Fassade gekapselt und objektorientiert erscheinen soll.

#### 4.6.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Man führt einen zusätzlichen Methodenaufruf ein. Bei einem einfachen Subsystem schafft man dadurch zusätzlichen Ballast.
- Meist wird eine Funktionalität einer Subklasse durch die Fassade eingeschränkt.
- Wenn sich die Schnittstellen der gekapselten Komponenten oft ändern, muss die Fassade eventuell ebenfalls häufig geändert werden.
- Das Fassade-Muster bietet eine einfache Schnittstelle zum Zugriff auf ein Subsystem, es lässt aber offen, ob direkte Zugriffe auf die Klassen des Subsystems weiterhin möglich sind. Dies muss im Einzelfall entschieden werden. Der direkte Zugriff ist unter Umständen performanter und bietet die gesamte, aber auch komplexere Funktionalität des Subsystems. Solange der Direktzugriff erlaubt ist, kann man im Subsystem nicht die Vorteile des Information Hiding nutzen.

#### 4.6.5 Einsatzgebiete

Das Fassade-Muster ist generell bei Subsystemen anwendbar, wenn die Schnittstelle frei entworfen werden kann. Das Muster kann eingesetzt werden, um die Komplexität eines Subsystems zu verbergen. Außerdem kann es zur Kapselung eines Legacy-Systems verwendet werden.

Von einer höheren Ebene aus betrachtet, kann eine Fassade auch mehrere Subsysteme kapseln.

Bei einem Schichtenmodell (siehe Architekturmuster **Layers**) kann eine neue Schicht mit Hilfe des Fassade-Musters eingeführt werden.

#### 4.6.6 Ähnliche Entwurfsmuster

Sowohl das Fassade-Muster als auch das **Adapter**-Muster kapseln vorhandene Programme. Fassade und Adapter sind Wrapper. Im Gegensatz zur Fassadenschnittstelle, die von einem Subsystem definiert wird, wird die Adapterschnittstelle vom Client

---

<sup>46</sup> Ein Legacy-System ist in der Informatik ein historisch gewachsenes System, oft in einer veralteten Technologie. Es stellt deshalb sozusagen oftmals eine "Altlast" dar.

vorgegeben. Hinter der Fassade verbergen sich Subsystemklassen, hinter dem Adapter verbirgt sich nur eine einzige Klasse. Für eine Fassadenschnittstelle gibt es keine scharfen Vorgaben von außerhalb des Subsystems. Sie kann vom Subsystem frei festgelegt werden. Meist wird die Schnittstelle eines Subsystems durch die Verwendung einer Fassade vereinfacht. Der Adapter dient zur genauen Anpassung einer Schnittstelle an die für den Client erforderliche Form.

Fassade und **Vermittler** abstrahieren von der Funktionalität existierender Klassen. Aufgabe eines Vermittlers ist es, die Kommunikation zwischen mehreren Objekten untereinander zu erleichtern, während es Aufgabe einer Fassade ist, die Verwendung von mehreren Objekten zu vereinfachen. Alle miteinander kommunizierenden Objekte kennen ihren Vermittler. Die Objekte, die über eine Fassade verwendet werden, wissen von der Fassade nichts.

Eine bei der Anwendung des Entwurfsmusters **Abstrakte Fabrik** gebildete konkrete Fabrik kann auch als Fassade angesehen werden. Allerdings beschränkt sich die Funktionalität einer Fabrik auf die Erzeugung von Produkten – sie kapselt nicht die Nutzung der Produkte. Das bedeutet, dass ein Client nicht nur auf die Fassade – also die Fabrik – zugreifen muss, sondern auch einen Direktzugriff auf die Produktklassen benötigt, um die von der Fabrik erzeugten Produkte zu nutzen.

## 4.7 Das Strukturmuster Kompositum

### 4.7.1 Name/Alternative Namen

Kompositum (engl. composite), Kompositionsmuster.

### 4.7.2 Problem

Man möchte Teil-Ganzes-Hierarchien erzeugen und dabei die Objekte in einer baumartigen Struktur gruppieren. Das folgende Bild gibt ein Beispiel einer solchen Teil-Ganzes-Hierarchie:

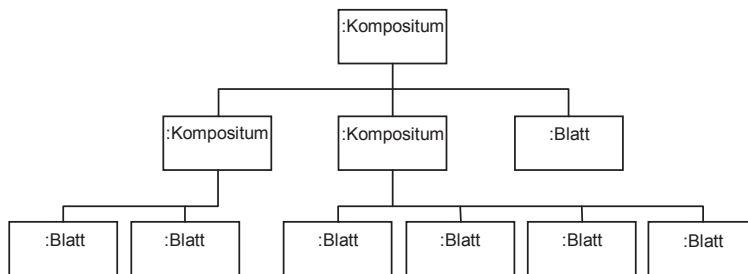


Bild 4-19 Struktur einer Teil-Ganzes-Hierarchie

Die Objekte in einer Baumstruktur werden auch als **Knoten** bezeichnet. Wie in Bild 4-19 zu sehen ist, können die Knoten der Baumstruktur dahingehend unterschieden werden, ob sie selber wieder zusammengesetzt sind (Kompositum-Objekte) oder ob es sich um einfache und nicht zusammengesetzte – also quasi atomare – Objekte (Blatt-Objekte) handelt. Ein Client-Programm soll für ausgesuchte Operationen mit einem Blatt-Objekt wie mit einem Kompositum-Objekt umgehen können, so dass für einen Client keine Unterscheidungen erforderlich werden. Ein Client soll nur die abstrakten Schnittstellen eines Knoten kennen, sei er zusammengesetzt oder nicht. Für ein Client-Programm soll es also verborgen sein, ob ein Knoten einfach oder zusammengesetzt ist.

Das **Kompositum-Muster** soll es erlauben, dass bei der Verarbeitung von Knoten in einer Baumstruktur einfache und zusammengesetzte Objekte gleich behandelt werden.



### 4.7.3 Lösung

Das Kompositum-Muster ist ein objektbasiertes Strukturmuster<sup>47</sup>. Durch den Einsatz des Kompositum-Musters wird es möglich, in einer Baumstruktur **zusammengesetzte**

<sup>47</sup> Kompositum ist zugleich auch der Name für eine Klasse, deren Objekte als Knoten in einem Baum auftreten können und deren Objekte selber auch weitere Knoten aggregieren können, um so die Baumstruktur aufzubauen,

**Objekte** (Gruppen von Objekten) gleich wie einzelne **einfache** oder **primitive** Objekte, sogenannte **Blätter**, zu behandeln. Dadurch wird der Aufwand im Client für die Verwaltung der resultierenden Baumstruktur verringert.

Die Grundlage des Kompositum-Musters ist die Definition einer abstrakten Klasse `Knoten`, deren Verhalten durch ihre Schnittstelle und ihre Verträge festgelegt wird. Eine Kompositum-Klasse ist von der abstrakten Klasse `Knoten` abgeleitet. Ein Kompositum-Objekt kann auf Grund des Klassendiagramms in Bild 4-20 gleichzeitig mehrere Objekte vom Typ `Knoten` aggregieren. An die Stelle einer Referenz auf ein Objekt der Klasse `Knoten` kann nach dem liskovschen Substitutionsprinzip eine Referenz auf ein Objekt einer von der Klasse `Knoten` abgeleiteten Klasse treten, wenn der Vertrag der Klasse `Knoten` beim Überschreiben eingehalten wird.

Eine Blatt-Klasse ist ebenfalls von der abstrakten Klasse `Knoten` abgeleitet, kann aber im Gegensatz zu einer Kompositum-Klasse keine Knoten aggregieren. Objekte von Blatt-Klassen – also **Blätter** – stellen somit Abschlusselemente einer Baumstruktur dar.

Damit ist es leicht, geschachtelte Objektstrukturen (Bäume) zu bilden.

Ein Baum besteht aus konkreten Objekten und ein Objekt ist entweder eine Instanz einer Kompositum- oder einer Blatt-Klasse. Das Kompositum-Muster ermöglicht es, eine Baumstruktur von Objekten aufzubauen.



Die Unterscheidung, ob es sich bei einer Klasse der Baumstruktur um eine Blatt-Klasse oder eine Kompositum-Klasse handelt, erfolgt lediglich anhand der Tatsache, ob ein Objekt dieser Klasse weitere Objekte der Baumstruktur aggregieren kann oder nicht.

Wird eine Nachricht an ein Kompositum versendet, so wird die Nachricht zum einen lokal für das zusammengesetzte Objekt ausgeführt und zum anderen an die Kinder weiterdelegiert (**Delegationsprinzip**). Ist der Empfänger hingegen ein Blatt, so wird die Operation direkt ausgeführt, da das Blatt keine Kinder hat.



#### 4.7.3.1 Klassendiagramm

Das Klassendiagramm in Bild 4-20 zeigt, dass eine Klasse `Blatt` und eine Klasse `Kompositum` von einer abstrakten Klasse `Knoten` abgeleitet sind und dass ein Objekt der Klasse `Kompositum` mehrere Objekte vom Typ `Knoten` aggregieren kann:

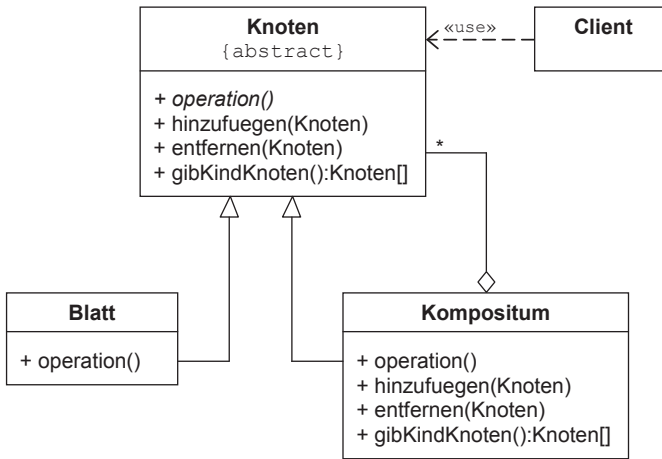


Bild 4-20 Klassendiagramm Kompositum

Wenn der Client nicht wissen soll, ob ein Objekt ein Blatt oder ein Kompositum ist, müssen beide Objekte dieselbe Schnittstelle haben. Die Implementierungen der Operation `operation()` können jedoch bei den verschiedenen Klassen voneinander abweichen.

Ein Client kann sowohl für ein Blatt als auch für ein Kompositum eine Kindoperation wie z. B. `gibKindKnoten()` aufrufen. Er soll ja nicht wissen, um was es sich bei einem Element handelt. Deshalb müssen die Operationen `hinzufuegen()`, `entfernen()` und `gibKindKnoten()` in der Wurzel der Klassenhierarchie bei der Komponente definiert werden. Es ist sinnvoll, in der Wurzelklasse `Knoten` ein Defaultverhalten für diese Kindoperationen zu implementieren, das für Blatt-Klassen geeignet ist und das von einer Kompositum-Klasse überschrieben wird.

Eine gängige Lösung ist, dass die abstrakte Klasse die Kindmethoden so implementiert, dass diese Methoden eine Exception werfen, wenn sie aufgerufen werden. Die Blatt-Klasse überschreibt dieses Verhalten nicht und wirft also eine Exception, wenn eine ihrer Kindmethoden aufgerufen wird. Die Kompositum-Klasse hingegen überschreibt die Kindmethoden in sinnvoller Weise und wirft dabei keine Exception. Dies bedeutet, dass die Nachbedingung einer Kindmethode in der Kompositum-Klasse verschärft wird, was den Vertrag einer Kindmethode nicht bricht.

Die Klassen `Kompositum` und `Blatt` sind im Bild 4-20 nur als Stellvertreter zu betrachten. In einer Baumstruktur kann es sowohl mehrere Kompositum-Klassen als auch mehrere Blatt-Klassen geben. Ein Beispiel für eine Baumstruktur mit mehreren Blatt-Klassen ist in Kapitel 4.7.5 zu sehen.

### 4.7.3.2 Teilnehmer

#### Knoten

Die abstrakte Klasse `Knoten` legt die Schnittstelle und das Verhalten der abgeleiteten Klassen `Kompositum` und `Blatt` fest. Es wird ein Defaultverhalten für die Kindoperationen implementiert.

#### Blatt

Die Klasse `Blatt` repräsentiert ein Abschlusselement in der Baumstruktur, das keine weiteren Knoten aggregiert und selbst immer nur Kind-Knoten sein kann.

#### Kompositum

Die Klasse `Kompositum` repräsentiert ein Knotenelement in der Baumstruktur, welches weitere Knoten aggregieren kann. Die Klasse `Kompositum` implementiert die kindbezogenen Operationen und überschreibt damit das Defaultverhalten, das in der Klasse `Knoten` implementiert ist.

### 4.7.3.3 Dynamisches Verhalten

Das dynamische Verhalten des Kompositum-Musters wird an einem Beispiel in Bild 4-21 verdeutlicht. Zuerst fügt der Client dem Objekt `k1` der Klasse `Kompositum` zwei Objekte `b1` und `b2` der Klasse `Blatt` sowie ein Objekt `k2` der Klasse `Kompositum` hinzu. Anschließend wird dem Objekt `k2` der Klasse `Kompositum` noch ein weiteres Blatt `b3` hinzugefügt. Die Baumstruktur sieht somit folgendermaßen aus:

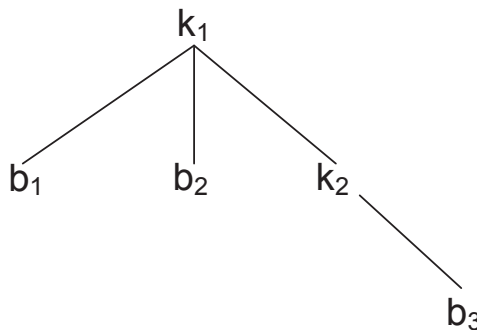


Bild 4-21 Baumstruktur des Beispiels

Nach dem Aufbau der Baumstruktur wird die Methode `operation()` des Objekts `k1` beispielhaft von einem Client aufgerufen. Der gesamte Ablauf dieses Aufrufs ist im folgenden Sequenzdiagramm dargestellt:

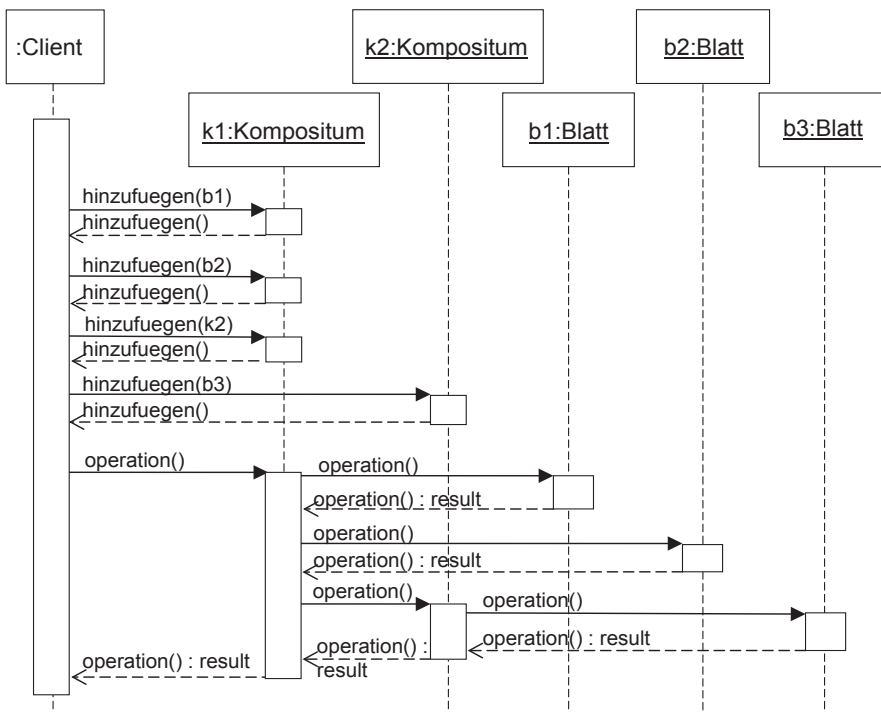


Bild 4-22 Sequenzdiagramm Kompositum-Muster

Wie in Bild 4-22 zu sehen ist, wird beim Aufruf der Methode `operation()` des Objekts `k1` die Anfrage rekursiv an alle untergeordneten Blätter und Kompositum-Instanzen weitergeleitet.

#### 4.7.3.4 Programmbeispiel

Die Klasse `Knoten` definiert eine abstrakte Basisklasse, von der die Klassen `Kompositum` und `Blatt` abgeleitet werden. Die Klasse `Knoten` definiert die abstrakte Methode `operation()`. Es folgt der Quellcode der Klasse `Knoten`:

```
// Datei: Knoten.java
import java.util.ArrayList;
public abstract class Knoten
{
    private String name = "";
    static int ebene = 0;           // Zaehler fuer Ausgabe-Ebene
    ArrayList<Knoten> kindelemente = new ArrayList<Knoten>();

    public Knoten (String name)
    {
        this.name = name;
    }

    public abstract void operation();
}
```



```
public void hinzufuegen (Knoten komp)
{
    System.out.println ("Kind-Methode nicht implementiert!");
}

public void entfernen (Knoten komp)
{
    System.out.println ("Kind-Methode nicht implementiert!");
}

public void gibKind()
{
    System.out.println ("Kind-Methode nicht implementiert!");
}

public String gibName()
{
    return this.name;
}
}
```

Die Klasse `Kompositum` ist von der Klasse `Knoten` abgeleitet. Sie überschreibt die kindbezogenen Methoden und implementiert die ausgesuchte Operation `operation()`. Hier die Klasse `Kompositum`:

```
// Datei: Kompositum.java
import java.util.Iterator;
public class Kompositum extends Knoten
{
    public Kompositum (String name)
    {
        super (name);
    }

    public void hinzufuegen (Knoten komp)
    {
        this.kindelemente.add (komp);
    }

    public void entfernen (Knoten komp)
    {
        //alle Kindelemente durchlaufen
        for (Iterator<Knoten> iter = kindelemente.iterator();
             iter.hasNext();)
        {
            Knoten f = (Knoten) iter.next();
            if (f instanceof Kompositum)
            {
                ((Kompositum) f).entfernen (komp);
            }
        }
        kindelemente.remove (komp);
    }
}
```

```

public void operation()
{
    String formatString;
    // Berechnen der neuen Ausgabe-Ebene
    ebene++;
    // Berechnen des Formatstrings fuer die Ausgabe von
    // Leerzeichen entsprechend der erreichten Ebene
    formatString = "%" + (ebene * 2) + "s";
    // Ausgabe der Leerzeichen
    System.out.printf (formatString, "");
    // Ausgabe eines Pluszeichens gefolgt vom Namen der Komponente
    System.out.println ("+ " + super.gibName() + "");
    // Aufruf der Operation fuer alle Kindelemente
    for (Iterator<Knoten> iter = kindelemente.iterator();
        iter.hasNext();)
    {
        Knoten f = (Knoten) (iter.next());
        f.operation();
    }
    // Ausgabe-Ebene wieder zuruecksetzen
    --ebene;
}
}

```

Die Klasse `Blatt` repräsentiert einfache und nicht zusammengesetzte Knoten einer Baumstruktur und hat im Gegensatz zur Klasse `Kompositum` keine untergeordneten Knoten:

```

// Datei: Blatt.java
public class Blatt extends Knoten
{
    public Blatt (String name)
    {
        super (name);
    }

    public void operation()
    {
        String formatString;
        // Berechnen des Formatstrings fuer die Ausgabe von
        // Leerzeichen entsprechend der erreichten Ebene
        formatString = "%" + (ebene * 2) + "s";
        // Ausgabe der Leerzeichen
        System.out.printf (formatString, "");
        // Ausgabe eines Minuszeichens gefolgt vom Namen des Knotens
        System.out.println (" - " + super.gibName());
    }
}

```

Die Klasse `TestKompositum` legt vier Kompositum-Objekte an. Um die Möglichkeit der Rekursion aufzuzeigen, wird `komp121` dem Kompositum `komp12` hinzugefügt. Anschließend werden sechs Instanzen der Klasse `Blatt` erstellt. Diese werden dann den Kompositum-Objekten hinzugefügt. Im nächsten Schritt wird der Inhalt der Baumstruktur ausgegeben. Zum Schluss werden ein `Blatt`- sowie ein `Kompositum`-Objekt

entfernt und der Baum erneut ausgegeben. Hier nun der Quellcode der Klasse `TestKompositum`:

```
// Datei: TestKompositum.java
public class TestKompositum
{
    public static void main (String[] args)
    {
        System.out.println ("Testprogramm zum Kompositum-Muster");
        System.out.println ("");

        // Aufbau der Objektstruktur
        Kompositum komp = new Kompositum ("Kompositum Ebene 1");
        Kompositum komp11 =
            new Kompositum ("Kompositum Ebene 11");
        Kompositum komp12 =
            new Kompositum ("Kompositum Ebene 12");
        Kompositum komp121 =
            new Kompositum ("Kompositum Ebene 121");

        komp.hinzufuegen (komp11);
        komp.hinzufuegen (komp12);

        komp12.hinzufuegen (komp121);

        Blatt blatt111 = new Blatt ("Blatt111");
        Blatt blatt112 = new Blatt ("Blatt112");

        Blatt blatt121 = new Blatt ("Blatt121");
        Blatt blatt122 = new Blatt ("Blatt122");
        Blatt blatt123 = new Blatt ("Blatt123");

        Blatt blatt1211 = new Blatt ("Blatt1211");

        komp11.hinzufuegen (blatt111);
        komp11.hinzufuegen (blatt112);

        komp12.hinzufuegen (blatt121);
        komp12.hinzufuegen (blatt122);
        komp12.hinzufuegen (blatt123);

        komp121.hinzufuegen (blatt1211);

        // Aufruf der Operation fuer das Kompositum
        System.out.println("Erste Ausgabe der Kompositum-Operation");
        System.out.println();
        komp.operation();

        // Modifikation der Objektstruktur
        komp12.entfernen (blatt122);
        komp12.entfernen (komp121);

        // erneuter Aufruf der Operation fuer das Kompositum
        System.out.println();
        System.out.println("Zweite Ausgabe der Kompositum-Operation");
        System.out.println();
        komp.operation();
    }
}
```



Hier das Protokoll des Programmlaufs:

Testprogramm zum Kompositum-Muster

Erste Ausgabe der Kompositum-Operation

```
+ Kompositum Ebene 1
  + Kompositum Ebene 11
    - Blatt111
    - Blatt112
  + Kompositum Ebene 12
    + Kompositum Ebene 121
      - Blatt1211
      - Blatt121
      - Blatt122
      - Blatt123
```

Zweite Ausgabe der Kompositum-Operation

```
+ Kompositum Ebene 1
  + Kompositum Ebene 11
    - Blatt111
    - Blatt112
  + Kompositum Ebene 12
    - Blatt121
    - Blatt123
```

## 4.7.4 Bewertung

### 4.7.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Da ein Objekt der Klasse `Blatt` dieselbe Schnittstelle hat wie ein Objekt der Klasse `Kompositum`, kann der Client `Blatt`-Objekte und zusammengesetzte `Kompositum`-Objekte einheitlich behandeln. Dies vereinfacht die Handhabung der Baumstruktur durch den Client.
- Das Strukturmuster `Kompositum` erlaubt es, verschachtelte Strukturen auf einfache Weise zu erzeugen bzw. um neue `Blatt`- bzw. `Kompositum`-Klassen zu erweitern. `Blatt`- und `Kompositum`-Klassen verwenden dieselbe Schnittstelle. Hierbei kann das zusammengesetzte Objekt mehrfach verschachtelt sein.

### 4.7.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Das Design und der Aufbau der Baumstruktur werden unübersichtlich, wenn man viele unterschiedliche `Blatt`- und `Kompositionsklassen` verwendet. Dieses Problem

entsteht aber durch die Art, wie das Strukturmuster Kompositum verwendet wird, und nicht durch das Strukturmuster selbst.

- Im Kompositum-Muster werden alle Knoten gleich behandelt. Bei Anwendungen, in denen der Aufbau einer Kompositum-Struktur gewissen Einschränkungen unterliegen soll, müssen diese Einschränkungen durch Typprüfungen zur Laufzeit gewährleistet werden. Als Beispiel hierzu wird folgendes Szenario betrachtet: Ordner können Ordner und Dateien enthalten. Source-Ordner sind ebenfalls Ordner, können aber nur bestimmte Dateien, nämlich Quellcode-Dateien enthalten. Eine solche Einschränkung kann über das Kompositum-Muster nicht umgesetzt werden.
- Das Entwurfsmuster Kompositum ist sehr mächtig. Sobald jedoch Änderungen an der Basisschnittstelle durchgeführt werden, wie z. B. das Hinzufügen einer neuen Methode, bedeutet dies auch, dass alle davon abgeleiteten Klassen potenziell ebenfalls geändert werden müssen.

#### 4.7.5 Einsatzgebiete

Das Kompositum-Muster kann im Zusammenhang mit dem Entwurfsmuster **Befehl** angewandt werden. Mit Hilfe des Kompositum-Musters kann ein Befehl aus mehreren Befehlen zusammengesetzt werden. Dadurch können Befehlsskripte bzw. Makrobefehle erstellt werden und trotzdem im Rahmen des Befehlsmusters wie einfache Befehle behandelt werden.

Zusammengesetzte Nachrichten, die über mehrere Schichten einer Schichtenarchitektur (siehe Kapitel 5.1) weitergereicht werden, sind typischerweise ineinander verschachtelt und können mit Hilfe des Kompositum-Musters zusammengesetzt werden. Diese Anwendung des Kompositum-Musters ist in der Literatur als eigenständiges Muster unter dem Namen **Composite-Message** [Bus98] bekannt.

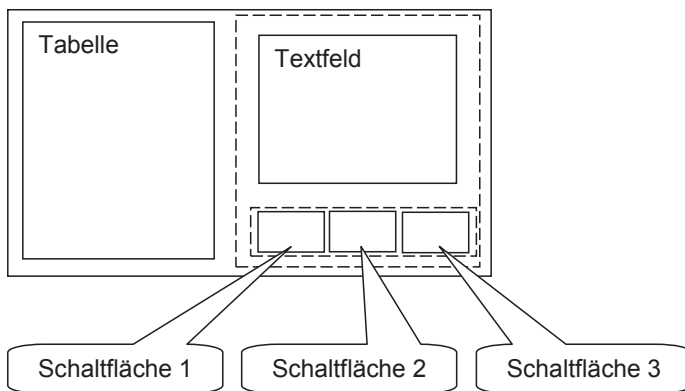
Beim Zusammensetzen von grafischen Oberflächen – wie es auch die Views im **MVC-Muster** sind –, ist das Kompositum-Muster weit verbreitet. Ein Fenster einer View wird dabei im Grafik-Paket Swing von Java aus Paneln aus verschachtelten Komponenten zusammengesetzt. Durch die Verschachtelung entsteht eine Baumstruktur aus Komponenten, die aus Komponenten zusammengesetzt sind (in Swing Panels<sup>48</sup>), und aus Blättern, die keine anderen Komponenten in sich tragen (z. B. Schaltflächen). Die Anwendung des Kompositum-Musters bei graphischen Oberflächen wird im folgenden Beispiel im Detail ausgeführt.

#### Anwendungsbeispiel: Grafische Oberflächen mit Swing

Es soll die in folgendem Bild dargestellte Oberfläche erzeugt werden:

---

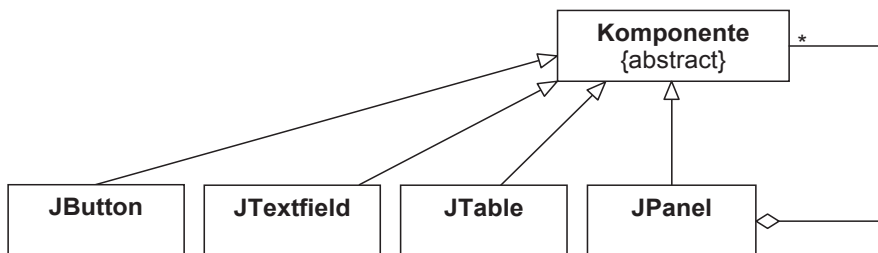
<sup>48</sup> Panels stellen unsichtbare Objektbehälter dar, mit deren Hilfe es möglich ist, grafische Komponenten zu Gruppen zusammenzufassen.



*Bild 4-23 Zu realisierende grafische Oberfläche*

Dabei repräsentieren die gestrichelten Linien unsichtbare Behälterobjekte (Panels), die andere grafische Komponenten gruppieren können. Der äußere Rahmen (Frame) stellt quasi das Wurzelement der verschachtelten Struktur dar.

In Grafikpaket Swing von Java gibt es zur Realisierung dieser Beispieloberfläche die folgenden Klassen: `JTextField` für Textfelder, `JButton` für Schaltflächen, `JTable` für Tabellen und `JPanel` für die Gruppierung von Gestaltungselementen. Dabei erben alle diese Klassen von derselben Basisklasse, nämlich von der Klasse `JComponent`. Die Vererbung geht bei einigen Klassen über mehrere Stufen, was im Klassendiagramm in Bild 4-24 nicht gezeigt ist, damit die Analogie zum Klassendiagramm des Kompositum-Musters in Bild 4-20 deutlicher wird:



*Bild 4-24 Klassendiagramm für eine Anwendung des Kompositum-Musters*

Die Klasse `JComponent` selbst ist abgeleitet von der Klasse `Container` aus dem Paket der grafischen Benutzerschnittstelle AWT (**A**bstr**ac**t **W**indow **T**oolkit). Damit haben alle in Bild 4-24 gezeigten Klassen die Eigenschaften der Klasse `Container` und können beliebig viele grafische Elemente aufnehmen. Diese Fähigkeiten werden aber im Bild nur von der Klasse `JPanel` genutzt, was durch die Aggregationsbeziehung im Bild 4-24 angedeutet wird. Die Klasse `JPanel` ist also im Sinne des Kompositum-Musters eine Kompositum-Klasse, während die Klassen `JTextField`, `JButton` und `JTable` Blatt-Klassen sind.

Daneben gibt es noch die Klasse `JFrame`, die ebenfalls von der Klasse `Container` abgeleitet ist, die aber nur als Wurzelement für den äußeren Rahmen einer grafischen Oberfläche genutzt werden kann und daher nicht als Kompositum-Klasse angesehen werden kann.

Das folgende Codefragment zeigt, wie man in Swing die in Bild 4-23 skizzierte Oberfläche mit Objekten der genannten Klassen zusammensetzen kann. Die Layoutmanager werden bei dieser Betrachtung außer Acht gelassen. Hier das bereits erwähnte Codefragment:

```
// Blatt-Objekte erzeugen
JTextField textField = new JTextField ("Textfeld");
JButton button1 = new JButton ("Schaltflaeche 1");
JButton button2 = new JButton ("Schaltflaeche 2");
JButton button3 = new JButton ("Schaltflaeche 3");
JTable table = new JTable();

// Kompositum-Objekte erzeugen
JPanel panelRechts = new JPanel();
JPanel panelUnten = new JPanel();

// Struktur aufbauen
panelRechts.add (textField);
panelRechts.add (panelUnten);
panelUnten.add (button1);
panelUnten.add (button2);
panelUnten.add (button3);

// in aeusseren Rahmen einhaengen
JFrame frame = new JFrame ("Mein Fenster");
frame.add (table);
frame.add (panelRechts);
```

Was passiert nun, wenn man die obige Klassenhierarchie zugrunde legt und den äußeren Rahmen vergrößert? Das `Frame`-Objekt vergrößert sich selbst und leitet den Aufruf an alle Objekte, die in ihm enthalten sind, weiter. Durch die Weiterleitung geht der Aufruf unter anderem an das Objekt `panelRechts`. Dieses Objekt ist selber ein Kompositum-Objekt und leitet wiederum den Aufruf an alle enthaltenen Objekte weiter. Dies geht rekursiv weiter, bis alle Elemente vergrößert sind.

## 4.7.6 Ähnliche Entwurfsmuster

Von der Struktur des Klassendiagramms her ist das Entwurfsmuster Kompositum sehr ähnlich zu dem **Dekorierer-Muster**. Ein Klassendiagramm des Dekorierers ist quasi ein Spezialfall des Klassendiagramms des Kompositum-Musters, dadurch dass ein Dekorierer-Objekt nur ein einziges Objekt aggregieren kann. Es zeigen sich jedoch Unterschiede in der Verwendung dieser beiden Muster: Während mit Hilfe des Dekorierer-Musters Funktionalität dynamisch gebildet werden kann, dient das Kompositum-Muster dazu, Objekte zu einer hierarchischen Struktur zusammenzusetzen.

Das Muster **Whole-Part** [Bus98] ist sowohl vom Aufbau als auch von der Funktionalität her sehr ähnlich dem Kompositum-Muster. Das Muster Whole-Part fordert aber im

Unterschied zum Kompositum-Muster, dass auf die Teilobjekte (Parts) einer Struktur nicht mehr zugegriffen werden darf, sondern von außen nur das Gesamtobjekt (Whole) angesprochen werden darf. Auf Grund dieser Forderung müssen die beim Muster Whole-Part beteiligten Teilobjekte nicht unbedingt alle die gleiche Schnittstelle anbieten wie beim Kompositum-Muster.

Als Variante von Whole-Part wird in [Bus98] das Muster **Assembly-Parts** vorgestellt. Im Muster Assembly-Parts ist die Struktur des Gesamtobjekts starr festgelegt und auch in diesem Muster können die Teilobjekte unterschiedliche Schnittstellen haben. Als Beispiel wird in [Bus98] angeführt, dass ein Auto aus Karosserie, Fahrwerk, Motor etc. besteht und dass diese Struktur sich zwischen verschiedenen Autotypen nicht grundsätzlich ändert. Das Kompositum-Muster hingegen schränkt die Flexibilität beim Aufbau einer Datenstruktur prinzipiell nicht ein und lässt auch nachträgliche Änderungen an der Datenstruktur zu. Ein weiterer Unterschied ist, dass das Muster Assembly-Parts keine gemeinsame Schnittstelle für die Teilobjekte voraussetzt, wie sie im Kompositum-Muster durch die Klasse `Knoten` (siehe Bild 4-20) vorgegeben ist.



## 4.8 Das Strukturmuster Proxy

### 4.8.1 Name/Alternative Namen

Stellvertreter (engl. proxy), Proxy, Surrogat<sup>49</sup> (engl. surrogate).

### 4.8.2 Problem

Ein Client-Programm soll nur indirekt über ein Stellvertreter-Objekt (Proxy-Objekt) auf ein bestimmtes echtes Objekt einer Anwendung zugreifen können. Das Stellvertreter-Objekt soll den Zugang zum echten Objekt vermitteln und soll ggf. eine zusätzliche Funktionalität vor dem echten Objekt einschieben können.

Das **Proxy-Muster** soll die Existenz eines echten Objekts hinter einem Stellvertreter mit demselben Interface verbergen. Der Stellvertreter soll die Kommunikation zum echten Objekt kapseln. Er soll den Aufruf einer Methode an das echte Objekt weiter delegieren und soll dabei ggf. eine Zusatzfunktionalität einschieben können.



### 4.8.3 Lösung

Ein Stellvertreter-Objekt (Proxy-Objekt oder kurz Proxy) erhält das Interface des echten Objekts. Der Zugang zum echten Objekt selbst erfolgt über den Proxy. Das Client-Programm greift dann direkt auf den entsprechenden Proxy als Stellvertreter des echten Objekts zu und der Proxy anschließend auf das echte Objekt. Das Client-Programm kennt nur den Proxy. Das echte Objekt ist dem Client verborgen. Der Zugang zum echten Objekt kann mit Aufwand verbunden sein, z. B. weil das echte Objekt nur über das Netzwerk erreichbar ist. Die gesamte Kommunikation des Proxys mit dem echten Objekt wird im Stellvertreter-Objekt gekapselt.

Das Proxy-Muster ist ein objektbasiertes Entwurfsmuster, bei dem das Proxy-Objekt den Zugriff auf das echte Objekt kontrolliert.



Der Proxy kann als zusätzliche Funktionen (siehe Kapitel 4.8.5) beispielsweise Sicherheitsfunktionen oder Protokollierfunktionen implementieren, die im ursprünglichen echten Objekt gar nicht vorhanden sind.

Durch das Muster wird nicht festgelegt, wie das Proxy-Objekt Kenntnis vom echten Objekt erlangt. Beispielsweise kann das Proxy-Objekt zur Laufzeit das echte Objekt bei Bedarf erzeugen.

<sup>49</sup> Surrogat bedeutet Ersatz.

### 4.8.3.1 Klassendiagramm

In Bild 4-25 wird das Proxy-Muster im Allgemeinen dargestellt:

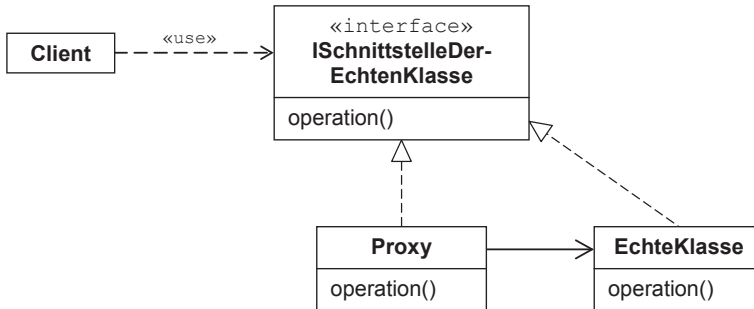


Bild 4-25 Klassendiagramm Proxy-Muster

Die Klasse `EchteKlasse` und die Klasse `Proxy` implementieren dasselbe Interface `ISchnittstelleDerEchtenKlasse`. Dadurch kann ein `Proxy`-Objekt als Stellvertreter eines echten Objekts fungieren. Über das Interface des echten Objekts stellt das `Proxy`-Objekt die Methoden des echten Objekts dem Client-Programm zur Verfügung. Das `Proxy`-Objekt schiebt sich einfach zwischen zwei vorhandene Objekte (`Client` und echtes Objekt). Wenn der `Client` eine Methode vom Typ des Interface bei einem Objekt einer Klasse aufruft, so bedeutet das, dass der Methodenaufwurf bei einem Objekt einer Klasse stattfindet, das dieses Interface implementiert. Das kann ein echtes Objekt sein, aber auch ein Stellvertreter-Objekt, das den Methodenaufwurf des Clients entgegennimmt und ihn an das echte Objekt weiterleitet.

Außer der Delegation kann das Stellvertreter-Objekt – wie schon gesagt – eine zusätzliche Funktionalität durchführen. Es ist auch möglich, dass das Stellvertreter-Objekt einen Methodenaufwurf komplett selbst ausführt, ohne das echte Objekt zu benutzen, wenn es dazu in der Lage ist. Dies ist z. B. der Fall, wenn Konstanten des echten Objekts abgefragt werden, die auch der `Proxy` hat. Beispiele für mögliche Varianten des `Proxy`-Musters sind in Kapitel 4.8.5 zu finden.

Ein `Client` hat eine Referenz vom Typ des Interface `ISchnittstelleDerEchtenKlasse`. Diese Referenz zeigt in der Regel auf ein Objekt der Klasse `Proxy`, also auf ein Stellvertreter-Objekt – was aber für einen `Client` verborgen ist. Ein `Client` weiß also nicht, wie Methodenaufwürfe, die er an das referenzierte Objekt stellt, in der Folge weiterverarbeitet bzw. weiterdelegiert werden.



### 4.8.3.2 Teilnehmer

#### ISchnittstelleDerEchtenKlasse

Das Interface `ISchnittstelleDerEchtenKlasse` stellt die Methodenköpfe einer echten Klasse zur Verfügung, die vom `Client` verwendet werden.

### Client

Der Client ruft die Methoden eines Objekts über eine Referenz auf das Interface `ISchnittstelleDerEchtenKlasse` der echten Klasse auf. Welches Objekt (Objekt der Klasse `Proxy` oder `EchteKlasse`) sich dahinter verbirgt, spielt für den Client keine Rolle. Wie die Referenz gesetzt wird, wird durch das Muster nicht festgelegt. Da der Client ein Interface als Typ für die Referenz nutzt, ist er nicht abhängig von der Ausprägung der Klassen, die dieses Interface implementieren. Zum Setzen der Referenz können beispielsweise Techniken der Dependency Injection (siehe Kapitel 1.10.2) eingesetzt werden.

### EchteKlasse

Die echte Klasse implementiert die Methoden des Interface `ISchnittstelleDerEchtenKlasse`.

### Proxy

Die Klasse `Proxy` implementiert die Methoden des Interface `ISchnittstelleDerEchtenKlasse` und hält eine Referenz auf ein echtes Objekt. Das Objekt der Proxy-Klasse delegiert bei der Implementierung des Interface in der Regel einen Aufruf einer Methode an das echte Objekt weiter. Je nach Art der Proxy-Variante werden außer der Delegation an das echte Objekt bei einem Methodenaufruf beispielsweise Statistiken erstellt, Rechte geprüft etc. (siehe Kapitel 4.8.5).

#### 4.8.3.3 Dynamisches Verhalten

Der Client ruft eine Methode des `Proxy`-Objekts auf. Der Aufruf wird in der Regel an das Objekt der Klasse `EchteKlasse` weitergeleitet und abgearbeitet. Das Ergebnis wird danach an den Client zurückgegeben.

Das folgende Bild zeigt ein Sequenzdiagramm des Proxy-Musters für ein bereits vorhandenes echtes Objekt:

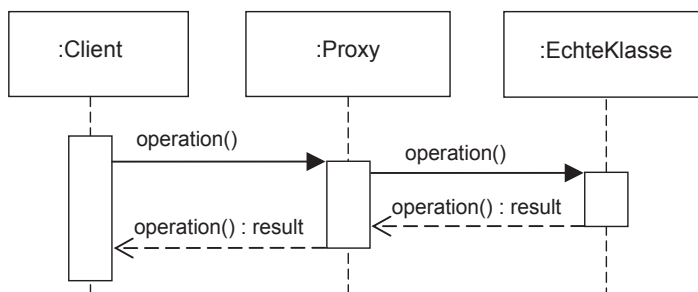


Bild 4-26 Sequenzdiagramm des Proxy-Musters

Da das Muster nicht festlegt, wie das Proxy-Objekt Kenntnis vom echten Objekt erlangt, kann das Proxy-Objekt beispielsweise zur Laufzeit das echte Objekt bei Bedarf erzeugen. Dies ist im folgenden Programmbeispiel ausgeführt.

#### 4.8.3.4 Programmbeispiel

Das Proxy-Muster soll an einem Beispiel zum Zugriff auf eine Datei vorgestellt werden. Das Interface `IDateiZugriff` definiert zwei Methoden `getName()` und `getInhalt()` zur Ausgabe des Dateinamens und des Inhalts der Datei. Um das Programm möglichst einfach zu halten, beschränkt sich der Dateizugriff auf eine simulierte Leseoperation. Hier das Interface `IDateiZugriff`:

```
// Datei: IDateiZugriff.java
public interface IDateiZugriff
{
    public String getName();
    public String getInhalt();
}
```

Die echte Klasse `DateiZugriff` implementiert dieses Interface, wobei die Leseoperation `getInhalt()` der Einfachheit halber in diesem Programm nur simuliert wird. Hier die Klasse `DateiZugriff`:

```
// Datei: DateiZugriff.java
public class DateiZugriff implements IDateiZugriff
{
    String name;

    public DateiZugriff (String name)
    {
        this.name = name;
        System.out.println ("Echtes Objekt instanziiert.");
    }

    public String getName()
    {
        return this.name;
    }

    public String getInhalt()
    {
        //Simulation einer Datei-Leseoperation
        return ("Daten von " + name);
    }
}
```

Ein Proxy speichert den Namen seiner zugehörigen Datei. Solange ein Client nur die Methode `getName()` aufruft, ist es nicht nötig, die Datei zu öffnen und den Inhalt zu laden. Erst wenn ein Client über `getInhalt()` den Inhalt der Datei lesen will, muss die relativ teure Operation über den Zugriff auf das echte Objekt ausgeführt werden. Dazu wird in diesem Fall das Originalobjekt erzeugt, das neben dem Namen auch den Inhalt kennt. Die Proxy-Klasse `ProxyDateiZugriff` sieht dann wie folgt aus:

```
// Datei: ProxyDateiZugriff.java
public class ProxyDateiZugriff implements IDateiZugriff
{
    String name;
    IDateiZugriff realeDatei;
```

```

public ProxyDateiZugriff(String name)
{
    this.name = name;
    System.out.println("Stellvertretendes Objekt instanziiert.");
}

public String getInhalt()
{
    if(realeDatei == null)
    {
        System.out.println("Inhalt liegt lokal nicht vor.");
        // Das echte Datei-Objekt wird erzeugt.
        realeDatei = new DateiZugriff(name);
    }
    return realeDatei.getInhalt();
}

public String getName()
{
    return name;
}
}

```

Der TestProxy erstellt ein Objekt der Klasse ProxyDateiZugriff, die das Interface IDateiZugriff implementiert, und ruft dann die Methoden getName() und getInhalt() des Interface beim Proxy auf:

```

// Datei: TestProxy.java
public class TestProxy
{
    public static void main (String[] args)
    {
        // Das Proxy-Objekt wird erzeugt.
        ProxyDateiZugriff pDatei =
            new ProxyDateiZugriff ("TestDatei.dat");
        System.out.println();
        System.out.println("Name: " + pDatei.getName());
        System.out.println("Inhalt: " + pDatei.getInhalt());
        System.out.println();
        System.out.println("Inhalt: " + pDatei.getInhalt());
    }
}

```



Die Ausgabe des Programms ist:

Stellvertretendes Objekt instanziiert.

Name: TestDatei.dat

Inhalt liegt lokal nicht vor.

Echtes Objekt instanziiert.

Inhalt: Daten von TestDatei.dat

Inhalt: Daten von TestDatei.dat

Beim Proxy in diesem Beispiel handelt es sich um eine Art Virtueller Proxy (siehe Kapitel 4.8.5). Solange ein Client nur den Namen wissen will, werden somit "teure" Dateioperationen vermieden. Erst wenn jemand auf den Inhalt der Datei zugreifen will, wird das eigentliche Dateiojekt erzeugt.

## 4.8.4 Bewertung

### 4.8.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Bereits bestehende Anwendungen können mit dem Proxy-Muster erweitert werden (siehe Kapitel 4.8.5).
- Der Funktionsumfang eines Proxys kann individuell gestaltet werden.
- Das Proxy-Muster verdeckt die Architektur des zugeordneten Systems, so dass der Client nur das Interface des echten Objekts kennen muss (Transparenz der Implementierung).
- Es wird ein Stellvertreter (Proxy) des echten Objektes erzeugt. Erst wenn bestimmte, im Wesentlichen von der Art der Proxy-Variante abhängige Operationen durchgeführt werden sollen, braucht man das echte Objekt.
- Durch die zusätzliche Zwischenschicht eines Proxy kann bei Operationen, die kein Original benötigen, z. B. beim Caching der Anfragen, ein Performance-Gewinn entstehen.

### 4.8.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Die Fehlersuche wird beim Zwischenschalten eines Proxys erschwert.
- Durch die zusätzliche Zwischenschicht eines Proxy entsteht beim Zugriff auf das Originalobjekt ein Performance-Verlust, da eine Weiterleitung (Delegation) erfolgt und dies mit Aufwand verbunden ist.

## 4.8.5 Einsatzgebiete

In folgenden Situationen kann das Proxy-Muster eingesetzt werden:

- Wenn der Ort des echten Objekts verborgen werden soll. Der entsprechende Proxy wird als Remote-Proxy bezeichnet und wird in diesem Kapitel im Detail erläutert. Ein Remote-Proxy ist beispielsweise ein zentraler Bestandteil des **Broker-Musters** (siehe Kapitel 5.4).
- Wenn das echte Objekt nicht verändert werden kann, aber eine Methode durch einen Proxy statisch erweitert werden soll. Der Proxy übernimmt dann die Funktionserweiterung und leitet anschließend den Aufruf an die eigentliche Methode des echten Objekts weiter. Die zusätzliche Funktionalität kann sich unter anderem auf die Verteilung von Ladezeiten, Statistiken, Filterung, Pufferung, Synchronisierung und Einführung einer Rechteverwaltung beziehen.

Daher gibt es das Proxy-Muster in mehreren Ausführungen, wobei jede Ausführung zwar eine ähnliche Vorgehensweise hat, aber stets andersartige Probleme behandelt. Man kann in einem Proxy Funktionen implementieren, die über die Funktionen des echten Objekts hinausgehen. Wesentlich sind die folgenden Varianten:

- **Virtueller Proxy**

Der virtuelle Proxy bietet eine Möglichkeit, Ladezeiten von Programmen auf einen größeren Zeitraum zu verteilen. Anwendungen sind beim Start häufig rechenintensiv, da alle Objekte der Anwendung instanziiert und initialisiert werden. Beim Virtuellen Proxy werden an Stelle der rechenintensiven echten Objekte Stellvertreter-Objekte instanziiert, die stellvertretend für die eigentlichen Objekte verwendet werden. Der virtuelle Proxy schiebt meist die Erzeugung des echten Objekts so lange hinaus, bis dieses gebraucht wird. Beim Aufruf einer Methode, für die das echte Objekt benötigt wird, wird das eigentliche Objekt zur Laufzeit instanziiert. Bei einem interaktiven System könnte der Proxy zum Beispiel eine "Bitte Warten"-Meldung ausgeben, um die rechenintensive Erzeugung des echten Objekts zu überbrücken. Wenn das echte Objekt dann erzeugt ist, kann der Proxy den Aufruf an dieses weiterleiten und den Wartezustand beenden.

- **Schutz-Proxy**

Mit dem Schutz-Proxy können Zugriffe auf die Objekte einer Anwendung überwacht werden. Der Schutz-Proxy überprüft beim Methodenaufruf, ob das aufrufende Client-Programm tatsächlich über die notwendige Zugriffsberechtigung verfügt. Auf diese Weise kann eine Rechteverwaltung nachträglich in ein bestehendes System eingeführt werden.

- **Synchronisierungs-Proxy**

Bei Anwendungsfällen, in denen mehrere Client-Programme auf ein gemeinsames Objekt zugreifen, können Methodenaufrufe über ein zwischengeschaltetes Stellvertreter-Objekt synchronisiert werden. Der Proxy nimmt alle Methodenaufrufe an das gemeinsame Objekt entgegen. Die Methodenaufrufe, die synchronisiert werden müssen, ordnet er in einer Queue an und arbeitet sie nacheinander ab. Andere Methodenaufrufe können direkt weitergeleitet werden.

- **Remote-Proxy**

Ein Remote-Proxy befindet sich lokal auf dem gleichen Rechner wie das Client-Programm und steht diesem als Stellvertreter der eigentlichen Komponente zur Verfügung, die auf einem entfernten (remote) Rechner läuft. Das Client-Programm kennt nur das Stellvertreter-Objekt und nicht die eigentliche Komponente (Transparenz). Das Client-Programm weiß nicht, wie Methodenaufrufe an das Stellvertreter-Objekt in der Folge weiterverarbeitet werden. Mit diesem Proxy können Anwendungen gekapselt, ausgetauscht und auf weitere Rechner ausgelagert werden, ohne, dass das Client-Programm verändert werden muss.

- **Firewall-Proxy**

Der Firewall-Proxy stellt eine zentrale Anlaufstelle auf einem Rechner für die Kommunikation zu mehreren Sub-Netzwerken dar. Ein klassisches Beispiel sind Unternehmensnetzwerke, bei denen die Internetanbindung häufig über Firewall-Proxys realisiert wird, um den Datenverkehr zu prüfen und ungewünschten Verkehr zu filtern.

- **Counting-Proxy**

Der Counting-Proxy ist ein Objekt, das eingeführt wird, um die Aktivitäten zwischen dem Client-Programm und den anderen Objekten einer Anwendung zu protokollieren. Diese Proxy-Variante findet vor allem bei der Erstellung von Statistiken Verwendung.

- **Cache-Proxy**

Der Cache-Proxy puffert Daten zwischen, so dass die Daten nicht bei jedem Aufruf von einem entfernten Rechner geladen werden müssen. Diese Proxy-Variante wird bei Browsern eingesetzt, um Webseiten zwischenspeichern. Ein Cache-Proxy hat oft viele Clients und kann deshalb vorausschauende Algorithmen effizienter als seine Clients nutzen, da er besseres Material für die Statistik und damit für die Anforderungsrate der zu cachenden Daten hat.

Das Proxy-Muster kann auch beim Testen<sup>50</sup> von Software eingesetzt werden. Die Stellvertreter-Objekte werden dann als **Mock**<sup>51</sup>-**Objekte** bezeichnet. Sie dienen dazu, eine noch nicht oder nicht vollständig vorhandene Umgebung für eine zu testende Komponente zu simulieren. Die Stellvertreter-Objekte verhalten sich, wie es von den eigentlichen Umgebungs-Objekten später im Gesamtsystem erwartet wird, liefern aber beispielsweise nur hart codierte, aber sinnvolle Ergebnisse zurück. Die Anwendung des Proxy-Musters geschieht hier außerhalb der Entwurfsphase.

## 4.8.6 Ähnliche Entwurfsmuster

Mit dem Proxy-, dem Dekorierer- und dem Adapter-Muster können Objekte um eine zusätzliche Funktionalität erweitert werden. Jedes dieser Muster führt ein neues Objekt ein<sup>52</sup>, das zwischen der Anwendung und dem eigentlichen Objekt steht und das eine Referenz auf das eigentliche Objekt enthält. Über diese Referenz kann das neue Objekt Anfragen und Befehle an das eigentliche Objekt delegieren. Vor oder nach bzw. vor und nach einer Delegation kann eine Zusatzfunktionalität eingefügt werden.

Der Proxy verwendet dieselbe Schnittstelle wie das eigentliche Objekt. Für den **Adapter** gilt diese Einschränkung nicht, dass die Schnittstelle gleich bleiben muss. Ganz im Gegenteil! Ein Adapter passt die vorhandene Schnittstelle des eigentlichen Objekts an die von der Anwendung gewünschte Schnittstelle an.

Die Aufgabe eines **Dekorierers** ist das dynamische Hinzufügen von neuer Funktionalität zu einem Objekt. Diese Aufgabe kann zwar prinzipiell auch ein Proxy erledigen. Aber bei einem Proxy steht eher eine andere Aufgabe – eine Art "Türsteher"-Funktion – im Vordergrund: ein Proxy ermöglicht oder verhindert den Zugriff auf die Funktionalität des eigentlichen Objekts. Beim Proxy-Muster geht man in der Regel davon aus, dass das eigentliche Objekt schon die richtige Funktionalität besitzt, beim Dekorierer-Muster soll die Funktionalität des eigentlichen Objekts dynamisch durch ein Dekorierer-Objekt erweitert werden.

---

<sup>50</sup> Für Muster, die in der Testphase angewendet werden können, hat sich der Begriff Test Pattern etabliert, zu denen beispielsweise das Mock Object Pattern [Mes07] gezählt wird.

<sup>51</sup> to mock (engl.) bedeutet nachahmen.

<sup>52</sup> Beim Adapter-Muster wird hier die objektbasierte Variante betrachtet.



## 4.9 Das Verhaltensmuster Schablonenmethode

### 4.9.1 Name/Alternative Namen

Schablonenmethode (engl. template method).

### 4.9.2 Problem

Die Struktur eines Algorithmus soll in der Basisklasse festgelegt werden, Einzelheiten in einer Unterklasse.

Das Muster **Schablonenmethode** soll bereits in einer Basisklasse die Struktur eines Algorithmus festlegen. Realisiert werden sollen variante Teile des Algorithmus in Unterklassen.



### 4.9.3 Lösung

Das Muster Schablonenmethode wird für Algorithmen eingesetzt, die in Einzeloperationen zerlegt werden können. Die Struktur eines solchen Algorithmus aus abstrakten Einzeloperationen und eventuell konkreten Einzeloperationen wird bereits in der Basisklasse festgelegt und in einer **Schablonenmethode** vorgegeben. Dabei werden bestimmte Einzeloperationen als **Einschubmethoden** (engl. **hooks**) in abstrakter Form in der Schablonenmethode eingeführt<sup>53</sup>. Das heißt, dass für diese abstrakten Einzeloperationen in der Basisklasse nur deren Schnittstellen und Verträge vorgegeben werden, die Implementierung wird an die Unterklassen delegiert.

Die Implementierung der Einschubmethoden in einer Unterklasse muss der Deklaration der entsprechenden Einschubmethode in der Basisklasse genügen. Es ist möglich, Einschubmethoden in verschiedenen abgeleiteten Klassen verschieden festzulegen und so verschiedene Varianten des Algorithmus zu erzeugen. Da das Muster Schablonenmethode auf statischer Vererbung basiert, ist es ein klassenbasiertes Entwurfsmuster.

Eine Schablonenmethode legt das Gerüst eines Algorithmus fest und benutzt dabei abstrakte Einschubmethoden. Einschubmethoden werden in Unterklassen implementiert, wobei jede Unterklasse die Einschubmethoden unterschiedlich implementieren kann. So kann jede Unterklasse eine andere Variante des Algorithmus realisieren und die Basisklasse wird von den Unterklassen unabhängig.



<sup>53</sup> In der Regel ist eine Einschubmethode abstrakt. Es gibt aber auch Varianten des Musters, bei denen Einschubmethoden nicht abstrakt sind, sondern mit einem leeren Rumpf oder mit einem Rumpf, der eine Standardimplementierung enthält, definiert werden. Diese Einschubmethoden werden dann in den Unterklassen überschrieben.

Da die Schablonenmethode den Vertrag der Einschubmethoden vorgibt und eine Unterklasse den Vertrag nach dem liskovschen Substitutionsprinzip einhalten muss, ist die Schablonenmethode nicht von der speziellen Implementierung einer Unterklasse abhängig. Man spricht daher auch von **Dependency Inversion**.

Einschubmethoden sind in der Regel speziell für den jeweiligen Algorithmus der Schablonenmethode geschaffen. Sie werden daher nicht von außerhalb der Schablonenmethode, sondern nur von der Schablonenmethode selbst, aufgerufen. Bei den **Einschubmethoden** handelt es sich also um **nicht öffentlich** zugängliche Servicemethoden.



Damit Einschubmethoden nicht von außerhalb benutzt werden können, können sie beispielsweise in Java als `protected` deklariert werden.

Eine Schablonenmethode enthält das Grundgerüst eines Algorithmus mit der Zerlegung des Algorithmus in Einschubmethoden und deren Aufruffreihenfolge. Diese Definition wird als invariant angesehen und soll daher von Subklassen nicht überschrieben werden. Um dies zu verdeutlichen, kann eine Schablonenmethode in Java als `final` deklariert werden bzw. in UML mit der Einschränkung `{leaf}` versehen werden. Soll ein Algorithmus komplett ausgetauscht werden, ist das Muster Schablonenmethode nicht anwendbar. Für den Zweck des Austauschs eines kompletten Algorithmus kann das **Strategie-Muster** eingesetzt werden.

Im folgenden Klassendiagramm und im anschließenden Programmbeispiel werden die Möglichkeiten von UML bzw. Java genutzt. In Java wird die Schablonenmethode als `final` und werden die Einschubmethoden als `protected` deklariert, sodass sie nur von Unterklassen überschrieben werden können.

#### 4.9.3.1 Klassendiagramm

Das Entwurfsmuster Schablonenmethode versucht, soviel Code wie möglich in einer abstrakten Basisklasse zu spezifizieren, und legt die Verträge der Einschubmethoden fest. Die Implementierung einer abstrakten Einschubmethode ist in der Vaterklasse noch nicht bekannt, nur ihr Vertrag. Die Implementierung der abstrakten Methoden der Basisklasse wird bewusst an die noch nicht existierenden Unterklassen delegiert.

Die Schablonenmethode der Klasse `AbstrakteKlasse` ist vollständig implementiert und benutzt in folgendem Beispiel für ihren Algorithmus die Einschubmethoden `einschubmethode1()` und `einschubmethode2()`. Das Besondere an diesen Einschubmethoden ist, dass sie abstrakt sind – die Einschubmethoden werden erst in einer spezialisierenden Unterklasse implementiert. Das folgende Klassendiagramm zeigt dieses Beispiel:

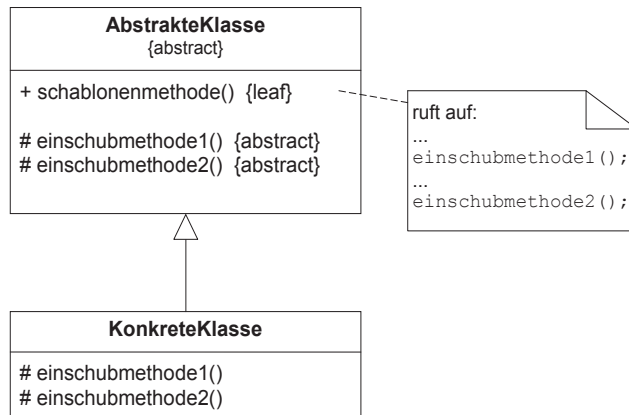


Bild 4-27 Beispiel für ein Klassendiagramm der Schablonenmethode

Die Einschubmethoden wurden in Bild 4-27 zusätzlich mit # als `protected` gekennzeichnet, damit sie nicht von außerhalb der Klassenhierarchie aufgerufen werden können. Auf diese Weise soll verdeutlicht werden, dass es sich um Servicemethoden für die Schablonenmethode handelt.

#### 4.9.3.2 Teilnehmer

##### AbstrakteKlasse

Die Klasse `AbstrakteKlasse` definiert das Skelett einer Schablonenmethode unter Verwendung abstrakter Einschubmethoden, um die einzelnen Schritte eines Algorithmus festzulegen.

##### KonkreteKlasse

Die Klasse `KonkreteKlasse` definiert die unterklassenspezifischen Schritte des Algorithmus in den entsprechenden konkreten Einschubmethoden.

#### 4.9.3.3 Dynamisches Verhalten

Das folgende Sequenzdiagramm zeigt, wie die Schablonenmethode einer konkreten Klasse von einer Client-Anwendung aufgerufen wird und wie die konkrete Klasse nun ihrerseits die Einschubmethoden verwendet, die in der Klasse `KonkreteKlasse` implementiert sind:

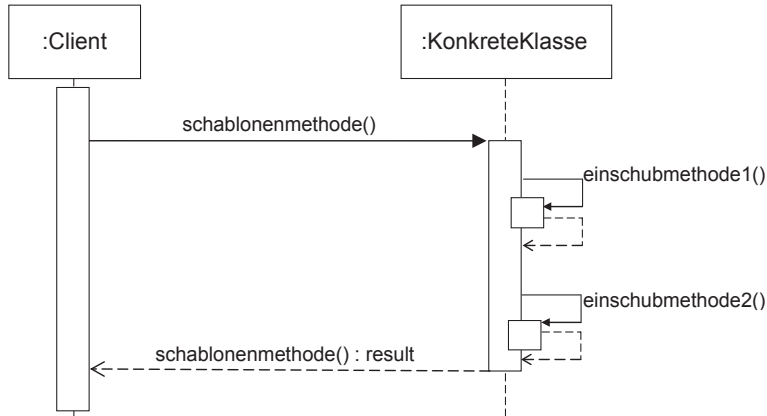


Bild 4-28 Sequenzdiagramm des Schablonenmethode-Musters

#### 4.9.3.4 Programmbeispiel

Im folgenden Beispiel enthält die Klasse `Urlaubskarte` als Schablonenmethode die Methode `karteSchreiben()`. Die Methode `karteSchreiben()` gibt einen Standardtext (Methode `textSchreiben()`) aus sowie einen Zusatztext, der je nach Verwendungszweck einer Karte unterschiedlich ausfallen kann. Die Ausgabe des Zusatztextes wird daher auf Subklassen verlagert. Zu diesem Zweck wird die Methode `zusatzSchreiben()` als abstrakte Einschubmethode in der Unterklasse `Urlaubskarte` definiert und in der Schablonenmethode benutzt. Die Klasse `Urlaubskarte` hat folgendes Aussehen:

```
// Datei: Urlaubskarte.java
abstract public class Urlaubskarte // abstrakte Klasse
{
    final public void karteSchreiben() // Schablonenmethode
    {
        textSchreiben();
        zusatzSchreiben(); // abstrakte Methode wird aufgerufen
    }

    private void textSchreiben() // normalen Text ausgeben
    {
        System.out.println (
            "Ich bin gut an meinem Urlaubsziel angekommen.");
        System.out.println (
            "Das Essen schmeckt gut und die Gegend gefaellt mir.");
    }

    abstract protected void zusatzSchreiben(); // Einschubmethode
}
```

Die Einschubmethode `zusatzSchreiben()` wird exemplarisch in den zwei Klassen `UrlaubskarteAnFreunde` und `UrlaubskarteAnFirma` implementiert: es wird ein Zusatztext für Urlaubskarten, die an Freunde geschickt werden, definiert, sowie ein

Zusatztext für Urlaubskarten, die an die Firma adressiert werden. Es folgt der Quelltext der beiden Klassen `UrlaubskarteAnFreunde` und `UrlaubskarteAnFirma`:

```
// Datei: UrlaubskarteAnFreunde.java
public class UrlaubskarteAnFreunde extends Urlaubskarte
{
    // Eine abgeleitete Klasse, in der
    public void zusatzSchreiben() // die abstrakte Methode
                                // überschrieben wird.
    {
        System.out.println ("Ich treibe viel Sport.");
    }
}

// Datei: UrlaubskarteAnFirma.java
public class UrlaubskarteAnFirma extends Urlaubskarte
{
    // Eine andere abgeleitete Klasse,
    public void zusatzSchreiben() // in der die abstrakte Methode
                                // überschrieben wird.
    {
        System.out.println ("Ich freue mich wieder auf die Arbeit.");
    }
}
```

Die Klasse `Test` zeigt, wie die Schablonenmethode `karteSchreiben()` in verschiedenen Unterklassen aufgerufen wird, dabei aber jeweils eine andere überschreibende Einschubmethode benutzt wird:

```
// Datei: Test.java // Testklasse
public class Test
{
    public static void main (String args[])
    {
        System.out.println ("Karte an die Freunde:");
        UrlaubskarteAnFreunde karteFreunde =
            new UrlaubskarteAnFreunde();
        karteFreunde.karteSchreiben(); // dabei erfolgt der Aufruf einer
                                      // überschreibenden Methode

        System.out.println();
        System.out.println ("Karte an die Firma:");
        UrlaubskarteAnFirma karteFirma = new UrlaubskarteAnFirma();
        karteFirma.karteSchreiben(); // dabei erfolgt der Aufruf einer
                                    // überschreibenden Methode
    }
}
```



Hier das Protokoll des Programmlaufs:

Karte an die Freunde:  
 Ich bin gut an meinem Urlaubsziel angekommen.  
 Das Essen schmeckt gut und die Gegend gefällt mir.  
 Ich treibe viel Sport.

Karte an die Firma:  
 Ich bin gut an meinem Urlaubsziel angekommen.  
 Das Essen schmeckt gut und die Gegend gefällt mir.  
 Ich freue mich wieder auf die Arbeit.

### 4.9.4 Einsatzgebiete

Die Schablonenmethode setzt voraus, dass mehrere Arbeitsschritte in mehreren Klassen gleichzeitig sind.

Das Muster der Schablonenmethode ist in allen objektorientierten Frameworks vorzufinden, weil damit ein hohes Maß an Wiederverwendbarkeit erreicht werden kann.



Ein Framework gibt die abstrakte Klasse vor und implementiert darin die grundlegenden Algorithmen. Der Nutzer des Frameworks muss die vom Framework geforderten Einschubmethoden implementieren. Durch den Einsatz der Vererbung kann die bereits im Framework implementierte Funktionalität wiederverwendet werden.

### Anwendungsbeispiel

Im Folgenden soll ein konkretes Beispiel betrachtet werden (siehe Bild 4-29). Es sollen verschiedene Klassen zur Sammlung von Objekten implementiert werden. Unabhängig von der verwendeten Datenstruktur (Liste, Baum etc.) soll festgestellt werden können, ob sich ein bestimmtes Objekt in der Sammlung befindet. Der dazu benötigte Algorithmus kann als Schablonenmethode `contains()` in einer gemeinsamen Basisklasse definiert werden. Jetzt das bereits erwähnte Bild:

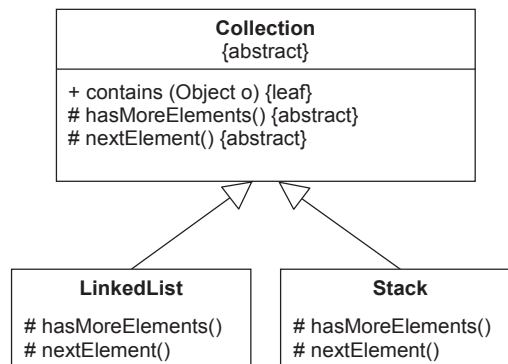


Bild 4-29 Container-Klassen zur Veranschaulichung der Schablonenmethode

Die Methode `contains()` der Klasse `Collection` ist vollständig implementiert und benutzt die beiden abstrakten Methoden `hasMoreElements()` und `nextElement()` derselben Klasse. Der folgende Programmausschnitt zeigt eine mögliche Implementierung der Methode `contains()`:

```

abstract class Collection
{
    // Schablonenmethode
    final boolean contains (Object o)
    {

```

```
while (hasMoreElements())
{
    if (o.equals (nextElement())
        return true;
    }
    return false;
}
// Einschubmethoden
abstract protected boolean hasMoreElements();
abstract protected Object nextElement();
}
```

Die abstrakten Methoden `hasMoreElements()` und `nextElement()` werden hier verwendet, obwohl noch keine konkreten Implementierungen vorhanden sind. Die Implementierung wird an die Subklassen delegiert. Die Subklassen können dabei den Algorithmus der Methode `contains()` mitbenutzen. Sie müssen den Algorithmus nicht nochmals implementieren. Die von der abstrakten Basisklasse abgeleiteten Klassen implementieren die Einschubmethoden in Abhängigkeit von der jeweiligen Datenstruktur der speziellen Ausprägung der Collection.

Einschubmethoden werden beispielsweise dann gebraucht, wenn in der Schablonenmethode Objekte erzeugt werden müssen, deren Typ aber noch nicht festgelegt werden kann. Die Objekterzeugung wird dann in die Implementierung der Einschubmethoden verlagert – analog zum Muster Fabrikmethode, bei dem die Objekterzeugung in die Implementierung der Fabrikmethode in einer Unterklasse verlagert wird.

## 4.9.5 Bewertung

### 4.9.5.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Durch das Muster Schablonenmethode wird eine Basisklasse nicht von ihren Unterklassen abhängig, obwohl die Basisklasse Einschubmethoden aufruft, die in den Unterklassen implementiert sind. Der Mechanismus des Musters Schablonenmethode führt zu einer Invertierung der Abhängigkeiten (**Dependency Inversion**).
- Vorteilhaft ist, dass der Algorithmus bereits in der Basisklasse auf grobem Niveau festgeschrieben werden kann, ohne die Details der Implementierung zu kennen. Es besteht somit die Möglichkeit zur Abstraktion von Verhalten.
- Die Schablonenmethode gewährleistet ein hohes Maß an Wiederverwendung der Spezifikation. Die Subklassen benutzen dabei den Algorithmus der Schablonenmethode mit und implementieren ihn nicht nochmals, d. h. sie überschreiben ihn nicht.

### 4.9.5.2 Nachteile

Aus diesem Verhaltensmuster resultieren – wenn sein Einsatz Sinn macht – keine Nachteile.

#### 4.9.6 Ähnliche Entwurfsmuster

Das Entwurfsmuster **Strategie** tauscht einen ganzen Algorithmus statt nur einzelner Teile aus. Es verwendet die Aggregation anstelle der Vererbung, ist also objektbasiert und nicht klassenbasiert wie das Entwurfsmuster Schablonenmethode. Ein Objekt einer Kontextklasse benutzt im Falle des Strategie-Musters den Algorithmus, ohne ihn überhaupt – also auch nicht schablonenhaft – selbst zu implementieren. Durch die Aggregation wird die Implementierung nicht zur Kompilierzeit, sondern zur Laufzeit austauschbar.



## 4.10 Das Verhaltensmuster Befehl

### 4.10.1 Name/Alternative Namen

Befehl, Kommando (engl. command), Aktion (engl. action).

### 4.10.2 Problem

Das Befehlsmuster soll einen Befehl, eine Aktion, eine Anfrage bzw. einen Methodenaufruf – im Folgenden kurz Befehl genannt – in einem Objekt kapseln.

Durch das Befehlsmuster soll die Erteilung und die Auslösung eines Befehls zeitlich entkoppelt werden. Das bedeutet, dass das Erzeugen eines Befehls und seine Ausführung zu verschiedenen Zeiten stattfinden können.

Der **Aufrufer** soll nicht von der Ausprägung eines konkreten Befehls abhängen. Er soll nur die Abstraktion eines Befehls kennen und somit sollen Details eines konkreten Befehls vor dem Aufrufer verborgen bleiben. Damit sollen verschiedene Implementierungen eines Befehls möglich sein, z. B. "rechts", "links", "oben", "unten". Außerdem soll der Aufrufer den Empfänger des Befehls nicht kennen.

Das **Befehlsmuster** soll die Details eines Befehls vor dem Aufrufer des Befehls verbergen. Es soll es erlauben, dass die Erzeugungszeit und die Ausführungszeit des Befehls getrennt werden können.



### 4.10.3 Lösung

Das Verhaltensmuster Befehl ist ein objektbasiertes Entwurfsmuster. Ein konkreter Befehl entspricht einem Methodenaufruf oder einer Folge von Methodenaufrufen auf einem Empfängerobjekt. Ein **konkreter Befehl** selbst wird zusammen mit der Referenz auf das Empfängerobjekt in einem auf dem Interface `IBefehl` basierenden Objekt **gekapselt** und wird einem **Aufrufer übergeben**. Der Aufrufer hängt so nicht von den Details des Befehls ab, sondern nur vom Interface `IBefehl`. Gibt der Aufrufer das Interface selbst vor, ist er natürlich nicht von ihm abhängig. Das Muster macht aber keine Aussage darüber, wer das Interface vorgibt.

Ein **Client** bzw. eine Anwendung erstellt konkrete Befehle und weist ihnen einen Empfänger zu.



Die Klasse `KonkreterBefehl` implementiert das Interface `IBefehl`. Der entsprechende Empfänger und die auszuführenden Methoden müssen dem konkreten Befehl jeweils bekannt sein.

Die Referenz auf den Empfänger und die Methoden sind im Objekt der Klasse `KonkreterBefehl` gespeichert.



Der **Aufrufer** erhält vom Client den Befehl und hat die Aufgabe, die Ausführung eines Befehls zu initiieren.



Auslöser zur Ausführung eines Befehls kann beispielsweise das Drücken eines Buttons (einer Schaltfläche) einer grafischen Oberfläche sein. Der Button übernimmt in diesem Falle die Rolle eines Aufrufers.

Der Aufrufer **entkoppelt** die Erzeugung eines Befehls von seiner Ausführung. Ein Client kann einen Befehl erzeugen und ihn an einen Aufrufer übergeben, der gleiche Client oder aber auch ein anderer kann zu einem späteren Zeitpunkt die Ausführung des Befehls veranlassen.

In der im Folgenden beschriebenen Grundform des Musters ist es die Aufgabe des Aufrufers, einen Befehl vom Client entgegenzunehmen, zu speichern und auf Anforderung auszuführen. Die Anforderung kann beispielsweise – wie bereits erwähnt – durch das Drücken einer Schaltfläche durch den Client erfolgen. Je nach Ausprägung des Befehlsmodells kann der Aufrufer sehr komplex sein (siehe Kapitel 4.10.5).

Der Aufrufer stellt in der Grundform des Musters eine Methode `speichereBefehl()` bereit, mit deren Hilfe ein Client einen Befehl übergibt. Außerdem besitzt der Aufrufer eine Methode `aktivieren()`. In dieser Methode `aktivieren()` wird die Methode `fuehreAus()` des Befehlsobjekts aufgerufen. Die Methode `fuehreAus()` sorgt dafür, dass die Methode `aktion()`, d. h. die Ausführung des Befehls, auf dem Empfängerobjekt aufgerufen wird.



Die Methode `aktion()` symbolisiert eine Methode des Empfängerobjekts zur Ausführung des Befehls. Wird der Befehl ausgeführt, so ruft das entsprechende Befehlsobjekt die Methode `aktion()` des Empfängerobjekts auf.

Auf Grund des liskovschen Substitutionsprinzips ist bei Einhaltung der Verträge ein Aufrufer in der Lage, auch Befehlsobjekte von anderen konkreten Befehlsklassen, die das Interface `IBefehl` implementieren, entgegenzunehmen, zu speichern und auszuführen. Ebenso können dann die Referenzen auf solche Befehlsobjekte in einer Liste aufgereiht werden, um beispielsweise die Historie ihrer Ausführung zu speichern.

#### 4.10.3.1 Klassendiagramm

Ein konkreter Befehl implementiert das Interface `IBefehl` und definiert dadurch die Methode `fuehreAus()`.

Ein Client erzeugt einen konkreten Befehl und speichert im erzeugten Befehlsobjekt eine Referenz auf den konkreten Empfänger. Der Client übergibt einem Aufrufer mit Hilfe der Methode `speichereBefehl()` eine Referenz auf den erzeugten konkreten Befehl. Ein Client muss den Empfänger also kennen, aber nicht der Empfänger den Client. Dies ist durch die Einschränkung der Navigationsrichtung der Assoziation zwischen Client und Empfänger im Klassendiagramm ausgedrückt:

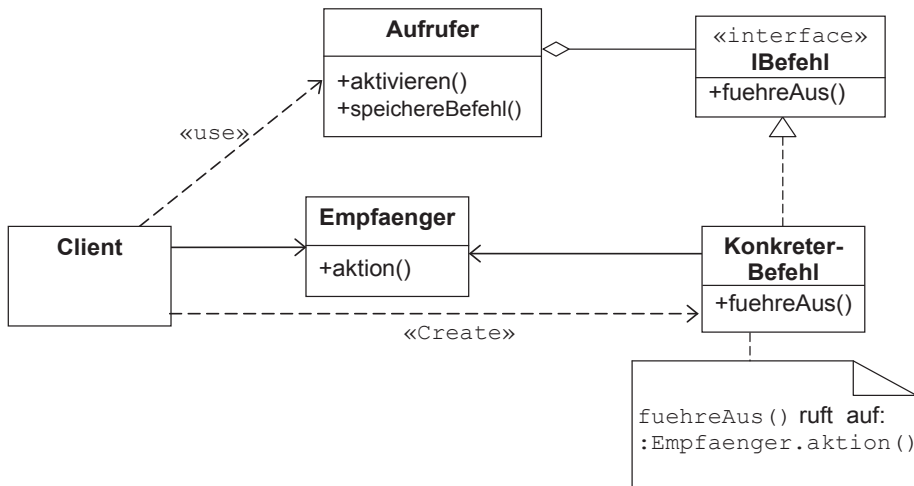


Bild 4-30 Klassendiagramm des Befehlsmusters

Ein Client kann beispielsweise die Ausführung eines von einem Aufrufer gespeicherten Befehls auslösen, indem er die Methode `aktivieren()` des Aufrufers aufruft. Der Aufrufer wird dadurch veranlasst, die Methode `fuehreAus()` des gespeicherten Befehls aufzurufen. In seiner Methode `fuehreAus()` ruft der konkrete Befehl die Methode `aktion()` des Empfängers auf, d. h. die Ausführung wird weiterdelegiert.

Die Methode `aktivieren()` wird hier beispielhaft benutzt. Das Muster lässt offen, wer für die Auslösung der Ausführung eines Befehls zuständig ist. Beispielsweise könnte ein Aufrufer zeitgesteuert arbeiten. Ein Client könnte die gewünschte Ausführungszeit bei der Übergabe eines Befehls an den Aufrufer spezifizieren. Der Aufrufer trägt dann dafür Sorge, dass der Befehl zu der gewünschten Zeit ausgeführt wird. Diese und weitere Varianten des Musters werden in Kapitel 4.10.5 erläutert.

Die Methode `aktion()` steht hier stellvertretend für eine Methode, die mit Hilfe eines Befehls ausgeführt werden soll. Eine solche Methode kann natürlich auch parametrisiert sein. Die Argumente für die Parameter können entweder durch einen entsprechenden Konstruktor oder durch set-Methoden, die die konkrete Befehlsklasse zur Verfügung stellen muss, an das Befehlsobjekt übergeben werden. Das Befehlsobjekt

kann dann diese Argumente beim Aufruf der Methode `aktion()` an das Empfängerobjekt weitergeben.

#### 4.10.3.2 Teilnehmer

##### **IBefehl**

Das Interface `IBefehl` definiert die Operation `fuehreAus()` für das Ausführen von Befehlen.

##### **KonkreterBefehl**

Die Klasse `KonkreterBefehl` implementiert das Interface `IBefehl`. Die Klasse `KonkreterBefehl` steht hier stellvertretend für alle möglichen Varianten von Befehlsklassen. Diese Klassen kapseln einen Befehl und speichern die zum Ausführen des Befehls nötigen Informationen wie etwa einen Parameter. Zu diesen Informationen gehört auch eine Referenz auf das Objekt der Klasse `Empfaenger`. Der Rumpf von `fuehreAus()` enthält den Aufruf der Methode `aktion()` des gespeicherten Empfängers.

##### **Client**

Der Client steht stellvertretend für die Anwendung. Er erzeugt ein Objekt der Klasse `KonkreterBefehl`, welches die auszuführende Aktion (Befehl) darstellt und eine Referenz auf ein Objekt der Klasse `Empfaenger` beinhaltet. Außerdem übergibt der Client einen konkreten Befehl an den Aufrufer mit Hilfe der Methode `speichereBefehl()`. Ein Client kann beispielsweise die Ausführung des Befehls auslösen, indem er die Methode `aktivieren()` des Aufrufers aufruft.

##### **Aufrufer**

Ein Aufrufer in der hier beschriebenen Grundform des Musters speichert eine Referenz auf einen Befehl. Der Typ dieser Referenz ist `IBefehl`. Nach dem liskovschen Substitutionsprinzip und bei Einhaltung der Verträge kann diese Referenz zur Laufzeit auf jedes Befehlsobjekt zeigen, dessen Klasse das Interface `IBefehl` implementiert. Soll die Ausführung des gespeicherten Befehls ausgelöst werden, ruft der Aufrufer die Methode `fuehreAus()` des gespeicherten Befehls auf.

##### **Empfaenger**

Die Klasse `Empfaenger` kennt als einzige die Details über die mit der Ausführung eines Befehls verknüpften Operationen. Die Klasse `Empfaenger` kann von Anwendung zu Anwendung des Befehlsmodells unterschiedlich sein. Hier wird beispielhaft eine Methode `aktion()` der Klasse `Empfaenger` benutzt, deren Ausführung durch ein Befehlsobjekt gekapselt wird. Wird also ein Befehl ausgeführt, wird die Methode `aktion()` des im Befehlsobjekt gespeicherten Empfängerobjekts ausgeführt.

#### 4.10.3.3 Dynamisches Verhalten

Nachdem der Client ein Objekt der Klasse `KonkreterBefehl` erzeugt und dem Aufrufer übergeben hat, erfolgt in diesem Beispiel der Aufruf der Methode `aktivieren()` des Aufruferobjekts. Damit wird die Ausführung des Befehls initiiert. Der Aufrufer ruft

die Methode `fuehreAus()` des gespeicherten Befehlsobjekts auf und dieses führt nun den Befehl aus, in dem es die Methode `aktion()` des Empfängerobjekts aufruft.

Das Sequenzdiagramm in Bild 4-31 zeigt den grundsätzlichen Ablauf des Befehlsmusters:

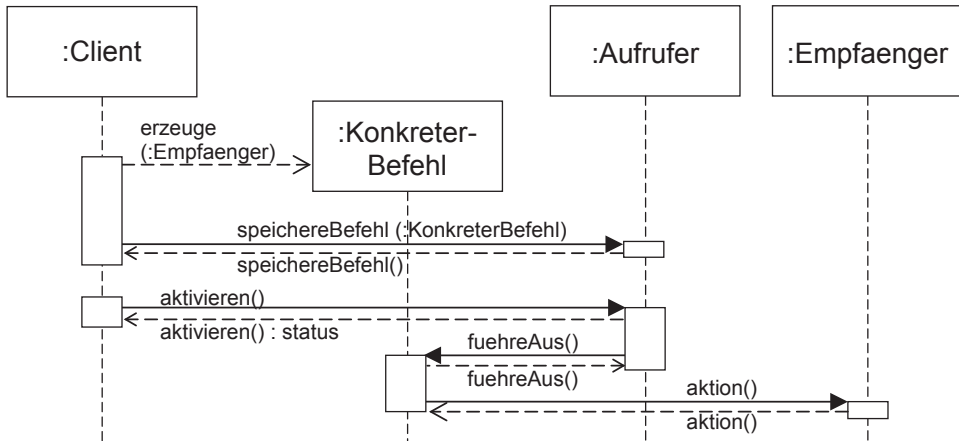


Bild 4-31 Sequenzdiagramm des Befehlsmusters

Wie bereits erwähnt wurde, legt das Entwurfsmuster nicht fest, wann und wie der Befehl ausgeführt wird. Beispielsweise könnte das Objekt der Klasse `Aufrufer` die Ausführung des konkreten Befehls zeitgesteuert veranlassen. Der Aufruf der Methode `aktivieren()` würde dann entfallen, da der Aufrufer selbst den Aufruf der Methode `fuehreAus()` auslöst. Es ist auch denkbar, dass der Aufrufer selbst ereignisgesteuert, oder dass ein anderer Client als der, der den Befehl erzeugt und einspeichert, die Methode `aktivieren()` aufruft und damit die Ausführung des gespeicherten Befehls auslöst.

#### 4.10.3.4 Programmbeispiel

Das Befehlsmuster wird im folgenden Beispiel verwendet, um Befehle für eine Lichtquelle zu kapseln. `Lichtquelle` ist also die Empfänger-Klasse und hat zwei Methoden, um ihren Status ändern zu können – also um das Licht ein- bzw. ausschalten zu können. Folglich gibt es Objekte zweier konkreten Befehlsklassen, nämlich der Klassen `LichtEinBefehl` und `LichtAusBefehl`. Ein Lichtschalter dient in diesem Beispiel dazu, eine Lichtquelle zu schalten. Die Klasse `LichtSchalter` fungiert somit als Aufrufer-Klasse. Sie speichert einen konkreten Befehl und kann dessen Ausführung veranlassen.

Zunächst das Klassendiagramm:

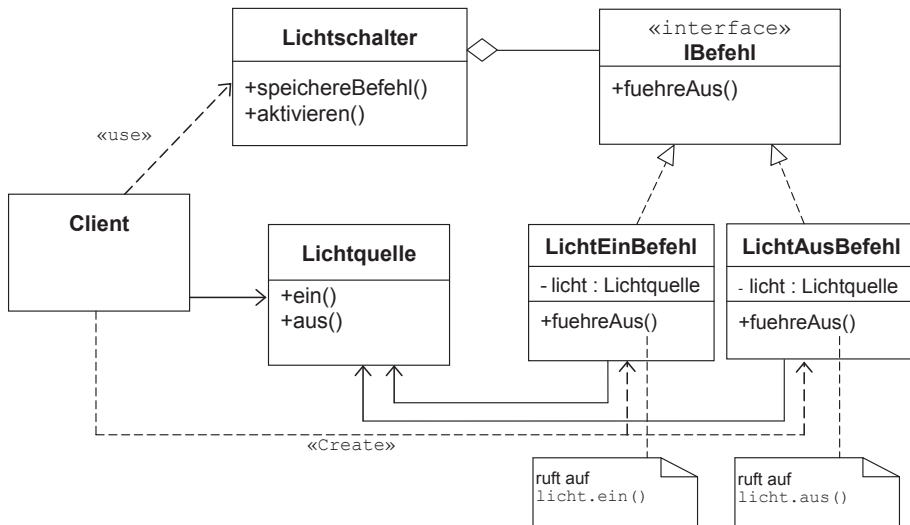


Bild 4-32 Klassendiagramm zum Programmbeispiel Befehlsmuster

Hier zunächst das Interface `IBefehl`, das die abstrakte Schnittstelle für die konkreten Befehlsklassen definiert:

```
// Datei: IBefehl.java
public interface IBefehl
{
    public void fuehreAus();
}
```

Die Klasse `LichtQuelle` ist in diesem Beispiel eine Empfänger-Klasse für konkrete Befehle. Sie stellt die Methoden `ein()` und `aus()` zur Verfügung, um eine Lichtquelle zu schalten:

```
// Datei: LichtQuelle.java
// LichtQuelle ist eine Empfaenger-Klasse mit zwei Aktionen
public class LichtQuelle
{
    public void ein()
    {
        System.out.println ("Licht wurde eingeschaltet!");
    }

    public void aus()
    {
        System.out.println ("Licht wurde ausgeschaltet!");
    }
}
```

Die Klasse `LichtSchalter` stellt in diesem Beispiel eine Aufrufer-Klasse dar. Sie enthält eine Referenz auf das Interface `IBefehl`, die durch Aufruf der Methode `speichereBefehl()` gesetzt werden kann. Die Referenz zeigt dann auf einen konkreten Befehl, dessen Ausführung ausgelöst werden kann. Hier die Klasse `LichtSchalter`:

```
// Datei: LichtSchalter.java
// Lichtschalter ist eine Aufrufer-Klasse und speichert einen Befehl
public class LichtSchalter
{
    private IBefehl befehl; // Referenz auf IBefehl

    public void speichereBefehl (IBefehl befehl)
    {
        this.befehl = befehl;
    }

    public void aktivieren()
    {
        befehl.fuehreAus();
    }
}
```

Die Klasse `LichtEinBefehl` besitzt eine Referenz auf eine Lichtquelle und eine Methode `fuehreAus()`, welche das Licht dieser Lichtquelle einschaltet:

```
// Datei: LichtEinBefehl.java
// Diese Klasse stellt eine konkrete Befehlsklasse dar
public class LichtEinBefehl implements IBefehl
{
    private LichtQuelle licht;

    public LichtEinBefehl (LichtQuelle licht)
    {
        this.licht = licht;
    }

    public void fuehreAus() //Definition der Methode fuehreAus()
    {
        licht.ein();
    }
}
```

Die Klasse `LichtAusBefehl` besitzt eine Referenz auf eine Lichtquelle und eine Methode `fuehreAus()`, welche das Licht dieser Lichtquelle ausschaltet:

```
// Datei: LichtAusBefehl.java
// Diese Klasse stellt eine konkrete Befehlsklasse dar
public class LichtAusBefehl implements IBefehl
{
    private LichtQuelle licht;

    public LichtAusBefehl (LichtQuelle licht)
    {
        this.licht = licht;
    }
}
```

```

    public void fuehreAus() //Definition der Methode fuehreAus()
    {
        licht.aus();
    }
}

```

Durch den Client wird ein konkreter Befehl erstellt und in einem Objekt der Klasse `LichtSchalter` abgespeichert. Durch den Aufruf der Methode `aktivieren()` des Schalters wird die Ausführung des gespeicherten Befehls ausgelöst. Hier die Klasse `Client`:

```

// Datei: Client.java
public class Client
{
    public static void main (String[] args)
    {
        LichtSchalter schalter = new LichtSchalter();
        LichtQuelle licht = new LichtQuelle();

        IBefehl lichtAn = new LichtEinBefehl (licht);
        IBefehl lichtAus = new LichtAusBefehl (licht);

        schalter.speichereBefehl (lichtAn);
        schalter.aktivieren();

        schalter.speichereBefehl (lichtAus);
        schalter.aktivieren();
    }
}

```



Die Ausgabe des Programms ist:

```

Licht wurde eingeschaltet!
Licht wurde ausgeschaltet!

```

Im konkreten, vorliegenden Beispiel ruft der Client die Methode `aktivieren()` aus Gründen der Einfachheit selbst auf. Beispielsweise könnte eine andere Aufrufer-Klasse, die eine Zeitschaltuhr implementiert, die Ausführung eines konkreten Befehls (z. B. `LichtEinBefehl`) zu einem bestimmten Zeitpunkt veranlassen. Das Licht könnte so abends automatisch eingeschaltet und am nächsten Morgen wieder automatisch ausgeschaltet werden.

## 4.10.4 Bewertung

### 4.10.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Das Erzeugen eines Befehls und seine Ausführung sind zeitlich entkoppelt.



- Durch die Kapselung eines Befehls in einem Befehlsobjekt wird eine Zwischenschicht eingeführt. Damit kann ein Befehl die Methode `aktion()` asynchron aufrufen.
- Der Aufrufer braucht die Details der konkreten Befehle nicht zu kennen. Damit sind Befehle austauschbar und man kann Aufrufer-Klassen unabhängig vom Rest der Applikation implementieren.
- Man kann Befehle dynamisch zur Laufzeit austauschen und damit das Verhalten eines Aufrufers wie beispielsweise eines Buttons einer grafischen Oberfläche verändern.
- Befehlsobjekte können wiederverwendet werden. So kann beispielsweise bei einem Befehlsobjekt die Referenz auf ein Empfängerobjekt geändert werden. Das Befehlsobjekt kann dann erneut ausgeführt werden, der Client muss nicht für jeden Empfänger ein neues Befehlsobjekt erzeugen.

#### 4.10.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Durch das Verpacken eines Befehls in einem Objekt entsteht ein zusätzlicher Aufwand beim Aufruf des Befehls (Verschachtelung der Aufrufe).
- Für jede Art von konkretem Befehl muss eine eigene Klasse erstellt werden. Die Schnittstelle `IBefehl` definiert die Signatur der Methode `fuehreAus()`. Dadurch sind die Parameter dieser Methode für alle Befehle gleich. Es gibt somit einen erhöhten Aufwand, um einem Befehl Parameter hinzuzufügen oder einen Befehl einer anderen Empfänger-Klasse zu implementieren.
- Die Implementierung dieses Musters in verteilten Systemen wird komplex.

#### 4.10.5 Einsatzgebiete

Man kann das Befehlsmuster z. B. in grafischen Oberflächen anwenden. Die Anwendung (hier Client genannt) könnte beispielsweise einen Button als Aufrufer erzeugen, der beim Anklicken die Methode `fuehreAus()` eines Objektes einer Klasse aufruft, die das Interface `IBefehl` implementiert. In ähnlicher Weise können die Aktionen, die durch Auswahl eines Menüpunktes in einem Menü – wie beispielsweise das Speichern eines Dokumentes – ausgelöst werden sollen, realisiert werden.

Die Verwendung des Befehlsmusters bietet sich in weiteren Fällen an:

##### 1. Logging – vereinfachte Protokollierung

Man kann die Referenzen auf die Befehlsobjekte in einem Stream (Kanal) aufreihen, wodurch man eine Logging-Möglichkeit für Befehle hat.

##### 2. Zeitversetzte Ausführung – Einreihen der Befehle in eine Warteschlange

Werden die Referenzen auf die Befehle in eine Warteschlange (engl. queue) eingereiht, so kann ein Server-Prozess sie von dort aus zu gegebener Zeit zur Ausführung bringen.

### 3. Undo-/Redo-Funktionalität – Rückgängigmachen von Befehlen

Ein Undo-/Redo-Mechanismus kann realisiert werden, indem die Referenzen auf ausgeführte Befehle gespeichert werden. Entsprechende `undo()`- und `redo()`-Methoden müssen in jeder einzelnen konkreten Befehlsklasse implementiert werden. Werden die Referenzen auf die ausgeführten Befehle dann in einem Befehlsstack abgelegt, so kann festgestellt werden, welcher Befehl als nächstes rückgängig gemacht oder erneut ausgeführt werden soll.

### 4. Zusammengesetzte Befehle – Makros

Mit Hilfe des **Kompositum-Musters** kann ein Befehl aus mehreren Befehlen zusammengesetzt werden. Dies erlaubt eine vereinfachte Behandlung von Operationen, die auf vielen Objekten agieren und dabei verschiedene Aktionen auf diesen Objekten durchführen. Das Interface `IBefehl` übernimmt die Rolle eines Knotens der Komponente im Kompositum-Muster, ein einfacher konkreter Befehl ist ein Blatt und ein Makro entspricht einem Kompositum.

### 5. Transaktionen – Rollback

Ein zusammengesetzter Befehl kann den Zustand der zu verändernden Objekte festhalten und danach erst seine Operationen ausführen. Wenn dann eine dieser Operationen fehlschlägt, kann der Befehl die Objekte wieder in den Zustand vor dem Befehl zurücksetzen, ganz analog zu einem Rollback einer Datenbank. Zur Speicherung des Zustands kann das in diesem Buch nicht behandelte **Memento-Pattern** (siehe beispielsweise [Gam95]) eingesetzt werden.

### 6. Recovery nach Systemcrash – persistente Speicherung der Befehle

Wenn die Referenzen auf die ausgeführten Befehle persistent gespeichert werden, können sie nach einem Systemcrash – wie bei einem Transaction Log – zurückgespielt werden.

## 4.10.6 Ähnliche Entwurfsmuster

Das Entwurfsmuster **Strategie** ist ähnlich dem Entwurfsmuster Befehl: beide kapseln einen Methodenaufwurf in einem eigenständigen Objekt. Die Teilnehmer haben andere Namen, entsprechen sich aber wie folgt: der Aufrufer heißt beim Strategie-Muster Kontext, das Interface `IBefehl` heißt dort `IStrategie`. Das Entwurfsmuster Befehl ist jedoch allgemeiner anwendbar: Ein Befehl ist allgemeiner definiert, während bei den Strategien davon ausgegangen wird, dass alle Strategien funktional gleich sind. Ein weiterer Unterschied ist, dass die Befehle von einem Empfänger ausgeführt werden, während eine Strategie immer nur vom Kontextobjekt selbst ausgeführt wird.

Das Entwurfsmuster **Command Processor** [Sta11] entspricht in etwa der in Kapitel 4.10.5 unter Punkt 2 genannten Variante des Befehlsmusters. Der Command Processor hat gegenüber dem Aufrufer im einfachen Befehlsmuster eine erweiterte Aufgabe: er verwaltet Befehlsobjekte und koordiniert deren Ausführung.

Das Entwurfsmuster **Active Object** [Sch00] stellt ebenfalls eine Erweiterung des Befehlsmusters dar. Im Entwurfsmuster Active Object werden Befehle in einem eigenen Thread ausgeführt.

## 4.11 Das Verhaltensmuster Beobachter

### 4.11.1 Name/Alternative Namen

Beobachter (engl. Observer), Publizieren/Abonnieren (engl. Publish/Subscribe), Zuhörer (engl. Listener)<sup>54</sup>.

### 4.11.2 Problem

Ändert sich der Zustand eines Objekts, sollen alle von diesem Objekt abhängigen Objekte benachrichtigt werden, so dass eine Aktualisierung dieser Objekte automatisch eingeleitet werden kann. Die beteiligten Objekte sollen lose gekoppelt sein, um eine Wiederverwendung der Klassen zu ermöglichen. Die Menge der Objekte, die bei einer Zustandsänderung benachrichtigt werden sollen, soll dynamisch veränderbar sein, d. h., abhängige Objekte können zu dieser Menge hinzukommen und zu einem späteren Zeitpunkt auch wieder entfernt werden.

Das Beobachter-Muster soll es erlauben, dass ein Objekt abhängige Objekte von einer Änderung seines Zustands informiert, so dass eine Aktualisierung automatisch eingeleitet werden kann.



### 4.11.3 Lösung

Das Beobachter-Muster ist ein objektbasiertes Entwurfsmuster. Ein Objekt vom Typ `Beobachtbar` kann von einer beliebigen Anzahl Objekte der Klasse `Beobachter` überwacht werden.

Die Struktur des Entwurfsmusters Beobachter enthält:

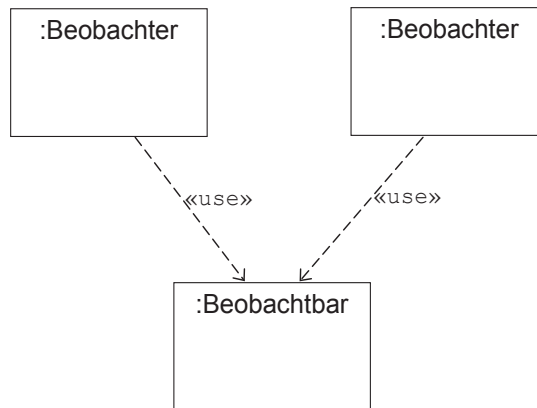
- ein Daten haltendes Objekt (**Beobachtbares Objekt/Observable/Publisher**), dessen Daten und damit dessen Zustand sich ändern können, und
- mehrere Interessenten für die gekapselten Daten (**Beobachter/Observer/Subscriber**), die gegebenenfalls auf Änderungen der Daten reagieren wollen.

Dabei soll ein Daten haltendes Objekt die Interessenten noch nicht zur Kompilierzeit, sondern erst zur Laufzeit nach ihrer Anmeldung kennen. Damit ein Beobachter über Änderungen des zu beobachtenden Objektes informiert wird, muss er sich zuvor beim beobachtbaren Objekt anmelden. Anschließend erhält der Beobachter solange bei jeder Zustandsänderung des zu beobachtenden Objekts eine Nachricht, bis er sich selbst wieder abmeldet. Das Muster geht davon aus, dass ein Beobachter das Objekt kennt, bei dem er sich anmelden und abmelden muss – schließlich will er es ja beobachten.

<sup>54</sup> In der deutschen Literatur ist praktisch nur der deutsche Name Beobachter geläufig, ansonsten alle genannten englischen Namen.

Klassisch würde ein Beobachter per Polling<sup>55</sup> anfragen, ob sich Daten geändert haben. Beim Beobachter-Muster ist jedoch die Kontrolle vertauscht – die Kontrolle hat der Beobachtbare. Bei einer Änderung der Daten generiert der Beobachtbare ein Ereignis für den Beobachter. Diese Vorgehensweise beim Beobachter-Muster entspricht also dem Prinzip "**Inversion of Control**" (siehe Kapitel 1.9.2).

Das folgende Bild zeigt die Abhängigkeit einer Beobachter-Klasse von der beobachtbaren Klasse:



*Bild 4-33 Abhängigkeit eines Beobachters vom beobachtbaren Objekt*

Eine **Verwendungsbeziehung** ist eine spezielle semantische Ausprägung einer **Abhängigkeitsbeziehung**. Eine Verwendungsbeziehung wird durch «use» charakterisiert. Der Benutzende ist der Abhängige. Der Benutzte ist der Unabhängige.

In der Regel ist ein Beobachter nicht nur daran interessiert, ob sich ein von ihm beobachtetes Objekt ändert, sondern auch daran, was sich bei dem Objekt geändert hat, bzw. wie dessen neuer Zustand ist. Um den Beobachtern den neuen Zustand mitzuteilen, gibt es im Rahmen dieses Musters zwei verschiedene Verfahren. Im **Push-Verfahren** wird der neue Zustand an die Aktualisierungs-Operation übergeben – die Daten werden einem Beobachter "zugeschoben". Im **Pull-Verfahren** ist der Beobachter hingegen selbst für das Abfragen des neuen Zustandes verantwortlich – er "zieht" sich die Daten, wenn er informiert wurde, dass die Daten sich geändert haben. Hierfür benötigt er eine Referenz auf den Beobachtbaren, der eine Methode für die Abfrage der geänderten Daten zur Verfügung stellen muss. Im Folgenden wird das Pull-Verfahren behandelt. Das Push-Verfahren wird dann in Kapitel 4.11.3.3 im Detail vorgestellt. Dort wird auch ein Vergleich zwischen beiden Verfahren durchgeführt.

#### 4.11.3.1 Klassendiagramm

Charakteristisch für das Beobachter-Muster in der **Pull-Variante** sind zwei Methoden, die Methode `aktualisieren()` und die Methode `gibZustand()`. Beim Aufruf der Methode `aktualisieren()` übergibt ein beobachtbares Objekt eine Referenz auf

<sup>55</sup> to poll (engl.) bedeutet abfragen. In der Informatik hat Polling die Bedeutung, dass mittels regelmäßiger zyklischer Abfragen der Zustand von Daten (Objekten, Geräten etc.) ermittelt wird.

sich selbst. Daraufhin kann der aufgerufene Beobachter mittels dieser Referenz die Methode `gibZustand()` bei dem beobachteten Objekt aufrufen und sich die Information über den neuen Zustand des beobachteten Objekts besorgen. Die Übergabe der Selbst-Referenz beim Aufruf der Methode `aktualisieren()` ist deshalb nötig, damit ein Beobachter, der mehrere Objekte gleichzeitig beobachtet, weiß, welches dieser Objekte seinen Zustand geändert hat. Die Methode `gibZustand()` steht hier stellvertretend für eine oder mehrere get-Methoden, die es einem Beobachter erlauben, sich über den Zustand eines beobachteten Objekts zu informieren. Die beiden genannten Methoden sind im folgenden Klassendiagramm des Beobachter-Musters dargestellt:

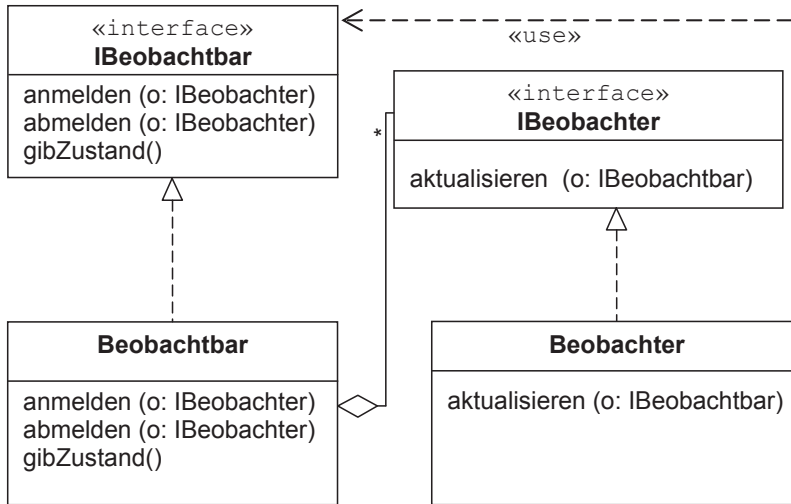


Bild 4-34 Klassendiagramm des Beobachter-Entwurfsmusters

Beim Beobachter-Muster (siehe Bild 4-34) werden Nachrichten in beide Richtungen zwischen den Objekten vom Typ `Beobachter` und dem Objekt vom Typ `Beobachtbar` ausgetauscht. Das beobachtbare Objekt verpflichtet den Aufrufer, ein spezielles Interface (**Callback-Interface**<sup>56</sup>) zu implementieren, das vom beobachtbaren Objekt vorgegeben wird. Ändert sich ein Beobachter, so hat dies keinen Einfluss auf das beobachtbare Objekt, solange das Callback-Interface `IBeobachter` und das Interface `IBeobachtbar` eingehalten werden. Dagegen wirken sich Änderungen an dem Callback-Interface `IBeobachter` oder an dem Interface `IBeobachtbar` auf die Beobachter aus.

Merkmale des Musters sind:

- Das zu beobachtende Objekt kennt zur Kompilierzeit nur das Beobachter-Interface `IBeobachter`, d. h. das Callback-Interface, und das Interface `IBeobachtbar` des Beobachtbaren, aber nicht die Implementierung eines Beobachters.
- Das zu beobachtende Objekt hält zur Laufzeit nach der Anmeldung eines Beobachters eine Referenz auf diesen Beobachter.

<sup>56</sup> Die Callback-Funktion (Rückruffunktion) wird vom Objekt der Klasse `Beobachtbar` aufgerufen.

- Der Beobachter kennt zur Kompilierzeit nur das Interface `IBeobachtbar` und das Callback-Interface, das er selbst implementiert.

Die Methodenaufrufe gehen also in beide Richtungen: Zum einen benutzt ein Objekt der Klasse `Beobachter` das Interface `IBeobachtbar` und ruft damit eine Methode eines Objektes auf, das dieses Interface implementiert. Zum anderen hält ein beobachtbares Objekt zur Laufzeit eine Referenz auf ein Objekt, das das Interface `IBeobachter` implementiert, und ruft Methoden auf diesem Objekt auf. Da aber das Interface `IBeobachter` vom Beobachtbaren vorgegeben wird und damit zum Beobachtbaren gehört, ist der Beobachtbare in der Tat nicht von dem Interface `IBeobachter` abhängig. Hingegen ist der Beobachter vom Beobachtbaren abhängig. Obwohl die Methodenaufrufe also in beide Richtungen gehen, besteht eine Abhängigkeit nur in einer einzigen Richtung, wie bereits in Bild 4-33 dargestellt wurde.

Ein Beobachter ist nur dann ein Beobachter, wenn er

- das Interface des beobachteten Objekts, `IBeobachtbar`, benutzt und
- selbst das vom beobachteten Objekt verlangte Callback-Interface, nämlich das Interface `IBeobachter`, implementiert.

Zum Beobachtbaren gehören deswegen:

- die Vorgabe des Interface des beobachteten Objekts, `IBeobachtbar`,
- die Implementierung dieses Interface und
- die Vorgabe des Interface `IBeobachter` des Beobachters (Callback-Interface).

#### 4.11.3.2 Teilnehmer

Für die beiden Rollen gibt es jeweils ein Interface<sup>57</sup>, nämlich die Interfaces `IBeobachtbar` und `IBeobachter`.

##### **IBeobachtbar**

Dieses Interface für das beobachtbare Objekt enthält die Operationen für das An- und Abmelden der Beobachter, sowie die abstrakte Methode `gibZustand()` zum Abruf der geänderten Daten. Beliebig viele Exemplare vom Typ `Beobachter` können ein Objekt vom Typ `Beobachtbar` beobachten.

##### **IBeobachter**

`IBeobachter` ist das Callback-Interface des Beobachtbaren. Der Beobachtbare kennt nur dieses Interface eines Beobachters. Dieses Interface wird von einem Beobachter realisiert. Es enthält den Methodenkopf der Methode `aktualisieren()`. Die Methode `aktualisieren()` wird vom Beobachtbaren für einen Rückruf, also zum Signalisieren einer Zustandsänderung, verwendet.

Ferner gibt es die Klassen `Beobachtbar` und `Beobachter`:

---

<sup>57</sup> Alternativ kann auch eine abstrakte Klasse verwendet werden.

### Beobachtbar

Ein Beobachtbarer hält eine Liste von Referenzen auf Objekte, die das Interface `IBeobachter` implementieren. Meldet sich ein Beobachter an, so wird ein neuer Eintrag zu dieser Liste hinzugefügt. Bei Abmeldung wird der Eintrag wieder aus der Liste entfernt. Hierdurch hat der Beobachtbare immer eine aktuelle Liste seiner angemeldeten Beobachter. Änderungen am Beobachter sind für den Beobachtbaren irrelevant, solange dieser das Callback-Interface einhält. Zur Kompilierzeit kennt der Beobachtbare vom Beobachter nur das Callback-Interface `IBeobachter`. Nach dem liskovschen Substitutionsprinzip kann bei Einhaltung der Verträge an die Stelle eines Interface stets ein Objekt treten, das dieses Interface implementiert. Demnach besteht zwar eine Abhängigkeit zu diesem Interface beim Kompilieren. Da das Interface aber vom Beobachtbaren vorgegeben wird, ist es de facto keine wirkliche Abhängigkeit.

Wenn sich sein Zustand ändert, benachrichtigt der Beobachtbare die bei ihm angemeldeten Beobachter durch Aufruf der Methode `aktualisieren()`.

### Beobachter

Ein Beobachter implementiert das Interface `IBeobachter` und damit auch die Methode `aktualisieren()`. Wenn die Beobachtbare über eine Änderung benachrichtigt werden sollen, iteriert der Beobachtbare in der Methode `benachrichtigen()` über die gespeicherten Referenzen der Beobachter. Dabei führt er für jeden Beobachter die Benachrichtigungs-Operation `aktualisieren()` aus.

#### 4.11.3.3 Dynamisches Verhalten

Jeder Beobachter meldet sich beim Objekt der Klasse `Beobachtbar`, das er beobachten will, an. Werden die Daten eines beobachtbaren Objekts geändert, was in Bild 4-35 beispielhaft durch einen Aufruf der Methode `setzeZustand()` angedeutet wird, dann ruft das beobachtbare Objekt seine eigene Methode `benachrichtigen()` (in einer neuen, verschobenen Ausführungsspezifikation) auf. Die Methode `benachrichtigen()` informiert alle angemeldeten Beobachter durch einen Aufruf der Methode `aktualisieren()` und übergibt dabei eine Referenz auf sich selbst. Ein Beobachter kann nun mit Hilfe der Methode `gibZustand()` vom beobachtbaren Objekt die Information über den neuen Zustand erhalten. Daher kommt es auch dort zu einem weiteren Balken der Ausführungsspezifikation<sup>58</sup>.

Die Methode `setzeZustand()` wurde bislang nicht erwähnt und spielt im Rahmen des Beobachter-Musters keine Rolle. Diese Methode steht stellvertretend für eine anwendungsspezifische Methode der konkreten Klasse der beobachtbaren Objekte, in der die für die Beobachtung relevanten Daten eines Objektes geändert werden.

An der Methode `gibZustand()` sieht man, dass in diesem Sequenzdiagramm das Pull-Verfahren gezeigt wird. Der Beobachter holt die Änderungen selbst ab. Die Methode `gibZustand()` stellt die Ausführung einer weiteren Methode im beobachtbaren

<sup>58</sup> Eine Ausführungsspezifikation – in UML 1 hieß der Begriff Aktivitätsbalken oder Steuerungsfokus (engl. focus of control) – zeigt sowohl die Zeitdauer, während der eine Ausführung aktiv ist, als auch die Kontrollbeziehung zwischen der Ausführung und dem dazugehörigen Aufrufer.

Objekt dar. Dies führt wiederum zu einer Schachtelung der Methodenaufrufe und zu einer neuen verschobenen Ausführungsspezifikation.

Das folgende Sequenzdiagramm zeigt exemplarisch die Anmeldung und Aktualisierung eines Beobachters:

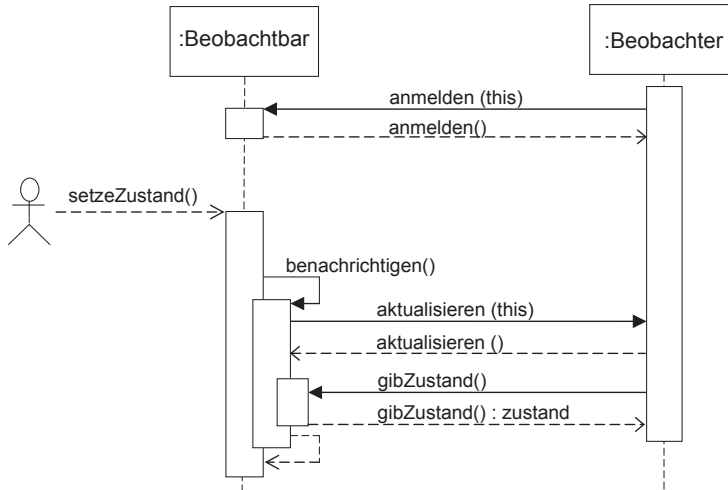


Bild 4-35 Sequenzdiagramm für die Anmeldung und Aktualisierung eines Beobachters

#### 4.11.3.4 Lösungsvariante mit dem Push-Verfahren

Die bisher beschriebene Pull-Variante ist dadurch charakterisiert, dass ein Beobachter zwar über eine Zustandsänderung informiert wird (Methode `aktualisieren()`), er aber danach sich die Informationen über den neuen Zustand beim beobachteten Objekt abholen muss (Methode `gibZustand()`).

In der Push-Variante werden die beiden genannten Methoden praktisch zu einer einzigen zusammengeführt: die Methode `aktualisieren()` erhält zusätzlich einen oder mehrere Parameter, die den neuen Zustand des Beobachtbaren beschreiben<sup>59</sup>. Durch einen Aufruf dieser modifizierten Methode `aktualisieren()` beim Beobachter "drückt" (push) das beobachtete Objekt gleich alle Informationen zu dem Beobachter hin. Die Methode `gibZustand()` kann entfallen.

Der Vorteil des Push-Verfahrens liegt damit auf der Hand: der Rückruf des Beobachters, um sich Informationen zu beschaffen, entfällt. Häufig sind im Pull-Verfahren für diese Informationsbeschaffung sogar mehrere Methodenaufrufe nötig – je nachdem wie die Schnittstelle vom Beobachtbaren konzipiert ist – die alle im Push-Verfahren überflüssig werden.

Ein Vorteil des Pull-Verfahrens ist darin zu sehen, dass die Schnittstelle zum Abholen der Information "breit" gemacht werden kann: Statt der bisher genannten Methode

<sup>59</sup> Damit verringert sich die Wiederverwendbarkeit der Methode `aktualisieren()` deutlich.



`gibZustand()` können mehrere `get`-Methoden zur Verfügung gestellt werden. Ein Beobachter kann dann entscheiden, welche Information er sich abholt. In der Push-Variante wird ein Beobachter immer mit der maximalen Information "geflutet".

Eine breite Schnittstelle beim Pull-Verfahren hat auch noch einen kleinen Vorteil bezüglich der Wartbarkeit: Wenn die Klasse `Beobachtbar` erweitert wird und zusätzliche Daten zu einem Zustandswechsel bereitstellen möchte, kann dies in manchen Fällen über zusätzliche `get`-Methoden im Interface `IBeobachtbar` erfolgen. Beobachter, die diese zusätzlichen Daten nicht benötigen, brauchen in diesen Fällen nicht geändert zu werden. In der Push-Variante würde die Methode `aktualisieren()` um zusätzliche Parameter erweitert, was in jedem Fall eine Änderung der Beobachter-Klassen erfordert. Es sei aber darauf hingewiesen, dass weitergehende Änderungen im Interface `IBeobachtbar` als die gerade beschriebenen auch in der Pull-Variante zu Änderungen in den Beobachter-Klassen führen.

Der Vorteil des Push-Verfahrens liegt auf der Hand: der Rückruf des Beobachters, um sich Informationen zu beschaffen, entfällt. Bei einer breiten Schnittstelle sind im Pull-Verfahren für diese Informationsbeschaffung sogar mehrere Methodenaufrufe nötig, die alle im Push-Verfahren überflüssig werden. Allerdings hat bei der Push-Schnittstelle jede Anwendung ihre eigenen Parameter. Damit kann die entsprechende Methode in der Regel nicht wiederverwendet werden.

#### 4.11.3.5 Programmbeispiel

In diesem Beispiel wird das Beobachter-Muster anhand eines Newsletter gezeigt. Dem Newsletter soll es möglich sein, seine Abonnenten zu verwalten und zu benachrichtigen, wenn es neue Informationen gibt. Das Beispiel ist gemäß dem Pull-Verfahren aufgebaut.

Das Interface `IBeobachter` definiert einen Beobachter, der durch die Methode `aktualisieren()` über Änderungen informiert werden kann:

```
// Datei: IBeobachter.java
public interface IBeobachter
{
    public void aktualisieren (IBeobachtbar b);
}
```

Das Interface `IBeobachtbar` wird durch ein beobachtbares Objekt implementiert, das seine Beobachter mit den Methoden `anmelden()` und `abmelden()` verwaltet. Mit der Methode `gibZustand()` können sich die angemeldeten Beobachter über die Änderungen informieren. Hier das Interface `IBeobachtbar`:

```
// Datei: IBeobachtbar.java
public interface IBeobachtbar
{
    public void anmelden (IBeobachter beobachter);
    public void abmelden (IBeobachter beobachter);
    public String gibZustand();
}
```

Die Klasse `Abonnent` stellt eine Implementierung des Interface `IBeobachter` dar:

**// Datei: Abonnent.java**

```
public class Abonnent implements IBeobachter
{
    private String name;

    public Abonnent (String name)
    {
        this.name = name;
    }

    public void aktualisieren (IBeobachtbar b)
    {
        System.out.println ("Neue Nachricht fuer " + name + ".");
        System.out.println ("Nachricht: " + b.gibZustand());
    }
}
```

Die Klasse `Newsletter` stellt eine Implementierung des Interface `IBeobachtbar` dar:

**// Datei: Newsletter.java**

```
import java.util.Vector;

public class Newsletter implements IBeobachtbar
{
    private Vector<IBeobachter> abonnenten = new Vector<IBeobachter>();

    private String nachricht;

    public void aendereNachricht (String neueNachricht)
    {
        nachricht = neueNachricht;
        benachrichtigen();
    }

    public void abmelden (IBeobachter beobachter)
    {
        abonnenten.remove (beobachter);
    }

    public void anmelden (IBeobachter beobachter)
    {
        abonnenten.add (beobachter);
    }

    private void benachrichtigen()
    {
        for (IBeobachter beobachter : abonnenten)
        {
            beobachter.aktualisieren (this);
        }
    }
}
```

```
public String gibZustand()
{
    return nachricht;
}
```

Die Klasse `TestBeobachter` zeigt die Funktionalität eines Newsletter:

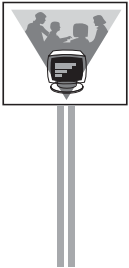
// Datei: `TestBeobachter.java`

```
public class TestBeobachter
{
    public static void main (String[] args)
    {
        Newsletter newsletter = new Newsletter();
        Abonnent andreas = new Abonnent ("Andreas");
        Abonnent birgit = new Abonnent ("Birgit");

        newsletter.anmelden (andreas);
        newsletter.anmelden (birgit);
        newsletter.aendereNachricht ("Neuigkeit 1");
        System.out.println();

        newsletter.abmelden (andreas);
        newsletter.aendereNachricht ("Neuigkeit 2");

        newsletter.abmelden (birgit);
        newsletter.aendereNachricht ("Neuigkeit 3");
    }
}
```



Die Ausgabe des Programms ist:

```
Neue Nachricht fuer Andreas.
Nachricht: Neuigkeit 1
Neue Nachricht fuer Birgit.
Nachricht: Neuigkeit 1

Neue Nachricht fuer Birgit.
Nachricht: Neuigkeit 2
```

## 4.11.4 Bewertung

### 4.11.4.1 Vorteile

Der Nutzen des Beobachter-Musters liegt in der hieraus resultierenden losen Koppelung. Hierbei ergeben sich unter anderem die folgenden Vorteile:

- Der Beobachtbare braucht zur Kompilierzeit die Beobachter nicht zu kennen, sondern nur das Interface, das aber von ihm selber vorgegeben wird.
- Neue Beobachter können jederzeit hinzugefügt werden.

- Der Beobachtbare muss nie modifiziert werden, um neue Beobachter zu unterstützen.
- Beobachter und Beobachtbarer können unabhängig voneinander wiederverwendet werden.
- Änderungen am Beobachtbaren, ohne dessen Interface zu ändern, haben keine Auswirkung auf die Gegenseite. Ein Beobachter kann auch beliebig geändert werden, solange er das Callback-Interface einhält.

#### 4.11.4.2 Nachteile

Es können jedoch auch Nachteile entstehen:

- Gibt es besonders viele Beobachter, so kann deren Benachrichtigung durch den Beobachtbaren sehr zeitaufwendig sein.
- Es muss darauf geachtet werden, dass durch die Behandlung einer Benachrichtigung vom Beobachter keine neuen Zustandsänderungen des Beobachteten ausgelöst werden, da dies sonst zu einer Endlosschleife führen kann.
- Jeder Beobachter wird informiert, auch wenn er eine Änderung nicht braucht.

#### 4.11.5 Einsatzgebiete

Das Beobachter-Muster kommt beispielsweise zum Einsatz, um mehrere Ansichten auf ein Datenmodell zu ermöglichen. Es ist in der Softwareentwicklung häufig notwendig, dass bei einer Zustandsänderung eines Objektes verschiedene andere Objekte informiert werden. Besonders häufig findet man eine solche Problemstellung bei der Programmierung grafischer Oberflächen. So ist beispielsweise eine View im Architekturmuster **Model-View-Controller** ein Beobachter im Sinne des Beobachter-Musters.

Das Beobachter-Muster kann auch Verwendung im **Vermittler-Muster** finden: Objekte der Klasse `Kollege` können damit den Vermittler benachrichtigen, der in diesem Falle die Rolle eines Beobachters einnimmt.

In der Sprache Java gibt es bereits Unterstützung für das Beobachter-Muster: Die Klasse `java.util.Observable` implementiert bereits die nötigen Funktionen zur Verwaltung von Beobachtern für ein beobachtbares Objekt. Es handelt sich dabei also nicht um ein Interface, wie dies in der bisherigen Beschreibung des Musters der Fall war, sondern um eine abstrakte Klasse. Um das Beobachter-Muster zu realisieren, muss man die Klasse eines beobachtbaren Objekts von der Klasse `java.util.Observable` ableiten. Des Weiteren wird das Interface `java.util.Observer` vorgegeben, das ein Beobachter implementieren muss.

#### 4.11.6 Ähnliche Entwurfsmuster

Sowohl über das Beobachter-Muster als auch über das **Vermittler-Muster** kann die Zusammenarbeit von Objekten gesteuert werden. Während beim Vermittler-Muster die Objekte (die Kollegen) gleichberechtigt sind und potentiell jedes Objekt mit jedem anderen über den Vermittler kommunizieren kann, haben die am Beobachter-Muster be-

teiligten Objekte bestimmte Rollen, die die Kommunikationsmöglichkeiten einschränken: nur beobachtbare Objekte können ihre Beobachter informieren und nicht umgekehrt. Das bedeutet, dass das Beobachter-Muster für einfachere Anwendungen besser geeignet ist. Würde man aber die komplexe Zusammenarbeit zwischen den Kollegen über das Beobachter-Muster realisieren, wäre jeder Kollege sowohl Beobachter als auch Beobachtbarer. Die daraus resultierende Kaskade von Benachrichtigungen wäre unüberschaubar und kaum nachzuvollziehen. Die Einschaltung eines Vermittlers "synchronisiert" in gewisser Weise auch die Zusammenarbeit. Denn ein benachrichtigter Kollege kann zwar sofort wieder den Vermittler anrufen, aber der Vermittler wird zuerst die alte Benachrichtigung noch komplett abarbeiten, bevor er sich dem neuen Anruf zuwendet.

## 4.12 Das Verhaltensmuster Strategie

### 4.12.1 Name/Alternative Namen

Strategie (engl. strategy oder policy).

### 4.12.2 Problem

Das Strategie-Muster soll es erlauben, dass eine Strategie, die in Form eines Algorithmus vorliegt, komplett ausgetauscht wird. Es soll also das Verhalten eines Objekts flexibilisiert werden und das jeweils passende Verhalten verwendet werden, ohne, dass die Klasse des den Algorithmus nutzenden Objekts geändert werden muss. Ein Objekt, das den auszutauschenden Algorithmus nutzt, wird im Folgenden als Kontextobjekt bezeichnet. Ein Kontextobjekt darf nicht von einer speziellen Implementierung des Algorithmus abhängen.

Das Strategie-Muster soll es erlauben, dass ein ganzer Algorithmus in Form einer Kapsel ausgetauscht wird.



### 4.12.3 Lösung

Damit die verschiedenen Algorithmen einer Gruppe von miteinander verwandten Algorithmen austauschbar werden, benötigen sie dasselbe Interface<sup>60</sup>, das im Folgenden mit `IStrategie` bezeichnet wird. Jeder einzelne Algorithmus wird nun als sogenannte Strategie separat in einer eigenen Klasse gekapselt, um ihn als Kapsel austauschbar zu machen. Bei Einhaltung der Verträge kann jede der konkreten Strategien an die Stelle des Interface `IStrategie` treten. Der jeweils zu einer bestimmten Zeit benötigte Algorithmus wird von der Umgebung des Kontextobjektes ausgewählt.

Das Strategie-Muster zieht die alternativen Strategien heraus, abstrahiert sie durch ein Interface vom Typ `IStrategie` und kapselt sie jeweils in einem eigenen Objekt.



Das Entwurfsmuster Strategie ist ein objektbasiertes Verhaltensmuster, da es einen Algorithmus in einem Objekt kapselt, das von einem anderen Objekt, dem Kontextobjekt, benutzt wird.

#### 4.12.3.1 Klassendiagramm

Beim Strategie-Muster wird eine Strategie, die ein Kontextobjekt benutzt, in ein eigenes Objekt vom Typ `IStrategie` ausgelagert. Das Kontextobjekt wird mit der Strategie

<sup>60</sup> Es besteht ebenfalls die Möglichkeit, das Strategie-Entwurfsmuster mit einer abstrakten Klasse anstatt einer Schnittstelle als Abstraktion zu entwerfen.

gie über eine Aggregationsbeziehung verbunden und kann so die Strategie nutzen. Die Aggregationsbeziehung erlaubt es nun dem Kontextobjekt, alle möglichen unterschiedlichen konkreten Strategien zu nutzen, solange die konkreten Strategien das Interface `IStrategie` realisieren und den Vertrag des Interface nicht brechen. Alle verschiedenen Algorithmen der Klassen `KonkreteStrategieX` ( $X = A..Z$ ) implementieren dabei das definierte Interface `IStrategie`. Das Interface `IStrategie` wird vom Kontext vorgegeben. Da jede konkrete Strategie dieses Interface erfüllen muss, hängt der Kontext nicht von der jeweiligen konkreten Strategie ab. Das wird als **Dependency Inversion** bezeichnet (siehe Kapitel 1.9.1). Das folgende Bild zeigt das Klassendiagramm:

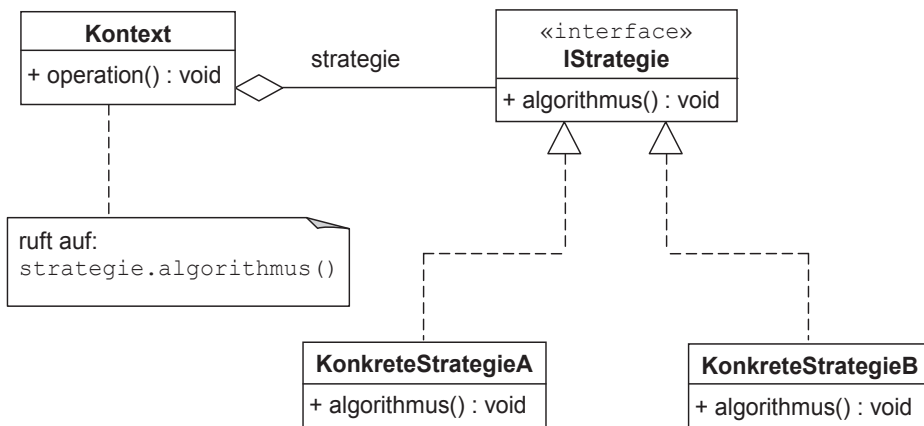


Bild 4-36 Klassendiagramm des Strategie-Musters

Die Klasse `Kontext` aggregiert das Interface `IStrategie`. Das dieses Interface implementierende Objekt ist damit Teil des Kontextobjekts und kann – falls gewünscht – jederzeit zur Laufzeit ausgetauscht werden. Der Kontext muss für seine Umgebung daher eine Möglichkeit bieten, um diese Referenz zu setzen. Dies wird auch als **Konfiguration des Kontextes** bezeichnet.

#### 4.12.3.2 Teilnehmer

Folgende Klassen bzw. Interfaces sind an dem Entwurfsmuster beteiligt:

##### Kontext

Die Klasse `Kontext` aggregiert als Abstraktion das Interface `IStrategie`. In Java enthält die Klasse `Kontext` dazu eine Referenz `strategie` vom Typ des Interface `IStrategie`. Durch diese Referenz wird die zu verwendende Strategie festgelegt. Durch `strategie.algorithmus()` wird die gewählte Strategie ausgeführt.

##### IStrategie

Das Interface `IStrategie` wird von allen unterstützten Algorithmen realisiert. Es wird dazu verwendet, um einen durch eine der Klassen `KonkreteStrategieX` implementierten Algorithmus aufzurufen.

### KonkreteStrategieX

Eine Klasse `KonkreteStrategieX` ( $X = A..Z$ ) implementiert einen konkreten Algorithmus, dessen Methodenkopf im Interface `IStrategie` deklariert wurde.

#### 4.12.3.3 Dynamisches Verhalten

Ein Objekt der Klasse `Kontext` kann beispielsweise im Konstruktor bzw. beim Aufruf von `setzeStrategie()` die Referenz auf ein Objekt der Klasse `KonkreteStrategieX` ( $X = A..Z$ ) erhalten. Damit kann es die Methoden des Objektes der Klasse `KonkreteStrategieX` aufrufen.

Ein Sequenzdiagramm für das Strategie-Muster wird in Bild 4-37 gezeigt:

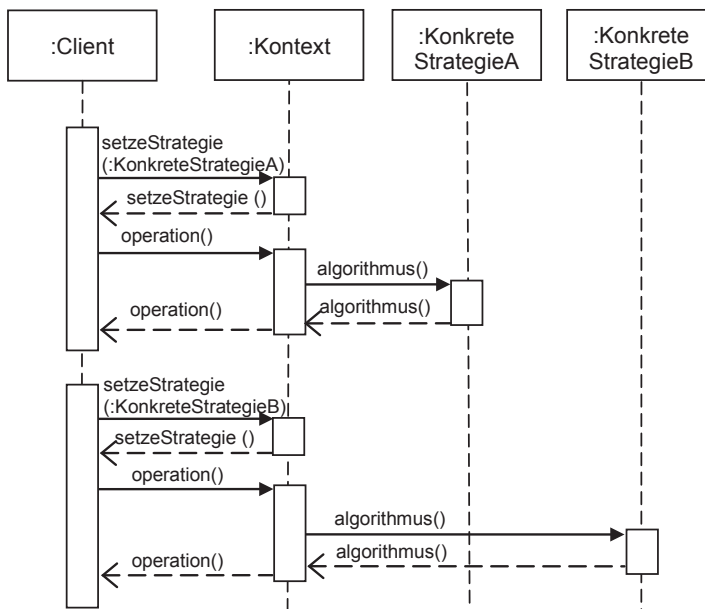


Bild 4-37 Sequenzdiagramm Strategie

Die Aufrufe der Methode `operation()` laufen im Kontextobjekt unterschiedlich ab, je nachdem, welche aktuelle Strategie vom Client gesetzt wurde. Beim ersten Aufruf wird die Methode `algorithmus()` eines Objekts der Klasse `KonkreteStrategieA` aufgerufen, beim zweiten Aufruf hingegen die Operation `algorithmus()` eines Objekts der Klasse `KonkreteStrategieB`.

#### 4.12.3.4 Programmbeispiel

In diesem Beispiel wird das Strategie-Muster dazu verwendet, einer Klasse, die zum Speichern eines Datums verwendet wird, zu ermöglichen, das Datum je nach gewählter Strategie in verschiedenen Datumsformaten auszugeben.



Die Schnittstelle `IDatumsFormat` definiert die Methode `datumAusgeben()`, die ein Datum in einem bestimmten Format ausgeben soll:

```
// Datei: IDatumsFormat.java
public interface IDatumsFormat
{
    public void datumAusgeben (int tag, int monat, int jahr);
}
```

Die Klasse `Datum` wird zum Speichern der Datumsinformationen verwendet. Sie spielt in diesem Beispiel die Rolle des Kontextes. Über die Methode `setzeFormat()` kann das Datumsformat, d. h. die Strategie, gesetzt werden. Mit der Methode `ausgeben()` wird das Datum ausgegeben und zwar entsprechend dem aktuell gesetzten Format. Hier der Quellcode der Klasse `Datum`:

```
// Datei: Datum.java
public class Datum
{
    private IDatumsFormat format = null;
    private int tag, monat, jahr = 0;

    public Datum (int tag, int monat, int jahr)
    {
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
    }

    public void setzeFormat (IDatumsFormat format)
    {
        this.format = format;
    }

    public void ausgeben()
    {
        // ruft die Methode datumAusgeben() des Objekts auf,
        // auf das format zeigt.
        format.datumAusgeben (tag, monat, jahr);
    }
}
```

Die beiden Klassen `EuropaeischesFormat` und `AmerikanischesFormat` definieren zwei Datumsformate mit den dazugehörigen Ausgabefunktionen. Zuerst wird die Klasse `EuropaeischesFormat` gezeigt, die ein Datum in der Form `tt.mm.jjjj` ausgibt:

```
// Datei: EuropaeischesFormat.java
public class EuropaeischesFormat implements IDatumsFormat
{
    public void datumAusgeben (int tag, int monat, int jahr)
    {
        System.out.println ("Europaeisches Format: "
            + (tag > 9 ? tag : "0" + tag)
            + ".")
    }
}
```

```

        + (monat > 9 ? monat : "0" + monat)
        + "."
        + jahr);
    }
}

```

Die Klasse `AmerikanischesFormat` gibt ein Datum in der Form `mm/dd/yyyy` aus:

```

// Datei: AmerikanischesFormat.java
public class AmerikanischesFormat implements IDatumsFormat
{
    public void datumAusgeben (int tag, int monat, int jahr)
    {
        System.out.println ("Amerikanisches Format: "
            + (monat > 9 ? monat : "0" + monat)
            + "/"
            + (tag > 9 ? tag : "0" + tag)
            + "/"
            + jahr);
    }
}

```

In der Klasse `TestStrategie` wird eine Instanz der Klasse `Datum` erzeugt. Daraufhin wird mit `setzeFormat()` die Formatierungsstrategie gesetzt und das Datum ausgegeben. Dann wird die Formatierungsstrategie auf ein anderes Format gesetzt und wieder das Datum ausgegeben. Hier die Klasse `Teststrategie`:

```

// Datei: TestStrategie.java
public class TestStrategie
{
    public static void main (String[] args)
    {
        Datum datum = new Datum (21, 9, 1985);

        datum.setzeFormat (new EuropaeischesFormat());
        datum.ausgeben();

        datum.setzeFormat (new AmerikanischesFormat());
        datum.ausgeben();
    }
}

```



Hier das Protokoll des Programmlaufs:

```

Europaeisches Format: 21.09.1985
Amerikanisches Format: 09/21/1985

```

#### 4.12.4 Einsatzgebiete

Das Entwurfsmuster Strategie ist einzusetzen, wenn eine Anwendung über mehrere alternative Strategien verfügt, zum Erreichen eines bestimmten Ergebnisses zu einer bestimmten Zeit aber nur eine davon benötigt. Der entsprechende Algorithmus soll von der Umgebung ausgetauscht werden können.

Ein gutes Beispiel aus der Praxis sind die Klassendefinitionen der grafischen Benutzeroberflächen von Java. Das Entwurfsmuster Strategie wird hierbei zur Delegation des Layouts von AWT- oder Swing-Komponenten an entsprechende Layoutmanager (`BorderLayout`, `FlowLayout` usw.) verwendet.

Die Beziehung zwischen View und Controller im **MVC-Muster** (siehe Kapitel 5.6) kann mit dem Strategie-Muster entworfen werden. Die Verwendung des Strategie-Musters erlaubt es der Umgebung, den Controller – und damit auch die Strategie der View – während der Laufzeit auszuwechseln. Dadurch kann die View ihr Verhalten ändern.

#### 4.12.5 Bewertung

##### 4.12.5.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Das Strategie-Muster kann wesentlich flexibler eingesetzt werden als eine Lösung, bei der die unterschiedlichen Algorithmen in Unterklassen der Kontextklasse realisiert werden. Bei dieser Lösung muss man eine Methode der Basisklasse in einer Unterklasse statisch überschreiben, wenn man sie variieren möchte.
- Durch die Verwendung einer Schnittstelle ist der Kontext nur vom Interface abhängig und nicht von dessen Implementierung (**Dependency Inversion**).
- Eine ganze Familie von Algorithmen ist möglich. Jeder Algorithmus ist für sich gekapselt und hat dieselbe Schnittstelle. Dadurch ist er flexibel zur Laufzeit austauschbar. Ein gekapselter Algorithmus kann leichter wiederverwendet werden.
- Es kann Code eingespart werden, wenn nicht alle Strategien gleichzeitig benötigt werden.
- Mehrfachverzweigungen können vermieden werden, was die Übersichtlichkeit des Programmtextes erhöht.

##### 4.12.5.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Von Nachteil ist, dass eine Applikation die unterschiedlichen Strategien eines Kontextobjekts kennen muss, um das Kontextobjekt passend initialisieren bzw. konfigurieren zu können. Diese Applikation muss also das Wissen besitzen, in welcher Situation welche Strategie angebracht ist.
- Es werden viele, oft kleine Klassen geschrieben.

- Beim Informationsfluss zwischen Kontext und Strategie entsteht ein höherer Kommunikationsaufwand gegenüber der herkömmlichen Implementierung der Algorithmen im Kontext selbst.

#### 4.12.6 Ähnliche Entwurfsmuster

Das Entwurfsmuster Strategie und das **Dekorierer-Muster** können das Verhalten eines Objekts ändern. Die Gesamtfunktionalität eines Kontextobjektes ändert sich durch den Austausch der Strategie nicht, weil alle Strategien die gleiche Funktionalität realisieren, allerdings hat jede Strategie ihre eigene Ausprägung. Beim Dekorierer-Muster hingegen wird die Funktionalität eines Objektes geändert bzw. erweitert. Alle Strategien realisieren eine gleichartige Funktionalität, hingegen kann jeder Dekorierer etwas ganz anderes machen. Ein Dekorierer behält das Verhalten des zu dekorierenden Objekts bei und versieht dieses Objekt mit einer Zusatzfunktionalität. Beim Strategiemuster hingegen wird von der Umgebung dieses Musters zielgerichtet das Kontextobjekt mit einer anderen Strategie versehen (kompletter Austausch eines Algorithmus). Damit wird keine Zusatzfunktionalität erzeugt, sondern es findet ein Austausch eines gesamten Algorithmus für das Kontextobjekt statt.

Das Entwurfsmuster **Schablonenmethode** befasst sich auch mit der Änderung eines Algorithmus. Eine Schablonenmethode legt aber das Grundgerüst eines Algorithmus statisch fest und erlaubt nur den Austausch einzelner Schritte des Algorithmus in verschiedenen Unterklassen. Eine Kontextklasse benutzt im Falle des Strategie-Musters den Algorithmus, ohne ihn überhaupt – also auch nicht schablonenhaft – selbst zu implementieren. Kurz und gut, der gesamte Algorithmus wird ausgetauscht, ohne, dass es eine Aussage über die Struktur des Algorithmus gibt. Das Strategie-Muster verwendet die Aggregation anstelle der statischen Vererbung, ist also objektbasiert und nicht klassenbasiert wie das Entwurfsmuster Schablonenmethode. Durch die Aggregation wird die Implementierung nicht zur Kompilierzeit, sondern zur Laufzeit austauschbar.

Das Entwurfsmuster Strategie ist dem Entwurfsmuster **Zustand** sehr ähnlich. Sowohl das Klassendiagramm als auch die Implementierung sind identisch, aber nicht die Interpretation. Während beim Zustandsmuster ein Kontextobjekt seine Anfragen an ein Objekt der Klasse `Zustand`, das seinen aktuellen Zustand repräsentiert, richtet, wird beim Strategie-Muster eine spezifische Anfrage eines Kontextobjekts an ein anderes Objekt gesandt, das eine Strategie (Algorithmus) zum Ausführen der Anfrage repräsentiert.

## 4.13 Das Verhaltensmuster Vermittler

Das Verhaltensmuster Vermittler oder Mediator darf nicht mit dem Architekturmuster Broker verwechselt werden. Ein Broker stellt umgangssprachlich auch einen Vermittler dar, aber keinen Vermittler im Sinne des Vermittler-Musters.

### 4.13.1 Name/Alternative Namen

Vermittler, Mediator (engl. mediator).

### 4.13.2 Problem

Ein zentraler Vermittler soll es erlauben, das Zusammenspiel zwischen vielen Objekten im Vermittler zu kapseln und zu steuern. Objekte sollen sich wechselseitig nicht mehr kennen, sondern nur noch den Vermittler. Der Vermittler soll das gewünschte Gesamtverhalten durch die Benachrichtigung der Kollegen erzeugen. Das Zusammenspiel der Objekte soll an zentraler Stelle im Vermittler abgeändert werden können, damit die einzelnen Objekte voneinander entkoppelt werden. Damit soll die Wiederverwendbarkeit der Objekte erhöht und außerdem das System übersichtlicher werden.

### 4.13.3 Lösung

Der Vermittler ist ein objektbasiertes Verhaltensmuster. Bei Änderungen eines Objekts benachrichtigt dieses den Vermittler. Der Vermittler wiederum benachrichtigt die anderen Objekte über die erfolgte Änderung. Zwischen einem aufrufenden Objekt und den anderen Objekten wird also ein Vermittler eingeführt, der auf Grund von eingehenden Nachrichten andere, davon betroffene Objekte benachrichtigt.

Die n-zu-m-Beziehung (vermaschtes Netz) zwischen den Objekten wird auf eine 1-zu-n-Beziehung (Sterntopologie) zwischen Vermittler und Objekten reduziert. Somit kann jedes Objekt mit jedem anderen in indirekter Art und Weise über einen Vermittler reden. Dadurch sind die Objekte nicht mehr wechselseitig voneinander abhängig, allerdings sind sie stark von ihrem Vermittler abhängig. Änderungen bei der Zustellung der Nachrichten erfolgen im Vermittler.

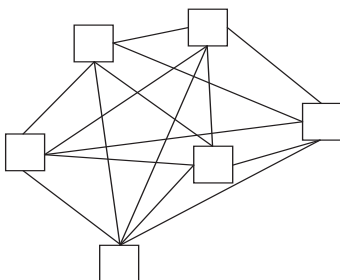


Bild 4-38 Vermaschtes Netz

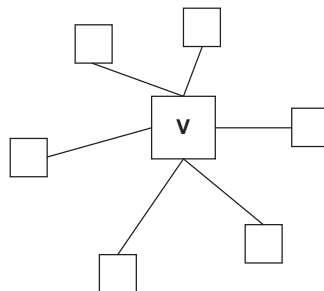


Bild 4-39 Sterntopologie

Beim Verhaltensmuster Vermittler kommunizieren Objekte nicht mehr direkt miteinander, sondern indirekt über einen Vermittler. Dieser informiert die von einer eingetroffenen Nachricht betroffenen Kollegen. Die wechselseitige Abhängigkeit der Objekte untereinander wird reduziert, die Objekte sind dafür vom Vermittler abhängig.



#### 4.13.3.1 Klassendiagramm

Die miteinander kommunizierenden Objekte werden als **Kollegen** bezeichnet. Das folgende Klassendiagramm des Vermittler-Musters verwendet für die Abstraktion eines Kollegen eine abstrakte Klasse. Diese abstrakte Klasse kann den Code, der zwischen den konkreten Kollegen geteilt wird und der beispielhaft im folgenden Bild durch die Methode `aenderung()` angedeutet ist, enthalten. Aber prinzipiell kann für die Abstraktion eines Kollegen sowohl eine abstrakte Klasse als auch ein Interface verwendet werden. Das Gleiche gilt auch auf der Vermittlerseite: Auch die Schnittstelle eines Vermittlers kann ebenso in einem Interface definiert werden anstatt in einer abstrakten Klasse.

Wie in Bild 4-40 zu sehen ist, hat die abstrakte Klasse `Kollege` eine Assoziation zu einem abstrakten Vermittler. Diese Assoziation erben alle konkreten Kollegen. Jeder Kollege soll also den Vermittler kennen.

Im Fall der Veränderung eines konkreten Kollegen-Objektes wird der assoziierte konkrete Vermittler benachrichtigt. Dieser informiert dann alle betroffenen weiteren konkreten Kollegen. Ein konkreter Vermittler muss also wissen, welche konkreten Kollegen er benachrichtigen soll. Das Klassendiagramm in Bild 4-40 zeigt daher eine gerichtete Assoziation zwischen der Klasse `KonkreterVermittler` und den Klassen `KonkreterKollegeX` ( $X = A..Z$ ).

Damit die Kommunikation zwischen Vermittler und Kollegen reibungslos funktioniert, muss in beiden Klassenhierarchien das liskovsche Substitutionsprinzip eingehalten werden. Hier das Klassendiagramm:

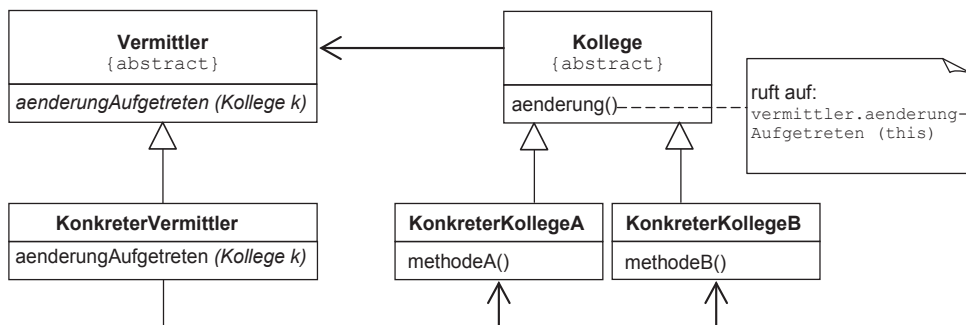


Bild 4-40 Klassendiagramm des Vermittler-Musters

Die Kommunikation aller konkreten Kollegen untereinander erfolgt nur zentral über den entsprechenden konkreten Vermittler.

Muss die Kommunikation zwischen den konkreten Kollegen abgeändert werden, weil beispielsweise Kollegen in eine andere Abteilung versetzt werden, muss nur der konkrete Vermittler geändert bzw. die Referenz eines Kollegen auf einen anderen konkreten Vermittler gesetzt werden. Ohne die Verwendung eines Vermittlers müsste man entweder die Kollegen-Klassen selbst abändern oder sie durch Unterklassenbildung an die neue Kommunikationsstruktur anpassen. Durch das Vermittler-Muster bleiben also die Kollegen-Klassen – bzw. deren Objekte – in unterschiedlichen Kommunikationsstrukturen unabhängig voneinander und damit wiederverwendbar.

#### 4.13.3.2 Teilnehmer

##### Vermittler

Die Klasse `Vermittler` ist abstrakt und definiert die Schnittstelle, über welche die Kollegen-Objekte Änderungen zur Weiterleitung an andere Kollegen einem Vermittler mitteilen können.

##### KonkreterVermittler

Ein Objekt der Klasse `KonkreterVermittler` realisiert einen Vermittler. Der konkrete Vermittler implementiert die Kommunikationsstruktur zu den konkreten Kollegen. So besitzt er beispielsweise eine Referenz auf alle Kollegen und ruft die entsprechenden Methoden der anderen gewünschten Kollegen auf, wenn er von einem der Kollegen benachrichtigt wird. Es kann mehrere Klassen für konkrete Vermittler geben, die an die jeweils unterschiedlichen Gegebenheiten der Kollegen-Objekte und deren Zusammenspiel angepasst sind.

##### Kollege

Die Klasse `Kollege` ist eine abstrakte Basisklasse oder auch ein Interface für alle konkreten Kollegen. Ein Kollege hält eine Referenz auf einen konkreten Vermittler. Er informiert den Vermittler, wenn bei ihm eine Änderung eingetreten ist. Ein Kollege arbeitet mit seinem Vermittler zusammen. Er spricht nicht direkt mit den anderen Kollegen, sondern nur indirekt über den Vermittler.

##### KonkreterKollegeX

Eine Klasse `KonkreterKollegeX` ( $X = A..Z$ ) leitet von der Klasse `Kollege` ab und definiert, wann eine Änderung eingetreten ist. Die Klasse verfügt über eine Methode `methodeX()`, die der Vermittler bei einer Aktualisierung aufruft, um einen Kollegen über eine erfolgte Änderung zu informieren.

#### 4.13.3.3 Dynamisches Verhalten

Bild 4-41 zeigt nun das dynamische Verhalten der Beteiligten am Vermittler-Muster:

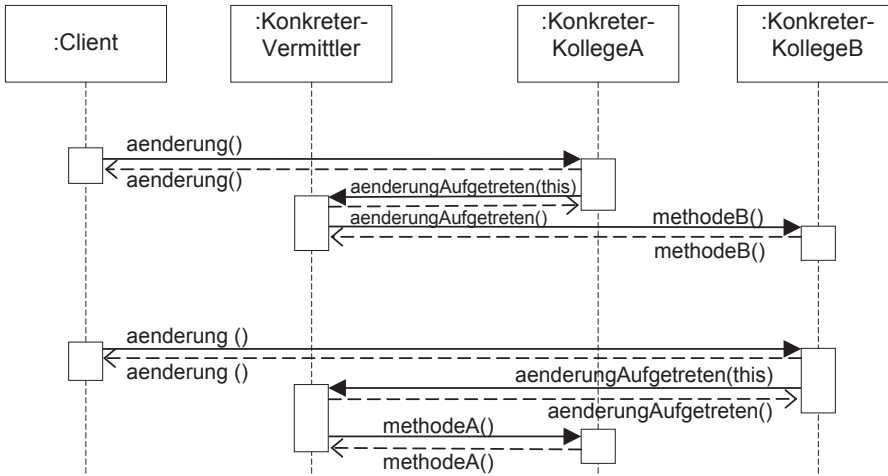


Bild 4-41 Sequenzdiagramm des Vermittler-Musters

In Bild 4-41 führt ein Client eine Veränderung am Objekt der Klasse `Konkreter-KollegeA` durch. Danach teilt dieses Objekt seinem konkreten Vermittler mit, dass eine Änderung eingetreten ist. Das Objekt der Klasse `KonkreterVermittler` nimmt diesen Methodenaufruf entgegen und informiert dann die weiteren Kollegen über die Veränderung, in diesem Falle das Objekt der Klasse `KonkreterKollegeB`.

Ferner wird im Sequenzdiagramm gezeigt, dass, wenn der Client das Objekt der Klasse `KonkreterKollegeB` ändert, dieses wiederum den Vermittler informiert und dieser dann die Methode `methodeA()` des Objekts der Klasse `KonkreterKollegeA` aufruft, sodass dieser Kollege Kenntnis über die Änderung erhält.

Es sei hier noch erwähnt, dass das Client-Objekt, das in Bild 4-41 dargestellt ist, nicht Bestandteil des Musters ist. Es steht hier stellvertretend für Objekte, die eine Änderung bei einem der Kollegen-Objekte bewirken und dadurch eine Benachrichtigung anderer Kollegen über den Vermittler auslösen.

### Weitergehende Lösungsvarianten

Die bisherige Beschreibung des Vermittler-Musters basierte auf einer sehr einfachen Form dieses Musters. Damit sollte das Prinzip des Musters zuerst einmal verständlich erklärt werden. Bei einer genaueren Betrachtung des Klassendiagramms in Bild 4-39 fällt aber ins Auge, dass Vermittler und Kollegen stark gekoppelt sind. Einmal gibt es eine Abhängigkeit auf der abstrakten Ebene und zum anderen gibt es eine umgekehrt gerichtete Abhängigkeit auf der konkreten Ebene. Hier ist der Vermittler sogar von jeder konkreten Kollegen-Klasse abhängig.

Weitere Punkte, die von der bisherigen Beschreibung nicht abgedeckt wurden, sind:

- Woher kennt der Vermittler die zu benachrichtigenden Kollegen?
- Woher weiß der Vermittler, welcher Kollege bei welchem Ereignis zu benachrichtigen ist?



- Warum müssen die Kollegen von einer gemeinsamen Basisklasse ableiten? Können die Objekte, die zu benachrichtigen sind, nicht von gänzlich unterschiedlichem Typ sein?

Diese Fragen müssen in der Regel anwendungsspezifisch beantwortet werden und sind nur schlecht durch eine allgemeine Herangehensweise im Rahmen des Vermittler-Musters zu lösen. In [Gr102] wird eine Lösung am Beispiel eines Vermittlers zwischen grafischen Objekten einer kleinen Anwendung gezeigt, bei der die oben genannten Punkte exemplarisch berücksichtigt sind.

#### 4.13.3.4 Programmbeispiel

Es handelt sich im Folgenden um ein sehr einfaches Beispiel: der konkrete Vermittler kennt nur den Nachrichtenaustausch zwischen zwei Objekten – einem Objekt der Klasse `KonkreterKollegeA` und einem Objekt der Klasse `KonkreterKollegeB`. Selbst an diesem einfachen Beispiel ist aber bereits der Effekt des Vermittler-Musters zusehen, nämlich dass die Klassen `KonkreterKollegeA` und `KonkreterKollegeB` nicht voneinander abhängig sind, obwohl ihre Objekte miteinander kommunizieren. Sie sind nur von dem Vermittler abhängig, über den sie miteinander kommunizieren. Das Beispiel orientiert sich an der einfachen Lösungsvariante im Bild 4-39.

Die abstrakte Klasse `Kollege` ist die Basisklasse für alle Kollegen:

```
// Datei: Kollege.java
public abstract class Kollege
{
    // Instanzvariable
    private Vermittler vermittler; // Referenz auf den Vermittler

    // Konstruktor
    public Kollege (Vermittler v)
    {
        vermittler = v;
    }

    // Wird von den ableitenden Klassen ueberschrieben
    public void aenderung()
    {
        vermittler.aenderungAufgetreten (this); // Vermittler informiert
    }
}
```

Die Klasse `KonkreterKollegeA` ist von der Klasse `Kollege` abgeleitet. Über die Methode `aenderung()` informiert sie einen Vermittler über Zustandsänderungen:

```
// Datei: KonkreterKollegeA.java
public class KonkreterKollegeA extends Kollege
{
    // Konstruktor
    public KonkreterKollegeA (Vermittler v)
    {
        super (v);
    }
}
```

```

        System.out.println ("KonkreterKollegeA: instanziiert");
    }

    // Wird aufgerufen, wenn sich ein anderer Kollege aendert
    public void methodeA()
    {
        System.out.println
            ("KonkreterKollegeA wird in methodeA() geaendert " +
             "als Folge der Aenderung eines Kollegen");
    }

    // Neuen Status setzen
    public void aenderung()
    {
        System.out.println
            ("KonkreterKollegeA wurde geaendert durch Aufruf" +
             " der Methode aenderung(). KonkreterKollegeA" +
             " informiert den Vermittler ");
        super.aenderung(); // informiert Vermittler
    }
}

```

Die Klasse `KonkreterKollegeB` ist ebenfalls von der Klasse `Kollege` abgeleitet und kann über die Methode `aenderung()` einen Vermittler informieren:

```

// Datei: KonkreterKollegeB.java
public class KonkreterKollegeB extends Kollege
{
    // Konstruktor
    public KonkreterKollegeB (Vermittler v)
    {
        super (v);
        System.out.println ("KonkreterKollegeB: instanziiert");
    }

    // Wird aufgerufen, wenn sich ein anderer Kollege aendert
    public void methodeB()
    {
        System.out.println
            ("KonkreterKollegeB wird in methodeB() geaendert" +
             " als Folge der Aenderung eines Kollegen");
    }

    // Neuen Status setzen
    public void aenderung()
    {
        System.out.println
            ("KonkreterKollegeB wurde geaendert durch Aufruf" +
             " der Methode aenderung(). KonkreterKollegeB" +
             " informiert den Vermittler ");
        super.aenderung(); // informiert Vermittler
    }
}

```

Die abstrakte Klasse `Vermittler` definiert die Schnittstelle, über die ein konkreter Vermittler über Änderungen von Kollegen informiert werden kann:

```
// Datei: Vermittler.java
public abstract class Vermittler
{
    // zur Information von Kollegen
    public abstract void aenderungAufgetreten(Kollege kollege);
}
```

Die Klasse `KonkreterVermittler` implementiert die abstrakten Methoden der Basisklasse `Vermittler`. Bei Änderung eines Kollegen informiert er den jeweils anderen Kollegen. Über `set`-Methoden erhält ein konkreter Vermittler Kenntnis über die beiden an der Kommunikation beteiligten Objekte. Hier nun der Quellcode der Klasse `KonkreterVermittler`:

```
// Datei: KonkreterVermittler.java
public class KonkreterVermittler extends Vermittler
{
    // Instanzvariablen
    private KonkreterKollegeA kollegeA;
    private KonkreterKollegeB kollegeB;

    // Konstruktor
    public KonkreterVermittler()
    {
        System.out.println("KonkreterVermittler: instanziiert");
    }

    // bei Aenderungen ruft der geaenderte Kollege diese
    // Vermittler-Methode auf
    public void aenderungAufgetreten (Kollege k)
    {
        if (k == (Kollege)kollegeA)
        {
            System.out.println
                ("KonkreterVermittler: informiere KollegeB");
            kollegeB.methodeB();
        }
        else if (k == (Kollege)kollegeB)
        {
            System.out.println
                ("KonkreterVermittler: informiere KollegeA");
            kollegeA.methodeA();
        }
    }

    //Set-Methoden fuer Kollegen
    public void setKollegeA (KonkreterKollegeA kka)
    {
        kollegeA = kka;
    }

    public void setKollegeB (KonkreterKollegeB kkb)
    {
        kollegeB = kkb;
    }
}
```

Das Client-Programm ist in zwei Phasen aufgeteilt. In der ersten Phase werden alle Objekte instanziiert. In der zweiten Phase führt der Client die Veränderungen an den Objekten der Klassen `KonkreterKollegeA` und `KonkreterKollegeB` durch:

```
// Datei: Client.java
public class Client
{
    public static void main (String[] args)
    {
        // Initialisierung
        System.out.println ("Initialisierung:");
        KonkreterVermittler konkreterVermittler =
            new KonkreterVermittler();
        KonkreterKollegeA kollegeA = new
            KonkreterKollegeA (konkreterVermittler);
        konkreterVermittler.setKollegeA (kollegeA);
        KonkreterKollegeB kollegeB = new
            KonkreterKollegeB (konkreterVermittler);
        konkreterVermittler.setKollegeB(kollegeB);

        // KollegeA aendern
        System.out.println ("\nKollegeA aendern:");
        kollegeA.aenderung();

        // KollegeB aendern
        System.out.println ("\nKollegeB aendern:");
        kollegeB.aenderung();
    }
}
```



Die Ausgabe ist:

```
Initialisierung:
KonkreterVermittler: instanziiert
KonkreterKollegeA: instanziiert
KonkreterKollegeB: instanziiert

KollegeA aendern:
KonkreterKollegeA wurde geaendert durch Aufruf der
Methode aenderung(). KonkreterKollegeA informiert den
Vermittler
KonkreterVermittler: informiere KollegeB
KonkreterKollegeB wird in methodeB() geaendert als
Folge der Aenderung eines Kollegen

KollegeB aendern:
KonkreterKollegeB wurde geaendert durch Aufruf der
Methode aenderung(). KonkreterKollegeB informiert den
Vermittler
KonkreterVermittler: informiere KollegeA
KonkreterKollegeA wird in methodeA() geaendert als
Folge der Aenderung eines Kollegen
```

Anzumerken ist, dass das gezeigte Beispiel untypisch ist, da es aus Platzgründen sehr vereinfacht ist. Die Kommunikation zwischen den beiden Kollegen ist hier im Vermittler hart codiert. Typischerweise muss der Vermittler eine Tabelle verwalten, in der die Kommunikationsverbindungen gespeichert sind, und die Verbindungen zwischen Objekten müssen tabellengesteuert durch Interpretation der Tabelle hergestellt werden.

#### **4.13.4 Bewertung**

##### **4.13.4.1 Vorteile**

Die folgenden Vorteile werden gesehen:

- Kollegen-Objekte sind untereinander lose gekoppelt und können wiederverwendet werden.
- Zwischen den Kollegen spannt sich ohne Vermittler eine n-zu-m-Beziehung auf. Der Vermittler reduziert dies jeweils auf eine 1-zu-n-Beziehung zwischen Objekten und Vermittler, was die Verständlichkeit, Verwaltung und Erweiterbarkeit verbessert.
- Die Steuerung der Kollegen-Objekte ist zentralisiert.
- Die Unterklassenbildung wird reduziert, da bei einer Änderung der Kommunikation zwischen Kollegen lediglich neue konkrete Vermittler erzeugt bzw. vorhandene Vermittler geändert werden müssen und keine neuen konkreten Kollegenklassen erzeugt werden müssen.
- Objekte können geändert werden, ohne den Kommunikationspartner anzupassen. (Rückwirkungsfreiheit auf Kommunikationspartner). Unter Umständen muss der Vermittler geändert werden. Damit sind Änderungen lokalisiert.

##### **4.13.4.2 Nachteile**

Die folgenden Nachteile werden festgestellt:

- Da der Vermittler die Kommunikation mit den Kollegen in sich kapselt, kann er unter Umständen sehr komplex werden.
- Der zentrale Vermittler ist fehleranfällig und bedarf fehlertoleranter Maßnahmen.
- Wenn sich die Kollegenschaft oder ihr Zusammenspiel ändert, muss der Vermittler angepasst werden.

#### **4.13.5 Einsatzgebiete**

Das Vermittler-Muster ist einzusetzen, wenn:

- Objekte zusammenarbeiten und miteinander kommunizieren, die Art und Weise der Zusammenarbeit und der Kommunikation aber flexibel änderbar sein soll, ohne dabei Unterklassen zu bilden (siehe folgendes Anwendungsbeispiel für grafische Oberflächen),
- ein Objekt aufgrund seiner engen Kopplung an andere Objekte schwer wiederzuverwenden ist oder
- Objekte ihre Kommunikationspartner nicht direkt kennen sollen.

### Anwendungsbeispiel: Grafische Oberflächen

Das Vermittler-Muster wird häufig bei grafischen Oberflächen eingesetzt, bei denen mehrere Elemente wie Eingabefelder, Drop-Down-Menüs und Buttons beispielsweise in einer Dialogbox zusammenspielen. Nach der Eingabe von Zeichen in einem Eingabefeld müssen etwa in einer Dialogbox Buttons aktiviert oder deaktiviert werden, in einer anderen Dialogbox dagegen kann es nötig sein, dass der eingegebene Text dahingehend geprüft werden muss, ob es sich bei der Eingabe um einen gültigen Datensatz handelt. Löst man das Problem dadurch, dass man das spezifische Verhalten eines Eingabefeldes in einer Unterklasse realisiert, entsteht eine ganze Reihe von Unterklassen, die nur noch selten wiederverwendbar sind.

Durch die Einschaltung eines Vermittlers wird die Lösung einfacher und die Klassen bleiben wiederverwendbar: Jede Dialogbox bekommt einen eigenen Vermittler, der – um im Beispiel zu bleiben – von einem Eingabefeld benachrichtigt wird, dass eine Texteingabe vorliegt. Nun entscheidet der Vermittler einer Dialogbox, welche anderen grafischen Elemente darüber informiert werden müssen.

Die grafischen Elemente aus diesem Beispiel entsprechen den Kollegen im Sinne des Vermittler-Musters. Sie brauchen nicht über Unterklassen an ein spezifisches Verhalten angepasst werden, sondern nur der Vermittler einer Dialogbox muss das jeweilige Verhalten realisieren. Das Beispiel stammt aus [Gam95] und wird dort ausführlich beschrieben.

### 4.13.6 Ähnliche Entwurfsmuster

Ein Broker ist umgangssprachlich auch ein Vermittler. Das **Broker-Muster** hat auch eine gewisse Ähnlichkeit mit dem Vermittler-Muster, dadurch, dass die Kommunikation zwischen Komponenten (Kollegen) über einen Broker bzw. Vermittler abläuft. Bei genauerer Betrachtung ergeben sich aber wesentliche Unterschiede:

- Beim Vermittler-Muster kommunizieren die Kollegen über den Vermittler, wobei der Vermittler auf Grund von bei ihm eingehenden Nachrichten betroffene Kollegen informiert. Alle Beteiligten befinden sich im selben System. Beim Broker-Muster spielt der Broker eine ähnliche Rolle wie ein Vermittler, der Broker leitet eine Nachricht nur an den einen gewünschten Empfänger weiter, jedoch können Clients, Server und Broker verteilt auf verschiedenen Rechnern ablaufen.
- Beim Architekturmuster Broker teilt eine Komponente dem Broker den Empfänger der Nachricht mit, der Broker ist für die Lokalisierung der Empfänger-Komponente im verteilten System, den Transport der Nachricht zum Empfänger und ggf. auch für den Rücktransport einer Antwort zuständig. Dieser letzte Aspekt fehlt beim Vermittler-Muster vollständig. Dies liegt auch daran, dass es beim Vermittler-Muster keine 1-zu-1-Zuordnung zwischen Sender und Empfänger wie bei der Anfrage eines Clients an einen Server beim Broker-Muster gibt, sondern eine 1-zu-n-Zuordnung. Außerdem erwartet ein Client in der Regel vom Server eine Antwort. Im Vermittler hingegen wird abgelegt, welche anderen Kollegen über eine Nachricht informiert werden sollen. Eine Antwort wird dabei von den benachrichtigten Kollegen nicht erwartet.

Sowohl über das Vermittler-Muster als auch über das **Beobachter-Muster** kann die Zusammenarbeit von Objekten gesteuert werden. Während beim Vermittler-Muster die Objekte (die Kollegen) gleichberechtigt sind und potentiell jedes Objekt mit jedem anderen über den Vermittler kommunizieren kann, haben die am Beobachter-Muster beteiligten Objekte bestimmte Rollen, die die Kommunikationsmöglichkeiten einschränken: nur beobachtbare Objekte können ihre Beobachter informieren und nicht umgekehrt. Das bedeutet, dass das Beobachter-Muster für einfachere Anwendungen besser geeignet ist. Würde man aber die komplexe Zusammenarbeit zwischen den Kollegen über das Beobachter-Muster realisieren, wäre jeder Kollege sowohl Beobachter als auch Beobachtbarer. Die daraus resultierende Kaskade von Benachrichtigungen wäre unüberschaubar und kaum nachzuvollziehen. Die Einschaltung eines Vermittlers "synchronisiert" in gewisser Weise auch die Zusammenarbeit. Denn ein benachrichtigter Kollege kann zwar sofort wieder den Vermittler anrufen, aber der Vermittler wird zuerst die alte Benachrichtigung noch komplett abarbeiten, bevor er sich dem neuen Anruf zuwendet.

## 4.14 Das Verhaltensmuster Zustand

### 4.14.1 Name/Alternative Namen

Zustand (engl. state).

### 4.14.2 Problem

Zustandsbehaftete Probleme werden oft in einem funktionsorientierten Ansatz durch die Programmierung einer Fallunterscheidung gelöst, die die aktuellen Zustände und das zugehörige Verhalten auflistet. Sollen dann neue Zustände hinzugefügt werden, muss die Fallunterscheidung erweitert werden, was schnell unübersichtlich und daher fehleranfällig wird.

Zustandsbehaftete Aufgaben sollen so gelöst werden, dass jeder Fall einer klassischen Bedingungsanweisung auf ein Objekt einer eigenen Klasse abgebildet wird.



### 4.14.3 Lösung

Eine Klasse, deren Objekte ein zustandsabhängiges Verhalten besitzen, wird im Zustandsmuster mit dem Klassennamen **Kontext** bezeichnet. Die Objekte dieser Klasse werden kurz Kontextobjekte genannt.

Für jeden möglichen Zustand eines Kontextobjekts gibt es in diesem Muster eine eigene Zustandsklasse, die das Verhalten im entsprechenden Zustand kapselt.

Im Entwurfsmuster Zustand hängt zur Kompilierzeit der Kontext nicht von den konkreten Zustandsklassen direkt ab, sondern nur von einem Interface bzw. von einer abstrakten Klasse, die das Kontextobjekt als Abstraktion vorgibt. Diese abstrakte Zustandschnittstelle wird von den konkreten Zustandsklassen implementiert.



Dadurch, dass – wie schon erwähnt – der Kontext die Zustandsschnittstelle vorgibt, wird eine **Dependency Inversion** erreicht, d. h. der Kontext hängt überhaupt nicht von der Ausprägung der Methoden des eingenommenen konkreten Zustands ab.

Objekte von Klassen, die die Zustandsschnittstelle implementieren, werden kurz Zustandsobjekte genannt. Zur Laufzeit tritt dann ein Zustandsobjekt bei Einhaltung der Verträge an die Stelle der Schnittstelle.



Die einzelnen **Zustände** werden in **eigene Klassen** ausgelagert, die die **gemeinsame Zustandsschnittstelle** implementieren.



Das Zustandsmuster lässt offen, welcher seiner Teilnehmer für die Bestimmung des Anfangszustandes verantwortlich ist.

Für eine Anwendung (**Client-Programm**) bietet ein Kontextobjekt verschiedene Operationen an, die je nach Zustand des Kontextobjektes ein anderes Verhalten zeigen sollen. Ein Kontextobjekt leitet den Aufruf einer solchen Operation an das **aktuelle Zustandsobjekt** weiter. Zu diesem Zweck hält die Kontextklasse eine Referenz auf das aktuelle Zustandsobjekt. Als Typ der Referenz wird das gemeinsame Interface (bzw. die gemeinsame abstrakte Basisklasse) der Zustandsklassen gewählt. Dies wird im Klassendiagramm in Bild 4-42 durch die Aggregationsbeziehung zwischen der Klasse `Kontext` und dem Interface `IZustand` ausgedrückt. Zur Laufzeit zeigt die Referenz auf das jeweils aktuelle Zustandsobjekt. Damit ändert sich das Verhalten des entsprechenden Kontextobjekts, da es die Anfragen des Client-Programms an das jeweils aktuelle Zustandsobjekt delegiert. Das Zustandsmuster dient zum dynamischen Austausch der Zustandsobjekte und wird daher zu den objektbasierten Entwurfsmustern gezählt.

Im Zustandsmuster werden die einzelnen Zustände durch eigene Klassen gekapselt, die von einer gemeinsamen abstrakten Basis-Klasse ableiten bzw. ein gemeinsames Interface implementieren. Ein zustandsabhängiges Kontextobjekt referenziert den aktuellen Zustand und führt Zustandsänderungen durch.



Das Zustandsmuster lässt offen, welcher seiner Teilnehmer für die Zustandsübergänge verantwortlich ist. In der **Grundform dieses Musters** ist das Kontextobjekt für die Zustandswechsel zuständig, da in ihm der Zustand gespeichert ist. Häufig ist die Bestimmung des Nachfolgezustandes aber auch eine zustandsabhängige Operation und sollte daher vom aktuellen Zustandsobjekt durchgeführt werden. Diese Alternativen werden in Kapitel 4.14.3.4 ausführlich diskutiert. Im Folgenden wird die Grundform des Musters vorgestellt.

Die **Anwendung** selbst kennt die Zustandsklassen und deren Objekte nicht, sondern arbeitet nur mit dem Kontextobjekt, das die Übergänge durchführt, zusammen. Zustandsabhängiges Verhalten wird in der Regel mittels **Zustandsautomaten** (engl. **state machines**) beschrieben. Diese können in UML in Form von Zustandsdiagrammen (siehe Beispiel in Kapitel 4.14.3.5) modelliert werden. In diesem Zusammenhang entspricht ein Kontextobjekt der Realisierung eines Zustandsautomaten.

#### 4.14.3.1 Klassendiagramm

Objekte der Klasse `Kontext` zeigen ein zustandsabhängiges Verhalten. Das bedeutet, dass es in dieser Klasse Methoden gibt, deren Wirkung unterschiedlich ist, je nach-

dem in welchem aktuellen Zustand sich ein Kontextobjekt befindet. Im Folgenden steht die Methode `operation()` beispielhaft für die zustandsabhängigen Methoden der Klasse `Kontext`.

Ein Kontextobjekt speichert seinen aktuellen Zustand, indem es ein Objekt einer **konkreten Zustandsklasse** aggregiert. Alle Zustandsklassen implementieren dabei dasselbe Interface bzw. sind von einer gemeinsamen **abstrakten Basisklasse** abgeleitet. Diese Struktur ist in folgendem Bild dargestellt:

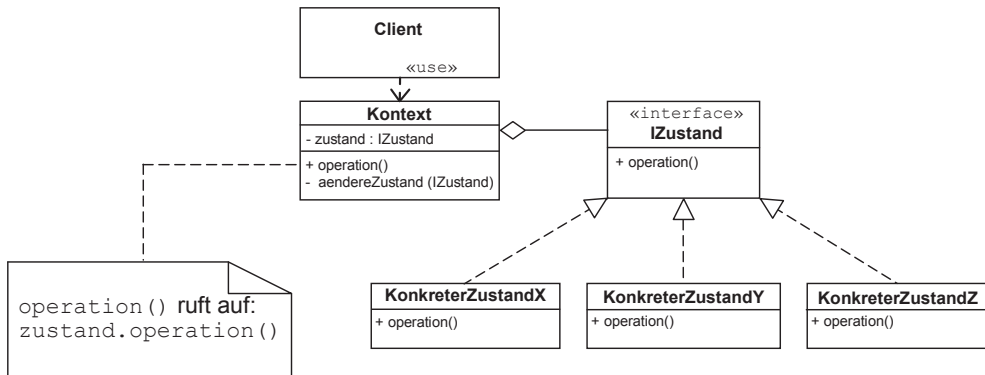


Bild 4-42 Klassendiagramm des Entwurfsmusters Zustand

Jede **konkrete Zustandsklasse** ist verantwortlich für das Verhalten eines Kontextobjekts in diesem Zustand. Hierzu realisiert die konkrete Zustandsklasse alle zustandsabhängigen Operationen, die in dem gemeinsamen Interface **IZustand** definiert sind. Wird nun die zustandsabhängige Operation `operation()` des Kontextobjekts aufgerufen, so delegiert dieses den Aufruf an die entsprechende Methode des aktuell referenzierten Zustandsobjektes.

Ein Client kann nicht direkt Zustandsänderungen durchführen, diese Aufgabe obliegt allein dem Kontextobjekt. Dazu dient die private Methode `aendereZustand()`. Das bedeutet, dass ein Client in diesem Muster unabhängig von dem Interface **IZustand** und von den konkreten Zustandsklassen ist.

#### 4.14.3.2 Teilnehmer

##### Client

Ein **Client** ruft die öffentlichen Methoden des Kontextobjekts auf. Er kennt dabei das Interface **IZustand** nicht und auch keine Klassen, die dieses realisieren.

##### Kontext

Objekte der Klasse **Kontext** können ihr Verhalten abhängig von ihrem internen Zustand ändern. Der interne Zustand wird dabei über eine Referenz auf ein Zustandsobjekt, das die Schnittstelle **IZustand** implementieren muss, gespeichert. Bei einem Zustandsübergang zeigt die Referenz auf ein neues Zustandsobjekt.

### IZustand

Das Interface `IZustand` definiert eine einheitliche Schnittstelle aller Zustandsklassen mit allen zustandsbehafteten Operationen.

### KonkreterZustandX

Eine Klasse `KonkreterZustandX` ( $X = A..Z$ ) implementiert das Verhalten, das mit einem einzigen Zustand eines Kontextobjekts verknüpft ist.

#### 4.14.3.3 Dynamisches Verhalten

Das dynamische Verhalten des Zustandsmusters soll in Bild 4-43 anhand von zwei Zustandsobjekten (`x:KonkreterZustandX` und `y:KonkreterZustandY`) vorgestellt werden:

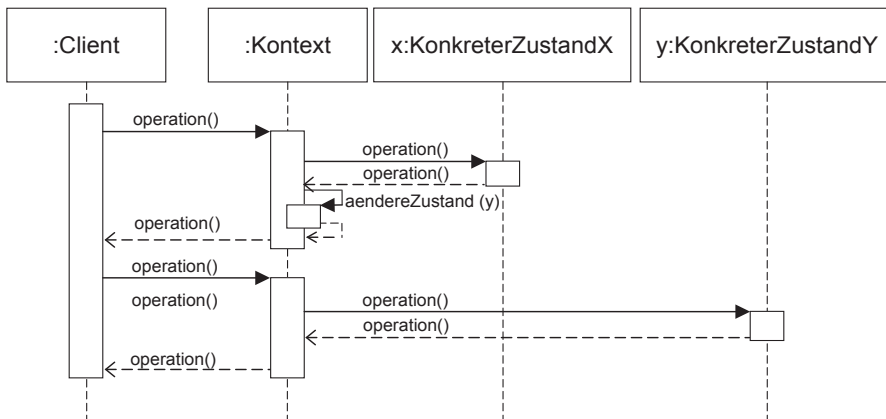


Bild 4-43 Sequenzdiagramm des Entwurfsmusters Zustand

Vom Client wird eine zustandsbehaftete Operation `operation()` des Kontextobjekts aufgerufen. Die Methode `operation()` der Klasse `Kontext` steht hier stellvertretend für solche zustandsbehaftete Operationen. Der Aufruf der Methode `operation()` wird an das aktuelle Zustandsobjekt weiterdelegiert (in diesem Fall `x:KonkreterZustandX`). Als Konsequenz ändert das Kontextobjekt seinen Zustand über einen Aufruf von `aendereZustand(y)`. Damit ist die Operation beendet und die Kontrolle geht zum Client zurück.

Setzt der Client später einen weiteren Aufruf der Methode `operation()` ab, wird dieser an das jetzt aktuelle Zustandsobjekt `y:KonkreterZustandY` delegiert. Damit ergibt sich ein anderes Verhalten beim Aufruf von `operation()`. Das Kontextobjekt kann nun wieder entscheiden, ob ein Zustandswechsel nötig ist, bevor die Kontrolle wieder zum Client wechselt.

#### 4.14.3.4 Implementierungsalternativen

In der bisher beschriebenen Grundform des Zustandsmusters obliegt bei einem Zustandswechsel die Verantwortung für das Bestimmen des Nachfolgezustands dem Kontextobjekt.

Bei vielen Zustandsautomaten ist aber die Bestimmung des Nachfolgezustands nicht nur abhängig von einem Ereignis, sondern auch vom aktuellen Zustand, in dem sich der Automat gerade befindet. In diesem Falle muss die Verantwortlichkeit für die Bestimmung des Nachfolgezustands dem aktuellen Zustandsobjekt übertragen werden. Denn dieses soll ja das gesamte Verhalten des Automaten in diesem Zustand kapseln.

Dafür gibt es zwei Möglichkeiten:

1. Jede Methode einer konkreten Zustandsklasse, die vom Kontextobjekt aufgerufen wird, um ein zustandsabhängiges Verhalten zu realisieren, wie beispielsweise die Methode `operation()`, liefert ein Zustandsobjekt als Ergebnis zurück, das vom Kontextobjekt für den Zustandswechsel benutzt wird.
2. Die Methode `aendereZustand()` der Kontextklasse wird öffentlich gemacht und die Methode `operation()` erhält einen Parameter vom Typ der Klasse `Kontext`. Ruft ein Kontextobjekt die Methode `operation()` auf, übergibt es über diesen Parameter eine Referenz auf sich selbst an das Zustandsobjekt. Auf diese Weise kennt ein Zustandsobjekt seinen Automaten – also das entsprechende Kontextobjekt – dessen Methode `aendereZustand()` es nun aufrufen kann, um den Nachfolgezustand zu setzen.

In beiden Fällen entstehen zusätzliche Abhängigkeiten: Ein konkreter Zustand muss die anderen Zustände kennen, um den Nachfolgezustand zu bestimmen. Im zweiten Fall muss ein konkreter Zustand zusätzlich noch das Kontextobjekt kennen.

Damit konkrete Zustände die anderen Zustände kennen, gibt es mehrere Implementierungsalternativen:

- Gibt es in einer Applikation nur einen einzigen Automaten einer bestimmten Klasse, dann kann die entsprechende Klasse als **Singleton-Klasse** (siehe Kapitel 4.20) realisiert werden. Die Zustandsobjekte können sich eine Referenz auf das Singleton-Objekt besorgen.
- Kennen die Zustände ihren Automaten, kann der Automat auch alle seine möglichen Zustände über get-Methoden (wie beispielsweise `getKonkreterZustandX()`) zur Verfügung stellen.
- Als letzte Möglichkeit wäre noch die Implementierung der konkreten Zustände in Form einer `enum`-Klasse zu nennen.

Diese Vorgehensweisen haben den weiteren Vorteil, dass zur Bestimmung des Nachfolgezustands nicht jedes Mal ein neues konkretes Zustandsobjekt erzeugt wird, sondern bereits existierende Zustandsobjekte wiederverwendet werden.

#### 4.14.3.5 Programmbeispiel

Als Beispiel soll hier die Alarmanlage einer Bank beschrieben werden. Befindet sich die Alarmanlage im Zustand `AlarmanlageAktiv` und wird eine Person erkannt, so wird ein akustischer Alarm als Aktion ausgegeben. Befindet sich die Anlage hingegen im Zustand `AlarmanlageInaktiv` (während der Geschäftszeit der Bank), so soll

kein akustisches Signal ausgegeben werden. Bild 4-44 zeigt das Zustandsdiagramm nach UML für dieses Beispiel:

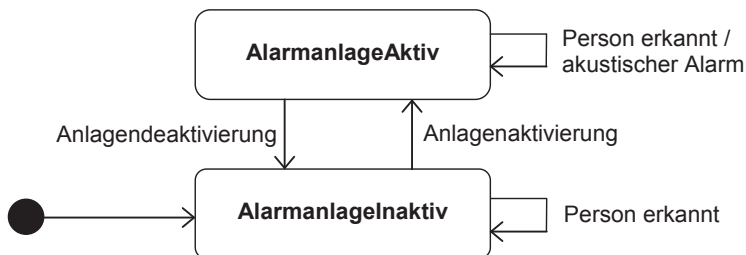


Bild 4-44 Zustandsdiagramm für die Alarmanlage

Es sind die beiden Zustände zu sehen, sowie die möglichen Übergänge zwischen ihnen. Außerdem wird dargestellt, dass das Ereignis `personErkannt` in dem einen Zustand eine Aktion auslöst, nämlich einen akustischen Alarm, während im anderen Zustand keine Aktion erfolgt. In beiden Fällen findet aber bei diesem Ereignis kein Zustandsübergang statt, sondern die Anlage verbleibt im aktuellen Zustand. Weiterhin wird gezeigt, dass als Startzustand der Zustand `AlarmanlageInaktiv` ausgewählt wird.

Bild 4-45 zeigt das Klassendiagramm dieses Programmbeispiels:

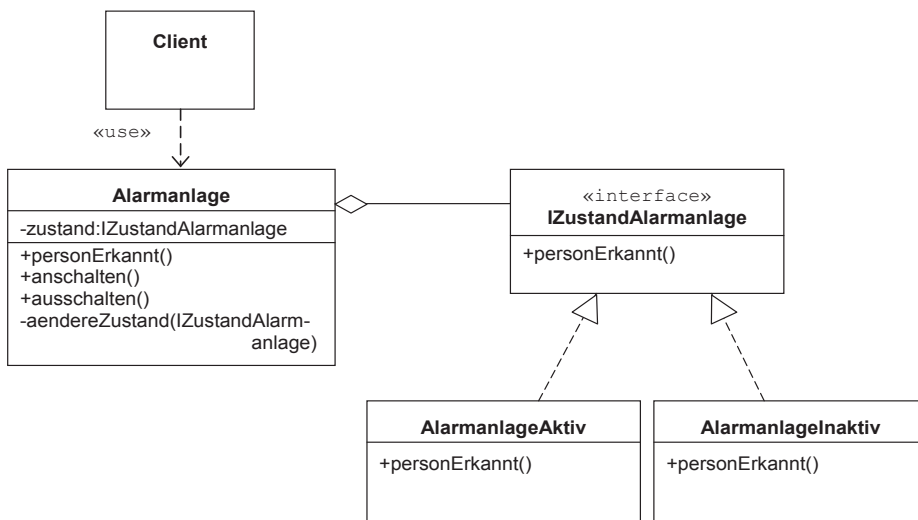


Bild 4-45 Klassendiagramm zum Programmbeispiel Alarmanlage

Die Schnittstelle `IZustandAlarmanlage` deklariert die Methoden aller Zustandsklassen:

```
// Datei: IZustandAlarmanlage.java
// Schnittstelle fuer alle Zustandsklassen
public interface IZustandAlarmanlage
{

```

```

    public void personErkannt();
}

```

Befindet sich eine Alarmanlage im konkreten Zustand `AlarmanlageAktiv`, so soll beim Erkennen einer Person (Aufruf der Methode `personErkannt()`) ein akustischer Alarm generiert werden:

```

// Datei: AlarmanlageAktiv.java
// Beschreibt das Verhalten der Alarmanlage im Zustand
// AlarmanlageAktiv
public class AlarmanlageAktiv implements IZustandAlarmanlage
{
    // Sofern eine Person erkannt wurde, ein akustisches Signal
    // ausgeben.
    public void personErkannt()
    {
        System.out.println ("RING RING");
    }
}

```

Im konkreten Zustand `AlarmanlageInaktiv` hingegen soll eine Alarmanlage keinen akustischen Alarm generieren:

```

// Datei: AlarmanlageInaktiv.java
// Beschreibt das Verhalten der Alarmanlage im Zustand
// AlarmanlageInaktiv
public class AlarmanlageInaktiv implements IZustandAlarmanlage
{
    // Sofern eine Person erkannt wurde, KEIN akustisches Signal
    // ausgeben, da dies im normalen Geschaeftsbetrieb nur stoerend
    // waere.
    public void personErkannt()
    {
        System.out.println ("Ruhig bleiben.");
    }
}

```

Die Klasse `Alarmanlage` definiert, wie die Alarmanlage von außen aufgerufen werden kann. Sie entspricht der Kontextklasse. Wird die Methode `personErkannt()` der Klasse `Alarmanlage` aufgerufen, so wird der Aufruf an den aktuellen konkreten Zustand weitergeleitet. Mit Hilfe der Methoden `anschalten()` und `ausschalten()` wird eine Zustandsänderung durchgeführt. Wie im Zustandsdiagramm in Bild 4-44 zu sehen ist, muss beim Erkennen einer Person kein Zustandswechsel erfolgen. Nach dem Zustandsmuster wird aber die Reaktion auf dieses Ereignis an das aktuelle Zustandsobjekt delegiert. Hier die Klasse `Alarmanlage`:

```

// Datei: Alarmanlage.java
// Kontext, ueber den die Alarmanlage gesteuert wird
public class Alarmanlage
{
    IZustandAlarmanlage aktiv = new AlarmanlageAktiv();

    IZustandAlarmanlage inaktiv = new AlarmanlageInaktiv();
    IZustandAlarmanlage zustand = null;
}

```

```
public Alarmanlage()
{
    zustand = inaktiv; // Startzustand
}

public void anschalten()
{
    aendereZustand(aktiv);
}

public void ausschalten()
{
    aendereZustand(inaktiv);
}

public void personErkannt()
{
    zustand.personErkannt();
}

private void aendereZustand (IZustandAlarmanlage neuerZustand)
{
    zustand = neuerZustand;
}
}
```

Im folgenden Beispielprogramm wird ein Objekt der Klasse `Alarmanlage` durch Aufruf der Methode `ausschalten()` zuerst in den Zustand `AlarmanlageInaktiv` versetzt und die Methode `personErkannt()` aufgerufen. Danach wird ein Zustandswechsel in den Zustand `AlarmanlageAktiv` durchgeführt und die Methode `personErkannt()` erneut aufgerufen:

**// Datei: Client.java**

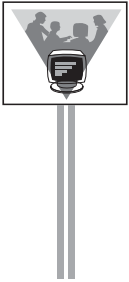
```
public class Client
{
    public static void main (String[] args)
    {
        Alarmanlage a = new Alarmanlage();

        System.out.println ("Anlage deaktivieren...");
        System.out.println ("Bei Kundentrieb stoert das.");
        a.ausschalten();

        System.out.println ("Person erkannt.");
        a.personErkannt();

        System.out.println ("Feierabend.");
        System.out.println ("Aktivierung der Alarmanlage.");
        a.anschalten();

        System.out.println ("Person erkannt.");
        a.personErkannt();
    }
}
```



Hier das Protokoll des Programmlaufs:

```
Anlage deaktivieren...
Bei Kundentrieb stoert das.
Person erkannt.
Ruhig bleiben.
Feierabend.
Aktivierung der Alarmanlage.
Person erkannt.
RING RING
```

Befindet sich die Alarmanlage im Zustand `AlarmanlageInaktiv`, so wird kein Alarm ausgegeben. Im Zustand `AlarmanlageAktiv` hingegen wird ein akustischer Alarm ("RING RING") ausgelöst.

Die Alarmanlage in diesem Beispiel ist relativ einfach, da die Bestimmung der Nachfolgezustände statisch erfolgt und von keinen Bedingungen abhängt. Daher konnte für die Realisierung der Zustandsübergänge die in Kapitel 4.14.3 beschriebene Grundform gewählt werden, in der das Kontextobjekt den Nachfolgezustand selbst bestimmt.

#### 4.14.4 Einsatzgebiete

Ein jedes zustandsabhängiges Verhalten kann durch dieses Entwurfsmuster beschrieben werden. Beispiele für zustandsbehaftete Probleme sind:

- Bedienelemente mit Zuständen einer grafischen Benutzeroberfläche und
- Zustände von parallelen Einheiten (Prozesssteuerung).

#### 4.14.5 Bewertung

##### 4.14.5.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Zustände werden in Form von Klassen realisiert. Das gesamte Verhalten für einen Zustand ist in einer einzigen Klasse, einer konkreten Zustandsklasse, konzentriert. Ein Kontextobjekt hat immer einen bestimmten Zustand und enthält damit immer ein Objekt einer solchen Zustandsklasse. Das Verhalten in einem Zustand ist gekapselt, was die Übersichtlichkeit erhöht.
- Die Erweiterbarkeit ist gegeben. Ein neuer Zustand entspricht einer neuen Klasse. Alles was getan werden muss, ist, auf jeden Fall eine neue Klasse für diesen Zustand zu implementieren. Je nach gewählter Implementierungsalternative sind keine oder nur geringe Änderungen an der Kontextklasse nötig.
- Die langwierigen Fallunterscheidungen, wie sie beim erwähnten funktionsorientierten Ansatz verwendet wurden, entfallen vollständig.
- Durch dieses Muster wird es möglich, Zustandsklassen eventuell auch in einem anderen Kontext wiederzuverwenden.



#### 4.14.5.2 Nachteile

Der folgende Nachteil wird festgestellt:

- Der Implementierungsaufwand kann bei einem einfachen zustandsbasierten Verhalten zu hoch gegenüber dem Nutzen sein, da viele Klassen erstellt und Objekte zur Laufzeit erzeugt werden müssen.

#### 4.14.6 Ähnliche Entwurfsmuster

Das Klassendiagramm des Zustandsmusters ist gleich aufgebaut wie das Klassendiagramm des **Strategie-Musters**. Der Unterschied liegt aber in der Verwendung der Muster. Während beim Zustandsmuster das Kontextobjekt die Anfrage an ein Zustandsobjekt weiterleitet, und damit ein zustandsabhängiges Verhalten erzeugt, erfolgt bei dem Strategie-Muster die Weiterleitung an einen speziellen Algorithmus, wobei alle Algorithmen dasselbe Verhalten zeigen, aber unterschiedlich implementiert sind.

## 4.15 Das Verhaltensmuster Rolle

### 4.15.1 Name/Alternative Namen

Rolle (engl. role oder auch rôle). Das hier vorgestellte Entwurfsmuster Rolle ist in der Literatur auch als **Role Object Pattern** (siehe etwa [Bäu97]) bekannt.

### 4.15.2 Problem

Objekte einer Software stehen in der Regel nicht alleine da. Sie wirken vielmehr mit anderen Objekten zusammen, indem sie dabei Rollen annehmen. Aus [Bäu97] ist die folgende Definition entnommen:

Eine **Rolle** ist ein wahrnehmbarer Verhaltensaspekt eines Objekts.



Weist ein Objekt zur Laufzeit verschiedene wahrnehmbare Verhaltensaspekte auf, so spielt es zur Laufzeit **dynamisch verschiedene Rollen**.

Der Name der aktuellen Rolle, die ein Objekt in einer Kollaboration spielt, kann formal in einem Objektdiagramm nach UML als Zustand hinter dem Objektnamen angegeben werden, wie in folgendem Bild gezeigt wird:

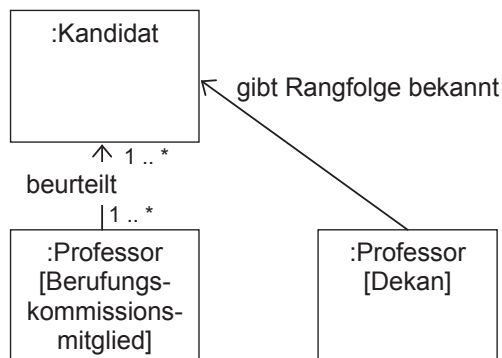


Bild 4-46 Rollen in einem Objektdiagramm

In diesem Beispiel können Objekte der Klasse `Professor` Mitglieder einer Berufungskommission sein. Jeder Professor als Mitglied einer Berufungskommission beurteilt die verschiedenen Kandidaten, die sich um eine neue Professorenstelle beworben haben. Hat die Berufungskommission ihre Entscheidung gefällt, gibt dasjenige Objekt der Klasse `Professor`, das die Rolle des Dekans der entsprechenden Fakultät spielt, die Rangfolge den Kandidaten bekannt. Objekte der Klasse `Professor` können also verschiedene Rollen spielen: nicht alle Professoren sind Mitglieder einer Berufungskommission sondern nur einige. In diesem Beispiel gibt es auch ein Objekt, das mehrere Rollen gleichzeitig spielt: Ein Professor als Dekan ist üblicherweise auch Mitglied einer

Berufungskommission. Darüberhinaus können die Rollen wechseln wie beispielsweise bei der Besetzung einer neuen Berufungskommission oder bei der Neuwahl des Dekans.

Aus diesem Beispiel heraus ergibt sich die folgende Anforderung an das Rollenmuster:

Das **Rollenmuster** soll es erlauben, dass ein Objekt dynamisch seine Rollen wechseln und mehrere Rollen gleichzeitig annehmen kann und dass mehrere Objekte die gleiche Rolle haben können.



### 4.15.3 Lösung

Soll ein Objekt in Gestalt mehrerer Rollen auftreten, wird dafür eine neue logische Ebene zwischen der Ebene der Klasse und der Ebene ihrer Instanzen benötigt, um die verschiedenen Verhaltensaspekte ein und desselben Objekts modellieren zu können.

Eine Klasse als statischer Typ beschreibt nur eine einzige statische Rolle, die für alle Instanzen der Klasse identisch ist. Rollen sind aber dynamisch und können von Objekten zu jeder Zeit angenommen und abgelegt werden. Rollen müssen daher mit Hilfe von eigenständigen Klassen bzw. Objekten realisiert werden.



Zur besseren Unterscheidung wird im Folgenden diejenige Klasse, deren Objekte zusätzliche Rollen annehmen können sollen, als **Kernklasse** bezeichnet. Die Klassen, deren Objekte die zusätzlichen Rollen für die Objekte der Kernklasse darstellen sollen, werden als **Rollenklassen** bezeichnet. Die erwähnte statische Rolle, die ein Objekt bereits auf Grund seiner Zugehörigkeit zur Kernklasse spielt, wird bei diesem Muster nicht betrachtet, sondern es geht hierbei nur um die zusätzlichen Rollen zur Rolle der Kernklasse.

Zur Darstellung und zur Diskussion verschiedener Lösungen wird das folgende Beispiel verwendet: Die Klasse `Mitarbeiter` sei die Kernklasse, die Klassen `Verkauf`, `Entwicklung`, `Controlling` seien Rollenklassen und `maier`, `mueller`, `schulze` seien die Namen von Instanzen der Kernklasse.

#### Lösungsansätze ohne Rollenmuster

Die einfachste Möglichkeit, um verschiedene Rollen eines Objekts zu realisieren, besteht darin, **alle Rollenfunktionen in die Kernklasse zu integrieren**. Im Beispiel müsste die Klasse `Mitarbeiter` um die Funktionen der Rollen `Verkauf`, `Entwicklung` und `Controlling` statisch erweitert werden. Ob ein Objekt eine bestimmte Rolle spielt, könnte beispielsweise über eine boolesche Variable dargestellt werden. Sind verschiedene booleschen Variablen gleichzeitig gesetzt, würde das bedeuten, dass das Objekt die entsprechenden Rollen gleichzeitig spielt. Ab einer gewissen Anzahl

von Rollen wäre aber eine solche Klasse auf Grund der Kombinationsmöglichkeiten bald zu komplex und zu unübersichtlich. Außerdem ist es für eine Anwendung nicht ohne weiteres ersichtlich, welche Rollen ein Objekt gerade einnimmt und welche Methoden bei dem Objekt auf Grund seiner Rollen sinnvoll aufgerufen werden können.

Eine weitere und innerhalb der Objektorientierung am häufigsten verwendete Technik zur Modellierung von Rollen ist die **statische Vererbung (Spezialisierung durch Ableitung)**. So würden in diesem Fall die Rollen `Verkauf`, `Entwicklung` und `Controlling` jeweils statisch von der Kernklasse `Mitarbeiter` erben. Solange die Instanzen jeweils nur eine einzige Rolle statisch übernehmen, wäre die Vererbung eine ausreichende Lösung. Sollte jedoch – beispielsweise in einer kleinen Firma – ein Mitarbeiter gleichzeitig mit Verkauf und Controlling beschäftigt sein, würde diese einfache Lösung nur bei einer Sprache, die Mehrfachvererbung unterstützt funktionieren – also nicht in Java. Der Lösungsansatz ist aber auch mit Mehrfachvererbung problematisch, da für jede mögliche Rollenkombination eine Klasse definiert werden muss. Ferner wäre es für einen Mitarbeiter nicht möglich, seine Rolle dynamisch zu wechseln.

Das dynamische Wechseln einer Rolle ist sehr schwierig zu realisieren. In den meisten objektorientierten Programmiersprachen<sup>61</sup> kann ein Objekt nicht mehr seine Klasse wechseln. Es sind implizit nur Typumwandlungen in die Oberklassen möglich, von denen die Klasse eines Objekts erbt (sogenannte **Up-Casts**). Bei einem Up-Cast gehen keine Informationen verloren und das Grundgerüst eines Objekts bleibt identisch – ein Up-Cast stellt nur eine eingeschränkte Sicht auf ein Objekt dar. Somit müsste für einen Rollenwechsel ein neues Objekt erzeugt werden, das alle relevanten Informationen von dem ursprünglichen Objekt kopiert. Alle im System vorhandenen Referenzen müssten so aktualisiert werden, dass sie auf das neue Objekt verweisen. Dies ist für eine praktikable Lösung zu umständlich.

Einen Lösungsansatz, der einen Rollenwechsel erlaubt, bietet das Verhaltensmuster **Zustand** (siehe Kapitel 4.14). Jedes Objekt der Klasse `Mitarbeiter` erhält – analog zu Bild 4-46 – jeweils einen Zustand, der beschreibt, welche Rolle von diesem Mitarbeiter gerade ausgeführt wird. Zur Laufzeit kann dann zwischen den einzelnen Zuständen – oder besser gesagt: zwischen den einzelnen Rollen – gewechselt werden. Allerdings ist es bei dieser Lösung für ein Objekt nicht so ohne weiteres möglich, verschiedene Rollen gleichzeitig anzunehmen. Ebenfalls nachteilig ist, dass alle Rollen die gleiche Schnittstelle haben müssten.

### Lösungsansatz des Rollenmusters

Das Entwurfsmuster Rolle basiert auf einem ähnlichen Grundgedanken wie das Entwurfsmuster Zustand: Analog zum Zustandsmuster, wo jeder Zustand in einem eigenen Objekt gekapselt wird, wird im Rollenmuster jede Rolle durch ein eigenes Objekt repräsentiert.

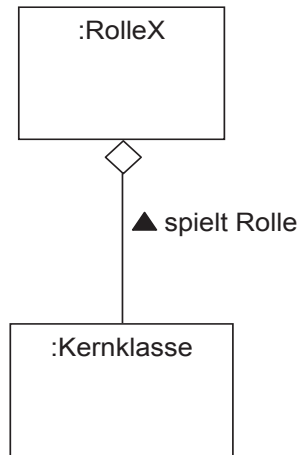
---

<sup>61</sup> Eine Ausnahme bilden typenlose Sprachen wie JavaScript.

Im Entwurfsmuster **Rolle** enthält eine Kernklasse die rollenunabhängige, d. h. statisch unveränderliche, Funktionalität eines Objekts. Die Rollen, die ein solches Objekt dynamisch spielen kann, werden in Rollenklassen bzw. Rollenobjekten implementiert. Ein Rollenobjekt aggregiert das Kernobjekt, welches diese Rolle gerade spielt.



Demnach besitzt jedes Rollenobjekt eine Referenz, die auf das Kernobjekt zeigt, das gerade diese Rolle spielt, wie das folgende Bild zeigt:



*Bild 4-47 Objektdiagramm des Rollen-Musters*

Wird eine Rolle für ein Objekt einer Kernklasse benötigt, so wird ein Objekt der Klasse `RolleX` erzeugt, welches das entsprechende Objekt der Kernklasse aggregiert. Soll ein Kernobjekt mehrere Rollen gleichzeitig annehmen, so stellt dies kein Hindernis dar. Die Objekte der verschiedenen Rollen aggregieren einfach alle dasselbe Kernobjekt.

Soll dieselbe Rolle von einem anderen Kernobjekt übernommen werden, muss nur die Referenz der Aggregation geändert werden, so dass sie auf das andere Kernobjekt zeigt. Soll ein Kernobjekt eine andere Rolle annehmen, dann muss ein – gegebenenfalls neu erzeugtes – entsprechendes Rollenobjekt dieses Kernobjekt referenzieren. Je nach Anwendungssituation kann das alte Rollenobjekt dann ganz gelöscht werden oder es wird nur dessen Referenz geändert und beispielsweise auf `null` gesetzt. Damit erfüllt dieser Lösungsansatz alle in Kapitel 4.15.2 gestellten Anforderungen.

Die folgenden Bilder zeigen in Form von Objektdiagrammen die Lösung durch das Rollenmuster. Zunächst werden verschiedene Mitarbeiter in derselben Rolle gezeigt:

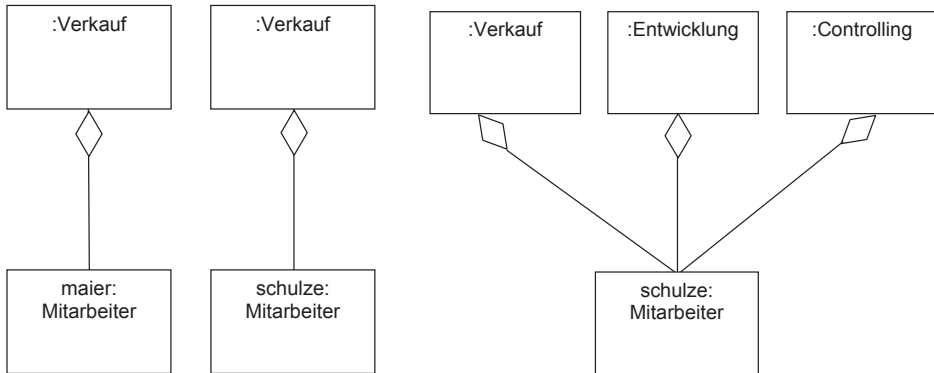


Bild 4-48 Mitarbeiter haben die gleiche Rolle    Bild 4-49 Ein Mitarbeiter hat mehrere Rollen

Das folgende Bild zeigt einen Mitarbeiter, der die Rolle wechselt:

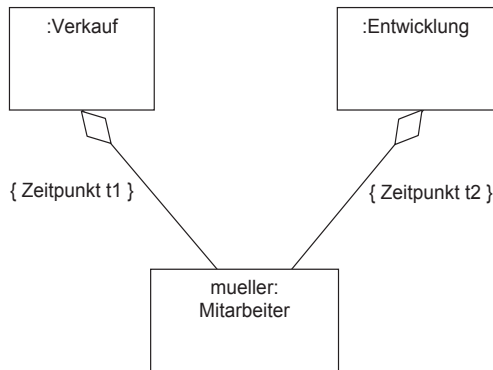


Bild 4-50 Ein Mitarbeiter hat zu verschiedenen Zeitpunkten  $t_1$  und  $t_2$  jeweils eine andere Rolle

Aus der bisherigen Lösungsbeschreibung wird klar, dass das Rollenmuster ein objekt-basiertes Muster ist.

Eine Anwendung interessiert sich in der Regel sowohl für die Datenfelder und Methoden des Kernobjekts selber als auch für die der Rolle(n), welche dieses gerade spielt. Beispielsweise möchte eine Anwendung sowohl den Namen eines Verkäufers wissen (Datenfeld des Kernobjekts) als auch dessen Umsatz im letzten Monat (Datenfeld des Rollenobjekts). Eigentlich sind dies Eigenschaften ein und desselben Objekts, die aber durch das Rollenmuster separiert wurden, damit das dynamische Wechseln von Rollen möglich wird. Um einer Anwendung den Zugriff sowohl auf die Datenfelder und Methoden des Rollenobjekts als auch auf die des Kernobjekts zu ermöglichen, bietet sich für den Entwurf der Rollenklassen folgende Lösung an:

- Eine Rollenklasse stellt eine Methode wie etwa `getKernObjekt()` zur Verfügung, die das gerade aggregierte Kernobjekt zurückgibt. Damit kann eine Anwendung dann auch Methoden des Kernobjekts aufrufen.

Diese Lösung ist einfach zu implementieren und wird als **Grundform des Entwurfsmusters** Rolle angesehen.

#### 4.15.3.1 Klassendiagramm

Die Klassen der verschiedenen Rollen werden mit `RolleX` ( $X = A..Z$ ) bezeichnet. Jede dieser Klassen enthält rollenspezifische Methoden, für die im folgenden Klassendiagramm exemplarisch in jeder Rollenklasse die Methode `operationX()` steht. Wie bereits erwähnt wurde, hat jede dieser Klassen eine Referenz auf ein Objekt der Klasse `KernKlasse` und bietet eine Methode `getKernObjekt()` an, mit der das aktuell referenzierte Kernobjekt abgefragt werden kann. Bild 4-51 zeigt das Klassendiagramm des Rollenmusters:

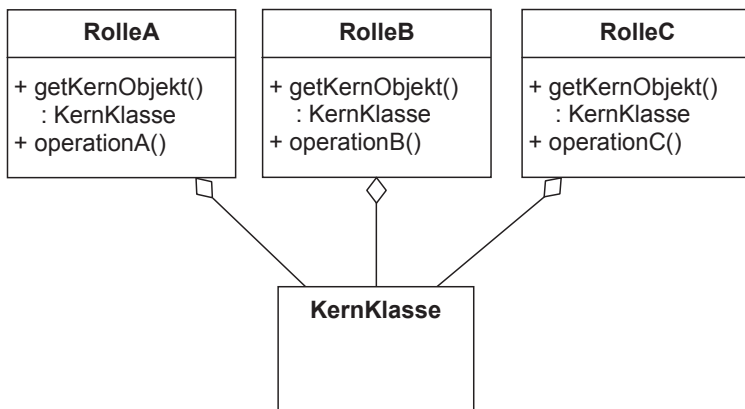


Bild 4-51 Klassendiagramm mit mehreren Rollen für eine Kernklasse

Die Klasse `KernKlasse` repräsentiert den rollenunabhängigen Teil der Objekte. Die Klasse `KernKlasse` wird hier nicht weiter detailliert, weil es sich im Wesentlichen um eine anwendungsspezifische Klasse handelt. Beispiele hierfür sind die bereits genannten Klassen `Mitarbeiter` und `Professor` oder `Entwicklung`, `Verkauf` und `Controlling`. Objekte der Klasse `KernKlasse` können die Rollen spielen, für welche im Bild 4-51 die Klassen `RolleA`, `RolleB` und `RolleC` stellvertretend stehen.

#### 4.15.3.2 Teilnehmer

##### Client

Ein Client steht stellvertretend für eine Anwendung, die mit Objekten in verschiedenen Rollen interagiert.

##### RolleX

Eine Klasse `RolleX` ( $X = A..Z$ ) beschreibt eine Rolle, die von einem Kernobjekt gespielt werden kann. Ein Objekt einer Klasse `RolleX` referenziert dasjenige Kernobjekt, das diese Rolle gerade spielt. Über diese Referenz kann auch auf die rollenunabhängigen Eigenschaften eines Kernobjekts zugegriffen werden.

### KernKlasse

Die Klasse `KernKlasse` repräsentiert die rollenunabhängigen Eigenschaften von Objekten. Eine Instanz dieser Klasse ist ein Objekt, das die Rollen `RolleX` annehmen kann. Spielt ein Kernobjekt eine bestimmte Rolle, dann wird es von einem entsprechenden Rollenobjekt referenziert.

#### 4.15.3.3 Dynamisches Verhalten

Das folgende Sequenzdiagramm zeigt, wie eine Anwendung – hier durch ein Objekt der Klasse `Client` repräsentiert – sowohl Rollenoperationen ausführen als auch rollenunabhängige Kernfunktionen bei dem referenzierten Kernobjekt aufrufen kann:

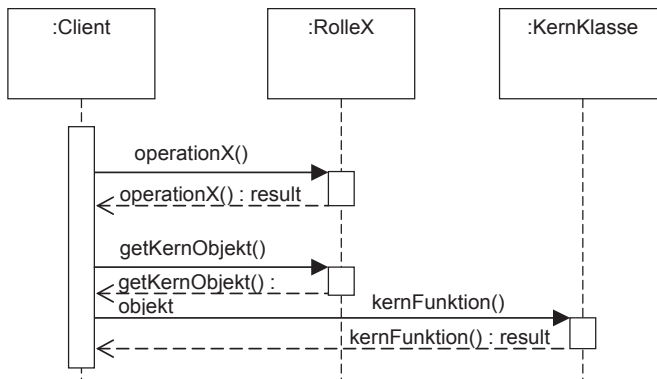


Bild 4-52 Sequenzdiagramm für eine Rollenoperation und eine Kernfunktion

Dem Client ist anfangs nur das Rollenobjekt bekannt. Rollenspezifische Operationen wie beispielsweise `operationX()` können daher vom Client direkt aufgerufen werden und werden von der Instanz der Klasse `RolleX` beantwortet. Aufrufe von Kernfunktionen kann der Client dadurch bewerkstelligen, dass er sich zuerst mit Hilfe der Methode `getKernObjekt()` eine Referenz auf das Kernobjekt besorgt, das diese Rolle gerade spielt. Über diese Referenz kann dann eine Methode des Kernobjekts wie beispielsweise `kernFunktion()` aufgerufen werden.

#### 4.15.3.4 Programmbeispiel

In dem folgenden Beispiel wird die in Kapitel 4.15.3 vorgestellte Grundform des Musters implementiert. Das bedeutet, dass ein Rollenobjekt den Zugriff auf das aggregierte Kernobjekt über eine `get`-Methode ermöglicht.

Das Programm greift das bisher schon benutzte Beispiel auf und realisiert einen kleinen Ausschnitt einer Unternehmensverwaltung. Die Klasse `Mitarbeiter` ist in diesem Beispiel die Kernklasse. Die Kernklasse hat im Rahmen des Entwurfsmusters Rolle nur eine untergeordnete Bedeutung. Die Klasse `Mitarbeiter` ist daher im Beispiel einfach und übersichtlich gehalten. Von der Kernklasse `Mitarbeiter` gibt es die Instanzen `maier` und `schulze`. Diese können in den Abteilungen Entwicklung und Verkauf arbeiten. Der Mitarbeiter `schulze` erhält Aufgaben in beiden Abteilungen und nimmt somit zwei Rollen gleichzeitig an.



Die Klassen `Entwicklung` und `Verkauf` repräsentieren die Aufgaben, die in der jeweiligen Abteilung durchzuführen sind. Diese beiden Klassen stellen im Beispiel Rollenklassen dar. Wie von der Grundform des Musters gefordert wird, stellen sie eine `get`-Methode zur Verfügung (`getMitarbeiter()`), mit der eine Anwendung ausgehend von einem Rollenobjekt auf das zugehörnde Kernobjekt zugreifen kann.

Die erste Rollenklasse ist die Klasse `Entwicklung`. Sie enthält die vom Muster vorgegebene Referenz auf einen `Mitarbeiter` und die gerade erwähnte Methode `getMitarbeiter()` sowie rollenspezifische Eigenschaften und Methoden – beispielhaft ist der Name des Projekts, an dem ein Entwickler arbeitet, und eine Methode zur Ausgabe des Projektnamens zu sehen. Hier nun der Quellcode der Klasse `Entwicklung`:

```
// Datei: Entwicklung.java
public class Entwicklung
{
    private Mitarbeiter mitarbeiter;
    private String projekt;

    public Entwicklung (String projekt)
    {
        this.projekt = projekt;
    }

    public String getAbteilungsBezeichnung()
    {
        return "Entwicklung";
    }

    public void printProjekt()
    {
        System.out.println (mitarbeiter.getName() + " "
            + "arbeitet momentan an: " + projekt);
    }

    public Mitarbeiter getMitarbeiter()
    {
        return mitarbeiter;
    }

    public void setMitarbeiter (Mitarbeiter mitarbeiter)
    {
        this.mitarbeiter = mitarbeiter;
    }
}
```

Es folgt nun als zweite Rollenklasse die Klasse `Verkauf`. Als Beispiel für eine rollenspezifische Eigenschaft wird hier eine Umsatzzahl benutzt. Als rollenspezifische Methode enthält die Klasse `Verkauf` eine Methode zur Ausgabe des Umsatzes des Mitarbeiters, der diese Rolle spielt. Daneben sind in der Klasse `Verkauf` wieder die durch das Muster vorgegebene Referenz auf einen `Mitarbeiter` und dazugehörnde `set`- und `get`-Methoden zu finden:

**// Datei: Verkauf.java**

```
public class Verkauf
{
    private Mitarbeiter mitarbeiter;
    int umsatz;

    public Verkauf (int umsatz)
    {
        this.umsatz = umsatz;
    }

    public String getAbteilungsBezeichnung()
    {
        return "Verkauf";
    }

    public void printUmsatz()
    {
        System.out.println ("Aktueller Umsatz von "
            + mitarbeiter.getName() + ": " + umsatz + " Euro.");
    }

    public Mitarbeiter getMitarbeiter()
    {
        return mitarbeiter;
    }

    public void setMitarbeiter (Mitarbeiter mitarbeiter)
    {
        this.mitarbeiter = mitarbeiter;
    }
}
```

Die Klasse `Mitarbeiter` ist in diesem Beispiel die Kernklasse. Um das Beispiel übersichtlich zu halten, speichert die Klasse `Mitarbeiter` nur den Namen eines Mitarbeiters:

**// Datei: Mitarbeiter.java**

```
public class Mitarbeiter
{
    private String name;

    public Mitarbeiter (String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Das folgende Hauptprogramm demonstriert nun das Rollenmuster. Es werden zuerst die beiden Kernobjekte mit den Namen `maier` und `schulze` erzeugt. Beiden Kernob-

jekten wird eine Rolle in der Abteilung Entwicklung zugewiesen. Nach einer Ausgabe dieses Zustands nimmt der Mitarbeiter `schulze` auch noch eine Rolle im Verkauf wahr. Der von dem Mitarbeiter `schulze` getätigte Umsatz wird am Ende ausgegeben. Die Klasse `TestRolle` enthält dieses Hauptprogramm als `main()`-Methode:

```
// Datei: TestRolle.java
public class TestRolle
{
    public static void main (String[] args)
    {
        // Kernobjekte werden erzeugt
        Mitarbeiter maier = new Mitarbeiter ("Maier");
        Mitarbeiter schulze = new Mitarbeiter ("Schulze");

        // Rollenobjekte der Abteilung Entwicklung
        Entwicklung entwicklung1 = new Entwicklung ("Produkt 2.0");
        Entwicklung entwicklung2 = new Entwicklung (
            "Produkt Addon 1.0");

        // Zuordnung der Rollen zu den Kernobjekten
        entwicklung1.setMitarbeiter (schulze);
        entwicklung2.setMitarbeiter (maier);

        // Ausgabe der Projekte
        System.out.println ("\nAktuelle Projekte der Mitarbeiter: ");
        entwicklung1.printProjekt();
        entwicklung2.printProjekt();

        // Ein Kernobjekt spielt eine weitere Rolle
        System.out.println ("\nSchulze erhaelt "
            + "zusätzliche Aufgaben im Verkauf:");
        Verkauf verkauf1 = new Verkauf (15000);
        verkauf1.setMitarbeiter (schulze);

        // Ausgabe der Umsatzzahlen
        System.out.println ("\nNur Schulze ist in der "
            + "Abteilung Verkauf:");
        verkauf1.printUmsatz();
    }
}
```



Hier das Protokoll des Programmlaufs:

```
Aktuelle Projekte der Mitarbeiter:
Schulze arbeitet momentan an: Produkt 2.0
Maier arbeitet momentan an: Produkt Addon 1.0
```

```
Schulze erhaelt zusätzliche Aufgaben im Verkauf:
```

```
Nur Schulze ist in der Abteilung Verkauf:
Aktueller Umsatz von Schulze: 15000 Euro.
```

Das Beispielprogramm zeigt, dass die beiden Kernobjekte die gleiche Rolle spielen können: beide Mitarbeiter sind Entwickler. Im zweiten Teil des Programms hat eines

der Kernobjekte zwei Rollen: der Mitarbeiter Schulze ist sowohl Entwickler als auch Verkäufer.

#### 4.15.4 Bewertung

##### 4.15.4.1 Vorteile

Folgende Vorteile werden gesehen:

- Rollen können zur Laufzeit dynamisch angenommen und abgelegt werden.
- Ein Objekt kann mehrere Rollen gleichzeitig spielen.

##### 4.15.4.2 Nachteile

Folgende Nachteile werden gesehen:

- Es ist in der Grundform des Rollenmusters etwas aufwendig, wenn man feststellen möchte, welche Rolle ein Kernobjekt einnimmt. Dann muss man alle Rollenobjekte daraufhin überprüfen, ob sie auf das gesuchte Kernobjekt zeigen.
- Es werden deutlich mehr Objekte erzeugt und diese müssen auch verwaltet werden.

#### 4.15.5 Einsatzgebiete

Das Rollenmuster bietet sich immer dann an, wenn Rollen zur Laufzeit dynamisch einem rollenunabhängigen Kernobjekt zugeordnet und aberkannt werden müssen und wo auch Kombinationen von Rollen möglich sein müssen. Durch Delegation kann das "in Rollen Schlüpfen" elegant nachmodelliert werden.

Rollen sind ein sehr gutes Mittel zur Umsetzung des **Single Responsibility-Prinzips** (siehe Kapitel 1.3). Das Single Responsibility-Prinzip fordert, dass die Abhängigkeiten von Klassen soweit zu reduzieren sind, dass die Methoden einer Klasse nur eine einzige Aufgabe erfüllen. Weitere Verantwortlichkeiten einer Klasse können mittels des Rollenmusters einfach als eigene Rollen ausgelagert und bei Bedarf angenommen werden.

Für das Identifizieren von Rollen während der Systemanalyse hat M. Fowler [Fow97] ein Rollenmuster entwickelt. In diesem Fall handelt es sich um das **Analysemuster Rolle**. M. Fowler zeigt in seiner Veröffentlichung mehrere Alternativen auf, wie eine während der Systemanalyse identifizierte Rolle in einen Entwurf umgesetzt werden kann. Das hier vorgestellte Entwurfsmuster Rolle ist eine dieser Alternativen.

#### 4.15.6 Ähnliche Entwurfsmuster

Im Gegensatz zum Rollenmuster ist die **Spezialisierung durch Ableitung** statisch und nicht flexibel. Sie ist aber sinnvoll, wenn ein Objekt unveränderliche Ausprägungen besitzt. Ist nämlich ein Objekt einer abgeleiteten Klasse erzeugt, so kann es nicht verändert werden. Die Klassen `Mann` oder `Frau` können beispielsweise gefahrlos von

`Person` abgeleitet werden, da das Geschlecht eine statische Eigenschaft darstellt. Wenn ein Objekt zu verschiedenen Zeiten unterschiedliche Rollen übernehmen kann, ist die Spezialisierung in abgeleiteten Objekten jedoch die falsche Vorgehensweise.

Das Entwurfsmuster **Zustand** kann eingesetzt werden, wenn das Verhalten eines Objekts sich dynamisch je nach eingenommenem Zustand ändert. Damit kann zwar ein Objekt eine Rolle zur Laufzeit ändern, aber nicht mehrere Rollen gleichzeitig einnehmen.

## 4.16 Das Verhaltensmuster Besucher

### 4.16.1 Namen/Alternative Namen

Besucher (engl. visitor).

### 4.16.2 Problem

Unter einer Objektstruktur wird im Folgenden ein – mehr oder weniger zusammenhängender – Satz von Objekten unterschiedlicher Klassen verstanden. Häufig tritt die Situation ein, dass Operationen realisiert werden müssen, die auf allen Objekten einer solchen heterogenen Objektstruktur aus verschiedenen Klassen arbeiten müssen und nicht nur auf einem einzelnen Objekt. Dabei sollen überdies lokale Daten aller Objekte der Objektstruktur zentral ausgewertet werden. Natürlich ist es möglich, Teilfunktionalitäten lokal in jedem einzelnen Objekt abzuhandeln. Es wird aber dennoch ein zentrales Objekt zum Zusammenführen der Daten aller Objekte benötigt.

Das Besuchermuster soll es erlauben, die Daten aller Objekte einer Objektstruktur zentral auszuwerten.



### 4.16.3 Lösung

Das Besucher-Muster kapselt die Logik einer Operation, die Daten von Objekten verschiedener Klassen, die sich innerhalb einer Objektstruktur befinden, benötigt, komplett in einem separaten Besucher-Objekt, einem sogenannten Kontrollobjekt. Das Besucher-Muster ist ein objektbasiertes Verhaltensmuster.

Die Daten tragenden Objekte, die im Rahmen der Operation besucht werden müssen, sind in einer Objektstruktur angeordnet. Als einfaches Beispiel sei hier eine Liste von Objekten genannt. Auf die Aspekte anderer, insbesondere von komplizierteren Objektstrukturen wird in Kapitel 4.16.3.3 eingegangen.

Ein wichtiges Kennzeichen des Besucher-Musters ist, dass die Objekte der Objektstruktur Instanzen von unterschiedlichen Klassen sein können. Die Klassen, von denen sich Instanzen in der Objektstruktur befinden, werden im Folgenden als **Element-Klassen** bezeichnet.



Die bereits bestehenden Element-Klassen müssen um eine `akzeptieren()`-Methode ergänzt werden, wenn sie diese nicht bereits schon aufweisen. Der Zweck der Methode `akzeptieren()` wird im Folgenden noch ausführlich erläutert. Sieht man von der Erstellung der `akzeptieren()`-Methode in den zu besuchenden Objekten ab, so bleiben die bestehenden Element-Klassen unverändert.

Für jede zu realisierende Operation wird eine **Besucher-Klasse** bereitgestellt, die einen Satz von überladenen `besuchen()`-Methoden enthält und zwar jeweils eine separate `besuchen()`-Methode für jede in der Objektstruktur vorhandene Element-Klasse.



Jede `besuchen()`-Methode hat einen Übergabeparameter vom Typ einer der Element-Klassen und kann über diesen Parameter auf die Daten des besuchten Objekts der Objektstruktur zugreifen. Eine Besucher-Klasse muss gegebenenfalls noch Methoden bereitstellen, mit deren Hilfe sich eine Anwendung die Ergebnisse einer Operation von einem Besucher-Objekt abholen kann.

Die `akzeptieren()`-Methode einer Element-Klasse hat als Übergabeparameter ein Besucher-Objekt. Das Objekt, dessen `akzeptieren()`-Methode aufgerufen wird, ruft im Rumpf der `akzeptieren()`-Methode die für eine Elementklasse spezifische `besuchen()`-Methode des übergebenen Besuchers auf und übergibt dabei eine Referenz auf sich selbst.



Dadurch kann die gerufene `besuchen()`-Methode – beispielsweise über öffentliche `get`-Methoden – auf die Daten des betreffenden Objekts der Objektstruktur zugreifen. Diese Hin- und Her-Aufrufe werden als simuliertes **Double-Dispatch** bezeichnet und in Kapitel 4.16.3.3 noch genauer beschrieben.

Zur Lösung wird eine abstrakte Klasse `Besucher` eingeführt, von der alle konkreten Besucher-Klassen abgeleitet sind. Die abstrakte Klasse `Besucher` wird als Typ für den jeweiligen Übergabeparameter der `akzeptieren()`-Methoden in den Element-Klassen gewählt. Die Übergabeparameter der `akzeptieren()`-Methoden können auf Grund des liskovschen Substitutionsprinzips auf jeden konkreten Besucher zeigen, sofern in den konkreten Besucher-Klassen die Verträge der abstrakten Klasse nicht gebrochen werden. Die folgende Beschreibung des Musters verwendet für die Abstraktion der Besucher-Klassen eine abstrakte Klasse. Die Abstraktion könnte aber genauso gut mittels eines Interface definiert werden. Die konkreten Besucher-Klassen müssten dann dieses Interface realisieren, statt von der abstrakten Klasse abzuleiten.

Sieht man von der erstmaligen Einführung einer solchen `akzeptieren()`-Methode in den Element-Klassen ab, so bleiben die Element-Klassen unverändert. Die Realisierung einer neuen Operation erfordert dann nur die Implementierung der `besuchen()`-Methoden in einer neuen Besucher-Klasse. Damit ist die Logik dieser neuen Operation selbst nicht über viele Objekte verteilt, sondern nur die Datenbeschaffung.



Dieser Ansatz weicht von dem Grundgedanken der Objektorientierung ab, die Daten eines Objekts und die zu den Daten gehörenden Prozeduren, die auf den Daten ope-

rieren, in ein Objekt zu kapseln. Beim Besucher-Muster dagegen verarbeiten und beschaffen Operationen, die jeweils in einer separaten Klasse gekapselt werden, die Daten von Objekten verschiedener Klassen mit unterschiedlichen Schnittstellen einer Objektstruktur. Diese werden dann zentral im Besucher-Objekt ausgewertet. Diese Vorgehensweise hat den Vorteil, dass die Klassen der Daten tragenden Objekte der Objektstruktur nicht für neu hinzukommende Operationen verändert werden müssen<sup>62</sup>.

#### 4.16.3.1 Klassendiagramm

Das folgende Klassendiagramm beschreibt die statische Struktur der Teilnehmer zueinander:

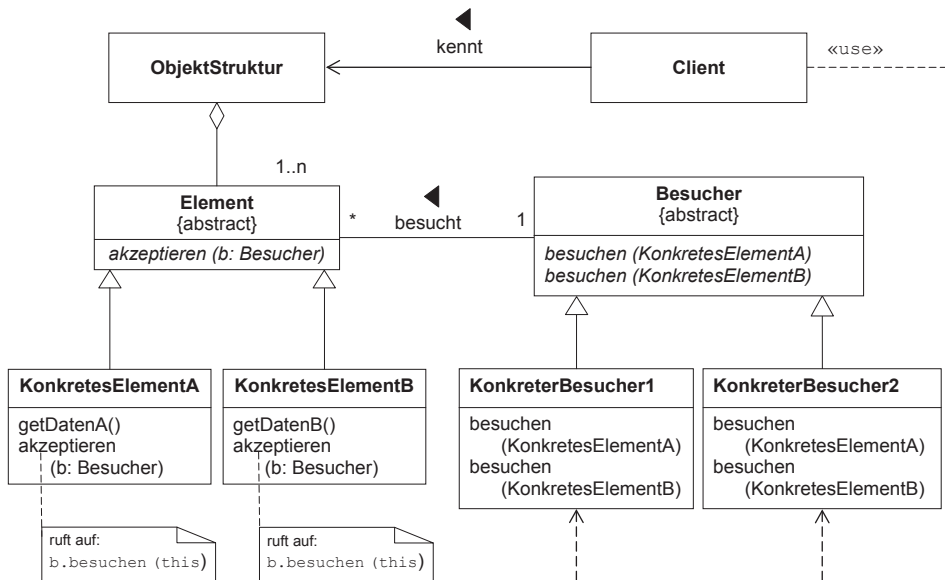


Bild 4-53 Klassendiagramm des Besucher-Musters unter Verwendung einer Objektstruktur

In der linken Hälfte von Bild 4-53 ist die Objektstruktur dargestellt. Die Klasse `Objektstruktur` ist hier nur stellvertretend zu sehen. Die Objektstruktur kann im einfachsten Fall aus einem Array von Elementen bestehen. Der Aufbau der Objektstruktur und die Art und Weise, wie auf deren Elemente zugegriffen werden kann, hat einen erheblichen Einfluss auf das dynamische Verhalten. Darauf wird in Kapitel 4.16.3.3 im Detail eingegangen.

Die Elemente der Objektstruktur sind Instanzen der Klassen `KonkretesElementY` ( $Y = A..Z$ ). Diese Klassen sind von der abstrakten Klasse `Element` abgeleitet, welche die Deklaration einer `akzeptieren()`-Methode mit einem Parameter vom Typ der abstrakten Basisklasse `Besucher` enthält. Die Methode `akzeptieren()` muss von allen Daten tragenden Klassen, deren Objekte besuchbar sein sollen, implementiert werden. Dadurch, dass als Typ des Parameters der `akzeptieren()`-Methode die

<sup>62</sup> Dies gilt natürlich nur, solange die neue Operation nur solche Daten benötigt, die bereits über die öffentliche Schnittstelle eines Objektes zur Verfügung stehen.



Klasse `Besucher` gewählt wird, braucht eine konkrete Element-Klasse die konkrete Klasse eines Besuchers nicht zu kennen, sondern kann auf Grund des liskovschen Substitutionsprinzips jedes Besucher-Objekt akzeptieren, dessen Klasse die Verträge der Basisklasse `Besucher` einhält.

Die Implementierung der `akzeptieren()`-Methode besteht nur aus dem Aufruf der `besuchen()`-Methode des als Parameter übergebenen Besuchers. Das akzeptierende Objekt übergibt an die `besuchen()`-Methode eine Referenz auf sich selbst, so dass der aufgerufene Besucher anschließend in der Lage ist, über die im Diagramm angedeuteten `get`-Methoden der konkreten Element-Klasse sich Daten vom aufrufenden Objekt zu besorgen, um seine zentrale Funktion zu realisieren.

Obwohl die Implementierung der `akzeptieren()`-Methode in jeder Klasse gleich ist, muss sie trotzdem in jeder konkreten Klasse `KonkretesElementY` stattfinden und darf nicht in eine Basis-Klasse verlagert werden, da dem Besucher sonst nur eine Referenz vom Typ der Basisklasse übergeben wird.



In der rechten Hälfte von Bild 4-53 befindet sich die Hierarchie der Besucher-Klassen.

Die Basisklasse aller Besucher-Klassen ist die abstrakte Klasse `Besucher`. Die Klasse `Besucher` definiert für jede konkrete Element-Klasse `KonkretesElementY` eine abstrakte `besuchen()`-Methode mit einem Parameter vom Typ der entsprechenden Element-Klasse<sup>63</sup>.



Die konkreten Besucher-Klassen `KonkreterBesucherX` ( $X = 1..n$ ) implementieren diese `besuchen()`-Methoden, indem sie im Rumpf einer `besuchen()`-Methode auf die Daten des übergebenen Objekts der Objektstruktur – beispielsweise über die `get`-Methoden der entsprechenden Element-Klasse – zugreifen und diese Daten zur Realisierung der gewünschten Funktionalität der jeweiligen Besucher-Klasse nutzen.

Die abstrakten Klassen `Element` und `Besucher` verbindet eine bidirektionale Assoziation, die es ermöglicht, dass jedes konkrete Besucher-Objekt jedes konkrete Daten tragende Objekt besucht und dass jeder Besuchte jeden Besucher akzeptieren muss.

Der Typ der Basisklasse `Element`, von der die Klassen `KonkretesElementY` abgeleitet sind, reicht als Parameter der `besuchen()`-Methode nicht aus, da auf Basis des übergebenen Typs `KonkretesElementY` die passende `besuchen()`-Methode ausgewählt wird.



<sup>63</sup> Man kann die `besuchen()`-Methoden der Klasse `Besucher` auch dafür nutzen, eine Default-Implementierung einzurichten, die von abgeleiteten Besucher-Klassen bei Bedarf genutzt werden kann.

In der hier vorgestellten Lösung sind die `besuchen()`-Methoden überladen: sie unterscheiden sich nicht durch ihren Namen, sondern nur durch den Typ des Parameters. Da aber jede `besuchen()`-Methode des Besuchers nur aus genau einer Element-Klasse heraus aufgerufen wird und durch die übergebene Selbstreferenz immer die zu dieser Element-Klasse passende `besuchen()`-Methode aufgerufen wird, wird der Overloading-Mechanismus eigentlich gar nicht benötigt, d. h., dass die `besuchen()`-Methoden sich auch im Namen unterscheiden und beispielsweise `besucheKonkretesElementY()` genannt werden könnten.

Der Client steht im Klassendiagramm in Bild 4-53 stellvertretend für eine Anwendung des Besucher-Musters. Der Client kennt die Objektstruktur und möchte auf dieser Objektstruktur eine Besuchoperation ausführen. Das Muster legt nicht fest, ob der Client ein Besucher-Objekt, das eine Operation realisiert, selber erzeugt oder ob ein Besucher-Objekt ihm von außen übergeben wird. Der Client stößt den Besuch der Objektstruktur durch das Besucher-Objekt an. Wie bereits erwähnt wurde, kann dieser Ablauf je nach Objektstruktur unterschiedlich sein (siehe Kapitel 4.16.3.3). Wenn die Objektstruktur beispielsweise ein einfaches Array ist, iteriert der Client über die Elemente des Arrays und ruft für jedes Element dessen `akzeptieren()`-Methode auf.

#### 4.16.3.2 Teilnehmer

Am Besucher-Entwurfsmuster nehmen die folgenden Klassen teil:

##### Besucher

Die abstrakte Klasse `Besucher` deklariert eine `besuchen()`-Methode für jede Klasse `KonkretesElementY` ( $Y = A..Z$ ). Jede `besuchen()`-Methode hat jeweils einen Parameter vom Typ einer Klasse `KonkretesElementY`.

##### KonkreterBesucherX ( $X = 1..n$ )

Instanzen einer Klasse `KonkreterBesucherX` sind die eigentlichen Besucher. Eine Klasse `KonkreterBesucherX` leitet von der abstrakten Klasse `Besucher` ab und implementiert in ihren `besuchen()`-Methoden jeweils die spezielle Operation, die mit den Daten der besuchten Objekte der Klasse `KonkretesElementY` ausgeführt wird.

##### Element

Die Klassen der Objekte, die besucht werden sollen, sind von der abstrakten Klasse `Element` abgeleitet<sup>64</sup>. Die Klasse `Element` enthält die Deklaration der `akzeptieren()`-Methode mit einem Parameter vom Typ der abstrakten Klasse `Besucher`.

##### KonkretesElementY ( $Y = A..Z$ )

Instanzen dieser Klassen, die von der abstrakten Klasse `Element` abgeleitet werden, sind die eigentlichen Daten tragenden Objekte und bilden die Objektstruktur. Sie werden von den Objekten vom Typ `KonkreterBesucherX` aufgesucht, die mit den Daten der besuchten Objekte ihre gewünschten Funktionalitäten erbringen. Die konkreten Element-Klassen implementieren die in der Basisklasse `Element` dekla-

<sup>64</sup> Das Besucher-Muster wird oftmals so implementiert. Das Muster verlangt jedoch streng genommen nur, dass die Klassen `KonkretesElementY` eine `akzeptieren()`-Methode haben.

rierte `akzeptieren()`-Methode, indem sie in deren Rumpf die `besuchen()`-Methode des Objekts vom Typ `KonkreterBesucherX` aufrufen und eine Referenz auf sich selbst als Parameter der Methode `besuchen()` übergeben.

### ObjektStruktur

In der Rolle der Klasse `ObjektStruktur` tritt das Objekt auf, das die Zugriffslogik für die konkreten Elemente enthält. Der Besuchsvorgang wird vom Objekt in der Rolle der `ObjektStruktur` eingeleitet, indem es die `akzeptieren()`-Methode der enthaltenen Elemente aufruft.

### Client

Mit Client wird das aufrufende Programm bezeichnet. Der Client kennt die `ObjektStruktur` und möchte eine Operation ausführen, die in einer Klasse `KonkreterBesucherX` implementiert ist. Dazu stößt der Client die Zusammenarbeit der beteiligten Klassen bzw. Objekte an, indem er die Methode `besucheElement(kbX)` des Objekts der Klasse `ObjektStruktur` aufruft.

#### 4.16.3.3 Dynamisches Verhalten

Das folgende Sequenzdiagramm zeigt das dynamische Verhalten der Teilnehmer:

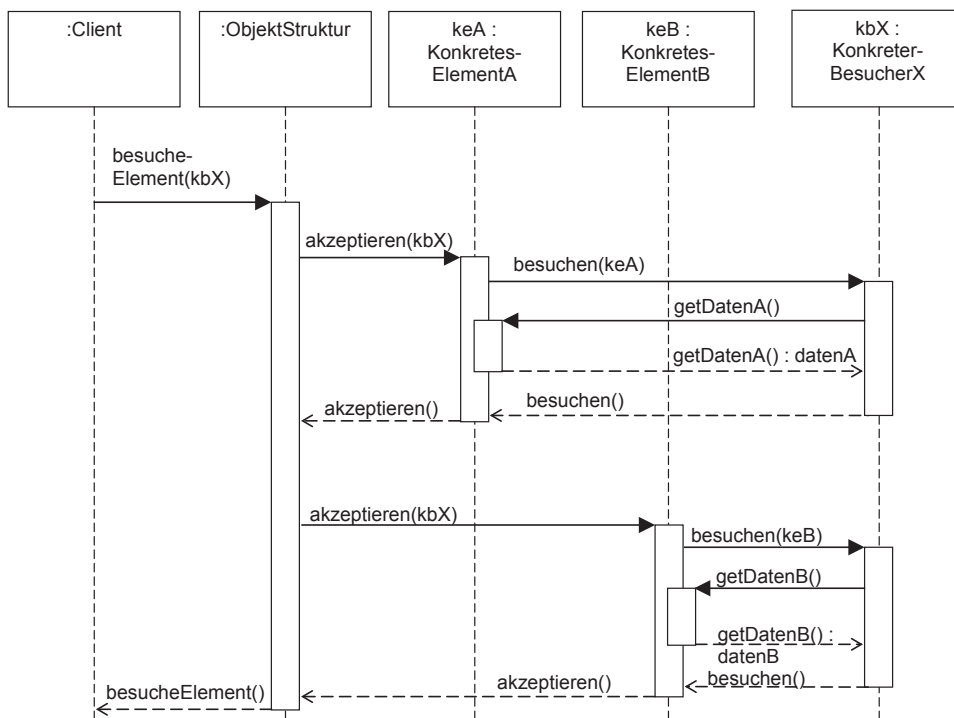


Bild 4-54 Sequenzdiagramm des Besucher-Musters

Der Client kennt die `ObjektStruktur` und möchte die Operation, die in einem Besucher-Objekt `kbX` vom Typ `KonkreterBesucherX` gekapselt ist, ausführen. Im Bild 4-54

ruft er dazu die Methode `besucheElement()` der Objektstruktur auf und übergibt dabei das Besucher-Objekt. Dieser Schritt ist nicht durch das Besucher-Muster festgelegt und kann – je nach Art der Objektstruktur – unterschiedlich sein. Dieser Aspekt wird im weiteren Verlauf dieses Kapitels noch ausführlich behandelt. Hier wird zuerst eine einfache Variante vorgestellt, bei der die Objektstruktur zwei Elemente – im Bild mit `keA` und `keB` bezeichnet – enthält.

Das Objekt der Klasse `ObjektStruktur` sorgt in dieser Variante dafür, dass die zwei Elemente besucht werden. Es ruft die `akzeptieren()`-Methode des Objekts `keA` vom Typ `KonkretesElementA` auf und übergibt das Besucher-Objekt `kbX`, das der Client vorgegeben hat. Das Objekt `keA` ruft in seiner `akzeptieren()`-Methode lediglich die `besuchen()`-Methode des übergebenen Objekts vom Typ `KonkreterBesucherX` auf und übergibt hierbei eine Referenz auf sich selbst.

Durch den Typ der übergebenen Referenz wird die für die Klasse des Objekts `keA` – also für die Klasse `KonkretesElementA` – bestimmte Implementierung der `besuchen()`-Methode ausgewählt und vom Besucher-Objekt `kbX` ausgeführt. In Bild 4-54 besteht diese Implementierung aus dem Aufruf der Methode `getDatenA()` bei dem übergebenen Objekt – in diesem Beispiel also beim Objekt `keA`. Mit den abgeholten Daten kann nun das Besucher-Objekt einen Teil seiner Gesamtfunktionalität erbringen. Danach ist der Besuch des ersten Elementes beendet.

Der Ablauf für das Element-Objekt der Klasse `keB` vom Typ `KonkretesElementB` ist identisch, nur, dass in diesem Fall die zur Klasse des Objekts `keB` passende Implementierung der `besuchen()`-Methode herangezogen wird: Es wird `getDatenB()` an Stelle von `getDatenA()` ausgeführt. Das Besucher-Objekt kann die vom Element `keB` abgeholten Daten verarbeiten und damit einen weiteren Teil seiner Funktionalität erbringen.

Der Aufruf von get-Methoden wie beispielsweise `getDatenA()` muss im Rumpf der jeweiligen `besuchen()`-Methode erfolgen, denn nur da kennt ein Besucher die Klasse des jeweiligen Elements. Dadurch entsteht ein Hin- und Her-Aufruf, der sich im Bild 4-54 durch einen verschobenen Aufrufbalken für die Aufrufe der get-Methoden bei den Element-Objekten ausdrückt. Durch diesen Hin- und Her-Aufruf kommt die bereits angesprochene bidirektionale Assoziation zustande, die es ermöglicht, dass jedes Besucher-Objekt jedes Element-Objekt besuchen kann.

Eigentlich hängt die Auswahl der auszuführenden Operation in einem solchen Zusammenhang von zwei Typen ab, dem Typ des Elementes und dem Typ des Besuchers. Diese Technik wird als **Double-Dispatch** bezeichnet und nur von wenigen Programmiersprachen unterstützt. Die meisten gängigen objektorientierten Sprachen – darunter auch Java – unterstützen nur ein **Single-Dispatch**: die Auswahl einer Operation hängt nur von einem einzigen Typ ab, dem Typ des Objektes, dessen Operation aufgerufen werden soll.



Double-Dispatch muss also in Java simuliert werden. Dies geschieht dadurch, dass die konkreten Elemente der Objektstruktur die in der Basisklasse deklarierte `akzeptieren()`-Methode – wie bereits beschrieben – implementieren. Im Rumpf dieser Methode rufen sie die `besuchen()`-Methode des Objektes vom Typ `KonkreterBesucherX` auf und übergeben hierbei eine Referenz auf sich selbst als Parameter. Damit ist – wie gewünscht – die Auswahl der "richtigen" `besuchen()`-Methode abhängig vom konkreten Besucher und dem konkreten Daten tragenden Element. Die Technik des Double-Dispatch wird ausführlich in [Bec08] beschrieben. Die Hin- und Her-Aufrufe kommen also durch die Simulation von Double-Dispatch in Java zustande.

### Varianten im Verhalten bedingt durch unterschiedliche Objektstrukturen

Es wurde bereits mehrfach erwähnt, dass die Beschreibung des dynamischen Verhaltens nur schematisch erfolgen kann, solange nicht die zu besuchende Objektstruktur konkret betrachtet wird. Bisher wurde die Objektstruktur als eigenständiges Objekt betrachtet, das die enthaltenen Elemente "kennt" und, wie im Bild 4-54 zu sehen ist, dafür sorgt, dass diese Elemente besucht werden. Diese Rolle kann aber auch von einem anderen Objekt übernommen werden – abhängig davon, wie die Objektstruktur konkret realisiert ist. Dadurch ergeben sich verschiedene Varianten des dynamischen Verhaltens des Besucher-Musters. Die Rolle, die in der bisherigen Beschreibung des Musters das Objekt der Klasse `ObjektStruktur` innehatte, kann alternativ auch:

- vom Client,
- von einem separaten Iterator-Objekt,
- einem Besucher-Objekt oder
- einer Datenstruktur, die die Elemente enthält (z. B. ein Baum),

übernommen werden. Diese Fälle werden im Folgenden vorgestellt:

#### • Client

Im einfachsten Fall nimmt der Client die Rolle der Objektstruktur an. Die Objekte vom Typ `KonkretesElementY` sind einzelne, verstreute Instanzen ohne Bezug zueinander oder befinden sich in einer einfachen Datenstruktur wie z. B. einem Array. Der Client nimmt ein Objekt der vorhandenen Typen `KonkretesElementY` ( $Y = A..Z$ ) nach dem anderen und sorgt dafür, dass es besucht wird, indem er jeweils die `akzeptieren()`-Methode dieses Objekts aufruft und dabei das Objekt vom Typ `KonkreterBesucherX` übergibt.

#### • Separates Iterator-Objekt

Wenn die Elemente der Objektstruktur gekapselt sind und die Objektstruktur ein Iterator-Objekt zum Zugriff auf die Elemente anbietet, so wird das Iterator-Objekt vom Client benutzt, um über die Menge der Objekte der vorhandenen Typen `KonkretesElementY` ( $Y = A..Z$ ) zu iterieren. Der Client ruft dabei die `akzeptieren()`-Methode jedes Objekts auf, auf das er vom Iterator-Objekt eine Referenz bekommt und übergibt hierbei das Objekt vom Typ `KonkreterBesucherX`. Die Reihenfolge der Besuche wird durch die Reihenfolge bestimmt, in der das Iterator-Objekt die einzelnen Referenzen liefert.

Beim Einsatz von Programmiersprachen wie Java oder C# kann diese Variante auch implizit auftreten, nämlich, wenn für die Speicherung der Objekte vom Typ `KonkretesElementY` eine Collection-Klasse verwendet wird und, wenn eine `foreach`-Schleife eingesetzt wird, um über diese Objekte zu iterieren. Bei dieser impliziten Variante sieht man praktisch keinen Unterschied zum ersten Fall, bei dem der Client die Rolle der Objektstruktur annimmt, da aus der Sicht eines Client eine Collection sich so trivial nutzen lässt wie eine einfache Objektstruktur.

- **Besucher-Objekt**

Das Besucher-Objekt übernimmt selbst die Rolle der Objektstruktur, wenn es Referenzen auf alle zu besuchenden Objekte enthält.

- **Datenstruktur der Elemente**

Sind die Objekte vom Typ `KonkretesElementY` in einer verketteten Form wie etwa einem Baum abgelegt, übernimmt diese Datenstruktur die Rolle der Objektstruktur und die Besuchsreihenfolge wird über diese Verkettung festgelegt. Der Client ruft die `akzeptieren()`-Methode eines Objekts vom Typ `KonkretesElementY` – im Fall eines Baums die des Wurzelknotens auf – und übergibt das Objekt vom Typ `KonkreterBesucherX`. Das besuchte Objekt sorgt in seiner `akzeptieren()`-Methode dafür, dass alle mit ihm verknüpften Objekte besucht werden.

Bei einer baumartigen Struktur entscheidet der besuchte Knoten dann über die Durchlaufstrategie durch den Baum. Er kann beispielsweise zuerst seine Kindknoten und anschließend sich selbst besuchen lassen, was zu einer Depth-First Strategie führt.

Besonders aufwendig gestaltet sich die Implementierung, wenn die Daten tragenden Objekte in einem Graphen angeordnet sind, der auch Zyklen enthalten kann. Die Möglichkeit, dass ein Objekt während eines Durchlaufens der Struktur mehrfach besucht wird, muss in der Regel ausgeschlossen werden.

Die hier beschriebene Variante gilt auch für Datenstrukturen, die mit Hilfe des Kompositum-Musters implementiert werden. Ein zusammengesetztes Element ruft in seiner `akzeptieren()`-Methode sowohl die `besuchen()`-Methode für sich selbst als auch die `akzeptieren()`-Methode aller von ihm referenzierten Elemente auf. Ein Blatt-Element ruft nur die `besuchen()`-Methode für sich auf.

#### 4.16.3.4 Programmbeispiel

Der nachfolgende Quellcode beschreibt ein Beispiel, bei dem ein Objekt vom Typ `Gehaltsdrucker` Elemente vom Typ `Teamleiter` und `Sachbearbeiter` besucht. Die Klasse `Gehaltsdrucker` nimmt hier die Funktion der Besucher-Klasse ein. In Abhängigkeit von der Klasse des besuchten Objekts wird die richtige Methode aufgerufen und die entsprechende Gehaltszeile ausgedruckt.

Zum besseren Verständnis des Beispielprogramms wird zu jeder Klasse separat mit angegeben, welcher Klasse im Bild 4-53 sie entspricht.

Die abstrakte Klasse `MitarbeiterBesucher` repräsentiert die abstrakte Besucher-Klasse und definiert für jede konkrete Element-Klasse, welche besucht werden soll, eine Methode `besuchen()`:

```
// Datei: MitarbeiterBesucher.java
abstract class MitarbeiterBesucher
{
    public abstract void besuchen (Teamleiter t);
    public abstract void besuchen (Sachbearbeiter s);
}
```

Die Klasse `Gehaltsdrucker` stellt einen konkreten Besucher dar und wird aus diesem Grund von der Klasse `MitarbeiterBesucher` abgeleitet. In den Implementierungen der jeweiligen `besuchen()`-Methoden werden Informationen über das jeweils gerade besuchte Objekt ausgegeben:

```
// Datei: Gehaltsdrucker.java
class Gehaltsdrucker extends MitarbeiterBesucher
{
    public Gehaltsdrucker()
    {
        System.out.print ("*****");
        System.out.println ("*****");
        System.out.println ("          Gehaltsliste");
        System.out.print ("Position" + "\t" + "Vorname" + "\t" + "\t" +
            "Name" + "\t" + "\t");
        System.out.println ("Gehalt" + "\t" + "Praemie");
        System.out.print ("*****");
        System.out.println ("*****");
    }

    public void besuchen (Teamleiter t)
    {
        String vorname;
        String name;
        if (t.getVorname().length() < 8)
            vorname = t.getVorname().concat ("\t");
        else vorname = t.getVorname();

        if (t.getName().length() < 8)
            name = t.getName().concat ("\t");
        else name = t.getName();

        System.out.print ("Leiter " + t.getTeambezeichnung() + "\t"
            + vorname + "\t" + name + "\t");
        System.out.printf ("%7.2f", t.getGrundgehalt());
        System.out.printf ("%3.2f", t.getPraemie());
        System.out.println();
    }

    public void besuchen (Sachbearbeiter s)
    {
        String vorname;
        String name;
```

```

        if (s.getVorname().length() < 8)
            vorname = s.getVorname().concat ("\\t");
        else vorname = s.getVorname();

        if (s.getName().length() < 8)
            name = s.getName().concat ("\\t");
        else name = s.getName();

        System.out.print ("Sachbearbeiter" + "\\t" + vorname + "\\t" +
            name + "\\t");
        System.out.printf ("%7.2f", s.getGehalt());
        System.out.println (" ---");
    }
}

```

Durch die Klasse `Gesellschaft` wird die Objektstruktur realisiert. Sie beinhaltet Beispielinstanzen der konkreten Elemente:

```

// Datei: Gesellschaft.java
import java.util.ArrayList;
import java.util.List;

// Diese Klasse repraesentiert eine Firma und enthaelt ihre
// Mitarbeiter
class Gesellschaft
{
    private List<Mitarbeiter> personal;

    public Gesellschaft()
    {
        this.personal = new ArrayList<Mitarbeiter>();
        initialisiereBeispieldaten();
    }

    private void initialisiereBeispieldaten()
    {
        // Sachbearbeiter Team 1
        ArrayList<Mitarbeiter> team1 = new ArrayList<Mitarbeiter>();

        team1.add
            (new Sachbearbeiter ("Markus","Mueller ", 48200.0f));
        team1.add
            (new Sachbearbeiter ("Silvia","Neustedt", 45500.0f));

        // Sachbearbeiter Team 2
        ArrayList<Mitarbeiter> team2 = new ArrayList<Mitarbeiter>();

        team2.add
            (new Sachbearbeiter ("Alexandra","Weiss", 37120.0f));
        team2.add
            (new Sachbearbeiter ("Michael","Kienzle", 35500.0f));

        // Teamleiter
        Teamleiter chef1, chef2;
    }
}

```



```
        chef1 =
            new Teamleiter ("Frank", "Hirschle", 40000.0f, 400.0f,
                           "Team 1");
        chef2 =
            new Teamleiter ("Corinna", "Steib", 35000.0f, 350.0f,
                           "Team 2");

        // alle Personen in die Personalliste
        this.personal.add (chef1);
        this.personal.add (chef2);
        this.personal.addAll (team1);
        this.personal.addAll (team2);
    }

    public List<Mitarbeiter> getPersonal()
    {
        return personal;
    }
}
```

Die abstrakte Klasse `Mitarbeiter` ist die Basisklasse für die konkreten Elemente dieses Beispiels. Sie entspricht somit der abstrakten Klasse `Element` aus der allgemeinen Beschreibung des Besucher-Musters. Die abstrakte Klasse `Mitarbeiter` gibt für konkrete Element-Klassen die Deklaration der abstrakten Methode `akzeptieren()` vor:

```
// Datei: Mitarbeiter.java
abstract class Mitarbeiter
{
    protected int personalnummer;
    private static int anzahlMitarbeiter = 0;
    protected String name;
    protected String vorname;

    Mitarbeiter (String vorname, String name)
    {
        this.personalnummer = anzahlMitarbeiter++;
        this.vorname = vorname;
        this.name = name;
    }

    public int getPersonalnummer()
    {
        return personalnummer;
    }

    public String getName()
    {
        return name;
    }

    public void setName (String name)
    {
        this.name = name;
    }
}
```

```

    public String getVorname()
    {
        return vorname;
    }

    public void setVorname (String vorname)
    {
        this.vorname = vorname;
    }

    public String toString()
    {
        return ("PersonalNr."+ this.personalnummer + "Name:" +
                this.vorname + " " + this.name);
    }

    public abstract void akzeptieren (MitarbeiterBesucher v);
}

```

Die Klasse `Sachbearbeiter` ist eine konkrete Element-Klasse. Sie ist von der abstrakten Klasse `Mitarbeiter` abgeleitet und implementiert die `akzeptieren()`-Methode:

```

// Datei: Sachbearbeiter.java
class Sachbearbeiter extends Mitarbeiter
{
    private float gehalt;

    public Sachbearbeiter (String vorname, String name, float gehalt)
    {
        super (vorname,name);
        this.gehalt = gehalt;
    }

    public float getGehalt()
    {
        return gehalt;
    }

    public void akzeptieren (MitarbeiterBesucher v)
    {
        // sich selbst besuchen lassen
        v.besuchen (this);
    }
}

```

Die Klasse `Teamleiter` stellt ebenfalls eine konkrete Element-Klasse dar. Sie ist von der abstrakten Klasse `Mitarbeiter` abgeleitet und implementiert die von der abstrakten Klasse `Mitarbeiter` vorgegebene Methode `akzeptieren()`:

```

// Datei: Teamleiter.java
class Teamleiter extends Mitarbeiter
{
    private String teambezeichnung;

```

```
private float grundgehalt;
private float praemie;

public Teamleiter (String vorname, String name,
                  float grundgehalt, float praemie,
                  String teambezeichnung)
{
    super (vorname,name);
    this.grundgehalt = grundgehalt;
    this.praemie = praemie;
    this.teambezeichnung = teambezeichnung;
}

public String getTeambezeichnung()
{
    return teambezeichnung;
}

public void setTeambezeichnung (String teambezeichnung)
{
    this.teambezeichnung = teambezeichnung;
}

public float getGrundgehalt()
{
    return this.grundgehalt;
}

public void setGrundgehalt (float grundgehalt)
{
    this.grundgehalt = grundgehalt;
}

public float getPraemie()
{
    return this.praemie;
}

public void setPraemie (float praemie)
{
    this.praemie = praemie;
}

public void akzeptieren (MitarbeiterBesucher v)
{
    // sich selbst besuchen lassen
    v.besuchen (this);
}
}
```

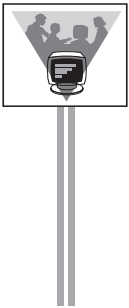
Innerhalb der `main()`-Methode der Klasse `PersonalVerwaltung` wird eine neue Objektstruktur als Objekt der Klasse `Gesellschaft` angelegt. Aus dieser kann die aktuelle Belegschaft in Form einer Mitarbeiterliste ermittelt werden. Die Belegschaft wird durchlaufen und die jeweiligen Mitarbeiter werden durch den Gehaltsdrucker besucht. Hier die Klasse `PersonalVerwaltung`:

```
// Datei: PersonalVerwaltung.java
import java.util.List;

public class PersonalVerwaltung
{
    public static void main (String[] args)
    {
        // Initialisierungen vornehmen
        Gesellschaft firma = new Gesellschaft();
        List<Mitarbeiter> belegschaft = firma.getPersonal();

        // Besucher-Objekt fuer die Liste erzeugen
        Gehaltsdrucker besucher = new Gehaltsdrucker();

        // Ueber die Liste iterieren und Besuche durchfuehren
        for (Mitarbeiter arbeiter: belegschaft)
        {
            arbeiter.akzeptieren (besucher);
        }
    }
}
```



Hier das Protokoll des Programmlaufs:

```
*****
                                Gehaltsliste
*****
Position      Vorname   Name      Gehalt   Praemie
*****
Leiter Team 1  Frank     Hirschle  40000.00  400.0
Leiter Team 2  Corinna  Steib     35000.00  350.0
Sachbearbeiter Markus   Mueller   48200,00  ---
Sachbearbeiter Silvia   Neustedt  45500,00  ---
Sachbearbeiter Alexandra Weiss      37120,00  ---
Sachbearbeiter Michael   Kienzle   35500,00  ---
```

## 4.16.4 Bewertung

### 4.16.4.1 Vorteile

Folgende Vorteile werden gesehen:

- **Einfaches Hinzufügen von neuer Funktionalität**

Mit dem Besucher-Muster kann eine neue Funktionalität sehr einfach zu einer bestehenden Datenstruktur hinzugefügt werden. Da die Daten tragenden Element-Klassen in der Regel die Schnittstelle für den Besuch beliebiger Besucher zur Verfügung stellen, genügt es meist, die gewünschte Funktionalität in einer zusätzlichen Besucher-Klasse zu implementieren.

- **Zentralisierung des Codes einer Operation**

Der Code einer Operation wird in einer Klasse zentralisiert und ist nicht über viele Klassen verteilt.

- **Möglichkeit Klassenhierarchien-übergreifender Besuche**

Besucher können Element-Objekte besuchen, die nicht Teil derselben Klassenhierarchie sind. Im Gegensatz zum Iterator-Muster setzt der Besucher nicht voraus, dass die zu besuchenden Objekte innerhalb einer Objektstruktur eine gemeinsame Basisklasse haben. Es wird nur eine passende `akzeptieren()`-Methode verlangt.

- **Sammeln von Informationen**

Das Besucher-Muster ermöglicht das Ansammeln von Zustandsinformationen der besuchten Objekte über mehrere Besuche hinweg.

- **Verbesserung der Wartbarkeit**

Muss eine Operation, die in einer Besucher-Klasse implementiert ist, angepasst werden, reicht es meist aus, die entsprechende Besucher-Klasse zu ändern, weil die Logik der Operation nicht über alle Element-Klassen verteilt ist.

- **Möglichkeit, Frameworks zu erweitern**

Steht ein Framework nur in Form einer Bibliothek zur Verfügung und nicht als Quellcode, so besteht das Vorgehen darin, von den Framework-Klassen abzuleiten und in den abgeleiteten Klassen jeweils eine `akzeptieren()`-Methode zu implementieren. Wirklich lohnenswert ist dieser Ansatz allerdings nur dann, wenn man dies durchführt, bevor Code geschrieben wird, der die Klassen des Frameworks verwendet, da sonst der gesamte, bereits existierende und möglicherweise getestete und freigegebene Code geändert werden muss.

#### 4.16.4.2 Nachteile

Folgende Nachteile werden gesehen:

- **Hoher Aufwand beim Hinzufügen von Element-Klassen**

Für jede neue zu besuchende Klasse `KonkretesElementY` muss eine neue `besuchen()`-Methode in der abstrakten Klasse `Besucher` definiert werden, die einen Parameter vom Typ der neuen Klasse hat. Ebenso müssen alle konkreten Besucher-Klassen um die Implementierung dieser Methode ergänzt werden. Das bedeutet, dass es schwierig ist, dieses Muster nachträglich zu implementieren.

- **Hoher Aufwand bei der nachträglichen Anwendung des Musters**

Wenn Element-Klassen bereits existieren und das Besucher-Muster nachträglich auf diese Element-Klassen angewendet werden soll, müssen zuerst alle Element-Klassen um eine `akzeptieren()`-Methode erweitert werden. Das bedeutet, dass es schwierig ist, dieses Muster nachträglich zu implementieren.

- **Overhead**

Durch das simulierte "double dispatch" in der Methode `akzeptieren()` entsteht zusätzlicher Aufwand, der die Performance verschlechtert.

- **Aufweichung der Kapselung privater Daten**

Ein konkreter Besucher braucht eine ganze Reihe von Attributen der besuchten Elemente. Er kann nur auf die öffentlichen Daten der Objekte zugreifen, die er besucht. Die Anwendung des Besucher-Musters kann daher dazu führen, dass man private Daten der Objekte öffentlich zugänglich macht.

### 4.16.5 Einsatzgebiete

Voraussetzung für den Einsatz des Besucher-Musters ist, dass die vorhandenen Element-Klassen nicht geändert werden sollen. Dieses Entwurfsmuster bietet sich an, wenn eine Objektstruktur mit vielen Element-Klassen und verschiedenen Schnittstellen zentral von einem Kontrollobjekt (Besucher-Objekt) bearbeitet werden soll, wobei eine einzelne Teil-Operation auf einem Objekt von dessen Daten abhängt. Da jede Besucher-Klasse geändert werden muss, wenn eine neue Element-Klasse für die Objektstruktur benötigt wird, ist es günstig, wenn sich die Menge der Element-Klassen der Objektstruktur so wenig wie möglich ändert. Neue Operationen – also Besucher-Klassen – können problemlos definiert werden.

Soll den Objekten einer Kompositum-Struktur eine neue Funktionalität hinzugefügt werden, die sich nicht nur auf ein einzelnes Objekt sondern auf alle Objekte der Struktur bezieht, dann müssen in der Regel ein oder mehrere Methoden in der Basisschnittstelle eingeführt werden. Eine Änderung an der Basisschnittstelle, wie z. B. das Hinzufügen einer neuen Methode, führt aber dazu, dass alle davon abgeleiteten Klassen potenziell ebenfalls geändert werden müssen.

Um diesem Effekt entgegenzuwirken, kann das Kompositum-Muster mit dem Entwurfsmuster Besucher kombiniert werden. Das Entwurfsmuster Besucher ermöglicht es, dass einer Kompositum-Struktur eine neue, objektübergreifende Funktionalität flexibel hinzugefügt werden kann, ohne die Klassen der Kompositum-Struktur ändern zu müssen. Diese Vorteile des Besucher-Musters können auch bei Objektstrukturen erzielt werden, die nach dem **Kompositum-Muster** (siehe Kapitel 4.7) aufgebaut wurden. Hierauf wurde bereits in Kapitel 4.16.3.3 eingegangen.

### 4.16.6 Ähnliche Entwurfsmuster

Ebenso wie ein Besucher realisiert ein **Dekorierer** eine neue Funktionalität. Das Besucher-Muster erlaubt es, zu einer Datenstruktur eine neue Funktion hinzuzufügen, die auf allen Objekten der Datenstruktur arbeitet. Mit dem Dekorierer-Muster können jedoch nur einzelne Objekte erweitert werden. Ein weiterer Unterschied ist, dass beim Besucher-Muster die zu besuchenden Objekte auf den Besuch "vorbereitet" sein müssen, dadurch dass sie eine entsprechende Methode dem Besucher zur Verfügung stellen.

Ein **Iterator** und ein Besucher sind sich insofern ähnlich, als dass sie sich über die Elemente einer Datenstruktur hinweg bewegen. Die Aufgabe eines Iterators beschränkt sich auf einen solchen Durchlauf der Datenstruktur, während ein Besucher zusätzlich während des Durchlaufs eine Funktionalität erbringt. Wie bereits erwähnt wurde, kann ein Besucher einen Iterator nutzen, um die Datenstruktur zu durchlaufen. Weiterhin setzt das Iterator-Muster voraus, dass die Objekte der Datenstruktur eine gemeinsame Basisklasse haben. Das Besucher-Muster verlangt hingegen nur, dass die Objekte der Datenstruktur eine passende `akzeptieren()`-Methode besitzen. Beim Besucher-Muster liegt außerdem der Fokus darauf, dass weitere Besucher mit anderer Funktionalität flexibel zu einer Datenstruktur hinzugefügt werden können.

## 4.17 Das Verhaltensmuster Iterator

### 4.17.1 Name/Alternative Namen

Iterator, Cursor (engl. cursor).

### 4.17.2 Problem

Mit einem Iterator soll nacheinander auf die einzelnen Objekte einer aus Objekten zusammengesetzten Datenstruktur in einer bestimmten Durchlaufstrategie zugegriffen werden, ohne dass der Aufbau der Datenstruktur für die Anwendung bekannt sein muss.

Das **Iterator-Muster** soll es erlauben, eine Datenstruktur in verschiedenen Durchlaufstrategien zu durchlaufen, ohne dass der Client den Aufbau der Datenstruktur kennt.



### 4.17.3 Lösung

Ein Iterator ist ein Objekt, mit dem von Objekt zu Objekt über alle Objekte einer Datenstruktur "iteriert"<sup>65</sup> werden kann. Ein Iterator zeigt nacheinander auf die einzelnen Objekte bzw. Elemente einer Datenstruktur. Der Name Iterator wird sowohl für ein iterierendes Objekt als auch für das entsprechende objektbasierte Verhaltensmuster verwendet.

Ein Iterator als Objekt trennt den Mechanismus des Traversierens<sup>66</sup> von der zu durchquerenden Datenstruktur. Diese Trennung ist der grundlegende Gedanke des Iterator-Musters. Denn dadurch braucht eine Anwendung den Aufbau einer Datenstruktur selbst nicht zu kennen.



Ein Iterator wandert über alle Elemente einer Datenstruktur nach einer bestimmten Durchlaufstrategie (Traversierungsart). Dabei besucht er jedes Element nur ein einziges Mal. Diese Beschreibung suggeriert, dass ein Iterator sequenziell auf die Elemente einer Datenstruktur zugreift. Ein sequenzieller Zugriff ist aber nicht zwingend vorgegeben – ein Iterator könnte auch wahlfrei auf die Elemente zugreifen.

<sup>65</sup> Iterieren bedeutet: es wird schrittweise auf jedes Objekt zugegriffen, das von der Datenstruktur referenziert wird.

<sup>66</sup> Mit Traversieren bezeichnet man das Durchlaufen einer Datenstruktur in einer bestimmten Reihenfolge.

Infolge der Trennung der Datenstruktur und des Mechanismus des Traversierens kann man Iteratoren für verschiedene **Traversierungsarten** schreiben, ohne dabei die Schnittstelle der Datenstruktur zu verändern



Die Realisierung der Traversierung wird aus der Datenstruktur herausgehalten. Dadurch wird die Formulierung der Datenstruktur einfacher. Eine Traversierung kann im Falle einer Liste zum Beispiel sequenziell vorwärts oder sequenziell rückwärts erfolgen. Bei einem Baum kann beispielsweise nach der Strategie Depth-First<sup>67</sup> traversiert werden.

Man unterscheidet externe und interne Iteratoren. Bei einem **externen Iterator** wird die Iteration vom Client selbst gesteuert, d. h., der Client muss das Weiterrücken des Iterators veranlassen. Ein **interner Iterator** rückt von selbst vor und kapselt so den Aufbau einer Iteration. Dadurch wird die Iteration wiederverwendbar. Sie muss also nicht wie bei einem externen Iterator stets neu programmiert werden.



Einem internen Iterator muss die Operation übergeben werden, die er während eines Durchlaufs ausführen soll. Interne Iteratoren werden ausführlich in "Design Patterns – Elements of Reusable Object-Oriented Software" [Gam95] der sogenannten "**Gang of Four**" (GoF) vorgestellt. Da interne Iteratoren den Aufbau der Iteratoren kapseln, ist von außen ihre Eigenschaft als Iterator nicht mehr erkennbar. Sie werden in dem vorliegenden Buch nicht weiter betrachtet.

Im Folgenden werden **externe Iteratoren** mit **sequenziellem Zugriff** behandelt. Solch ein Iterator hat grundsätzlich die beiden Methoden `next()` und `hasNext()`. Diese beiden Methoden sind typisch für externe Iteratoren. Die Methode `hasNext()` stellt fest, ob es noch ein nächstes Element gibt oder nicht, die Methode `next()` beschafft das nächste Element. Gibt es noch weitere Elemente in der Datenstruktur, gibt die Methode `hasNext()` als Ergebnis `true` zurück. Das folgende Bild zeigt die Trennung von Iterator und der zu durchquerenden Datenstruktur für den Fall eines externen Iterators:

<sup>67</sup> Depth-First bedeutet "Tiefe zuerst" – bildlich gesprochen geht man in einem Baum erst in die Tiefe, dann in die Breite. Das bedeutet, dass bei der Depth-First Traversierung ausgehend vom Startknoten ein Unterzweig eines jeden auftretenden Kindknotens bis zu dessen letzten Kindknoten durchlaufen wird und erst anschließend zurückgekehrt wird, um einen Nachbarknoten zu besuchen.



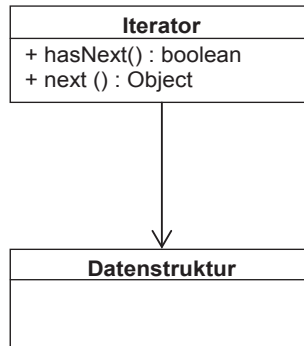


Bild 4-55 Trennung von Iterator und der zu durchquerenden Datenstruktur

Die in Bild 4-55 gezeigte Schnittstelle eines Iterators ist quasi minimal<sup>68</sup>. Die Schnittstelle kann von Anwendung zu Anwendung oder von Datenstruktur zu Datenstruktur variieren. Beispielsweise kann ein Iterator auch Methoden zum Zurücksetzen auf das vorherige Element oder zum Finden eines bestimmten Elements anbieten. Ein Iterator kann aber auch mit einer Methode `remove()` so erweitert werden, dass er die Objekte der Datenstruktur löschen kann.

#### 4.17.3.1 Klassendiagramm

Da die hier besprochenen **externen Iteratoren mit sequenziellem Zugriff** alle die gleiche Schnittstelle haben, wird ein Interface `IIterator` eingeführt, das von allen konkreten Iteratoren implementiert werden muss. Im folgenden Klassendiagramm wird aus Gründen der Übersichtlichkeit nur ein einziger konkreter Iterator gezeigt:

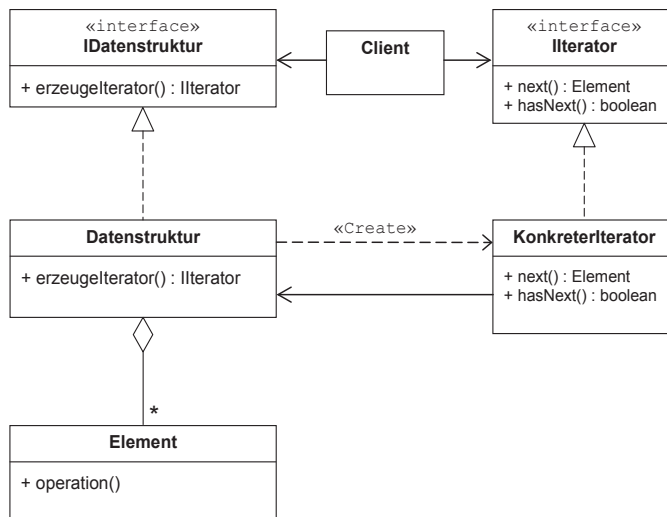


Bild 4-56 Klassendiagramm des Iterator-Musters

<sup>68</sup> Theoretisch könnten die beiden Methoden der Schnittstelle noch zu einer einzigen verschmolzen werden.

Der Aufbau einer Datenstruktur ist für die Anwendung irrelevant, solange die Anwendung zum Traversieren der Datenstruktur einen Iterator benutzt. Als Symbol für alle möglichen Datenstrukturen enthält das Klassendiagramm eine Klasse `Datenstruktur`. Zum besseren Verständnis des Musters muss aber die Funktionsweise eines Iterators gezeigt werden. Dazu wird zunächst eine sehr einfache Datenstruktur verwendet. Wie im Klassendiagramm in Bild 4-56 zu sehen ist, aggregiert die Klasse `Datenstruktur` eine Menge von Elementen beispielsweise in Form eines Arrays. Die Klasse `Element` steht hier stellvertretend für weitere Elementklassen, die von der Klasse `Element` abgeleitet sind. Die Klasse `Element` ist somit die Basisklasse aller Klassen, deren Objekte in der Datenstruktur vorkommen können. Diese Forderung ergibt sich daraus, dass die Methode `next()` im Interface `IIterator` als Ergebnistyp den Typ `Element` hat.

Eine Anwendung – hier Client genannt – nutzt das Interface `IIterator`. Er kann aber dadurch auf Grund des liskovschen Substitutionsprinzips jeden konkreten Iterator nutzen, dessen Klasse das Interface `IIterator` implementiert und dessen Verträge einhält.

Das Interface `IDatenstruktur` definiert den Kopf einer Methode `erzeugeIterator()`. Diese Methode muss in jeder Datenstruktur implementiert werden und einen zu der Datenstruktur passenden Iterator an den Aufrufer zurückliefern. Bei der Methode `erzeugeIterator()` handelt es sich um eine **Fabrikmethode** (siehe Kapitel 4.18).

Die Datenstruktur selbst muss keine Methoden zum Traversieren zur Verfügung stellen, sondern muss nur einen passenden Iterator anbieten. Der Iterator übernimmt die Aufgabe der Traversierung. Dadurch dass die Aufgabe der Traversierung aus der Datenstruktur herausgezogen wird (**Separation of Concerns**), wird die Implementierung der Datenstruktur vereinfacht. Wie das Klassendiagramm in Bild 4-56 zeigt, sind allerdings die Klasse `Datenstruktur` und die Klasse `KonkreterIterator` stark gekoppelt. Dafür muss der Client den Aufbau der Datenstruktur nicht kennen und muss sich nicht mehr um die Art und Weise der Traversierung der Datenstruktur kümmern.

Wie bereits erwähnt wurde, gibt es Datenstrukturen, bei denen verschiedene Traversierungsarten und damit verschiedene Iteratoren zum Einsatz kommen können. In diesem Falle kann die Methode `erzeugeIterator()` parametrisiert werden, damit ein Client über den Parameter den gewünschten Iterator auswählen kann. Alternativ kann eine Datenstruktur auch mehrere Methoden zum Erzeugen von Iteratoren zur Verfügung stellen. Hierauf wird im Folgenden nicht weiter eingegangen.

#### 4.17.3.2 Teilnehmer

##### IIterator

Das Interface `IIterator` definiert eine Methode `next()` zum Traversieren und zum Zugriff auf die Elemente der Datenstruktur sowie eine Methode `hasNext()` zum Überprüfen der Existenz eines nächsten Elements.

### KonkreterIterator

Die Klasse `KonkreterIterator` implementiert das Interface `IIterator` und verwaltet die aktuelle Position beim Durchqueren der Datenstruktur. Diese Klasse steht stellvertretend für verschiedene Klassen von konkreten Iteratoren, die zur Datenstruktur passen.

### IDatenstruktur

Das Interface `IDatenstruktur` definiert den Kopf der Methode `erzeugeIterator()` zum Erzeugen eines Iterators.

### Datenstruktur

Die Klasse `Datenstruktur` implementiert die Methode `erzeugeIterator()`, die vom Interface `IDatenstruktur` vorgegeben ist. Die Methode `erzeugeIterator()` gibt ein Objekt der zu der Klasse `Datenstruktur` passenden Klasse `KonkreterIterator` zurück.

### Element

Die Klasse `Element` ist die Basisklasse aller Klassen, deren Objekte in der Datenstruktur vorkommen können. Die Methode `operation()` steht stellvertretend für Operationen, die ein Client auf allen Elementen der Datenstruktur ausführen kann.

#### 4.17.3.3 Dynamisches Verhalten

Ein Client möchte eine Operation auf jedem Element einer Datenstruktur ausführen, kennt aber den Aufbau der Datenstruktur nicht. Die Datenstruktur erzeugt auf Anfrage durch den Client in der Methode `erzeugeIterator()` einen für diese Datenstruktur speziell implementierten Iterator und gibt diesen an den Client zurück. Alle weiteren Operationen, die das Traversieren der Datenstruktur betreffen, werden über den Iterator durchgeführt und nicht direkt auf der Datenstruktur selbst.

Der Client kann nun in einer Schleife durch Aufruf der Methode `next()` sich vom Iterator das nächste Element der Datenstruktur besorgen und die gewünschte Operation auf diesem Element ausführen – hier angedeutet durch den Aufruf einer Methode `operation()`. Die Schleife bricht ab, wenn der Aufruf der Methode `hasNext()` beim Iterator-Objekt als Ergebnis `false` liefert.

Das folgende Bild zeigt das Sequenzdiagramm für das Iterator-Muster:

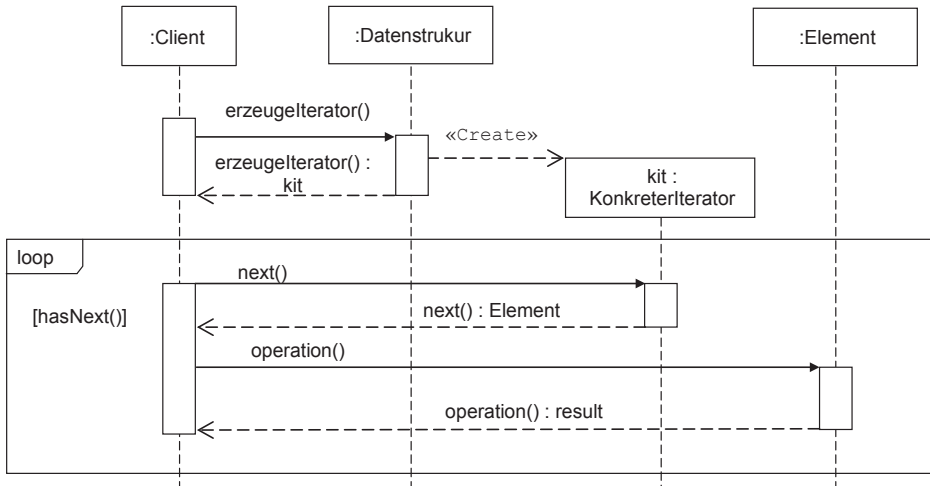


Bild 4-57 Sequenzdiagramm für das Iterator-Muster

Das Sequenzdiagramm zeigt nicht, wie das Iterator-Objekt und die Datenstruktur interagieren. Aus Sicht des Clients ist diese Zusammenarbeit auch irrelevant. Außerdem könnte diese Interaktion nur anhand einer konkreten Datenstruktur und einer konkreten Durchlaufstrategie dargestellt werden und nicht anhand einer nur schematischen Klassenstruktur des Musters.

Durch das Iterator-Muster ist nicht definiert, ob ein Iterator-Objekt für einen weiteren Durchlauf durch die Datenstruktur wiederbenutzt werden kann oder ob jedes Mal ein neues Iterator-Objekt erzeugt werden muss. Nach dem Erzeugen eines Iterator-Objekts "zeigt" nämlich die Methode `next()` quasi vor das erste Element der Datenstruktur und nach einem Durchlauf durch die Datenstruktur auf das letzte. Zur Wiederverwendung in einem weiteren Durchlauf müsste ein Iterator-Objekt also in seinen Initialzustand versetzt werden können.

#### 4.17.3.4 Programmbeispiel

In folgendem Programmbeispiel soll eine Liste bestehend aus Mitarbeitern erzeugt und darüber iteriert werden. Diese Liste wird zwar der Einfachheit halber mit Hilfe eines Arrays implementiert, die interne Struktur wird aber für eine Anwendung, die die Mitarbeiterliste über einen Iterator nutzt, nicht ersichtlich. Eine solche Anwendung des Iterator-Musters ist in der Klasse `TestClient` am Ende des Beispiels zu sehen.

Die Klasse `Mitarbeiter` spielt in diesem Beispiel die Rolle der Klasse `Element` aus der Beschreibung des Iterator-Musters. Die Details dieser Klasse sind aber aus Sicht des Iterator-Musters irrelevant. Die Klasse `Mitarbeiter` wurde daher für das Beispiel relativ einfach und übersichtlich gehalten:

```
// Datei: Mitarbeiter.java
class Mitarbeiter
{
    private String vorname;
    private String nachname;

    private double gehalt;
    private int persNr;

    private String abteilung;

    public Mitarbeiter (String vorname, String nachname, String
        abteilung, double gehalt, int persNr)
    {
        this.vorname = vorname;
        this.nachname = nachname;
        this.abteilung = abteilung;
        this.gehalt = gehalt;
        this.persNr = persNr;
    }

    // Methoden zum Setzen und Auslesen der Mitarbeiterattribute
    // Hier beispielhaft nur die Methode print():
    public void print()
    {
        System.out.println (vorname + " " + nachname + ", " +
            persNr + ", " + abteilung + ", " + gehalt);
    }
}
```

Das Interface `IIterator` definiert die Schnittstelle für einen Iterator passend zur Klasse `Mitarbeiter`:

```
// Datei: IIterator.java
interface IIterator
{
    public boolean hasNext();
    public Mitarbeiter next();
}
```

In der Klasse `MitarbeiterIterator` wird nun das Interface `IIterator` implementiert und die für die Datenstruktur `MitarbeiterArray` spezifische Funktionalität bereitgestellt:

```
// Datei: MitarbeiterIterator.java
class MitarbeiterIterator implements IIterator
{
    int size=0;
    int index=0;
    Mitarbeiter data[];
```

```

//Konstruktor mit Array und Anzahl der Elemente
public MitarbeiterIterator (Mitarbeiter ma[], int s)
{
    data = ma;
    size = s;
}

//Implementierung von hasNext()
public boolean hasNext()
{
    return index < size;
}

//Implementierung von next()
public Mitarbeiter next()
{
    return data[index++];
}
}

```

Das Interface `IDatenstruktur` definiert die Schnittstelle für iterierbare Datenstrukturen:

```

// Datei: IDatenstruktur.java
// Definiert das Interface fuer Datenstrukturen, die mit einem
// Iterator durchlaufen werden sollen
interface IDatenstruktur
{
    public IIterator erzeugeIterator();
}

```

Nun folgt die konkrete Datenstruktur in der Klasse `MitarbeiterArray`, die das Interface `IDatenstruktur` implementiert. Zur Vereinfachung wird ein statisches Array mit fester Größe verwendet, um die Daten zu halten:

```

// Datei: MitarbeiterArray.java
// Implementierung von IDatenstruktur
class MitarbeiterArray implements IDatenstruktur
{
    final int max = 10;
    Mitarbeiter[] data = new Mitarbeiter[max];
    int index = 0;

    //Implementierung der erzeugeIterator()-Funktion
    //Gibt als Rueckgabewert den fuer diese Datenstruktur spezifischen
    //Iterator zurueck
    public IIterator erzeugeIterator()
    {
        return new MitarbeiterIterator (data, index);
    }
}

```

```

//Funktion zum Hinzufuegen eines neuen Mitarbeiters
public void add (Mitarbeiter ma)
{
    if(index < max - 1)
    {
        data[index++] = ma;
    }
}
}

```

Im statischen Initialisierungsblock der Klasse `TestClient` wird eine Liste von Mitarbeitern in Form eines Objekts der Klasse `MitarbeiterArray` erzeugt und gefüllt. In der sich anschließenden `main()`-Methode wird über diese Liste iteriert und es werden die Details des jeweiligen Mitarbeiters ausgegeben. Diese `main()`-Methode stellt sozusagen die Anwendung dar. Der Quellcode dieser Methode zeigt, dass durch den Einsatz eines Iterators keine Kenntnisse über den Aufbau der konkreten Datenstruktur `maListe` benötigt werden. Im Folgenden die Klasse `TestClient`:

```

// Datei: TestClient.java
public class TestClient
{
    private static MitarbeiterArray maListe;

    static
    {
        //Mitarbeiterliste erzeugen
        maListe = new MitarbeiterArray();

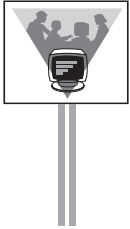
        //Mitarbeiter zur Liste hinzufuegen
        maListe.add (new Mitarbeiter ("Herman", "Hinz",
                                     "MMI-Entwicklung", 3250.00, 1));
        maListe.add (new Mitarbeiter ("Thomas", "Kunz",
                                     "MMI-Entwicklung", 3050.00, 2));
        maListe.add (new Mitarbeiter ("Heinz", "Mueller",
                                     "Unit Tests", 3450.00, 3));
        maListe.add (new Mitarbeiter ("Hans", "Maier",
                                     "Unit Tests", 3400.00, 4));
        maListe.add (new Mitarbeiter ("Max", "Muster",
                                     "Unit Tests", 3500.00, 5));
        maListe.add (new Mitarbeiter ("Peter", "Schmidt",
                                     "Requirements Engineering", 3700.00, 6));
    }

    public static void main(String[] args)
    {
        //Iterator holen und Liste durchlaufen
        IIterator iter = maListe.erzeugeIterator();

        while (iter.hasNext())
        {
            Mitarbeiter elem = iter.next();

```

```
        elem.print();  
    }  
}
```



Hier das Protokoll des Programmlaufs:

Hermann Hinz, 1, MMI-Entwicklung, 3250.0  
Thomas Kunz, 2, MMI-Entwicklung, 3050.0  
Heinz Mueller, 3, Unit Tests, 3450.0  
Hans Maier, 4, Unit Tests, 3400.0  
Max Muster, 5, Unit Tests, 3500.0  
Peter Schmidt, 6, System Engineering, 3700.0

## 4.17.4 Bewertung

### 4.17.4.1 Vorteile

Folgende Vorteile ergeben sich durch die Verwendung des Musters:

- Der Quellcode einer Anwendung zur Traversierung einer Datenstruktur ist unabhängig von der konkreten Datenstruktur. Man kann also auf Objekte verschiedener Datenstrukturen in einheitlicher Weise zugreifen und sie durchlaufen.
- Der Aufbau der Datenstruktur, die die zu durchlaufenden Objekte enthält, wird nicht sichtbar.
- Die Iteration erfolgt in einem eigenen Objekt.
- Die Datenstruktur kann je nach Iterator auf verschiedene Arten durchlaufen werden.
- Mehrere Iteratoren können gleichzeitig über eine Datenstruktur laufen, da jeder den Zustand der Traversierung für sich selbst verwaltet.

### 4.17.4.2 Nachteile

Durch die Verwendung des Musters können auch folgende Nachteile entstehen:

- Die konkreten Datenstrukturobjekte müssen ihren Iterator selber erzeugen. Datenstruktur und Iterator sind dadurch stark gekoppelt.
- Externe Iteratoren sind potentiell anfällig gegen Änderungen an der Datenstruktur während eines Durchlaufs: Wird beispielsweise ein Element eingefügt, kann es der Iterator "übersehen". Wird ein Element aus der Datenstruktur entfernt, besucht es der Iterator möglicherweise trotzdem noch. Sogenannte **robuste Iteratoren** sind stabil gegenüber solchen Änderungen der Datenstruktur, aber aufwendig zu implementieren.



### 4.17.5 Einsatzgebiete

Der Einsatz des Iterator-Musters ist immer dann sinnvoll, wenn ein einheitlicher Zugriff auf Datenstrukturen mit verschiedenen Typen wie Arrays, Bäume oder Listen zur Verfügung gestellt werden soll.

Iteratoren können auf rekursiven Datenstrukturen, die durch das **Kompositum-Muster** erzeugt wurden, zum Einsatz kommen.

Mit einem Iterator kann die Objektstruktur des **Besucher-Musters** durchlaufen werden.

Die Datenstrukturen, die von der Standard Template Library (STL) von C++ zur Verfügung gestellt werden oder als Collections in der Java Bibliothek `java.util` implementiert sind, bieten Iteratoren an, mit deren Hilfe die Datenstrukturen durchlaufen werden können.

#### Iteratoren in Java

Das Iterator-Muster, das von Java unterstützt wird, soll im Folgenden kurz skizziert werden, da dessen Möglichkeiten über den bisher im Kapitel vorgestellten Mechanismus hinausgehen (siehe dazu auch Kapitel 18 in [Hei10]).

Betrachtet man sich das Interface `IIterator` im Klassendiagramm Bild 4-56 genauer, so stellt man fest, dass es den Typ `Element` benutzt. Dieses Interface muss also für jeden Elementtyp neu definiert werden. In Java werden stattdessen die Möglichkeiten der generischen Programmierung (im Englischen kurz mit **Generics** bezeichnet) eingesetzt. Die Bibliothek `java.util` enthält das Interface `Iterator<E>`, das bis auf den generischen Typparameter ein ähnliches Aussehen wie das hier im Buch vorgestellte Interface `IIterator` hat und die Methoden `next()` und `hasNext()` definiert.

Dieses Interface `Iterator<E>` könnte man auch beim vorhergehenden Programmbeispiel nutzen und in der Klasse `MitarbeiterArray` den Iterator wie folgt erzeugen:

```
public Iterator<Mitarbeiter> erzeugeIterator()
{
    return new MitarbeiterIterator(data, index);
}
```

Die Klasse `MitarbeiterIterator` muss nun natürlich das Interface `Iterator<Mitarbeiter>` implementieren<sup>69</sup>. Weiterhin muss die Nutzung des erzeugten Iterators in der Klasse `TestClient` leicht angepasst werden:

```
Iterator<Mitarbeiter> iter = maListe.erzeugeIterator();
```

<sup>69</sup> Da das Interface `Iterator<E>` eine Methode `remove()` definiert, muss die Klasse `MitarbeiterIterator` noch um eine leere Methode `remove()` ergänzt werden, damit das modifizierte Beispiel lauffähig wird. Der vollständige Quellcode des modifizierten Beispiels ist auf dem begleitenden Webaufttritt zu finden.

Java definiert im Zusammenhang mit Iteratoren noch ein weiteres Interface: das Interface `Iterable<E>` in der Bibliothek `java.lang`. Dieses Interface entspricht in seiner Rolle in etwa dem Interface `IDatenstruktur`. Klassen, die das Interface `Iterable<E>` implementieren, müssen eine Methode `iterator()` zur Verfügung stellen, die als Ergebnis einen Iterator vom Typ `Iterator<E>` liefert. Der Clou an dem Interface `Iterable` ist, dass über Datenstrukturen, die dieses Interface implementieren, mit einer sogenannten `foreach`-Schleife iteriert werden kann.

Wird also im Programmbeispiel die Klasse `MitarbeiterArray` mit dem Zusatz `implements Iterable<Mitarbeiter>` versehen und die Methode `erzeugeIterator()` umbenannt in `iterator()`, dann könnte die `main()`-Methode der Klasse `Testclient` alternativ auch wie folgt formuliert werden:

```
public static void main (String[] args)
{
    for (Mitarbeiter elem : maListe)
        elem.print();
}
```

Hier ist nun die Nutzung eines Iterators komplett aus dem Quelltext verschwunden. Sollen aber zusätzliche Funktionen des Iterators benutzt werden, wie etwa Elemente einfügen oder entfernen, dann ist die explizite Schreibweise zur Nutzung eines Iterators weiterhin erforderlich.

#### 4.17.6 Ähnliche Entwurfsmuster

Ein Iterator und ein Besucher des **Besucher-Musters** sind sich insofern ähnlich, als dass sie sich über die Elemente einer Datenstruktur hinweg bewegen. Die Aufgabe eines Iterators beschränkt sich auf einen solchen Durchlauf der Datenstruktur, während ein Besucher zusätzlich während des Durchlaufs eine zusätzliche Funktionalität erbringt. Wie bereits erwähnt wurde, kann ein Besucher einen Iterator nutzen, um die Datenstruktur zu durchlaufen. Weiterhin setzt das Iterator-Muster voraus, dass die Objekte der Datenstruktur eine gemeinsame Basisklasse haben. Das Besucher-Muster verlangt hingegen nur, dass die Objekte der Datenstruktur eine passende `akzeptieren()`-Methode besitzen. Beim Besucher-Muster liegt außerdem der Fokus darauf, dass weitere Besucher mit anderer Funktionalität flexibel zu einer Datenstruktur hinzugefügt werden können.

## 4.18 Das Erzeugungsmuster Fabrikmethode

### 4.18.1 Name/Alternative Namen

Fabrikmethode (engl. factory method), Virtueller Konstruktor.

### 4.18.2 Problem

Eine wiederverwendbare Anwendung wie ein Framework soll nur Basisklassen bzw. abstrakte Basisklassen oder Interfaces enthalten. Das bedeutet, dass eine wiederverwendbare Anwendung Objekte nicht hart codiert erzeugen kann, da sie die eigentlichen Klassen ja gar nicht kennt. Diese Klassen sollen erst zur Laufzeit von dem Programm, das die wiederverwendbare Anwendung benutzt, zur Verfügung gestellt werden.

Eine Klasse soll ein Objekt erzeugen, dessen Typ sie nicht kennt, bzw. dessen Typ erst zur Laufzeit des Programms bekannt ist. Die erzeugende Klasse kennt nur die Basisklasse des zu erzeugenden Objekts zur Kompilierzeit, weiß aber nicht, von welcher Unterklasse das entsprechende Objekt zur Laufzeit im lauffähigen Programm später sein soll.



In stark typisierten Sprachen wie Java wird beim Aufruf des `new`-Operators der Typ des zu erzeugenden Objekts im Programmtext statisch festgelegt. Es gibt keine Möglichkeit, den Typ eines erzeugten Objekts im Nachhinein zu verändern. Zur Erzeugung und Initialisierung eines Objektes wird in Java der `new`-Operator mit dem Konstruktor, der dem Klassennamen entspricht, verwendet. Diese Vorgehensweise ist natürlich sehr inflexibel. Die Klasse eines zu erzeugenden Objekts soll hier eben nicht im Programmtext festgelegt werden. Damit kann beispielsweise in Java die Erzeugung nicht direkt mit `new` erfolgen. Ein Programm soll überdies stabil bleiben, auch wenn sich der Typ des zu erzeugenden Objekts ändert.

Der alternative Name "Virtueller Konstruktor" beschreibt das Problem sehr gut. Er stammt aus dem Kontext der Sprache C++. In C++ müssen alle Methoden, die polymorph sein sollen, mit dem Schlüsselwort `virtual` gekennzeichnet werden. Daher spricht man auch von virtuellen Methoden. Wann immer die Erzeugung eines Objektes polymorph sein soll – also erst zur Laufzeit entschieden werden soll, welcher Konstruktor zu wählen ist, um ein Objekt des entsprechenden Typs zu erzeugen – hätte man folglich gerne einen virtuellen Konstruktor. Konstruktoren können aber nicht überschrieben werden und sind daher nicht polymorph. Das Muster Fabrikmethode stellt also quasi einen Ersatz für einen virtuellen Konstruktor dar: die Erzeugung eines Objekts wird in eine Methode verpackt, die polymorph ist. Diese Methode wird in der Lösung als Fabrikmethode bezeichnet.

Das **Fabrikmethode-Muster** soll es erlauben, dass die Erzeugung einer konkreten Instanz in der Methode einer Unterklasse gekapselt wird.



### 4.18.3 Lösung

Das Muster Fabrikmethode basiert auf einem abstrakten Grundgerüst, das von konkreten Klassen überschrieben wird und genutzt werden kann. Das Muster Fabrikmethode verwendet in der **höheren, abstrakten Ebene** eine Klasse `Erzeuger` und eine Klasse `Produkt`. Die Klasse `Erzeuger` enthält zur Erzeugung von Produkten eine Fabrikmethode, die abstrakt bleiben muss, da der konkrete Produkttyp noch nicht bekannt ist.

Auf der **tieferen, konkreten Ebene** gibt es **konkrete Produkte** und **konkrete Erzeuger**. Eine Unterklasse `KonkreterErzeugerX` der Klasse `Erzeuger` überschreibt die abstrakte Fabrikmethode so, dass in der überschreibenden Fabrikmethode ein Objekt der Klasse `KonkretesProduktX`, das von der abstrakten Klasse `Produkt` abgeleitet ist, erzeugt wird. Die Erzeugung eines konkreten Produkts wird also auf eine konkrete Unterklasse der Klasse `Erzeuger` und deren konkrete Fabrikmethode, die das gewünschte Produkt erzeugt, verlagert.

Die Erzeugung von Objekten wird an die entsprechende Unterklasse delegiert, die das Wissen über die jeweilige Klasse der zu erzeugenden Objekte besitzt. Eine Klasse einer höheren Ebene ist damit nicht mehr abhängig von der Ausprägung einer Klasse einer tieferen Ebene (**Dependency Inversion**). Die konkreten Klassen können damit zu einem späteren Zeitpunkt geschrieben werden.



Diese Vorgehensweise erlaubt es, eine wiederverwendbare Anwendung in Form einer abstrakten Basisklasse oder Schnittstelle sowohl für das Produkt als auch für die Erzeuger-Klasse zu schreiben und mit davon abgeleiteten Klassen auszuführen. Erst zur Laufzeit des Programms, das die wiederverwendbare Anwendung nutzt, wird entschieden, welche Unterklasse der Klasse `Erzeuger` instanziiert werden soll, damit diese nach dem liskovschen Substitutionsprinzip an die Stelle eines Objekts der Klasse `Erzeuger` tritt, um ein konkretes Produkt zu erzeugen. Beim Erzeugungsmuster Fabrikmethode legen Unterklassen die zu erzeugenden Objekte statisch fest. Dieses Muster zählt somit zu den klassenbasierten Mustern.

#### 4.18.3.1 Klassendiagramm

Das Fabrikmethode-Muster setzt voraus, dass die zu erzeugenden Produkte eine einheitliche Schnittstelle haben. Im Folgenden wird dazu eine abstrakte Klasse namens `Produkt` eingeführt. Sie muss nicht unbedingt abstrakt sein, sie muss nur die Basis-

klasse aller konkreten Produkte sein. Es könnte auch eine Schnittstelle `IProdukt` sein, die von den konkreten Produkten implementiert wird. Für jede Produktklasse wird eine Klasse `KonkretesProduktX` eingeführt. Von diesen konkreten Produktklassen sollen Objekte erzeugt werden können.

Die Klasse `Erzeuger` definiert eine sogenannte Fabrikmethode, die dem Muster den Namen gab. In einer Fabrikmethode werden Objekte erzeugt und als Ergebnis zurückgegeben. In der Klasse `Erzeuger` wird nur die Schnittstelle der Fabrikmethode definiert: Der Rückgabotyp der Fabrikmethode ist vom Typ `Produkt` und die Fabrikmethode ist abstrakt, da hier der Typ des zu erzeugenden Produkts noch nicht bekannt ist.

Konkrete Produkte werden durch den Aufruf der überschreibenden Fabrikmethode einer Unterklasse der Klasse `Erzeuger` geschaffen. Die Fabrikmethode einer Unterklasse der Klasse `Erzeuger` kapselt die Erzeugung eines entsprechenden konkreten Produkts.

Für jede Produktklasse `KonkretesProduktX` wird eine Unterklasse `KonkreterErzeugerX` der Klasse `Erzeuger` eingeführt, welche die abstrakte Fabrikmethode überschreibt.



Somit können verschiedenartige Erzeugungsprozesse durchgeführt werden. Die Anwendung kennt nur die abstrakten Klassen und kennt die zu generierende konkrete Ausprägung des Produkts nicht. Der Typ der zu erzeugenden Objekte ist also zur Kompilierzeit nicht bekannt.

Das folgende Bild zeigt das Klassendiagramm des Fabrikmethode-Musters:

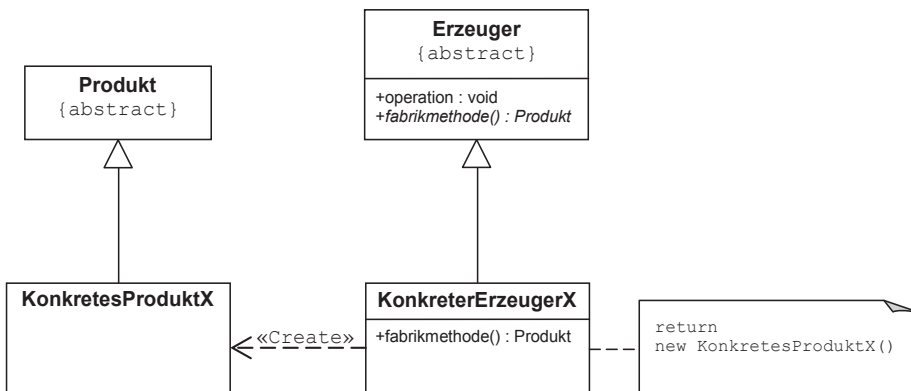


Bild 4-58 Klassendiagramm des Fabrikmethode-Musters mit abstrakten Klassen

Eine Anwendung des Fabrikmethode-Musters, die im Klassendiagramm nicht dargestellt ist, besitzt typischerweise eine Referenz vom Typ des abstrakten Erzeugers. Wie diese Referenz gesetzt wird, ist durch das Muster nicht festgelegt. Zeigt die Referenz zur Laufzeit der Anwendung auf einen konkreten Erzeuger und wird die Fabrikmethode des referenzierten konkreten Erzeugers aufgerufen, so wird die entsprechende überschreibende Fabrikmethode aufgerufen. Diese erzeugt nun die entsprechenden kon-

kreten Produkte, die bei Einhaltung der Verträge in der Anwendung nach dem liskovschen Substitutionsprinzip an die Stelle eines abstrakten Produkts treten.

#### 4.18.3.2 Teilnehmer

Folgende Klassen sind an dem Entwurfsmuster Fabrikmethode beteiligt:

##### Produkt

Die abstrakte Klasse `Produkt` definiert die Schnittstelle der Objekte, die durch die Fabrikmethode erzeugt werden. Alle konkreten Produkte müssen von der Klasse `Produkt` abgeleitet sein und müssen damit diese Schnittstelle implementieren.

##### KonkretesProduktX

Die Klasse `KonkretesProduktX` ist von der abstrakten Klasse `Produkt` abgeleitet und stellt ein zu erzeugendes konkretes Produkt dar. Typischerweise gibt es mehrere Klassen von konkreten Produkten.

##### Erzeuger

Die Klasse `Erzeuger` definiert eine abstrakte Fabrikmethode `fabrikmethode()`, die eine Referenz auf ein Objekt vom Typ `Produkt` zurückgibt. Diese Methode kann auch von einem Objekt der Klasse `Erzeuger` selbst aufgerufen werden (beispielsweise in der Methode `operation()`).

##### KonkreterErzeugerX

Die Klasse `KonkreterErzeugerX` überschreibt die Methode `fabrikmethode()` der Klasse `Erzeuger`, um ein neues Objekt vom Typ `KonkretesProduktX` zu erzeugen und eine Referenz darauf zurückzugeben. Die zurückgegebene Referenz ist zwar vom Typ `Produkt`, kann aber auf ein Objekt vom Typ `KonkretesProduktX` verweisen (liskovsches Substitutionprinzip).

#### 4.18.3.3 Dynamisches Verhalten

Das folgende Bild gibt ein Beispiel für die Erzeugung konkreter Produkte:

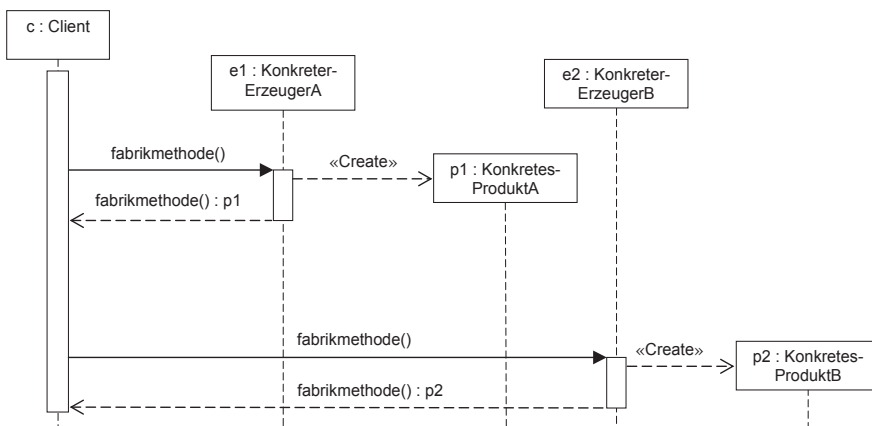


Bild 4-59 Sequenzdiagramm Fabrikmethode

Im Bild 4-59 ruft ein Client die Methode `fabrikmethode()` des ihm bekannten Erzeugers `e1` auf. Daraufhin wird vom Objekt `e1` der Klasse `KonkreterErzeugerA` eine Instanz vom Typ `KonkretesProduktA` erzeugt und an den Client zurückgegeben. Wird zu einem späteren Zeitpunkt vom Client beispielsweise die Fabrikmethode des Objekts `e2` der Klasse `KonkreterErzeugerB` aufgerufen, wird eine Instanz vom Typ `KonkretesProduktB` erzeugt. Je nachdem, welcher konkrete Erzeuger verwendet wird, werden also verschiedene Objekte erzeugt.

Der Client steht im Bild 4-59 stellvertretend für eine Anwendung (beispielsweise für ein Framework), die konkrete Produkte flexibel erzeugen und nutzen will. Er ist nicht Bestandteil des Musters. Das Muster lässt es auch offen, wie ein Client Kenntnis über einen konkreten Erzeuger erlangt.

#### 4.18.3.4 Programmbeispiel

In diesem Programmbeispiel entsprechen die Klassennamen den genannten Teilnehmern. Die abstrakte Klasse `Erzeuger` definiert mit der `Fabrikmethode` die Schnittstelle zur Erzeugung von konkreten Produkten:

```
// Datei: Erzeuger.java
public abstract class Erzeuger
{
    // Kopf der Fabrikmethode
    public abstract Produkt erzeugeProdukt();
}
```

In der Unterklasse `KonkreterErzeugerA` werden Objekte der Klasse `KonkretesProduktA` erzeugt:

```
// Datei: KonkreterErzeugerA.java
public class KonkreterErzeugerA extends Erzeuger
{
    public Produkt erzeugeProdukt() // Ueberschreiben der Fabrikmethode
    {
        return new KonkretesProduktA();
    }
}
```

In der Unterklasse `KonkreterErzeugerB` werden Objekte der Klasse `KonkretesProduktB` erzeugt:

```
// Datei: KonkreterErzeugerB.java
public class KonkreterErzeugerB extends Erzeuger
{
    public Produkt erzeugeProdukt() // Ueberschreiben der Fabrikmethode
    {
        return new KonkretesProduktB();
    }
}
```

Die abstrakte Klasse `Produkt` deklariert eine abstrakte Methode `print()` zur Ausgabe von Informationen über ein Produkt:

```
// Datei: Produkt.java
public abstract class Produkt // abstrakte Klasse
{
    abstract void print(); // abstrakte Methode
}
```

Die Klasse `KonkretesProduktA` stellt eine konkrete Klasse des abstrakten Produkts dar und muss die Methode `print()` implementieren:

```
// Datei: KonkretesProduktA.java
public class KonkretesProduktA extends Produkt
{
    private String x = "A";

    void print()
    {
        System.out.println ("x = " + x);
    }
}
```

Die Klasse `KonkretesProduktB` stellt eine andere konkrete Klasse des abstrakten Produkts dar und muss ebenfalls die Methode `print()` implementieren:

```
// Datei: KonkretesProduktB.java
public class KonkretesProduktB extends Produkt
{
    private String x = "B";

    void print()
    {
        System.out.println ("x = " + x);
    }
}
```

In der Klasse `Test` werden zwei konkrete Erzeuger instanziiert, mit deren Hilfe konkrete Produkte durch die Fabrikmethode erstellt werden:

```
// Datei: Test.java
public class Test
{
    public static void main (String args[])
    {
        Erzeuger erzeuger;
        Produkt produkt;

        System.out.println ("Hallo");

        erzeuger = new KonkreterErzeugerA();
        produkt = erzeuger.erzeugeProdukt();
        produkt.print();

        erzeuger = new KonkreterErzeugerB();
        produkt = erzeuger.erzeugeProdukt();
    }
}
```



```
    produkt.print();  
}  
}
```



Hier das Protokoll des Programmlaufs:

```
Hallo  
x = A  
x = B
```

Bei diesem Client handelt es sich, wie der Name schon sagt, um einen Test. Bei einer "richtigen" Anwendung wäre es eher untypisch, dass sie sich ihre Erzeuger nach Belieben selbst erzeugt. Die Anwendung sollte ja gerade in der Erzeugung von Produkten flexibel bleiben. In der Regel besitzt ein Client nur eine Referenz vom Typ der abstrakten Erzeuger-Klasse. Das Muster lässt offen, wie diese Referenz so gesetzt werden kann, dass sie auf ein Objekt einer konkreten Erzeuger-Klasse zeigt. Hier kann beispielsweise die Technik der **Dependency Injection** (siehe Kapitel 1.10.2) eingesetzt werden.

## 4.18.4 Bewertung

### 4.18.4.1 Vorteile

Folgende Vorteile werden gesehen:

- Eine Klasse muss die Klassen der Objekte, die sie erzeugen muss, nicht von vornherein kennen. Transparent für das Programm können durch Austausch der unabhängigen konkreten Erzeuger andere konkrete Produkte erzeugt werden, d. h., es können verschiedenartige konkrete Produkte generiert werden. Eine explizite Erzeugung der konkreten Produkte ist nicht in der Anwendung enthalten. Der Einsatz von Fabrikmethoden ist damit flexibler als explizite Erzeugungsanweisungen.
- Sollen die Basisklassen nicht nur abstrakte Methoden enthalten, kann man in der Basisklasse `Erzeuger` eine Default-Erzeugung definieren. Dadurch sind die Unterklassen frei, ob sie eine eigene überschreibende Fabrikmethode bereitstellen oder nicht.
- Durch die Verwendung von Schnittstellen (Interfaces bzw. abstrakte Klassen) für Produkte kann das Einbinden von "falschen" anwendungsspezifischen Klassen verhindert werden.
- Basierend auf den Schnittstellen (Interfaces bzw. abstrakten Klassen) für Produkte kann ein Framework bzw. eine Klassenbibliothek entwickelt werden, welches bzw. welche Produkte bearbeitet, auch wenn noch gar keine konkreten Produktklassen vorhanden sind. Eine Anwendung des Frameworks kann dann konkrete Produkte, die die abstrakten Schnittstellen implementieren, in eigenen spezifischen Unterklassen erzeugen, ohne Operationen der Oberklassen des Frameworks abändern zu müssen (Dependency Inversion). Ein solches Framework ist also unabhängig von der Ausprägung der konkreten Produkte.

#### 4.18.4.2 Nachteile

Folgende Nachteile werden gesehen:

- Für jede neue Produktklasse muss eine weitere Unterklasse eingeführt werden, um die Fabrikmethode entsprechend zu redefinieren. Damit erhöht sich die Komplexität.
- Aus Performance-Gründen kann es sinnvoller sein, Objekte direkt zu erzeugen und nicht die Fabrikmethode einzusetzen.

#### 4.18.5 Einsatzgebiete

Es müssen Objekte einer Klasse erzeugt werden, deren Typ von vornherein nicht bekannt ist. Dieses Erzeugungsmuster wird in vielen Frameworks (Klassenbibliotheken) eingesetzt.

Fabrikmethoden werden oftmals im Muster **Schablonenmethode** eingesetzt. Fabrikmethoden spielen in diesen Fällen dann die Rolle von Einschubmethoden, die aus Schablonenmethoden heraus aufgerufen werden. Wird beispielsweise die Fabrikmethode der abstrakten Erzeugerklasse in der Methode `operation()` der Erzeugerklasse selbst (siehe Bild 4-58) aufgerufen, dann stellt in dieser Situation die Fabrikmethode eine Einschubmethode und die Methode `operation()` eine Schablonenmethode aus dem Muster Schablonenmethode dar.

Die Methode `erzeugeIterator()` aus Kapitel 4.17, mit der eine Datenstruktur einen zu ihr passenden **Iterator** erzeugt, ist ein Beispiel für den Einsatz des Erzeugungsmusters Fabrikmethode.

Eine **Abstrakte Fabrik** verwendet in der Regel Fabrikmethoden.

#### 4.18.6 Ähnliche Entwurfsmuster

Die **Abstrakte Fabrik** hat im Gegensatz zu der Fabrikmethode eine Produktfamilie und nicht ein einzelnes Produkt zum Inhalt. Die Erzeugung eines konkreten Objekts erfolgt in beiden Fällen in Unterklassen.

## 4.19 Das Erzeugungsmuster Abstrakte Fabrik

### 4.19.1 Name/Alternative Namen

Abstrakte Fabrik (engl. abstract factory, kit oder toolkit).

### 4.19.2 Problem

Die Problemstellung soll zunächst anhand von zwei Beispielen verdeutlicht werden: Im ersten Beispiel werden Schrauben und Muttern, die es in unterschiedlichen Größen und Gewindearten gibt, produziert und verpackt. Da es aber keinen Sinn macht, 10-er Schrauben und 6-er Muttern miteinander zu verpacken, sollen immer nur passende Produkte – also 6-er Schrauben und 6-er Muttern bzw. 10-er Schrauben und 10-er Muttern – erzeugt werden können. 6-er Schrauben und 6-er Muttern bilden in diesem Beispiel eine zusammengehörige Produktgruppe, eine andere Produktgruppe besteht aus 10-er Schrauben und 10-er Muttern. Als zweites Beispiel wird eine Anwendung betrachtet, die mehrere Objekte benötigt, deren Klassen betriebssystemabhängig implementiert werden müssen. Um diese Anwendung lauffähig zu machen, müssen alle betriebssystemabhängigen Klassen in der richtigen Variante instanziiert werden. Es muss ausgeschlossen werden, dass eine Klasse beispielsweise in der Linux-Variante instanziiert wird und eine andere hingegen in der Windows-Variante.

Eine Menge oder Gruppe von Produkten, die miteinander verwandt sind, zueinander passen oder – allgemein gesprochen – voneinander abhängig sind, wird in diesem Muster als **Produktfamilie** bezeichnet.

Es soll gewährleistet werden, dass immer nur Produkte, die zu derselben Produktfamilie gehören, erzeugt werden. Die angestrebte Lösung soll flexibel sein, d. h., dass eine Anwendung selber nicht geändert werden muss, wenn eine andere Variante der Produktfamilie erzeugt werden soll.

### 4.19.3 Lösung

Damit die Lösung wie gefordert flexibel wird, enthält das Entwurfsmuster Abstrakte Fabrik eine abstrakte und eine konkrete Ebene. Die folgende Beschreibung des Musters verwendet auf der abstrakten Ebene Referenzen auf Interfaces.

Die Abstraktionen können aber genauso gut mittels abstrakter Klassen definiert werden. Die konkreten Klassen müssen dann von den entsprechenden abstrakten Klassen abgeleitet werden, statt die Interfaces zu realisieren.

Das Client-Programm kennt zur Kompilierzeit nur Referenzen auf die Abstrakte Fabrik und auf die abstrakten Produkte. Jede konkrete Fabrik erzeugt eine jeweils andere Produktfamilie von konkreten Produkten. Je nachdem, welche konkrete Fabrik zur Laufzeit an die Stelle der Abstrakten Fabrik tritt, wird die eine oder andere konkrete Produktfamilie erzeugt.

Das Muster Abstrakte Fabrik erlaubt es, dass durch Wahl der entsprechenden konkreten Fabrik eine zu erzeugende Produktfamilie zur Laufzeit ausgewählt werden kann.



Das Entwurfsmuster Abstrakte Fabrik ist ein objektbasiertes Muster, da die Erzeugung einer Produktfamilie in einem einzigen Objekt gekapselt ist. Um eine andere Produktfamilie zu erzeugen, muss dieses Objekt ausgetauscht werden.

#### 4.19.3.1 Klassendiagramm

Das Klassendiagramm in Bild 4-60 enthält das Interface `IAbstrakteFabrik`, das die Schnittstelle für die Erzeugung der Produkte einer Produktfamilie darstellt. Zusätzlich dazu gibt es konkrete Fabriken, die das Interface `IAbstrakteFabrik` implementieren. Jede dieser konkreten Fabriken erzeugt Produkte einer bestimmten Produktfamilie. Im vorliegenden Fall erzeugt z. B. die Klasse `KonkreteFabrik1` die Produkte der Klassen `KonkretesProduktA1` und `KonkretesProduktB1`. Diese beiden Produkte gehören also zu einer Produktfamilie. Die einzelnen Erzeugungsmethoden des Musters Abstrakte Fabrik wie beispielsweise die Methoden `erzeugeProduktA()` und `erzeugeProduktB()` können als **Fabrikmethoden** (siehe Kapitel 4.18) angesehen werden. Ähnlich wie beim Muster Fabrikmethode wird der Typ der zu erzeugenden Objekte in Implementierungsklassen festgelegt. Beim Muster Abstrakte Fabrik sind es die konkreten Fabriken, die die konkreten Objekte erzeugen und damit den Typ der zu erzeugenden Produkte festlegen. Hier das Klassendiagramm:

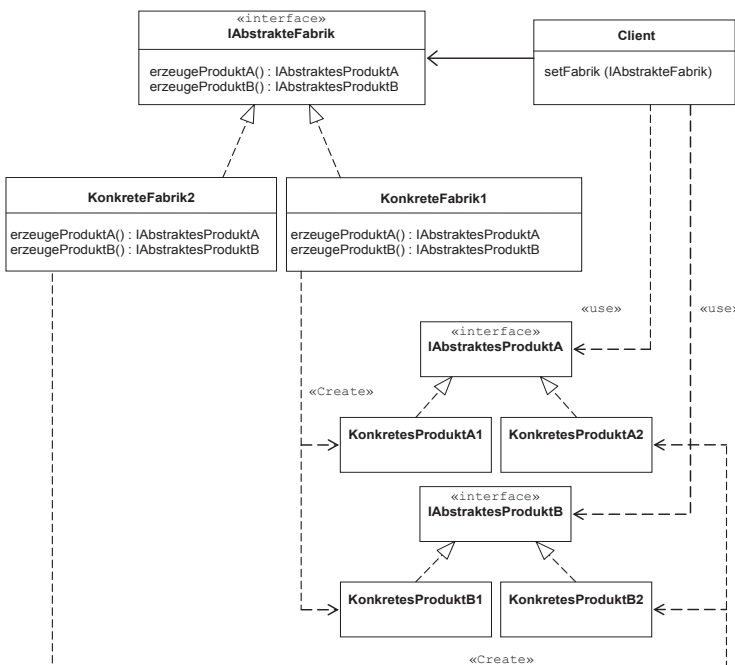


Bild 4-60 Klassendiagramm der Abstrakten Fabrik

Die konkreten Produktklassen `KonkretesProduktA1` und `KonkretesProduktA2` realisieren das Interface `IAbstraktesProduktA`. Analog dazu gibt es die Produktklassen `KonkretesProduktB1` und `KonkretesProduktB2`, die das Interface `IAbstraktesProduktB` realisieren. Wie im Klassendiagramm zu sehen ist, kennt ein Client nur die Interfaces der Produkte. Aufgrund des liskovschen Substitutionsprinzips werden jedoch die Methoden der konkreten Produktklassen aufgerufen.

Das Muster schreibt nicht vor, wie ein Client Kenntnis von einer konkreten Fabrik erlangt. Damit ein Client aber nicht von einer konkreten Fabrik abhängig wird, sondern flexibel mit verschiedenen konkreten Fabriken arbeiten kann, wurde im Klassendiagramm die Methode `setFabrik()` des Clients eingeführt. Damit kann von außen dem Client eine konkrete Fabrik übergeben werden, mit der er dann arbeitet, um Produkte zu erzeugen. Diese Vorgehensweise entspricht dem Prinzip **Dependency Injection** mit Hilfe einer Setter-Methode (siehe Kapitel 1.10.2). Das Programm kennt die entsprechende konkrete Fabrik nicht und hängt damit nicht von deren Ausprägung ab.

#### 4.19.3.2 Teilnehmer

Die Teilnehmer des Entwurfsmusters Abstrakte Fabrik werden im Folgenden kurz vorgestellt:

##### **IAbstraktesProduktX**

Das Interface `IAbstraktesProduktX` ( $X = A..Z$ ) definiert die Schnittstelle einer Produktart  $X$  und damit die Schnittstelle für ein bestimmtes konkretes Produkt dieser Art.

##### **IAbstrakteFabrik**

Das Interface `IAbstrakteFabrik` definiert die Schnittstelle für konkrete Fabriken. Für jede Produktart, die zu einer Produktfamilie gehört, wird in der Schnittstelle eine Erzeugungsmethode vorgesehen.

##### **KonkretesProduktXY**

Eine Klasse `KonkretesProduktXY` ( $X = A..Z, Y = 1..n$ ) definiert eine Variante  $Y$  der Produktart  $X$  und realisiert das Interface `IAbstraktesProduktX`. Ein konkretes Produkt  $XY$  gehört zur Produktfamilie  $Y$  und wird durch die entsprechende konkrete Fabrik  $Y$  erzeugt.

##### **KonkreteFabrikY**

Eine Klasse `KonkreteFabrikY` ( $Y = 1..n$ ) implementiert das Interface `IAbstrakteFabrik` und instanziiert in den Erzeugungsmethoden konkrete Produkte  $XY$ , die zur Produktfamilie  $Y$  gehören.

##### **Client**

Ein Client benutzt die Schnittstellen der abstrakten Fabrik und der abstrakten Produkte. Er besitzt eine Referenz, die zur Laufzeit auf eine konkrete Fabrik zeigt, um die Produkte einer Produktfamilie zu erzeugen.

### 4.19.3.3 Dynamisches Verhalten

Das dynamische Verhalten der Teilnehmer des Musters Abstrakte Fabrik wird mit Hilfe des Sequenzdiagramms in Bild 4-61 beschrieben:

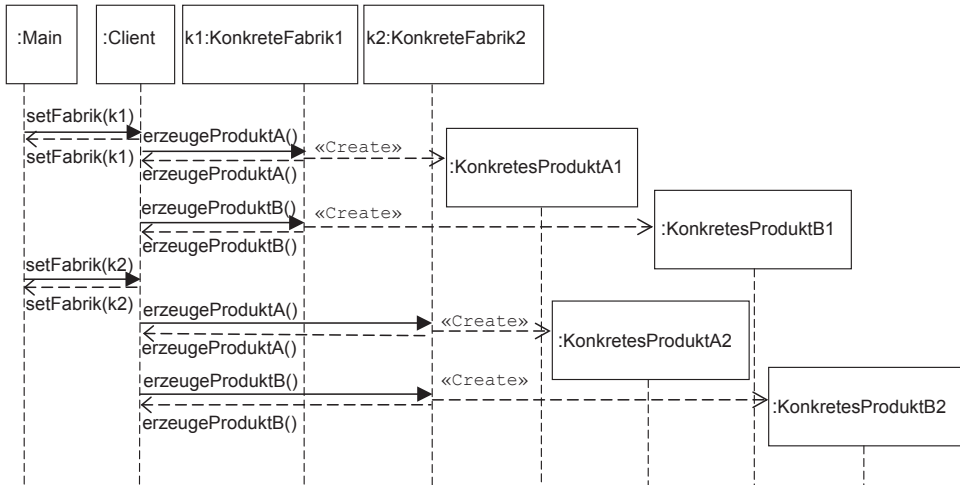


Bild 4-61 Sequenzdiagramm des Musters Abstrakte Fabrik

Zuerst wird dem Client die zu verwendende konkrete Fabrik mit Hilfe der Methode `setFabrik()` übergeben. Der Client kennt nur die Schnittstelle `IAbstrakteFabrik`, weiß also nicht, welche konkrete Fabrik übergeben wurde. Er erzeugt anschließend zwei Produkte vom Typ `IAbstraktesProduktA` und `IAbstraktesProduktB` durch Nutzung der Schnittstellenmethoden `erzeugeProduktA()` und `erzeugeProduktB()`. Da dem Client ein Objekt `k1` der Klasse `KonkreteFabrik1` als aktuell zu nutzende Fabrik übergeben wurde, werden Instanzen der Klassen `KonkretesProduktA1` und `KonkretesProduktB1` erzeugt. Der Client arbeitet also zu diesem Zeitpunkt mit Produkten der Produktfamilie 1.

Im nächsten Teil des Sequenzdiagramms wird vom Client ein Objekt der Klasse `KonkreteFabrik2` genutzt, da die Referenz des Clients durch Aufruf der Methode `setFabrik()` geändert wurde und nun auf das Objekt `k2` zeigt. Weil der Client ab diesem Zeitpunkt also die konkrete Fabrik `k2` benutzt, werden durch die Aufrufe der entsprechenden Methoden Instanzen der Klassen `KonkretesProduktA2` und `KonkretesProduktB2` erzeugt. Jetzt arbeitet der Client also mit Produkten der Produktfamilie 2.

### 4.19.3.4 Programmbeispiel

Das Muster Abstrakte Fabrik soll anhand eines einfachen Beispiels beschrieben werden, das bereits in der Einleitung zu diesem Muster kurz skizziert wurde: Es werden die zwei Produktarten Schrauben und Muttern betrachtet. Konkrete Produkte müssen jeweils das gleiche Gewinde haben, um miteinander genutzt werden zu können. Daher muss eine Fabrik immer die gleiche Produktfamilie erzeugen, d. h. entweder M6-

Schrauben und M6-Muttern oder aber M10-Schrauben und die dazugehörigen Muttern mit dem Gewinde M10.

Nachfolgend werden das Interface `IMutter` und die konkreten Produkte, nämlich Muttern mit Gewinde M6 bzw. mit Gewinde M10, vorgestellt:

**// Datei: IMutter.java**

```
public interface IMutter
{
    public void print();
}
```

**// Datei: MutterM6.java**

```
public class MutterM6 implements IMutter
{
    public void print()
    {
        System.out.println ("Mutter mit M6 Gewinde.");
    }
}
```

**// Datei: MutterM10.java**

```
public class MutterM10 implements IMutter
{
    public void print()
    {
        System.out.println ("Mutter mit M10 Gewinde.");
    }
}
```

Um die Muttern auch entsprechend nutzen zu können, sind Schrauben erforderlich. Zuerst wird das Interface `ISchraube` gezeigt und dann die konkreten Produkte, nämlich Schrauben mit Gewinde M6 und mit M10:

**// Datei: ISchraube.java**

```
public interface ISchraube
{
    public void print();
}
```

**// Datei: SchraubeM6.java**

```
public class SchraubeM6 implements ISchraube
{
    public void print()
    {
        System.out.println ("Schraube mit M6 Gewinde.");
    }
}
```

**// Datei: SchraubeM10.java**

```
public class SchraubeM10 implements ISchraube
{
```

```

    public void print()
    {
        System.out.println ("Schraube mit M10 Gewinde.");
    }
}

```

Für die Erzeugung von Schrauben und Muttern dient die Abstrakte Fabrik, die hier in Form eines Interface definiert wird:

```

// Datei: IAbstrakteFabrik.java
// IAbstrakteFabrik hat zwei Methoden. Jede erzeugt
// eine Instanz eines anderen Produktes (IMutter und ISchraube).
public interface IAbstrakteFabrik
{
    public ISchraube erzeugeSchraube();
    public IMutter erzeugeMutter();
}

```

Für die Erzeugung einer konkreten Produktfamilie, d. h. von Schrauben und Muttern mit gleichem Gewinde, wird eine konkrete Fabrik benötigt. Die beiden entsprechenden konkreten Fabriken für die Gewindegrößen M6 und M10 werden im Folgenden gezeigt:

```

// Datei: KonkreteFabrikM6.java
public class KonkreteFabrikM6 implements IAbstrakteFabrik
{
    public ISchraube erzeugeSchraube()
    {
        return new SchraubeM6();
    }

    public IMutter erzeugeMutter()
    {
        return new MutterM6();
    }
}

// Datei: KonkreteFabrikM10.java
public class KonkreteFabrikM10 implements IAbstrakteFabrik
{
    public ISchraube erzeugeSchraube()
    {
        return new SchraubeM10();
    }

    public IMutter erzeugeMutter()
    {
        return new MutterM10();
    }
}

```

Die beiden Klassen `ProduktionsMaschine` und `Schachtel` stellen die Anwendung des Musters `Abstrakte Fabrik` dar. Objekte der Klasse `Schachtel` stellen Behälter für Schrauben und Muttern dar. In dieser Klasse gibt es keine Bezüge auf konkrete Pro-



dukte. Prinzipiell kann eine Schachtel beliebige Kombinationen von Schrauben und Muttern in unterschiedlichen Größen enthalten. Um eine Schachtel aber für einen Käufer praktikabel zu machen, sollte sie nur mit Schrauben und Muttern gleicher Größe gefüllt werden. Dies leistet die Klasse `ProduktionsMaschine` mit Hilfe des Musters `Abstrakte Fabrik`. Zuerst folgt nun der Quellcode der Klasse `Schachtel`:

**// Datei: Schachtel.java**

```
public class Schachtel
{
    private int anzahl;
    private int anzahlSchrauben = 0;
    private int anzahlMuttern = 0;
    private ISchraube[] schrauben;
    private IMutter[] muttern;

    Schachtel (int groesse)
    {
        anzahl = groesse;
        schrauben = new ISchraube[anzahl];
        muttern = new IMutter[anzahl];
    }

    int anzahl()
    {
        return anzahl;
    }

    public void legeSchraubeHinein (ISchraube schraube)
    {
        if (anzahlSchrauben == anzahl) // wenn schon voll
        {
            return;
        }
        else
        {
            anzahlSchrauben++;
            schrauben[anzahlSchrauben-1] = schraube;
        }
    }

    public void legeMutterHinein (IMutter mutter)
    {
        if (anzahlMuttern == anzahl) // wenn schon voll
        {
            return;
        }
        else
        {
            anzahlMuttern++;
            muttern[anzahlMuttern-1] = mutter;
        }
    }

    public void zeigeInhalt()
    {
        int i;
```

```

        for (i = 0; i < anzahlSchrauben; i++)
        {
            schrauben[i].print();
        }
        for (i = 0; i < anzahlMuttern; i++)
        {
            muttern[i].print();
        }
    }
}

```

In der Klasse `ProduktionsMaschine` ist die Arbeitsweise des Musters Abstrakte Fabrik zu sehen. Eine Produktionsmaschine besitzt eine Referenz auf die abstrakte Fabrik. Mittels der konkreten Fabrik, auf welche die Referenz zur Laufzeit zeigt, werden konkrete Produkte erzeugt und von der Maschine in eine Schachtel gefüllt. Der Quellcode der Klasse `ProduktionsMaschine` enthält keine Bezüge auf konkrete Produkte:

```

public class ProduktionsMaschine
{
    private IAbstrakteFabrik fabrik = null;

    public void setFabrik (IAbstrakteFabrik fabrikRef)
    {
        this.fabrik = fabrikRef;
    }

    public void fuelleSchachtel (Schachtel schachtel)
    {
        int i;

        for (i = 0; i < schachtel.anzahl(); i++)
        {
            schachtel.legeSchraubeHinein (fabrik.erzeugeSchraube());
            schachtel.legeMutterHinein (fabrik.erzeugeMutter());
        }
    }
}

```

Mit Hilfe der Klasse `TestProduktion` wird die Arbeitsweise einer einzigen Produktionsmaschine getestet und geprüft, ob Schachteln mit den Produkten nur einer einzigen Produktfamilie erzeugt werden. Die Testklasse enthält ein Objekt der Klasse `ProduktionsMaschine` und weist diesem Objekt zuerst eine konkrete Fabrik (Instanz von `KonkreteFabrikM6`) für die Erzeugung zu. Im nächsten Schritt wird die Maschine aufgefordert, eine Schachtel zu füllen. Anschließend wird der Inhalt der Schachtel angezeigt.

Im zweiten Teil des Programms werden diese Schritte wiederholt. Allerdings wird die Referenz der Produktionsmaschine mit Hilfe der Methode `setFabrik()` auf ein Objekt der Klasse `KonkreteFabrikM10` gesetzt. In der Programmausgabe ist zu sehen, dass daraufhin nur Produkte der Produktfamilie mit M10-Gewinde erzeugt und in eine Schachtel verpackt werden. Hier die Klasse `Testproduktion`:

```
// Datei: TestProduktion.java
public class TestProduktion
{
    static ProduktionsMaschine maschine = new ProduktionsMaschine();
    static Schachtel sch1 = new Schachtel (5);
    static Schachtel sch2 = new Schachtel (3);

    public static void main (String[] args)
    {
        maschine.setFabrik (new KonkreteFabrikM6());
        maschine.fuelleSchachtel (sch1);
        sch1.zeigeInhalt();

        System.out.println();

        maschine.setFabrik (new KonkreteFabrikM10());
        maschine.fuelleSchachtel (sch2);
        sch2.zeigeInhalt();
    }
}
```



Hier das Protokoll des Programmlaufs:

Schraube mit M6 Gewinde.  
 Schraube mit M6 Gewinde.  
 Schraube mit M6 Gewinde.  
 Schraube mit M6 Gewinde.  
 Schraube mit M6 Gewinde.  
 Mutter mit M6 Gewinde.  
 Mutter mit M6 Gewinde.  
 Mutter mit M6 Gewinde.  
 Mutter mit M6 Gewinde.  
 Mutter mit M6 Gewinde.

Schraube mit M10 Gewinde.  
 Schraube mit M10 Gewinde.  
 Schraube mit M10 Gewinde.  
 Mutter mit M10 Gewinde.  
 Mutter mit M10 Gewinde.  
 Mutter mit M10 Gewinde.

## 4.19.4 Bewertung

### 4.19.4.1 Vorteile

Folgende Vorteile werden gesehen:

- Die konkreten Klassen werden durch abstrakte Klassen bzw. Interfaces isoliert. Der Vorteil ist, dass eine Anwendung, die die erstellten Objekte benutzt, lediglich deren Schnittstellen kennt und so eine Abkopplung von den konkreten Implementierungen möglich ist.

- Produktfamilien können leicht ausgetauscht werden. Es muss nur eine einzige Referenz so geändert werden, dass sie auf eine andere konkrete Fabrik zeigt.
- Solange die Referenz auf die konkrete Fabrik nicht geändert wird, ist sichergestellt, dass in diesem Zeitraum nur eine einzige Produktfamilie mit zusammenpassenden Produkten verwendet wird.
- Eine neue Produktfamilie kann flexibel hinzugefügt werden, ohne dass die Anwendung geändert werden muss – vorausgesetzt die Interfaces werden nicht geändert. Es muss nur eine neue konkrete Fabrik implementiert werden.

#### 4.19.4.2 Nachteile

Der folgende Nachteil wird gesehen:

- Neue Produktarten lassen sich nur mit Aufwand hinzufügen, da eine neue Produktart im Interface der Abstrakten Fabrik berücksichtigt werden muss und für diese Produktart eine weitere Erzeugungsmethode eingeführt werden muss. Als Konsequenz müssen auch alle konkreten Fabriken an die erweiterte Abstrakte Fabrik angepasst werden.

#### 4.19.5 Einsatzgebiete

Das Muster Abstrakte Fabrik kann immer dann eingesetzt werden, wenn ein System unabhängig von der Erzeugung seiner Produkte sein soll, das System aber mit einer zur Laufzeit ausgewählten Produktfamilie konfiguriert werden kann. Ein Beispiel hierfür ist ein Framework, das zur Laufzeit einerseits lauffähige Produktfamilien für Linux und andererseits für Windows erzeugen soll.

Bei Nutzung des Musters Abstrakte Fabrik möchte der Client seine Objekte – sprich Produkte – erzeugen können, ohne zur Kompilierzeit die konkreten Produkten bzw. deren Klassen zu kennen. Die entsprechenden konkreten Fabriken und konkreten Produkte werden erst zur Laufzeit festgelegt. Zur Kompilierzeit greift ein Client nur auf die Interfaces `IAbstrakteFabrik` und `IAbstraktesProduktX` zu.

Beim Strukturmuster **Brücke** wird oftmals das Muster Abstrakte Fabrik eingesetzt, um zueinander passende Abstraktions- und Implementierungsobjekte zu erzeugen.

#### 4.19.6 Ähnliche Entwurfsmuster

Eine Abstrakte Fabrik hat eine analoge Wirkung wie eine **Fassade**: die Abstrakte Fabrik bietet einen vereinfachten Zugang zum Erzeugen von Gruppen von untereinander abhängigen Produkten an. Eine Anwendung muss sich nicht mehr um die Abhängigkeiten der Produkte untereinander kümmern. Eine Fassade ist im Allgemeinen aber nicht auf das Erzeugen von Objekten beschränkt, sondern kann alle möglichen Zugriffe auf die Objekte vereinfachen.

Beim Muster Abstrakte Fabrik und beim Muster **Fabrikmethode** erfolgt die Erzeugung eines konkreten Objekts in Unterklassen. In den Unterklassen wird auch der Typ der zu erzeugenden Objekte festgelegt. Im Gegensatz zu einer Abstrakten Fabrik hat eine

Fabrikmethode ein einzelnes Produkt und nicht eine Produktfamilie zum Inhalt. Die Erzeugungsmethoden einer Abstrakten Fabrik können als Fabrikmethoden angesehen werden. Ein weiterer Unterschied zwischen beiden Muster ist, dass das Muster Fabrikmethode ein klassenbasiertes Muster ist und das Muster Abstrakte Fabrik ein objektbasiertes Muster.

Der Begriff Fabrik wird in der Objektorientierung relativ allgemein benutzt. So wird häufig schon von einem Fabrikmuster gesprochen, wenn Erzeugungsmethoden in einer Klasse gesammelt werden. Präziser definiert ist das Muster **Statische Fabrik**, siehe beispielsweise [Lah09]. Bei einer Statischen Fabrik kommt meist eine klassenbezogene d. h. statische Fabrikmethode zum Einsatz. Wenn die Klasse des zu erzeugenden Objekts vom Wert eines Parameters abhängig ist, spricht man von einer parametrisierten Statischen Fabrik. Bei einer konfigurierbaren Statischen Fabrik liest die Statische Fabrik aus einer Konfigurationsdatei ein, welche Produktfamilie erzeugt werden soll. Auch mit diesen beiden Möglichkeiten erreicht eine Statische Fabrik nicht die Offenheit und Erweiterbarkeit einer Abstrakten Fabrik.

## 4.20 Das Erzeugungsmuster Singleton

### 4.20.1 Name/Alternative Namen

Singleton.

### 4.20.2 Problem

Es soll eine Klasse geben, von der sichergestellt werden muss, dass nur eine einzige Instanz von ihr existiert. Dabei soll es für ein Client-Programm irrelevant sein, ob die einzige Instanz bereits erzeugt worden ist oder nicht. Für das Erzeugen der einzigen Instanz soll die genannte Klasse selbst verantwortlich sein.

Das **Singleton-Muster** soll gewährleisten, dass eine Klasse nur ein einziges Mal instanziiert werden kann.



### 4.20.3 Lösung

Die Klasse, von der es nur ein einziges Objekt geben darf, wird im Folgenden `Singleton` genannt. Ein Client, der mit dem Singleton-Objekt arbeiten will, bekommt eine Referenz auf dieses Singleton-Objekt. Das Singleton-Muster ist somit ein objektbasiertes Entwurfsmuster.

Ein Singleton-Objekt kann nur von der Klasse `Singleton` selbst erzeugt werden. Alle Konstruktoren dieser Klasse werden mit dem Zugriffsmodifizierer `private` gekennzeichnet, so dass andere Klassen kein Objekt unter Verwendung eines Konstruktors erzeugen können. Objekte, die die Klasse `Singleton` verwenden möchten, erhalten von der Klassenmethode `getInstance()` der Klasse `Singleton` eine Referenz auf das einzig existierende Objekt der Klasse `Singleton` zurück.

Für die Erzeugung des einzigen Exemplars der Klasse `Singleton` durch die Klasse `Singleton` selbst gibt es zwei unterschiedliche Vorgehensweisen.

Bei der ersten Variante, wie sie in [Gam95] zu finden ist, wird das Singleton-Objekt erst erzeugt, wenn ein Client-Objekt zum ersten Mal die Methode `getInstance()` aufruft. Wird diese Methode jedoch aus mehreren Threads gleichzeitig aufgerufen, so könnte dies zur Erzeugung mehrerer Objekte führen. Um dies zu vermeiden, müssen entsprechende Synchronisationsmechanismen eingesetzt werden, die sich jedoch je nach Programmiersprache unterscheiden.

Die zweite Methode ist unter anderem in [Gr102] zu finden. Dabei wird das Singleton-Objekt bereits beim Laden der Klasse erzeugt, unabhängig davon, ob das Singleton-Objekt jemals gebraucht wird oder nicht.

Diese beiden Varianten haben je nach verwendeter Programmiersprache meist unterschiedliche Implementierungen, die Aspekte wie Geschwindigkeit, Threadsicherheit und Speicherbedarf berücksichtigen (siehe Kapitel 4.20.3.4). Hier zeigt sich, dass sich das Singleton-Muster aufgrund seiner einfachen Struktur zwar leicht erklären lässt, seine Implementierung aber dennoch komplex sein kann.

#### 4.20.3.1 Klassendiagramm

Die Klasse `Singleton` erzeugt das einzige Singleton-Objekt als Klassenattribut und stellt dieses über eine Klassenmethode `getInstance()` zur Verfügung. Der für die Erzeugung benutzte Konstruktor wird als `private` deklariert, so dass er außerhalb der Klasse `Singleton` nicht zur Verfügung steht. Die Methode `operation()` steht im Klassendiagramm stellvertretend für alle möglichen Operationen, die auf dem Singleton-Objekt ausgeführt werden können. Das folgende Bild zeigt das Klassendiagramm:

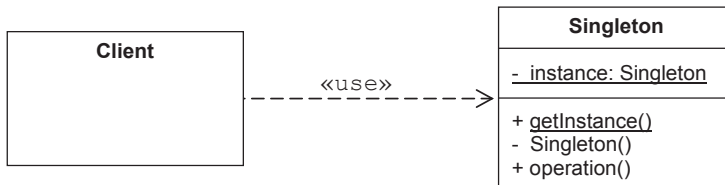


Bild 4-62 Klassendiagramm für das Singleton-Muster

Ein Client nutzt das Singleton-Objekt und ruft Operationen des Singleton-Objekts auf. Die Klasse `Singleton` muss für den Client sichtbar sein. Man kann nur vom Client zum Singleton gelangen, die Klasse `Singleton` kennt aber den Client nicht.

#### 4.20.3.2 Teilnehmer

##### Client-Klasse

Der `Client` ruft die Klasse `Singleton` auf.

##### Singleton-Klasse

Die Klasse `Singleton` ist diejenige Klasse, von der es nur eine einzige Instanz geben darf.

#### 4.20.3.3 Dynamisches Verhalten

Ein Client, der mit dem Singleton-Objekt arbeiten will, ruft die Klassenmethode `getInstance()` der Klasse `Singleton` auf. Wie Bild 4-63 zeigt, ist der Rückgabewert der Methode `getInstance()` eine Referenz auf die neu erstellte Instanz der Klasse `Singleton` oder – wenn die Instanz bereits existiert – auf das einzige bereits vorhandene Exemplar der Klasse `Singleton`.

Nachdem der Client eine Referenz auf die Instanz der Klasse `Singleton` hat, kann er Operationen auf dieser Instanz aufrufen, hier dargestellt durch den Aufruf der Methode `operation()`:

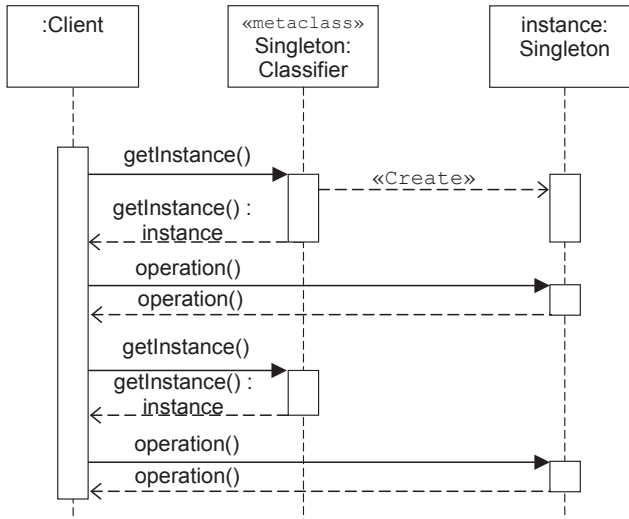


Bild 4-63 Sequenzdiagramm für das Abholen einer Referenz auf das Singleton-Objekt

Die Instanz `Singleton:Classifier` stellt die Klasse `Singleton` als Instanz der Metaklasse `Classifier` dar.

#### 4.20.3.4 Programmbeispiel

Überall da, wo nur ein einziges Exemplar einer Klasse erzeugt werden darf wie z. B. bei einem authentifizierenden Objekt, ist das hier beschriebene Muster die richtige Wahl. Im folgenden Beispiel wird zuerst die Variante 1 des Singleton-Musters gezeigt. Dann wird das Beispiel modifiziert zur Variante 2. Anschließend werden beide Varianten diskutiert.

##### Programmierbeispiel Variante 1 – `ToolTipManager`

Um die Vor- und Nachteile beider Methoden besser verstehen zu können, soll nun zuerst der Quellcode einer typischen Singleton-Klasse vorgestellt werden, nämlich der Klasse `ToolTipManager`. Ein `ToolTipManager` bestimmt beispielsweise, welche Erläuterung (engl. tool tip) wie lange angezeigt wird, wenn die Maus auf ein grafisches Element zeigt. Es darf aber nur eine einzige Anzeige und keine konkurrierenden `ToolTipManager` geben. Daher muss die Klasse `ToolTipManager` als Singleton ausgelegt werden. Hier die Klasse `ToolTipManager`:

```
// Datei: ToolTipManager.java
final class ToolTipManager
{
    private static ToolTipManager instance;
```



```
private ToolTipManager()
{
    System.out.println ("Neues Singleton erzeugt.");
}

public static ToolTipManager getInstance()
{
    if (instance == null)
    {
        instance = new ToolTipManager();
    }
    return instance;
}

public void operation()
{
    // eigentliche Funktionalitaet des Singleton
    System.out.println ("operation() aufgerufen.");
}
}
```

Dieser Code ist nicht multi-threading-fähig. Der gleichzeitige Aufruf von `getInstance()` aus mehreren Threads kann zu Problemen führen.



Auf die Synchronisierung eines derartigen Codes wird in der weiteren Behandlung dieses Musters noch eingegangen.

Dieses Beispiel zeigt, dass die Singleton-Klasse zur Verwirklichung der Singleton-Funktionalität eine Referenz `instance` auf das Singleton-Objekt beinhalten muss. Diese Referenz muss eine Klassenvariable sein (`static` in Java), da auf sie in der Klassenmethode `getInstance()` zugegriffen wird.

Die Methode `getInstance()` muss eine Klassenmethode sein, da Client-Objekte auf diese Methode zugreifen, bevor überhaupt ein Singleton-Objekt existiert.



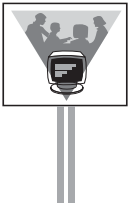
Ein Aufruf der Methode `operation()` sieht damit folgendermaßen aus:

```
ToolTipManager.getInstance().operation();
```

Hierbei wird über den Klassennamen `ToolTipManager` auf die Klassenmethode `getInstance()` zugegriffen, um eine Referenz auf das Singleton-Objekt zu erhalten. Über diese Referenz wird dann die Instanzmethode `operation()` aufgerufen, die die eigentliche Funktionalität des Singleton-Objekts bereitstellt.

Die im Folgenden dargestellte Klasse `Client` ruft im Programm die Methode `ToolTipManager.getInstance().operation()` dreimal hintereinander auf:

```
// Datei: Client.java
public class Client
{
    public static void main (String[] args)
    {
        ToolTipManager.getInstance().operation();
        ToolTipManager.getInstance().operation();
        ToolTipManager.getInstance().operation();
    }
}
```



In der Programmausgabe sieht man, dass das Objekt nur ein einziges Mal erzeugt wird:

```
Neues Singleton erzeugt.
operation() wird aufgerufen.
operation() wird aufgerufen.
operation() wird aufgerufen.
```

Der interessanteste Teil einer Singleton-Implementierung ist jedoch der Rumpf der Methode `getInstance()` selbst, sowie die Definition der Referenz `instance`. An diesen beiden Stellen unterscheiden sich die oben genannten zwei Varianten zur Erzeugung des Singleton-Objekts. Die gerade vorgestellte Variante erzeugt – wie in [Gam95] – das Singleton-Objekt erst dann, wenn die Methode `getInstance()` aufgerufen wird. Der Vorteil bei dieser Variante liegt darin, dass die Objekterzeugung wirklich nur dann erfolgt, wenn tatsächlich ein Client-Objekt das Singleton-Objekt benötigt. Der Nachteil ist eine leichte Verschlechterung der Performance bei weiteren Zugriffen auf die Methode `getInstance()`, da jedes Mal der Vergleich (`instance == null`) ausgeführt werden muss, der nach dem ersten Aufruf sowieso immer negativ ausfällt.

## Programmierbeispiel Variante 2

Ist der (fast immer überflüssige) Vergleich und die damit einhergehende Verschlechterung der Performance unerwünscht, so kann auf die zweite Variante des Singleton-Musters [Gr102] zurückgegriffen werden.

In der zweiten Variante erfolgt die Erzeugung des Singleton-Objekts "statisch", d. h. sofort bei Programmstart oder im Falle von Java beim Laden der Singleton-Klasse durch den Klassenlader, wenn die Klasse zum ersten Mal benutzt wird.



Der Vorteil bei dieser Variante ist, dass der oben genannte Vergleich in der Methode `getInstance()` gänzlich entfällt. Folgendes Beispiel zeigt diese Variante:

```
// Datei: ToolTipManager2.java
final public class ToolTipManager2
{
    private static ToolTipManager2 instance = new ToolTipManager2();

    private ToolTipManager2()
    {
        System.out.println ("ToolTipManager2 erzeugt.");
    }

    public static ToolTipManager2 getInstance()
    {
        System.out.println ("ToolTipManager2::getInstance()");
        return instance;
    }

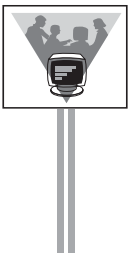
    public void operation()
    {
        // eigentliche Funktionalität des Singleton
        System.out.println ("operation() aufgerufen.");
    }
}
```

Es ist zu sehen, dass sich die Vereinbarung der Klassenvariable `instance` von der ersten Variante unterscheidet.

In der zweiten Variante wird sofort beim Programmstart ein Exemplar der Klasse `ToolTipManager` erzeugt und nicht erst beim Aufruf von `getInstance()`.

Der Zugriff erfolgt dabei – wie in der ersten Variante – mit Hilfe der Methode `getInstance()`. Im Folgenden die Klasse `Client2`:

```
// Datei: Client2.java
public class Client2
{
    public static void main (String[] args)
    {
        ToolTipManager2.getInstance().operation();
        ToolTipManager2.getInstance().operation();
        ToolTipManager2.getInstance().operation();
    }
}
```



Die Ausgabe des Programms ist:

```
ToolTipManager2 erzeugt.
ToolTipManager2::getInstance()
operation() aufgerufen.
ToolTipManager2::getInstance()
operation() aufgerufen.
ToolTipManager2::getInstance()
operation() aufgerufen.
```

In diesem Fall vereinfacht sich der Rumpf der Methode `getInstance()`, da der Vergleich entfällt und die Referenz auf das Singleton-Objekt sofort zurückgegeben werden kann. Dadurch ist in diesem Beispiel die Methode `getInstance()` nicht mehr durch einen Threadwechsel unterbrechbar und somit im Gegensatz zum vorherigen Beispiel bereits threadsicher, d. h., die statische Methode `getInstance()` kann von verschiedenen Threads aufgerufen werden. Die Performance leidet so nicht, allerdings wird je nach verwendeter Programmiersprache unnötig Speicher verbraucht für den Fall, dass das Singleton-Objekt nie gebraucht wird. Im Falle von Java wird die Klasse wirklich nur dann geladen, wenn sie wirklich genutzt wird, also das Objekt gebraucht wird. Dann wird automatisch auch das Objekt erstellt.

### Vergleich der beiden Varianten

Es ist – wie so oft – zwischen Performance und Speicherbedarf abzuwägen und die richtige Variante für die aktuell vorliegende Situation auszuwählen. Hat man nur einen einzigen Thread und ist sich nicht sicher, ob das Singleton-Objekt überhaupt jemals benötigt wird, so ist sicherlich die Variante 1 aus [Gam95] geeigneter. Sie verspricht einen unter Umständen geringeren Speicherverbrauch, benötigt aber einen höheren Rechenaufwand. Wenn die Lösung threadsicher sein soll und das Singleton-Objekt in jedem Fall gebraucht wird, so ist die Variante 2 aus [Gr102] die bessere Lösung. Unter den genannten Umständen fällt der Rechenaufwand für diese Variante geringer aus. Diese Lösung kann sogar noch weiter optimiert werden: Wenn `instance` zu einer öffentlichen Konstanten der Klasse `Singleton` gemacht wird, kann die Klassenmethode `getInstance()` entfallen, da auf die öffentlichen Konstanten direkt zugegriffen werden kann.

Das zweite Beispiel ist threadsicher, d. h., es kann problemlos von mehreren Threads aufgerufen werden, da das Singleton-Objekt bereits beim Laden der Klasse angelegt wird. Dies soll durch den mehrmaligen Aufruf der Methode `getInstance()` in verschiedenen Threads aufgezeigt werden:

```
// Datei: SingletonTestThread.java
public class SingletonTestThread extends Thread
{
    public String threadName;
    public SingletonTestThread (String tName)
    {
        threadName = tName;
    }

    public void run()
    {
        try
        {
            Thread.sleep (500);
            System.out.println (threadName + " - call 1");
            ToolTipManager2.getInstance().operation();
            Thread.sleep (500);
            System.out.println (threadName + " - call 2");
        }
    }
}
```

```
        TooltipManager2.getInstance().operation();
        Thread.sleep (500);
        System.out.println (threadName + " - call 3");
        TooltipManager2.getInstance().operation();
    }
    catch (InterruptedException ie)
    {
        System.out.println (threadName + " - interrupted.");
    }
}
}
```

In dem folgenden Beispiel werden 3 Threads angelegt und gestartet. Erzeugt wird ein Thread mit Hilfe des `new`-Operators und zum Starten eines Threads wird seine Methode `start()` aufgerufen. Hier die Klasse `TestSingletonMultipleThreads` mit der `main()`-Methode:

```
// Datei: TestSingletonMultipleThreads.java
public class TestSingletonMultipleThreads
{
    public static void main (String[] args)
    {
        SingletonTestThread s1 = new SingletonTestThread ("Thread 1");
        SingletonTestThread s2 = new SingletonTestThread ("Thread 2");
        SingletonTestThread s3 = new SingletonTestThread ("Thread 3");

        s1.start();
        s2.start();
        s3.start();
    }
}
```

Die Methode `start()` **reserviert die Systemressourcen**, welche notwendig sind, um den Thread zu starten. Außerdem **ruft sie die Methode `run()` auf**. Wie in der `run()`-Methode der Klasse `SingletonTestThread` zu sehen ist, holt sich jeder Thread nach dem Start dreimal hintereinander eine Instanz auf die Klasse `TooltipManager2` und ruft für jede geholt Instanz die Methode `operation()` auf.



Die nicht-deterministische<sup>70</sup> Ausgabe des Programms ist:

```
Thread 2 - call 1
ToolTipManager2 erzeugt.
ToolTipManager2::getInstance()
operation() aufgerufen.
Thread 3 - call 1
ToolTipManager2::getInstance()
operation() aufgerufen.
Thread 1 - call 1
ToolTipManager2::getInstance()
operation() aufgerufen.
Thread 2 - call 2
ToolTipManager2::getInstance()
operation() aufgerufen.
Thread 3 - call 2
ToolTipManager2::getInstance()
operation() aufgerufen.
Thread 1 - call 2
ToolTipManager2::getInstance()
operation() aufgerufen.
Thread 2 - call 3
ToolTipManager2::getInstance()
operation() aufgerufen.
Thread 3 - call 3
ToolTipManager2::getInstance()
operation() aufgerufen.
Thread 1 - call 3
ToolTipManager2::getInstance()
operation() aufgerufen.
```

Um beim ersten Beispiel Threadsicherheit zu gewährleisten, müsste die Methode `getInstance()` mit `synchronized` gekennzeichnet werden. Dies wird im Folgenden gezeigt:

```
public synchronized static ToolTipManager getInstance()
{
    if (instance == null)
    {
        instance = new ToolTipManager();
        System.out.println ("Neues Singleton erzeugt");
    }
    return instance;
}
```

Die Kennzeichnung mit dem Schlüsselwort `synchronized` hat allerdings den Nachteil, dass die Aufrufe für `getInstance()` relativ langsam sind [shespi].

<sup>70</sup> Nicht-deterministisch bedeutet, dass die Ausgabe bei einem erneuten Aufruf nicht zwangsläufig die gleiche Ausgabe erzeugt, da mehrere Threads gleichzeitig ablaufen.

## 4.20.4 Bewertung

### 4.20.4.1 Vorteile

Der folgende Vorteil wird gesehen:

- Es wird gewährleistet, dass nur ein einziges Exemplar erzeugt wird.

### 4.20.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Bei der Verwendung mehrerer Threads ist auf die Synchronisation zu achten.
- Während es einfach ist, eine Klasse pro Applikation nur einmal zu verwenden, ist es schwierig, das Singleton-Muster über mehrere virtuelle Maschinen oder Rechner hinweg zu gewährleisten.

## 4.20.5 Einsatzgebiete

Das Singleton-Muster wird dann verwendet, wenn eine Instanz einer Klasse innerhalb einer Applikation nur in einer einzigen Ausprägung zum Einsatz kommen soll. Beispiele hierfür sind z. B. eine Logging-Klasse, die einen Anwendungsablauf protokolliert, oder aber eine Klasse, die zentrale Applikations-Einstellungen verwaltet.

Das Singleton-Muster kann auch in verschiedenen anderen Entwurfsmustern eingesetzt werden. Beispielsweise ist ein **Objektpool** üblicherweise nur einmal vorhanden. Ein weiteres Beispiel ist das **Fassade-Muster**, bei dem eine Fassade meist als Singleton realisiert wird.

## 4.20.6 Ähnliche Entwurfsmuster

Wenn die Anzahl der Singleton-Objekte nicht auf eins begrenzt werden soll, sondern auf eine beliebige andere Anzahl, kann das Entwurfsmuster **Objektpool** [Gr102] eingesetzt werden.

## 4.21 Das Erzeugungsmuster Objektpool

### 4.21.1 Name/Alternative Namen

Objektpool (engl. object pool).

### 4.21.2 Problem

Die Erzeugung bestimmter Ressourcen wie Datenbankverbindungen oder Threads erweist sich oftmals als rechenleistungs- und speicherintensiv. Daher sollen solche Instanzen als wiederverwendbare Ressourcen betrachtet werden und aus einem Pool abgerufen werden können. In speziellen Fällen soll auch die Anzahl der existierenden Instanzen eines Typs bewusst beschränkt werden.

Das Muster **Objektpool** soll es ermöglichen, Objekte aus einem Pool zur Verfügung zu stellen und wiederzuverwenden, statt sie jedesmal aufwendig bei jeder Anforderung neu zu erzeugen.



### 4.21.3 Lösung

Für ausgesuchte Typen, deren Instanziierung zu aufwendig ist oder bei denen die Zahl der Instanzen beschränkt werden soll, verwendet ein Objektpool die auszugebenden Objekte wieder. Der Objektpool hält die Objekte bereit und teilt sie den Interessenten zu. Die auszugebenden Objekte werden im Pool zwischengespeichert. Wie bei einem Pfandflaschensystem sorgt der Objektpool für eine Wiederverwendung. Damit werden Ressourcen gespart. Solche Objekte werden nach Gebrauch nicht zerstört, sondern dem Pool für eine erneute Verwendung wieder zugeführt. Wie die Flaschen im Pfandflaschensystem müssen auch die Objekte "gereinigt", d. h. in einen "sauberen" einheitlichen Zustand gebracht werden. Alle Objekte im Pool müssen nämlich denselben Zustand aufweisen, damit es keine Rolle spielt, welches dieser Objekte verwendet wird. Das Muster Objektpool ist ein objektbasiertes Muster.

Die Effizienz eines Objektpools hängt natürlich von der Güte der Implementierung und der Art der im Objektpool gespeicherten Objekte ab.

Ein Objektpool erbringt die folgenden Leistungen:

- zurückgegebene Objekte annehmen, zwischenspeichern sowie initialisieren und
- angeforderte Objekte herausgeben.

Die verschiedenen Nutzer eines Pools greifen in der Rolle von Clients auf den Objektpool als Server zu. Da der Objektpool eine einzige Zentralstelle für alle Clients sein soll, wird er als Singleton angelegt. Das Singleton-Muster (siehe Kapitel 4.20) garantiert, dass von einer bestimmten Klasse nur eine einzige Instanz angelegt wird, auf die von mehreren Clients zugegriffen werden kann. Durch die Auslegung als Singleton wird sichergestellt, dass die auszugebenden Objekte nur von einer einzigen Instanz



eines Objektpools bezogen werden, so dass es keine konkurrierenden Alternativen geben kann.

Um mehrere nebenläufige Clients gleichzeitig durch einen Objektpool bedienen zu können, müssen die kritischen Abschnitte der Objektanforderung und Objektrückgabe des Objektpools threadsicher ausgeführt werden. Threadsicher bedeutet, dass der Objektpool auch bei Vorhandensein mehrerer Threads als Clients immer in einem gültigen Zustand ist.

Ist die Kapazität des Pools erschöpft, so kann der Objektpool je nach Strategie ein neues Objekt erzeugen, einen Client warten lassen, bis wieder ein zurückgegebenes und "gereinigtes" Objekt zur Verfügung steht, oder aber dem Client die Entscheidung bei einem ihm mitgeteilten Engpass überlassen.

Bei der Beschreibung des Musters wird im Folgenden davon ausgegangen, dass ein Client nur ein einziges Objekt zur gleichen Zeit benötigt. Zur Anforderung eines einzelnen Objekts wird eine Methode `gibObjekt()` vom Objektpool zur Verfügung gestellt. Gibt es Client-Threads, die nicht nur ein einziges, sondern mehrere Objekte des Objektpools gleichzeitig verwenden wollen, und ist die Zahl der verfügbaren Objekte im Objektpool beschränkt, kann es durchaus vorkommen, dass die Objekte anfordernden Threads aufeinander warten müssen und sich damit gegenseitig blockieren. Zur Lösung dieses Problems müsste der Objektpool Methoden zur Verfügung stellen, mit denen mehrere Objekte gleichzeitig angefordert werden können.

#### 4.21.3.1 Klassendiagramm

Das folgende Klassendiagramm stellt den Objektpool, den Client und die wiederverwendbare Klasse der Pool-Objekte dar:

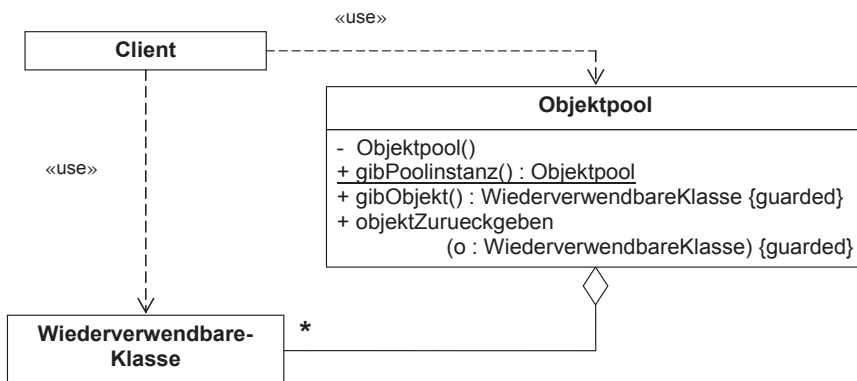


Bild 4-64 Klassendiagramm des Objektpool-Musters

Die Klasse `Objektpool` ist als Singleton (siehe Kapitel 4.20) ausgelegt. Zur Realisierung des Entwurfsmusters **Singleton** wird die Klassenmethode `gibPoolinstanz()` zur Verfügung gestellt und der Konstruktor als `private` deklariert.

Die Methoden `gibObjekt()` und `objektZurueckgeben()` sind mit dem Zusatz `{guarded}` versehen, um anzudeuten, dass die beiden Methoden threadsicher ausgeführt werden müssen.

Damit das Entwurfsmuster Objektpool sinnvoll eingesetzt werden kann, sollten einige Regeln eingehalten werden. Diese Regeln lauten:

- **Regel 1: Objekte sollten zurückgesetzt werden.**  
Enthalten die vom Pool verwalteten Objekte Daten, die durch die Clients verändert werden können, so müssen sie in einen definierten Ursprungszustand zurückversetzt werden. In der Regel geschieht dies beim Zurücklegen in den Pool.
- **Regel 2: Der Pool sollte eine minimale und eine maximale Größe haben.**  
Der Pool darf nicht zu klein sein, da die Clients sonst unter Umständen lange warten müssen, ehe sie ein Objekt aus dem Objektpool erhalten. Ein zu großer Pool verschwendet Ressourcen. Der Pool könnte sich im Idealfall der jeweiligen Situation anpassen, d. h. dynamisch neue Objekte zur Laufzeit anlegen oder bereits bestehende Objekte zur Laufzeit löschen.
- **Regel 3: Der Pool sollte zu Beginn mit einer gewissen Mindestanzahl von Objekten gefüllt sein.**  
Hat der Objektpool eine anfängliche Mindestanzahl an bereits initialisierten Objekten, so kann vermieden werden, dass beim Starten des Objektpools ein Client auf die Erzeugung der ersten initialisierten Objekte warten muss.

#### 4.21.3.2 Teilnehmer

Das Entwurfsmuster Objektpool umfasst drei verschiedene Klassen:

##### Objektpool

Diese Klasse verwaltet die Instanzen der Klasse `WiederverwendbareKlasse`. Ein Client kann nur über die einzige Instanz vom Typ `Objektpool` Objekte vom Typ `WiederverwendbareKlasse` erhalten.

##### Wiederverwendbare Klasse

Instanzen von diesem Typ sind aufwendig zu erstellen. Ein Client wird ein Objekt dieser Klasse nicht zerstören, sondern nach Verwendung dieses wieder in den Pool zurücklegen.

##### Client

Die Klasse Client und ihre Objekte stehen stellvertretend für alle möglichen Objekte, die den Objektpool verwenden, um an Instanzen des Typs `WiederverwendbareKlasse` zu gelangen.

#### 4.21.3.3 Dynamisches Verhalten

Mit dem Aufruf `Objektpool.gibPoolinstanz()` erhält ein Client eine Instanz des Pools. Neben der Klassenmethode `gibPoolinstanz()` verfügt der Objektpool noch über zwei weitere Methoden, die der Client verwenden kann:

```
gibObjekt(): WiederverwendbareKlasse,
objektZurueckgeben (o : WiederverwendbareKlasse).
```

Mit `gibObjekt()` fordert ein Client vom Pool ein Objekt vom Typ `WiederverwendbareKlasse` an, das er exklusiv nutzen will. Sind im Objektpool freie Objekte vom Typ `WiederverwendbareKlasse` vorhanden, so erhält der Client eines dieser Objekte. Damit kann nun der Client beliebige Operationen auf diesem Objekt ausführen. Wenn der Client dieses Objekt nicht mehr benötigt, ruft der Client `objektZurueckgeben()` für den Pool auf. Dies ist in dem folgenden Sequenzdiagramm dargestellt:

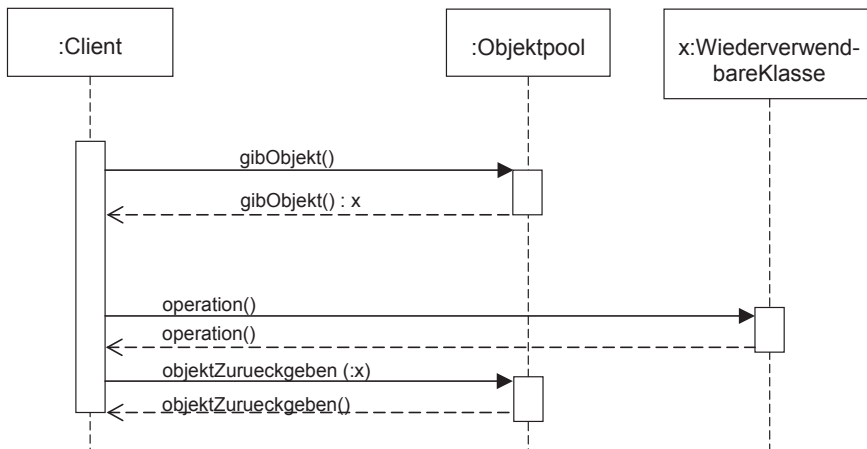


Bild 4-65 Abholen eines Objekts im Objektpool und Rückgabe

Das Verhalten des Objektpools im Falle, dass bei einer Anfrage `gibObjekt()` kein freies Objekt mehr im Pool zur Verfügung steht, wird durch das Muster nicht festgelegt. So könnte etwa der Objektpool ein neues Objekt erzeugen und dem Client das neu erstellte Objekt übergeben. Diese Situation ist in Bild 4-65 nicht dargestellt, sondern das gezeigte Sequenzdiagramm geht davon aus, dass noch freie Objekte verfügbar sind.

#### 4.21.3.4 Programmbeispiel

Als Beispiel für einen Objektpool wird eine Taxizentrale vorgestellt. Die Taxizentrale repräsentiert hierbei einen Objektpool, dessen Objekte Taxis sind. Die Zentrale vermittelt freie Taxis an Passagiere, die die Taxis verwenden und nach Gebrauch wieder an den Objektpool zurückgeben.

Die Klasse `Taxi` stellt zwei wichtige Methoden zur Verfügung, nämlich dass ein Passagier einsteigen und am Ziel wieder aussteigen kann:

```
// Datei: Taxi.java
public class Taxi
{
    private Passagier passagier;
    private int nummer = 0;
```

```

public Taxi (int nummer)
{
    this.nummer = nummer;
}

public int getNummer()
{
    return nummer;
}

public void passagierSteigtAus()
{
    System.out.println ("Aus Taxi " + this.getNummer()
        + " ist Passagier " + passagier.getName()
        + " ausgestiegen.");
    passagier = null;
}

public void passagierSteigtEin (Passagier passagier)
{
    this.passagier = passagier;

    System.out.println ("In Taxi " + this.getNummer()
        + " ist Passagier " + passagier.getName()
        + " eingestiegen.");
}
}

```

Passagiere sind in diesem Beispiel die Kunden der Taxizentrale. Ein Passagier kann ein Taxi betreten und am Ziel wieder verlassen. Um ein Taxi betreten zu können, wird ein Taxi von der Zentrale angefordert. Beim Verlassen wird das Taxi als frei an die Zentrale gemeldet. Dies spiegelt sich in den Methoden der Klasse `Passagier` wieder:

**// Datei: Passagier.java**

```

public class Passagier
{
    private String name = "";
    private Taxi taxi;

    public Passagier (String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

    public void taxiBetreten (TaxiZentrale taxiZentrale)
    {
        taxi = taxiZentrale.taxiAnfordern();
    }
}

```

```

        if (taxi == null) // Es ist kein freies Taxi vorhanden
            System.out.println ("Fuer Passagier " + this.getName()
                + " ist kein freies Taxi vorhanden.");
        else
            taxi.passagierSteigtEin (this);
    }

    public void taxiVerlassen (TaxiZentrale taxiZentrale)
    {
        if (taxi != null)
        {
            taxi.passagierSteigtAus();
            taxiZentrale.taxiFreigeben (taxi);
        }
    }
}

```

Die Klasse `TaxiZentrale` stellt den eigentlichen Objektpool dar. Diese Klasse verwaltet ein Objekt der Klasse `Vector` zum Speichern der Taxis. Die beiden Methoden `taxiAnfordern()` und `taxiFreigeben()` dienen zum Anfordern bzw. zum Zurückgeben eines Taxis an den Objektpool. Da diese Methoden die Liste verfügbarer Taxis bearbeiten, sollten sie threadsicher gemacht werden. Dies erfolgt über das Schlüsselwort `synchronized`. Um sicherzustellen, dass der Taxi-Pool eine einzige, zentrale Verwaltungsstelle für Taxis darstellt, ist die Klasse `TaxiZentrale` zusätzlich als Singleton (siehe Kapitel 4.20) implementiert. Hier der Quellcode der Klasse `TaxiZentrale`:

```

// Datei: TaxiZentrale.java
import java.util.Vector;

public class TaxiZentrale
{
    private int size = 2; // Es gibt leider nur 2 Taxis
    private static TaxiZentrale taxiZentrale = new TaxiZentrale();
    private Vector<Taxi> taxis;

    private TaxiZentrale()
    {
        taxis = new Vector<Taxi> (size);

        for (int i = 1; i <= size; i++)
            taxis.add (new Taxi (i));
        System.out.println ("Neue Taxizentrale mit " + size
            + " verwalteten Taxis erzeugt.");
    }

    public static TaxiZentrale gibZentrale()
    {
        return taxiZentrale;
    }

    public synchronized Taxi taxiAnfordern()
    {

```

```

        if (taxis.size() > 0)
        {
            Taxi taxi = taxis.get (0);
            taxis.remove (taxi);
            return taxi;
        }

        // Kein Taxi frei
        return null;
    }

    public synchronized void taxiFreigegeben (Taxi taxi)
    {
        taxis.add (taxi);
    }
}

```

Die Klasse `TestTaxiZentrale` erzeugt drei Instanzen der Klasse `Passagier`. Diese Passagiere fordern nacheinander ein Taxi bei der `TaxiZentrale` an. Da die `TaxiZentrale` im Beispiel nur zwei Taxis bereitstellt, bekommt der dritte Passagier Klaus erst dann ein Taxi, wenn ein anderer Passagier sein Taxi wieder verlassen hat. Hier die Klasse `TestTaxiZentrale`:

**// Datei: TestTaxiZentrale.java**

```

public class TestTaxiZentrale
{
    public static void main (String[] args)
    {
        TaxiZentrale taxiZentrale = TaxiZentrale.gibZentrale();

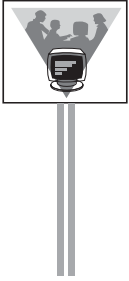
        Passagier hans = new Passagier ("Hans");
        Passagier anna = new Passagier ("Anna");
        Passagier klaus = new Passagier ("Klaus");

        hans.taxiBetreten (taxiZentrale);
        anna.taxiBetreten (taxiZentrale);
        klaus.taxiBetreten (taxiZentrale);

        hans.taxiVerlassen (taxiZentrale);
        klaus.taxiBetreten (taxiZentrale);

        anna.taxiVerlassen (taxiZentrale);
        klaus.taxiVerlassen (taxiZentrale);
    }
}

```



#### Hier das Protokoll des Programmlaufs:

```
Neue Taxizentrale mit 2 verwalteten Taxis erzeugt.  
In Taxi 1 ist Passagier Hans eingestiegen.  
In Taxi 2 ist Passagier Anna eingestiegen.  
Für Passagier Klaus ist kein freies Taxi vorhanden.  
Aus Taxi 1 ist Passagier Hans ausgestiegen.  
In Taxi 1 ist Passagier Klaus eingestiegen.  
Aus Taxi 2 ist Passagier Anna ausgestiegen.  
Aus Taxi 1 ist Passagier Klaus ausgestiegen.
```

## 4.21.4 Bewertung

### 4.21.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Wird der Mechanismus eines Objektpools eingesetzt, so erspart man sich durch die Wiederverwendung von Objekten ein ständiges Erzeugen dieser Objekte zur Laufzeit. Die Ressourcen werden geschont.
- Durch Beobachtung des Pools können Rückschlüsse auf die aktuelle Systemlast gezogen werden.

### 4.21.4.2 Nachteile

Folgende Nachteile können durch den Einsatz eines Objektpools entstehen:

- Die Rückgabe der Objekte ist ein zentrales Problem dieses Musters. Der Pool ist auf den "guten Willen" der Clients angewiesen, dass diese ein Objekt nach Benutzung auch tatsächlich wieder zurückgeben. Besonders im Falle von Ausnahmen (Exceptions) kann leicht vergessen werden, ein Objekt an den Pool zurückzugeben.
- Die wiederzuverwendenden Objekte müssen sich vor ihrer Wiederverwendung immer im gleichen Zustand befinden.
- In nebenläufigen Umgebungen synchronisiert der Pool die nebenläufigen Einheiten. Dies kann zu **Deadlocks** führen.

## 4.21.5 Einsatzgebiete

Das Entwurfsmuster eines Objektpools wird häufig bei Datenbankzugriffen verwendet, um Verbindungen zur Datenbank zwischenspeichern und wiederzuverwenden. Es wird dann in der Regel von einem **Connection Pool** gesprochen. Dieser Connection Pool ist insbesondere auch bei Web-Technologien sinnvoll, da hierbei Verbindungen oftmals nur für eine kurze Dauer genutzt werden [db2cop].

Bei parallelen Systemen werden oft sogenannte **Thread Pools** eingesetzt. So müssen Threads nicht ständig neu erstellt werden, sondern sie werden einfach wiederverwen-

det. Ebenso wie das Erstellen von Datenbankverbindungen ist auch das Erstellen von Threads aufwendig.

#### 4.21.6 Ähnliche Entwurfsmuster

Das Muster Objektpool hat eine gewisse Ähnlichkeit mit dem Entwurfsmuster **Singleton**, da durch beide Muster die Anzahl der Objekte kontrolliert wird. Beim Objektpool ist allerdings die Anzahl der verfügbaren Objekte nicht wie beim Singleton auf eins beschränkt. Ein weiterer Unterschied ist, dass ein Singleton-Objekt nicht exklusiv von einem einzigen Klienten angefordert und genutzt wird und daher auch nicht wieder zurückgegeben werden muss, sondern dass ein Singleton mehreren Klienten gleichzeitig über eine Referenz zur Verfügung stehen kann.

Für einen Client wirkt die Methode `gibObjekt()` der Klasse `Objektpool` wie eine **Fabrikmethode**, da der Pool die Clients mit Objekten vom Typ `WiederverwendbareKlasse` versorgt. Diese Objekte werden allerdings nicht in der Methode `gibObjekt()` erzeugt, sondern sie werden im Pool vorgehalten und über die Methode `gibObjekt()` nur zur Verfügung gestellt.

In [Gr302] wird das Muster **Thread Pool** als eigenständiges Entwurfsmuster beschrieben. Es wurde im vorherigen Abschnitt als Variante des Musters Objektpool vorgestellt.



## 4.22 Zusammenfassung

Entwurfsmuster sind bewährte Ansätze für die Lösung bestimmter Probleme beim Entwurf. Praktisch alle Entwurfsmuster ermöglichen mit ihrem Lösungsansatz eine gewisse Erweiterbarkeit und damit eine Wiederverwendbarkeit. Darüber hinaus bieten diese Lösungsansätze eine größere Flexibilität, beispielsweise dadurch, dass Implementierungsentscheidungen auf Unterklassen verschoben oder in Objekten gekapselt werden, womit die Entscheidungen austauschbar werden. Die genannten Vorteile werden allerdings mit Aufwand, d. h. Zeitverbrauch und Speicherbedarf, erkauft. Beim Einsatz eines Entwurfsmusters muss man stets prüfen, ob für die Erweiterbarkeit oder die Flexibilität überhaupt Bedarf besteht.

Jedes der Muster ist für eine ganz bestimmte Problemstellung anwendbar. Die Kunst besteht deshalb in der Eingrenzung des Problems. Diese Eingrenzung erlaubt es, zwischen ähnlichen Entwurfsmustern die richtige Wahl zu treffen.

Man kann nach [Gam95] bei den Entwurfsmustern nach den Klassen Strukturmuster, Verhaltensmuster und Erzeugungsmuster unterscheiden.

Entwurfsmuster können ferner in zwei weitere Kategorien eingeteilt werden: **klassenbasierte** und **objektbasierte Muster**. Bei klassenbasierten Entwurfsmustern wird das Muster zur Kompilierzeit festgelegt. Die klassenbasierten Muster verwenden hierbei die statische Bildung von Unterklassen. Bei objektbasierten Mustern können Objekte zur Laufzeit oft ausgetauscht werden. Die Bildung von Objekten von Unterklassen erfolgt hierbei dynamisch zur Laufzeit. Der allergrößte Teil der vorgestellten Entwurfsmuster ist objektbasiert. Lediglich die hier vorgestellten Muster Schablonenmethode und Fabrikmethode sind klassenbasiert. Das Adapter-Muster kann sowohl klassenbasiert als auch objektbasiert gelöst werden.

**Strukturmuster** (siehe Kapitel 4.2.1) befassen sich mit dem Aufbau von Strukturen aus Klassen und Objekten. Im Folgenden werden die einzelnen Strukturmuster, die in diesem Buch besprochen wurden, kurz beschrieben:

- Das **Adapter-Muster** (siehe Kapitel 4.3) passt eine vorhandene "falsche" Schnittstelle an die gewünschte Form an.
- Das **Brücke-Muster** (siehe Kapitel 4.4) trennt die Klassenhierarchie einer Abstraktion und ihrer Implementierung. Dadurch wird erreicht, dass diese beiden Teile getrennt verändert und erweitert werden können. Man verbindet beide Teile durch eine "Brücke". Für den Client ist nur die Hierarchie der Abstraktion sichtbar.
- Das **Dekorierer-Muster** (siehe Kapitel 4.5) soll es erlauben, zur Laufzeit eine zusätzliche Funktionalität zu einem vorhandenen Objekt einer Klasse oder Subklasse in dynamischer Weise hinzuzufügen.
- Das **Fassade-Muster** (siehe Kapitel 4.6) soll eine meist vereinfachte abstrakte Schnittstelle zum Zugriff auf die Klassen eines Subsystems bereitstellen.
- Das **Kompositum-Muster** (siehe Kapitel 4.7) erlaubt es, dass bei der Verarbeitung von Elementen in einer Baumstruktur einfache und zusammengesetzte Objekte gleich behandelt werden. Eine Typprüfung entfällt.

- Das **Proxy-Muster** (siehe Kapitel 4.8) verbirgt die Existenz eines Objekts hinter einem Stellvertreter mit derselben Schnittstelle. Der Stellvertreter kapselt die Kommunikation zum echten Objekt. Er kann Funktionen an das echte Objekt weiterdelegieren und dabei auch Zusatzfunktionalität hinzufügen.

**Verhaltensmuster** (siehe Kapitel 4.2.2) beschreiben Interaktionen zwischen Objekten in bestimmten Rollen zur gemeinsamen Lösung eines bestimmten Problems. Die folgenden Verhaltensmuster wurden in diesem Kapitel ausführlich beschrieben:

- Das Muster **Schablonenmethode** (siehe Kapitel 4.9) legt bereits in einer Basisklasse die Struktur eines Algorithmus fest. Realisiert werden variante Teile des Algorithmus statisch in Unterklassen.
- Das **Befehlsmuster** (siehe Kapitel 4.10) soll die Details eines Befehls vor dem Aufrufer des Befehls verbergen. Es soll es erlauben, dass die Erzeugungszeit und die Ausführungszeit des Befehls getrennt werden können.
- Das **Beobachter-Muster** (siehe Kapitel 4.11) soll es erlauben, dass ein Objekt abhängige Objekte von einer Änderung seines Zustands automatisch informiert, so dass eine Aktualisierung automatisch eingeleitet werden kann.
- Das **Strategie-Muster** (siehe Kapitel 4.12) soll es gestatten, dass ein ganzer Algorithmus in Form einer Kapsel ausgetauscht wird.
- Das **Vermittler-Muster** (siehe Kapitel 4.13) soll es erlauben, dass Objekte miteinander über einen Vermittler reden, der alle von einem eingehenden Methodenaufruf betroffenen Kollegen informiert. Dadurch bleiben die einzelnen Objekte unabhängig voneinander und sind austauschbar. Das Vermittler-Muster stellt die Koordination einer ganzen Gruppe von Objekten durch den zentralen Vermittler in den Vordergrund.
- Im **Zustandsmuster** (siehe Kapitel 4.14) werden die einzelnen Zustände durch eigene Klassen gekapselt, die von einer gemeinsamen abstrakten Basisklasse ableiten bzw. eine gemeinsame Schnittstelle als Abstraktion implementieren. Ein zustandsabhängiges Kontextobjekt referenziert den aktuellen Zustand und führt im Rahmen des Grundmusters Zustandsänderungen durch.
- Das **Rollen-Muster** (siehe Kapitel 4.15) soll es erlauben, dass ein Objekt dynamisch die Rollen wechseln und mehrere Rollen gleichzeitig annehmen kann, sowie, dass mehrere Objekte dieselbe Rolle haben können.
- Das **Besucher-Muster** (siehe Kapitel 4.16) soll es gestatten, eine neue Operation auf einer Datenstruktur aus Objekten durchzuführen. Beim Besucher-Muster müssen die zu besuchenden Objekte auf den Besuch "vorbereitet" sein, dadurch dass sie eine entsprechende Methode dem Besucher zur Verfügung stellen.
- Das **Iterator-Muster** (siehe Kapitel 4.17) soll es erlauben, eine Datenstruktur in verschiedenen Durchlaufstrategien zu durchlaufen, ohne dass der Client den Aufbau der Datenstruktur kennt.

**Erzeugungsmuster** (siehe Kapitel 4.2.3) machen ein System unabhängig davon, auf welche Weise seine Objekte erzeugt werden. Die folgenden kurzen Zusammenfassungen beschreiben die einzelnen Erzeugungsmuster, die in diesem Buch vorgestellt wurden:

- Mit dem **Fabrikmethode-Muster** (siehe Kapitel 4.18) soll die Erzeugung einer konkreten Instanz in der Methode einer Unterklasse gekapselt werden. Die Unterklassen werden durch Vererbung statisch gewonnen.
- Das Muster **Abstrakte Fabrik** (siehe Kapitel 4.19) soll es erlauben, dass durch Wahl der entsprechenden konkreten Fabrik diejenige Produktfamilie zur Laufzeit ausgewählt werden kann, aus der Objekte erzeugt werden sollen.
- Das **Singleton-Muster** (siehe Kapitel 4.20) soll es gewährleisten, dass eine Klasse nur ein einziges Mal instanziiert werden kann.
- Das Muster **Objektpool** (siehe Kapitel 4.21) soll es ermöglichen, Instanzen zur Verfügung zu stellen, dabei diese wiederzuverwenden und so Ressourcen zu schonen, anstatt die Instanzen immer wieder neu zu erzeugen.

## 4.23 Aufgaben

### Aufgaben 4.23.1: Allgemeine Fragen

- 4.23.1.1 Welche beiden Entwurfsmuster haben dasselbe Klassendiagramm, das aber jeweils anders interpretiert wird?
- 4.23.1.2 Wann zieht man eine abstrakte Basisklasse einem Interface vor?
- 4.23.1.3 Was ist ein Strukturmuster?
- 4.23.1.4 Was ist ein Verhaltensmuster?
- 4.23.1.5 Was ist ein Erzeugungsmuster?

### Aufgaben 4.23.2: Adapter

- 4.23.2.1 Welche Ziel hat das Adaptermuster?
- 4.23.2.2 Was ist der Unterschied zwischen dem Proxy- und dem Adapter-Muster?

### Aufgaben 4.23.3: Brücke

- 4.23.3.1 Wie funktioniert das Brücke-Muster?
- 4.23.3.2 Wo ist die Brücke zu sehen?

### Aufgaben 4.23.4: Dekorierer

- 4.23.4.1 Welches Entwurfsmuster hat bis auf eine andere Multiplizität das gleiche Klassendiagramm wie das Dekorierer-Muster?
- 4.23.4.2 Wozu dient das Dekorierer-Muster?

### Aufgaben 4.23.5: Fassade

- 4.23.5.1 Wozu dient das Fassade-Muster?
- 4.23.5.2 Kann das Fassade-Muster den Zugriff auf die Subsystemklassen verhindern? Begründen Sie Ihre Antwort.

### Aufgaben 4.23.6: Kompositum

- 4.23.6.1 Welche Rollen spielen die Klassen `Knoten`, `Kompositum` und `Blatt` im Rahmen des Kompositum-Patterns.
- 4.23.6.2 Eine `Blatt`-Klasse erbt von der Basisklasse `Knoten` die Kindfunktionen, hat aber keine Kinder. Wie geht man üblicherweise vor?

### Aufgaben 4.23.7: Proxy

- 4.23.7.1 Was ist das Ziel des Proxy-Musters?
- 4.23.7.2 Was ist ein Schutz-Proxy?

### Aufgaben 4.23.8: Schablonenmethode

- 4.23.8.1 Was hat das Muster Schablonenmethode zum Inhalt?
- 4.23.8.2 Arbeitet das Muster Schablonenmethode klassenbasiert oder objektbasiert?

**Aufgaben 4.23.9: Befehl**

- 4.23.9.1 Was ist das Ziel des Befehlsamusters?
- 4.23.9.2 Nennen Sie 2 Anwendungsgebiete für das Befehlsamuster.

**Aufgaben 4.23.10: Beobachter**

- 4.23.10.1 Erklären Sie das Prinzip einer Callback-Schnittstelle anhand des Beobachter-Musters.
- 4.23.10.2 Kennt der Beobachtete beim Beobachter-Muster die komplette Implementierung seiner Beobachter?

**Aufgaben 4.23.11: Strategie**

- 4.23.11.1 Wozu wird das Strategie-Muster eingesetzt?
- 4.23.11.2 Erklären Sie den Unterschied zwischen den Verhaltensmustern Strategie und Schablonenmethode.

**Aufgaben 4.23.12: Vermittler**

- 4.23.12.1 Was ist das Ziel des Vermittler-Musters?
- 4.23.12.2 Zeichnen Sie die für das Vermittler-Muster zutreffende Topologie.

**Aufgaben 4.23.13: Zustand**

- 4.23.13.1 Was ist das Ziel des Zustandsamusters?
- 4.23.13.2 Hängt der Kontext zur Kompilierzeit (statisch) von den konkreten Zustandsklassen ab?

**Aufgaben 4.23.14: Rolle**

- 4.23.14.1 Definieren Sie den Begriff der Rolle.
- 4.23.14.2 Was ist das Ziel des Rollen-Musters?
- 4.23.14.3 Wie wird beim Rollen-Muster erreicht, dass ein Objekt zur Laufzeit verschiedene Rollen annehmen kann?
- 4.23.14.4 Was ist der Unterschied zwischen dem Rollen-Muster und der Spezialisierung durch Ableitung?

**Aufgaben 4.23.15: Besucher**

- 4.23.15.1 Was ist das Ziel des Besucher-Musters?
- 4.23.15.2 Beschreiben Sie die Aufgaben eines Besucher-Objekts.

**Aufgaben 4.23.16: Iterator**

- 4.23.16.1 Was ist das Ziel des Iterator-Musters?
- 4.23.16.2 Nennen Sie ein Entwurfsmuster, das dem Iterator-Muster ähnelt.
- 4.23.16.3 Was ist der grundlegende Gedanke des Iterator-Musters?

**Aufgaben 4.23.17: Fabrikmethode**

- 4.23.17.1 Was ist das Ziel des Musters Fabrikmethode?
- 4.23.17.2 Wann wird eine Fabrikmethode eingesetzt?
- 4.23.17.3 Welche Nachteile ergeben sich durch die Verwendung des Fabrikmethode-Musters?

**Aufgaben 4.23.18: Abstrakte Fabrik**

- 4.23.18.1 Was ist das Ziel des Musters Abstrakte Fabrik?
- 4.23.18.2 Was ist der Unterschied zwischen den beiden Erzeugungsmustern Fabrikmethode und Abstrakte Fabrik?

**Aufgaben 4.23.19: Singleton**

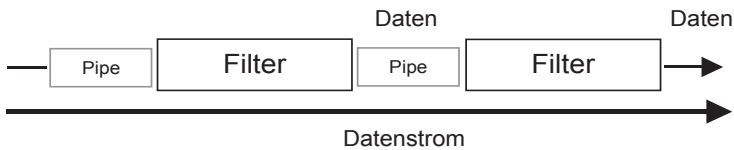
- 4.23.19.1 Wozu wird das Singleton-Muster eingesetzt?
- 4.23.19.2 Skizzieren Sie die Lösung beim Entwurfsmuster Singleton.

**Aufgaben 4.23.20: Objektpool**

- 4.23.20.1 Was ist das Ziel des Musters Objektpool?
- 4.23.20.2 Wofür wird im Zusammenhang mit Datenbanken das Objektpool-Muster eingesetzt?

# Kapitel 5

## Architekturmuster



- 5.1 Das Architekturmuster Layers
- 5.2 Das Architekturmuster Pipes and Filters
- 5.3 Das Architekturmuster Plug-in
- 5.4 Das Architekturmuster Broker
- 5.5 Das Architekturmuster Service-Oriented Architecture
- 5.6 Das Architekturmuster Model-View-Controller
- 5.7 Zusammenfassung
- 5.8 Aufgaben

## 5 Architekturmuster

Entwurfsmuster stellen feinkörnige Muster dar, während Architekturmuster grobkörnig sind. Architekturmuster lösen nicht ein Teilproblem, sondern beeinflussen die Grundzüge der Architektur eines Systems.



Die in diesem Kapitel vorgestellten Architekturmuster sind den Autoren in der Praxis besonders oft begegnet. Sie können in die folgenden Kategorien – angelehnt an [Bus98] – eingeteilt werden:

- **Struktur eines Systems:** Layers, Pipes and Filters,
- **Adaptive Systeme:** Plug-in,
- **Verteilte Systeme:** Broker, Service-Oriented Architecture,
- **Interaktive Systeme:** Model-View-Controller.

Es folgt eine kurze Übersicht über diese Muster:

Das Architekturmuster **Layers** (siehe Kapitel 5.1) schneidet die Architektur eines Systems in Schichten, wobei jede Schicht auf die Dienste der darunterliegenden Schicht zugreifen kann (horizontale Schichtung eines Systems). Diese Architektur setzt voraus, dass sich ein sinnvoller Ablauf ergibt, wenn ein Client die oberste Schicht des Systems als Server ruft, diese in der Rolle eines Clients wieder die nächst tiefere Schicht als Server usw. bis hin zur letzten Schicht.

Das Architekturmuster **Pipes and Filters** (siehe Kapitel 5.2) strukturiert in seiner Grundform eine Anwendung in eine Kette von sequenziellen Verarbeitungsprozessen (Filtern), die über ihre Ausgabe bzw. Eingabe gekoppelt sind: Die Ausgabe eines Prozesses ist die Eingabe des nächsten Prozesses (**datenflussorientierte Architektur eines Systems**). Diese Architektur macht Sinn, wenn sich ein System in eine Kette von Prozessen gliedern lässt, wobei jeder Prozess das Ergebnis des vorangehenden Prozesses übernimmt und weiterverarbeitet. Eine Pipe dient hierbei zur asynchronen Entkopplung zweier benachbarter Prozesse.

Das Architekturmuster **Plug-in** (siehe Kapitel 5.3) strukturiert eine spezielle Anwendung so, dass sie über Erweiterungspunkte in Form von Schnittstellen verfügt, an denen Plug-ins, die diese Schnittstellen implementieren, vom Plug-in-Manager (unter Verwendung der Laufzeitumgebung) instanziiert und eingehängt werden können. Die Anwendung ist aber ohne diese zusätzlichen Erweiterungen lauffähig. Eine solche Architektur lohnt sich, wenn die Basisanwendung ohne die Erweiterungen einem breiten Anwenderkreis zur Verfügung steht und die speziellen Erweiterungen für bestimmte Benutzergruppen gedacht sind.



Das Architekturmuster **Broker** (siehe Kapitel 5.4) strukturiert ein System aus mehreren Clients und Servern in eine Architektur mit einem Broker als vermittelnde Instanz für das Finden des passenden Servers für eine Anfrage des Clients nach einem Service und das Zurückliefern der Antwort des Servers an den Client. Client- und Server-Komponenten kommunizieren untereinander nur über einen Broker. In einem verteilten System ist der Broker selbst als Middleware auf alle Knotenrechner des Systems verteilt.

Das Architekturmuster **Service-Oriented Architecture**<sup>71</sup> (siehe Kapitel 5.5) gliedert die Architektur eines Systems in Komponenten bzw. Teilkomponenten, die den Anwendungsfällen bzw. Teilen von Anwendungsfällen aus Sicht der Systemanalyse entsprechen und deren Leistungen als Service zur Verfügung stellen. Damit sollen bei einer Abänderung der Geschäftsprozesse nur:

- die entsprechenden Komponenten als Verkörperung der Anwendungsfälle abgeändert werden müssen oder
- ggfs. unter Verwendung bestehender elementarer Services neue Komponenten erzeugt werden.

Das Architekturmuster **Model-View-Controller**<sup>72</sup> (siehe Kapitel 5.6) trennt eine Anwendung in die Komponenten Model (Verarbeitung/Datenhaltung), View (Ausgabe) und Controller (Eingabe). Das Model hängt dabei nicht von der Ein- und Ausgabe ab. Mit dieser Strategie ist es leicht möglich, View und Controller im System auszutauschen (Trennung der Oberflächenkomponenten von der Verarbeitung). Damit kann der Kern einer Anwendung länger leben als die Programmteile für die Ein- und Ausgabe.

Ein **Architekturmuster** kann verschiedene Entwurfsmuster enthalten. Ob überhaupt und wie viele Entwurfsmuster miteinander kombiniert werden müssen, damit ein Architekturmuster entsteht, ist von Fall zu Fall verschieden. Das Architekturmuster Model-View-Controller (MVC) enthält in der Regel das Beobachter-Muster, das Strategie-Muster sowie das Kompositum-Muster. Das Architekturmuster Layers verwendet hingegen kein einziges Entwurfsmuster.

Während alle im vorherigen Kapitel gezeigten Entwurfsmuster objektorientiert sind, können Muster generell auch einen nicht objektorientierten Charakter haben. So ist beispielsweise das Architekturmuster Layers oder das Muster Plug-in nicht an die Verwendung objektorientierter Techniken geknüpft.

---

<sup>71</sup> Service Oriented Architecture wird abgekürzt durch SOA.

<sup>72</sup> Model-View-Controller wird abgekürzt durch MVC.

## 5.1 Das Architekturmuster Layers

Das Architekturmuster Layers modelliert die **Struktur eines Systems** in Schichten mit Client/Server-Beziehungen zwischen den Schichten.

### 5.1.1 Name/Alternative Namen

Schichtenmodell, Layers.

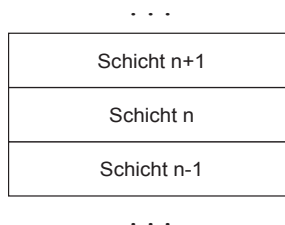
### 5.1.2 Problem

Ein Softwaresystem soll in eine überschaubare Zahl von Teilsystemen als horizontale Schichten geschnitten werden. Miteinander verwandte Aspekte des Systems sollen jeweils in einer eigenen Schicht konzentriert werden. Jede Schicht soll eine Abstraktion der in ihr enthaltenen Funktionalität sein. Funktionen einer höheren Schicht sollen sich auf die Funktionen der nächst tieferen Schicht beziehen, diese aufrufen und die Antwort zurückerhalten. Zwischen jeweils zwei Schichten soll also das Client-Server-Prinzip gelten, wobei die Server-Schicht wiederum als Client auftreten soll, wenn sie auf die nächst tiefere Schicht zugreift.

Auf diese Weise soll die Zahl der Abhängigkeitsbeziehungen eingeschränkt werden. Die Entwicklung der einzelnen Funktionen soll dadurch erleichtert werden und mögliche Code-Änderungen sollen lokal in ihrer Auswirkung bleiben.

### 5.1.3 Lösung

Aufgrund zu hoher Komplexität eines Systems wird dessen Softwarearchitektur in mehrere übereinanderliegende horizontale Schichten eines sogenannten Schichtenmodells gegliedert. Dabei enthält jede Schicht (engl. **tier** oder **layer**) konzeptionell eine bestimmte Unteraufgabe eines Systems in abstrakter Form (z. B. Kommunikation, Datenhaltung). Das Aufteilen in Schichten erfolgt in der Regel nach funktionalen Gesichtspunkten. Im Folgenden die Skizze eines Schichtenmodells:



*Bild 5-1 Schichtenmodell*

Dabei soll eine tiefere Schicht einer höheren Schicht Funktionalität bereitstellen, indem sie Dienste (engl. services) anbietet und ein Ergebnis an die aufrufende Schicht liefert. Sie soll also auf Dienstanforderungen antworten. Für die Schichten gilt das Client-Server-Prinzip.

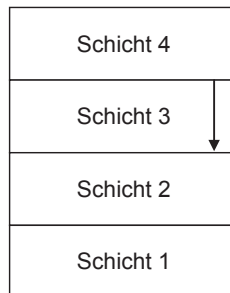
Es gilt der folgende Satz von Regeln:

1. Eine Schicht  $n$  hängt nur von der jeweils tiefer liegenden Schicht  $n - 1$  ab.
2. Eine Schicht hängt nicht von den höheren Schichten ab.
3. Jede Schicht bietet Dienste für die jeweils höher liegende Schicht an.
4. Der Zugriff auf einen Dienst der nächst tieferen Schicht erfolgt über die Schnittstellen dieser Schicht.

Eine tiefere Schicht kann die Dienste einer höheren Schicht nicht aufrufen. Eine höhere Schicht kann jedoch auf die Dienste der nächst tieferen Schichten zugreifen.



Jede Schicht verfügt über Schnittstellen zu Services der nächst tieferen Schicht. Die Zahl der Schichten ist nicht von vorneherein festgelegt. Sie sollte aber überschaubar sein. Eine höhere Schicht darf generell nur tiefere Schichten verwenden. In der Grundform des Musters darf eine Schicht nur auf die direkt darunter liegende Schicht zugreifen. Dies wird auch **Strict Layering** genannt. Der Begriff **Layer Bridging** wird dagegen verwendet, wenn eine Schicht auf alle unter ihr liegenden Schichten zugreifen darf. Mit Layer Bridging wird Regel 1 aufgeweicht. Das folgende Bild gibt ein Beispiel für Layer Bridging:



*Bild 5-2 Beispiel für Layer Bridging*

In Bild 5-2 greift die Schicht 4 direkt auf die Schicht 2 zu.

Beim **Strict Layering** darf eine Schicht nur auf die direkt darunter liegende Schicht zugreifen, beim **Layer Bridging** hingegen darf sie auf alle unter ihr liegenden Schichten direkt zugreifen.



Jede Schicht stellt eine Abstraktionsebene im System dar. In der älteren Literatur (Beispiel: [Dij68]) wird eine Schicht auch als abstrakte Maschine<sup>73</sup> bezeichnet: Die über die Schnittstelle einer Schicht nach oben zur Verfügung gestellten Funktionen werden da-

<sup>73</sup> Der Begriff abstrakte Maschine ist hier im Gegensatz zu einer realen Maschine zu verstehen, die tatsächlich in Hardware realisiert vorliegt. Heute würde man eher den Begriff virtuelle Maschine verwenden.

bei auch als Operationen einer abstrakten Maschine angesehen. Insgesamt entsteht somit eine Hierarchie von abstrakten Maschinen.

### 5.1.3.1 Klassendiagramm

In diesem Kapitel werden die Schichten und ihre Funktionen als objektorientierte Lösung mit Klassen vorgestellt. Das Konzept der Schichten ist aber unabhängig von der Objektorientierung. Auf die Darstellung einer alternativen, nicht objektorientierten Variante wird verzichtet. Das Schichtenmodell soll hier in seiner Grundform diskutiert werden.

Eine Klasse einer Schicht  $n$  benutzt eine Klasse der Schicht  $n - 1$ . Eine Schicht wird als Komponente (Subsystem) dargestellt. Eine Komponente wird wiederum durch Klassen implementiert. Dies zeigt Bild 5-3:

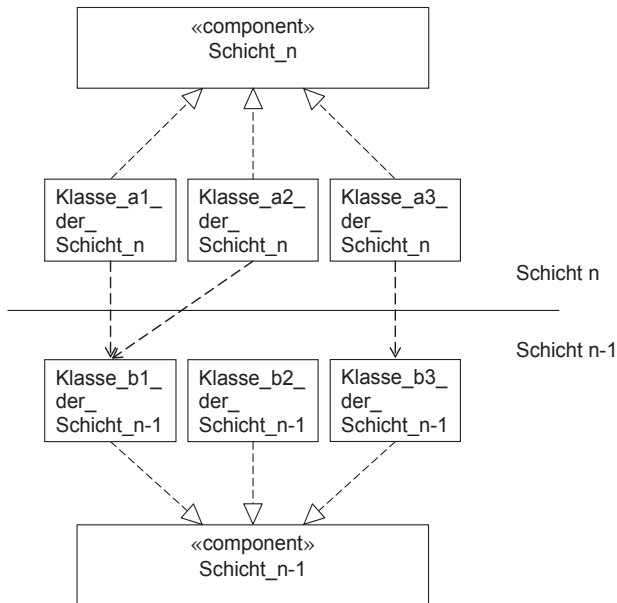


Bild 5-3 Beispiel für die Abhängigkeitsbeziehung zwischen Objekten benachbarter Komponenten

Instanziiert wird hierbei nicht die Komponente `Schicht_n`, sondern nur die Klassen, welche die Komponente realisieren. Die Komponente hier ist eine Abstraktion, die von Klassen implementiert wird. Es handelt sich um eine sogenannte **indirekte Implementierung** [Hit05, Seite 145].

### 5.1.3.2 Teilnehmer

Teilnehmer an diesem Architekturmuster sind die Objekte einer Schicht und die Objekte der darunterliegenden Schicht:

**Klasse\_aX\_der\_Schicht\_n**

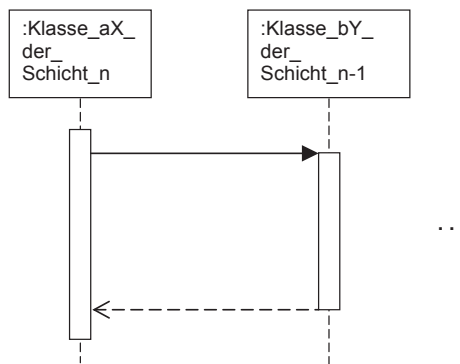
Ein Objekt dieser Klasse fordert einen Service eines Objektes der Klasse `Klasse_bY_der_Schicht_n-1` an.

**Klasse\_bY\_der\_Schicht\_n-1**

Ein Objekt dieser Klasse erbringt einen Service. Dabei kann es wiederum Objekte der direkt unter ihm liegenden Schicht aufrufen, falls eine solche Schicht existiert.

**5.1.3.3 Dynamisches Verhalten**

Das folgende Bild zeigt das Client-Server-Verhalten der Schichten exemplarisch:



*Bild 5-4 Aufruf und Antwort eines Objekts der jeweils benachbarten tieferen Schicht*

Eine Schicht  $n - 1$  stellt Dienste für die Schicht  $n$  bereit und nutzt selbst wieder die Dienste der Schicht  $n - 2$ .

**5.1.4 Bewertung**

Nach dem Prinzip des "Teile und herrsche" wird ein schwer beherrschbares, komplexeres Problem in kleinere, möglichst unabhängige Teilprobleme zerlegt, die dann besser verständlich sind und einfacher gelöst werden können. Das Schichten-Muster folgt genau diesem Prinzip. Damit wird das zu lösende Problem durch die Schichtenbildung mit jeder Schicht kleiner. Die Strukturierung in Schichten erfolgt nach dem Client-Server-Prinzip.

**5.1.4.1 Vorteile**

Das Architekturmuster Layers hat die folgenden Vorteile:

- Jede Schicht stellt eine Abstraktion einer bestimmten Funktionalität dar und kann leichter verstanden werden als das Ganze.
- Schichten können unter Umständen wiederverwendet werden.
- Schichten sind stabil und können standardisiert werden.

- Code-Abhängigkeiten können minimiert werden, d. h. Änderungen am Quellcode sollten wenige Ebenen betreffen.
- Nach Festlegung der Schnittstellen können die einzelnen Ebenen parallel entwickelt werden.
- Die Schichten können sehr gut bottom-up integriert und getestet werden.

#### 5.1.4.2 Nachteile

Die folgenden Nachteile treten beim Architekturmuster Layers auf:

- Die Regel des Strict Layering, dass ein Zugriff nur auf die benachbarte Ebene erfolgt, ist oft zu einschränkend. Diese Regel kann aber umgangen werden, wenn die Performance gesteigert werden soll (Layer Bridging).
- Es liegt auf der Hand, dass eine Anfrage, die von Schicht zu Schicht weitergereicht wird, zeitaufwendiger als ein direkter Zugriff ist.
- Änderungen können sich über Schichten hinaus auswirken. Manche Arbeiten wie z. B. eine Korrektur von Fehlern können alle Schichten betreffen. Dies bedeutet Mehraufwand.
- Es kann schwierig sein, die "richtige" Anzahl von Schichten zu finden.

#### 5.1.5 Einsatzgebiete

Der Einsatz des Musters Layers kann überall da erfolgen, wo nach dem Client-Server-Prinzip modelliert wird. Zusammengehörige Funktionen werden in einer eigenen Schicht abstrahiert. Oftmals stellt aber auch eine höhere Schicht eine Abstraktion einer tieferen Schicht dar. So abstrahiert z. B. die Schicht Betriebssystem die Schicht der Hardware eines Rechners.

Beispiele für Schichtenmodelle sind die Zusammenarbeit zwischen Rechner-Hardware, Betriebssystem und Anwendungssoftware (siehe Kapitel 5.1.5.1), das Schichtenmodell für die Anwendungssoftware von Rechnern (siehe Kapitel 5.1.5.2) oder das in Kapitel 5.1.5.4 gezeigte ISO/OSI-Modell für das Kommunikationssystem eines Rechners.

Praktisch jedes Application Programming Interface (API) stellt eine Schicht dar, die von der darunterliegenden Schicht abstrahiert. Als Beispiel hierfür seien die Standardbibliotheken der Programmiersprache C genannt: sie abstrahieren von den speziellen Gegebenheiten des darunter liegenden Betriebssystems. Speziell bei der Ein- und Ausgabe werden sogar noch zwei Schichten unterschieden: die low-level sowie die high-level Ein- und Ausgabe.

Auch virtuelle Maschinen stellen ein Anwendungsgebiet des Architekturmusters Layers dar. Beispielsweise abstrahiert die Java-Virtuelle Maschine von der zugrunde liegenden realen Hardware. Die Java-Virtuelle Maschine ist in der Lage, einen sogenannten Bytecode zu verarbeiten, der wesentlich kompakter und abstrakter ist als normaler Maschinencode.

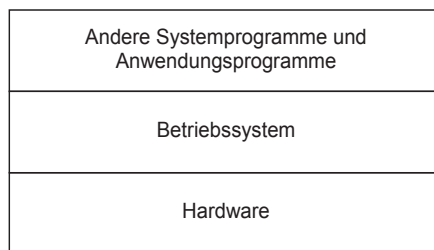
Die **Fassade** führt eine zusätzliche Schicht ein, welche die Klassen eines Teilsystems kapselt.

#### 5.1.5.1 Schichtenmodell für Hardware, Betriebssystem und Anwendungssoftware

Ein Betriebssystem ist ein Systemprogramm und hat zwei grundsätzliche Aufgaben:

- Es abstrahiert und verwaltet die Betriebsmittel des Rechners und
- es hat den Nutzer zu unterstützen.

Dies sieht man am besten am folgenden Schichtenmodell:



*Bild 5-5 Schichtenstruktur eines Rechners*

Zu den "anderen Systemprogrammen" gehört beispielsweise die Kommandoschnittstelle Shell bei Unix oder ein DBMS<sup>74</sup>.

Das Betriebssystem hat sowohl Schnittstellen zur Hardware als auch Schnittstellen zu den Programmen des Anwenders. Beide Schnittstellen muss das Betriebssystem "bedienen".

Das Betriebssystem verbirgt also die Hardware gegenüber den anderen System- und Anwendungsprogrammen. Es abstrahiert die Hardware und zeigt sich dem Benutzer gegenüber als **virtuelle Maschine**, die leichter zu programmieren ist als die zugrunde liegende Hardware. Die Ziele eines Betriebssystems sind folgende:

- Die Ressourcen eines Rechners sollen gerecht und fehlerfrei den Anwendern zur Verfügung gestellt werden. Das bedeutet, dass Prozessoren, Speicher und I/O-Geräte den konkurrierenden Programmen gerecht und geordnet zugeteilt werden sollen.
- Ein Softwareentwickler soll mit vernünftigen Aufwand Programme schreiben können, ohne dass er über detaillierte Kenntnisse der zugrundeliegenden Hardware verfügt.

Das Betriebssystem stellt die Verbindung zwischen den anderen System- und Anwendungsprogrammen und der Hardware des Rechners her. Es stellt seine Leistungen

<sup>74</sup> DBMS = Database Management System.

diesen Programmen zur Verfügung und ist eine Softwareschicht, die über der Hardware liegt.

### 5.1.5.2 Schichtenmodell für die Anwendungssoftware eines Rechners

Schichtenmodelle für Rechner finden nicht nur bei Informationssystemen, sondern auch bei eingebetteten Systemen<sup>75</sup> (engl. embedded systems) Verwendung. Im Folgenden soll nur auf Schichtenmodelle für Informationssysteme eingegangen werden. Das Betriebssystem wird dabei nicht berücksichtigt.

Die Anwendungssoftware eines Rechners beinhaltet im Allgemeinen folgende Funktionalitäten:

- Ein- und Ausgabe (Benutzerschnittstelle),
- Verarbeitung und
- persistente Datenhaltung.

Im Falle der Verwendung eines kommerziellen DBMS besteht die persistente Datenhaltung aus

- Datenzugriff (der Schnittstelle des DBMS und der dazugehörigen API für die verwendete Programmiersprache),
- dem DBMS-Kern,
- den Funktionen des Betriebssystems und
- der persistenten Datenhaltung auf der Festplatte.

Hierbei verbirgt das DBMS das Betriebssystem.

Im Falle des direkten Zugriffs auf die Festplatte ohne die Verwendung eines kommerziellen DBMS besteht die persistente Datenhaltung aus

- Datenzugriff (Schnittstelle des Dateisystems und der dazugehörigen API für die verwendete Programmiersprache),
- den Funktionen des Betriebssystems und
- der persistenten Datenhaltung auf der Festplatte.

Im weiteren Verlauf dieses Kapitels soll nur der Einsatz eines kommerziellen DBMS betrachtet werden und nicht das direkte Schreiben vom Programm aus auf die Festplatte bzw. das Lesen von der Festplatte.

Aus den zu Beginn dieses Kapitels genannten Funktionalitäten werden bei Einsatz eines DBMS die folgenden vier Schichten:

---

<sup>75</sup> Siehe beispielsweise die AUTOSAR-Architektur für die Automobilindustrie [Kin09].



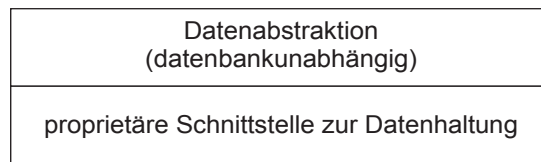
- I/O-Schicht,
- Verarbeitungsschicht,
- Datenzugriffsschicht und
- DBMS-Kern



Im Folgenden werden diese Schichten erläutert:

- Die **Schnittstelle zur Ein- und Ausgabe** (engl. **man-machine interface**, MMI) befindet sich innerhalb der sogenannten I/O-Schicht und kann die ereignisorientierten Anwendungsfälle anstoßen.
- Die Objekte der **Verarbeitungsschicht** stellen die Kontrollobjekte und die Entity-Objekte im Arbeitsspeicher (**transiente Datenhaltung**) dar.
- Die **Datenzugriffsschicht** abstrahiert die persistente Datenhaltungsschicht. Die Datenzugriffsschicht muss nicht stets selbst implementiert werden, da ein proprietäres DBMS datenbankspezifische Zugriffsfunktionen zur Datenbank enthält. Die Klassen der Datenzugriffsschicht sorgen für das persistente Speichern und Laden der Daten bei Bedarf. Es ist aber möglich, in einer höheren Schicht der Datenzugriffsschicht zusätzlich eine selbst programmierte, datenbankunabhängige Schnittstelle anzubieten (siehe Bild 5-6). Dadurch kann das DBMS unterhalb der datenbankunabhängigen Datenzugriffsschicht ausgetauscht werden, ohne dass die Objekte der Verarbeitungsschicht betroffen sind.

Falls das DBMS nie ausgetauscht wird, kann auf eine Kapselung der proprietären Schnittstelle in einer solchen **Datenabstraktionsschicht** verzichtet werden. In diesem Fall existiert nur der tiefere **herstellerabhängige Anteil der Datenzugriffsschicht**. Das folgende Bild zeigt die mögliche Struktur der Datenbankzugriffsschicht:



*Bild 5-6 Struktur der Datenzugriffsschicht*

- Im **DBMS-Kern** befindet sich der Kern des DBMS. Er speichert die Daten der Entity-Objekte persistent auf Festplatte bzw. lädt sie.

### 5.1.5.3 Vergleich zwischen Einrechner- und Mehrrechner-Systemen

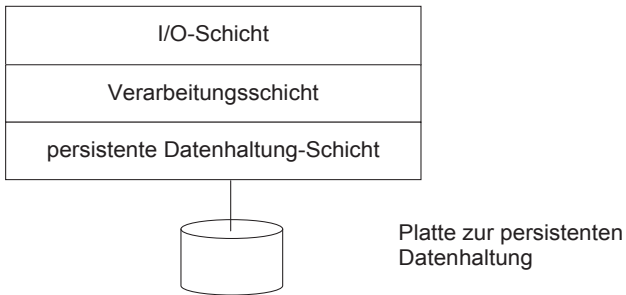
Für die Einteilung in Schichten ist es ferner wichtig, ob es sich um ein Einrechner-System (engl. **stand-alone system**) oder ein verteiltes System handelt, da bei einem verteilten System noch die Kommunikation zwischen den Systemen hinzukommt. Bei Client-Server-Systemen sind Zwei-Schichten-Architekturen (engl. **two-tier architectures**) und Drei-Schichten-Architekturen (engl. **three-tier architectures**) üblich. Die Zwei-Schichten-Architektur wird vorwiegend für eine Kombination aus Client- und Ser-

ver-Rechner eingesetzt. Bei einer Drei-Schichten-Architektur dagegen treten ein Client-Rechner, ein Anwendungsserver-Rechner und ein Datenbankserver-Rechner auf. Diese Modelle werden im Folgenden genauer vorgestellt:

- **Schichtenmodell eines Einrechner-Systems**

Im Rahmen eines Schichtenmodells werden die Funktionsklassen auf einem Rechner in Schichten angeordnet. Die oberste Schicht, die I/O-Schicht, ist die **Ein- und Ausgabe**, in anderen Worten die **Benutzerschnittstelle**. Die Benutzerschnittstelle kann Anwendungsfälle der **Verarbeitungsschicht** anstoßen. Die Verarbeitungsschicht enthält Kontroll- und Entity-Objekte, die zu Beginn der Systemanalyse zunächst alleine bei der Durchführung der Anwendungsfälle betrachtet werden. Die transienten Entity-Objekte der Verarbeitungsschicht werden durch die persistente Datenhaltungsschicht (**Datenzugriffsschicht + DBMS-Kern**) von der Festplatte zur Verfügung gestellt bzw. auf der Festplatte abgespeichert.

Damit ergibt sich in einem ersten Ansatz das folgende Schichtenmodell:



*Bild 5-7 Schichtenmodell eines Einrechner-Systems*

Eine Schicht kann bei Bedarf in weitere feinere Schichten zerlegt werden.

- **Schichtenmodell eines Client/Server-Systems**

Im Rahmen einer **Zwei-Schichten-Client/Server-Architektur** werden die folgenden Schichten auf zwei getrennte Rechner verteilt:

- I/O-Schicht,
- Verarbeitungsschicht,
- persistente Datenhaltungsschicht.

Durch die Verteilung auf zwei Rechner wird jeweils eine zusätzliche Schicht zur Datenübertragung (Kommunikation) auf jedem Rechner benötigt, damit die Rechner miteinander kommunizieren können:

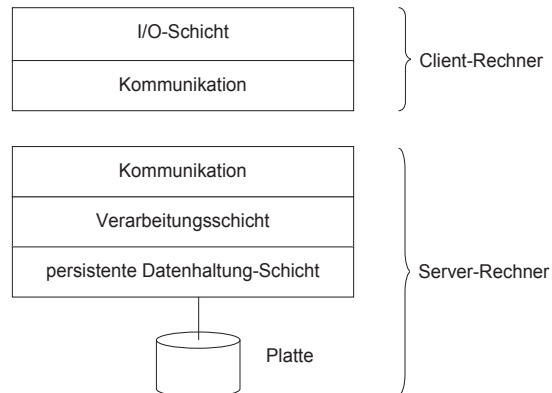


Bild 5-8 Beispiel für ein Schichtenmodell eines Client/Server-Rechners

Abhängig davon, ob sich die Verarbeitungsschicht auf dem Client-Rechner oder dem Server-Rechner befindet, wird zwischen einer **Thin-Client-Architektur** (siehe Bild 5-9) und einer **Fat-Client-Architektur** (Bild 5-10) unterschieden. Dies wird im Folgenden gezeigt:

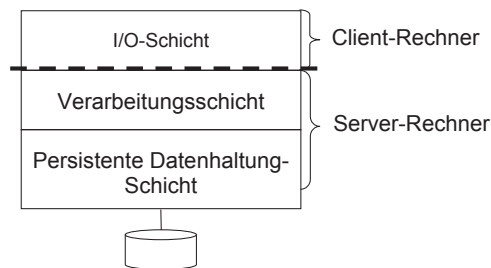


Bild 5-9 Thin Client<sup>76</sup>

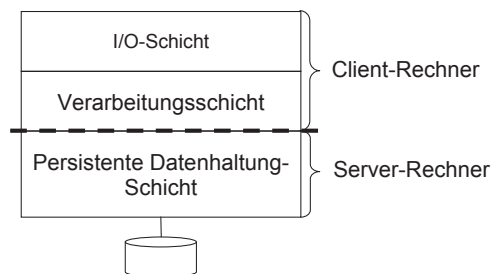


Bild 5-10 Fat Client<sup>77</sup>

Auf jedem Rechner wird zusätzlich eine **Kommunikationsschicht** erforderlich, damit die Rechner miteinander kommunizieren können. Diese Kommunikationsschicht

<sup>76</sup> Bei der Thin-Client-Architektur werden nur die Objekte der I/O-Schicht auf dem Client-Rechner ausgeführt.

<sup>77</sup> Bei der Fat-Client-Architektur befinden sich nur die persistente Datenhaltungsschicht auf dem Server-Rechner.

ist jeweils über und unter der gestrichelten fetten Linie einzuziehen.

- **Drei-Schichten-Architektur**

Im Rahmen einer **Drei-Schichten-Client/Server-Architektur** befinden sich die folgenden Schichten auf jeweils einem eigenen Rechner:

- I/O-Schicht,
- Verarbeitungsschicht und
- Datenhaltungsschicht.

An den Stellen, an denen ein System aufgetrennt und auf verschiedene Rechner gelegt wird, ist wieder eine beidseitige Kommunikationsschicht einzuziehen:

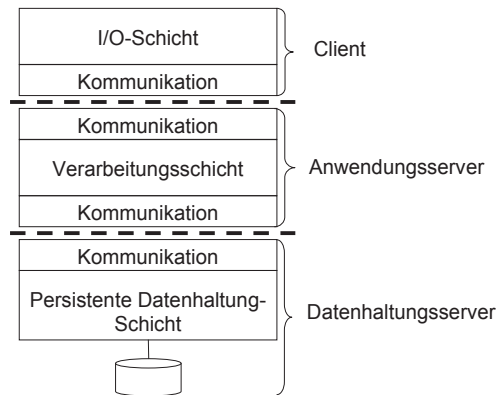


Bild 5-11 Modell der Drei-Schichten-Architektur mit Kommunikation

## Vorteile der Zwei- und Drei-Schichten-Architektur

Eine Zwei-Schichten-Architektur hat den Vorteil, dass bei Verwendung eines Thin-Client-Rechners bis auf die Benutzer-Schnittstelle alle Funktionen auf dem Server-Rechner liegen. Damit sind sie zentral für die verschiedenen Clients.

Der Vorteil der Drei-Schichten-Architektur ist, dass sie skalierbar ist. Bei Bedarf wird der Rechner des Anwendungsservers repliziert. Die Daten werden nur in einer einzigen Ausprägung an einer einzigen Stelle zentral gehalten, ggf. auf mehrere Rechner partitioniert.

### 5.1.5.4 Das ISO/OSI-Schichtenmodell

Ein typisches Beispiel für ein Schichtenmodell ist das ISO/OSI-Schichtenmodell (siehe [Ros90]). Es beschreibt das Kommunikationssystem eines Rechners, in anderen Worten das Netzwerkprotokoll. Ein Kommunikationssystem erlaubt Anwendungen, über ein Netzwerk miteinander zu kommunizieren. Das **ISO/OSI Basic Reference Model** strukturiert das Kommunikationssystem in sieben Schichten. Jede Schicht ist für die Lösung eines spezifischen Problems zuständig. Die Probleme sind jeweils mit Hilfe der darunterliegenden Schicht zu lösen:

- **Anwendungsschicht** (engl. **application layer**)  
Die Anwendungsschicht, die höchste Schicht des Referenzmodells, stellt die Schnittstelle der Anwendungsprozesse zum Kommunikationssystem dar. Es handelt sich hierbei um allgemeine Hilfsdienste für die Kommunikation oder aber um spezielle Kommunikationsdienste wie File Transfer, Message Handling System (MHS) etc.
- **Darstellungsschicht** (engl. **presentation layer**)  
Die Aufgabe der sechsten Schicht ist das **Aushandeln einer Transfersyntax**, die für beide Partner verständlich ist. Verstehen beide Rechner dieselbe lokale Syntax (z. B. bei Rechnern desselben Herstellers), so können sie sich auf eine Transfersyntax auf Basis der lokalen Syntax einigen. Ist dies nicht der Fall, so müssen sie sich auf die neutrale Transfersyntax ASN.1/BER<sup>78</sup> einigen. Die weiteren Aufgaben dieser Schicht sind die Durchführung der Wandlung der lokalen Syntax in die Transfersyntax bzw. aus der Transfersyntax in die lokale Syntax.
- **Kommunikationssteuerungsschicht** (engl. **session layer**)  
Die Kommunikationssteuerungsschicht regelt die **Synchronisation zwischen Prozessen in verschiedenen Systemen**.
- **Transportschicht** (engl. **transport layer**)  
Die Transportschicht befasst sich mit **Ende-zu-Ende-Verbindungen zwischen Prozessen** in den Endsystemen und zerlegt Nachrichten in Pakete bzw. fügt sie wieder zusammen.
- **Vermittlungsschicht** (engl. **network layer**)  
Die Vermittlungsschicht organisiert eine **Ende-zu-Ende-Verbindung zwischen kommunizierenden Endsystemen** eines Netzwerks. Sie ist damit für die **Routenwahl** vom Sender zum Empfänger zuständig.
- **Sicherungsschicht** (engl. **data link layer**)  
Die zweite Schicht stellt eine gesicherte Punkt-zu-Punkt-Verbindung zwischen zwei Systemen zur Verfügung. Sie führt die **Fehlerkorrektur in Bitsequenzen** durch.
- **Bitübertragungsschicht** (engl. **physical layer**)  
Die Bitübertragungsschicht stellt ungesicherte Verbindungen zwischen Systemen für die **Übertragung von Bits** zur Verfügung. Als Fehlerfall kann nur der Ausfall der Verbindung gemeldet werden.

Bild 5-12 zeigt die Kommunikationssysteme zweier wechselwirkender Rechner:

---

<sup>78</sup> Die neutrale Syntax ist die Abstract Syntax Notation 1 (ASN.1). Sie bildet Teil 1 der Transfersyntax und betrifft die Datentypen. Teil 2 sind die Basic Encoding Rules for ASN.1 (BER), die einen genormten Satz von Regeln zur Codierung der Datenwerte in die Transfersyntax beinhalten.

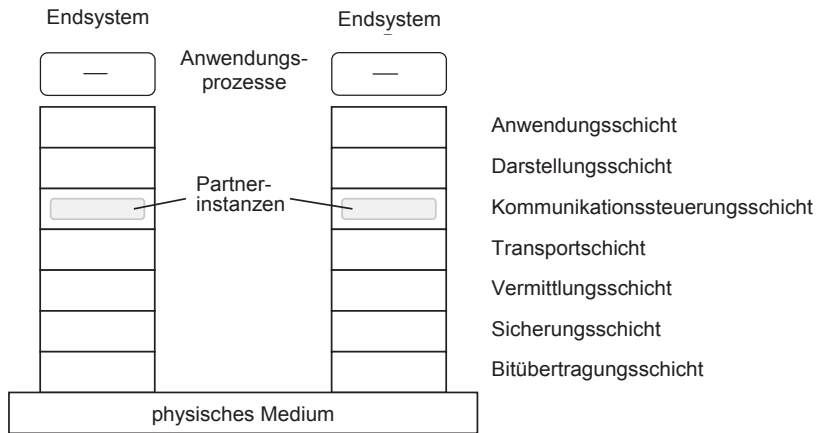


Bild 5-12 Prinzipieller Aufbau des OSI-Schichtenmodells

Auf hoher Ebene überwiegt im Protokollstapel die Anwenderfunktionalität, auf tiefer Ebene die Ansteuerung der Hardware. Die Aktivitäten auf hoher Ebene des Protokollstapels sollen durch Aktivitäten auf tieferer Ebene des Protokollstapels realisiert werden.

Wenn eine Schicht nach außen passend reagieren soll, so muss sie auch im Inneren des Rechners gewisse Arbeitseinheiten (Instanzen) haben, die für das Außenverhalten verantwortlich sind. OSI beschreibt ein logisches Modell von **Arbeitseinheiten (Instanzen) in den Schichten eines Rechners** und trifft dabei eine Aussage darüber, welche Funktionalitäten einer Arbeitseinheit nach außen bereitgestellt werden. Jeder Rechner, mit dem kommuniziert wird und der dem OSI-Modell genügt, soll die gleichen Instanzen haben. Dabei verhandeln die Partnerinstanzen einer Schicht, d. h. die entsprechenden Schichteninstanzen beim Sender und Empfänger, miteinander, um die Aufgabe der Schicht gemeinsam zu erfüllen.

Für das ISO/OSI-System ist das folgende Szenario von Bedeutung:

Zwei Endsysteme kommunizieren miteinander. Dabei unterhalten sich zwei Instanzen einer Schicht  $n$  über das Protokoll der  $n$ -ten Schicht. Eine Nachricht dieser Schicht wird in dem Protokollstapel des zugeordneten Rechners bis auf das Übertragungsmedium durchgereicht, dann übertragen und im Protokollstapel des Empfängers wieder bis auf die  $n$ -te Ebene hochgereicht. Dabei werden auf jeder durchlaufenen Schicht beim Sender zusätzliche Informationen angefügt, die dann beim Empfänger auf der entsprechenden Schicht wieder entfernt werden. Dies ist im folgenden Bild gezeigt:

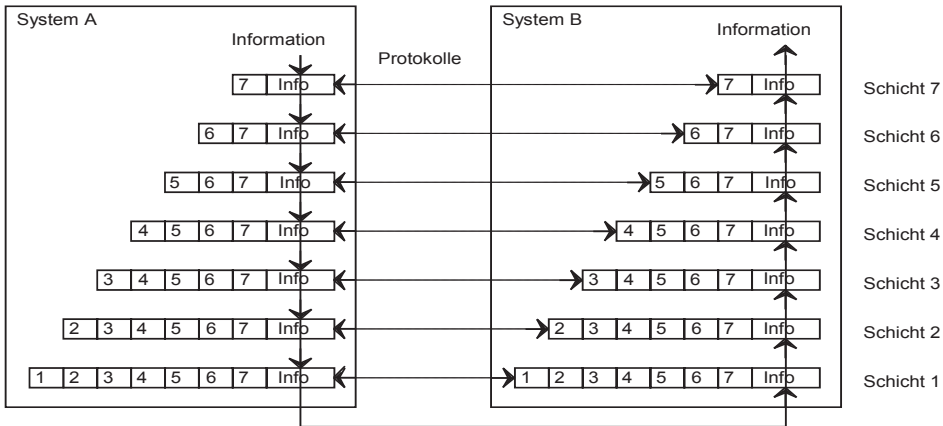


Bild 5-13 Protokollweg durch die OSI-Schichten

Es ist nicht sofort ersichtlich, warum das ISO/OSI-Modell gerade sieben Schichten hat. Die Architekten von TCP/IP erfanden vier Schichten, diejenigen von DECnet<sup>79</sup> erfanden acht Schichten. SNA<sup>80</sup> aus der IBM-Welt hatte auch acht Schichten, hatte aber wiederum die Schichten anders aufgeteilt als DECnet. Die Zahl der Schichten ist letztendlich nicht relevant. Entscheidend ist jedoch, dass die Nutzer einer Standardisierung (verschiedene Hersteller) die Schichten gemäß demselben Standard implementieren. Damit ist eine Interoperabilität zwischen Rechnern verschiedener Hersteller gewährleistet.

### 5.1.6 Ähnliche Muster

Sowohl beim Architekturmuster Layers als auch bei **Pipes and Filters** wird eine Anwendung funktional in Teilsysteme geschnitten. Das Architekturmuster Pipes and Filters ist datenstromorientiert und hat dabei eine Vorzugsrichtung für die Weiterleitung der Ausgabe, nämlich hin zum nächsten Filter. Ein Filter nach dem anderen wird dabei in der Grundform des Musters der Reihe nach von den Daten durchlaufen, wobei in jedem Filter die Daten verändert werden. Ein Schichtenmodell hingegen ist serviceorientiert und stellt in Aufrufschnittstellen der jeweils höheren Schicht die Ergebnisse der aufgerufenen Dienste zur Verfügung. Ein Dienst gibt eine Antwort also an den Aufrufer zurück, während bei Pipes and Filters ein Filter die Antwort an das nächste Element der Filterkette gibt.

Ähnlich sind **horizontale Schichtenmodelle mit vertikalen Strukturen** wie die Management Information Base (MIB) beim Netzwerkmanagement [Ros93, S. 76] oder die Client- und Server-Fabriken bei einem objektorientierten Client-Server-System (siehe [Gol12]).

<sup>79</sup> DECnet war das Kommunikationssystem der Firma Digital Equipment Corporation.

<sup>80</sup> SNA steht für Systems Network Architecture und wurde von der Fa. IBM in den 1970er Jahren entwickelt.

## 5.2 Das Architekturmuster Pipes and Filters

Das Architekturmuster Pipes and Filters modelliert die **Struktur eines Systems** in eine Kette von sequenziellen Verarbeitungsprozessen (Filtern), die über ihre Ausgabe bzw. Eingabe gekoppelt sind: Die Ausgabe eines Prozesses ist die Eingabe des nächsten Prozesses (datenflussorientierte Architektur eines Systems).

### 5.2.1 Name/Alternative Namen

Kanäle und Filter (engl. pipes and filters), es wird auch die Kurzform Filter-Muster (engl. filter pattern) benutzt.

### 5.2.2 Problem

Ein datenstromorientiertes System der Datenverarbeitung soll nicht als monolithischer Block gebaut werden. Es soll leicht erweiterbar sein und eine möglichst parallele Verarbeitung erlauben.

### 5.2.3 Lösung

Ein System, das Datenströme verarbeitet, kann mithilfe dieses Musters in Systemkomponenten strukturiert werden.

Die Aufgabe des gesamten Systems wird in einzelne Verarbeitungsstufen zerlegt. Jeder Verarbeitungsschritt wird in Form eines **Filters** implementiert. Jeweils zwei Filter werden durch eine **Pipe** verbunden. **Filter** lesen die Daten und bearbeiten sie sequenziell. Ein Filter wandelt eine Dateneingabe in eine Datenausgabe um.



Die Eingabedaten eines Filters werden als Datenstrom empfangen. Daraufhin führt der Filter eine Funktion auf den Eingabedaten durch und stellt danach die Ausgabedaten wieder als Datenstrom zur Verfügung.

Zur Bearbeitung eines Datenstroms hat ein **Filter** die folgenden Möglichkeiten:

- **Teile der Daten entnehmen.**
- **Teile den Daten hinzufügen.**  
Dabei werden z. B. weitere Informationen berechnet und angehängt.
- **Teile der Daten modifizieren.**  
Dabei werden z. B. vorhandene Informationen konzentriert oder in eine andere Darstellung überführt.

Die hier genannten Verarbeitungsarten können beliebig kombiniert werden.

Eine **Pipe** transferiert und puffert die Daten. Sie ist ein **passives Element**, das die Implementierung des Datenstroms darstellt.



Da eine Pipe Daten zwischenspeichern kann, kann ein Prozess Daten zu einer bestimmten Zeit in eine Pipe schreiben und ein anderer Prozess kann dann diese Daten zu einer anderen Zeit entnehmen. Eine Pipe kann also **asynchron entkoppeln**.



Die komplette Abfolge von Verarbeitungsstufen bezeichnet man als **Pipeline** oder **Filterkette**. Am Anfang einer Pipeline befindet sich eine Datenquelle, das Ende einer Pipeline bildet eine Datensenke. Die **Datenquelle** sendet in eine Pipe, die **Datensenke** empfängt Daten aus einer Pipe.

Datenquelle und Datensenke stellen stets die Endstücke einer Pipeline und die Schnittstelle zur Außenwelt dar.



Das folgende Bild zeigt den Aufbau einer Pipeline:

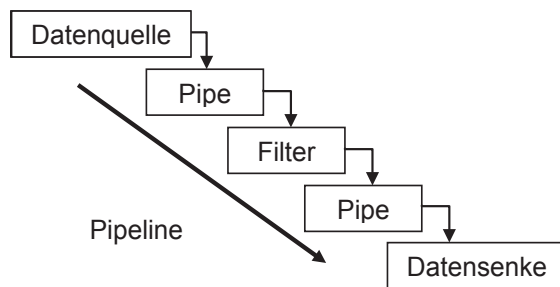


Bild 5-14 Aufbau einer Pipeline beim Muster Pipes and Filters

Eine Pipe ermöglicht den Zugriff auf eine Datenquelle bzw. Datensenke genauso wie die Kommunikation zwischen zwei Filtern.

Eine aktive **Datenquelle** liefert Daten an eine Pipe. Ist eine Datenquelle hingegen passiv, so wartet sie, bis der nächste Filter Daten von ihr anfordert. Analog dazu fordert eine aktive **Datensenke** Daten an. Ist sie passiv, so wartet sie auf Daten.



Die Aktivität eines Filters kann auf unterschiedliche Art und Weise ausgelöst werden:

- Ein **aktiver Filter** holt die Daten selbst aus der ihm aus Sicht des Datenstroms vorangehenden Pipe und schickt seine Ergebnisse in die ihm aus Sicht des Datenstroms nachfolgende Pipe.
- Von einem **passiven Filter** werden seine Daten über die ihm aus Sicht des Datenstroms nachfolgende Pipe von einem aktiven Filter geholt.
- Einem **passiven Filter** werden seine Daten über die ihm aus Sicht des Datenstroms vorangehende Pipe von einem aktiven Filter gesendet.

Ein **aktiver Filter** stellt einen eigenständigen parallelen Prozess<sup>81</sup> dar. Ein aktiver Filter ist der typische Fall. Er wird von einem übergeordneten Programm aufgerufen und läuft immer als parallele Einheit.



Aktive Filter sind vom Betriebssystem Unix bekannt. Sie holen ihre Eingabedaten aus der dem Filter aus Sicht des Datenflusses vorangehenden Pipe und senden ihre Ausgabedaten in die dem Filter aus Sicht des Datenflusses nachfolgende Pipe. Eine Pipe entkoppelt – wie bereits erwähnt – benachbarte aktive Filter asynchron.

Aktive Filter sind leichter zu kombinieren als passive Filter. Ein aktiver Filter liest und verarbeitet seine Eingabedaten kontinuierlich. Mit der Ausgabe von Daten als Datenstrom wird nach Möglichkeit sofort begonnen, ehe die Eingabe vollständig gelesen ist. Die Bearbeitung der eingelesenen Daten und die Ausgabe der entsprechenden erzeugten Daten erfolgt stückweise, wenn der eingehende Datenstrom stückweise verarbeitet werden kann. Damit kann zum einen die Latenzzeit verkürzt werden und zum anderen eine gewisse Parallelität erzeugt werden, wenn Filter schon Teilergebnisse liefern, ehe sie fertig sind.

Es gibt auch **passive Filter**, die von einem dem passiven Filter benachbarten Filterelement aufgerufen und damit aktiviert werden. Passive Filter, die erst durch einen Aufruf eines diesem Filter benachbarten Filterelements aktiviert werden, sind nicht typisch.



Ein passiver Filter wird nach dem Push-Prinzip von seinem Vorgänger zum Übergeben von Daten oder nach dem Pull-Prinzip von seinem Nachfolger zum Abholen von Daten aufgerufen.

Die bisherige Lösungsbeschreibung des Musters Pipes and Filters nutzte an keiner Stelle spezielle objektorientierte Möglichkeiten. Daraus kann man schließen, dass das Muster sehr gut auch nicht objektorientiert realisiert werden kann. Im Folgenden wird jedoch eine objektorientierte Lösung vorgestellt und zunächst das Muster Pipes and Filters mittels eines Kommunikationsdiagramms skizziert. Anschließend werden die Teilnehmer vorgestellt. Das anschließende Unterkapitel verdeutlicht das dynamische Verhalten.

### 5.2.3.1 Kommunikationsdiagramm

Anstelle eines sonst hier üblichen Klassendiagramms wird im Folgenden ein Kommunikationsdiagramm nach UML für Objekte vom Typ `Datenquelle`, `Pipe`, `Filter` und `Datensenke` gezeigt, wobei hier nur ein einziger Filter dargestellt ist. Es ist ein sogenanntes Kommunikationsdiagramm der Analyse (nach [Gol12]), in dem nur die Nachrichten und noch nicht die Methodenaufrufe eingetragen sind. Das Kommuni-

<sup>81</sup> Betriebssystem-Prozess oder Thread.

kationsdiagramm visualisiert die Vorzugsrichtung für den Datenfluss beim Muster Pipes and Filters.

Die Klassen `Datenquelle`, `Filter` und `Datensenke` sind in diesem Beispiel aktive Klassen. Eine aktive Klasse ist eine Klasse, deren Objekte einen oder mehrere Threads bzw. Betriebssystem-Prozesse beinhalten. Eine aktive Klasse hat seit UML 2.0 jeweils eine doppelte Linie an der linken und an der rechten Seite.

Im Folgenden das Kommunikationsdiagramm:

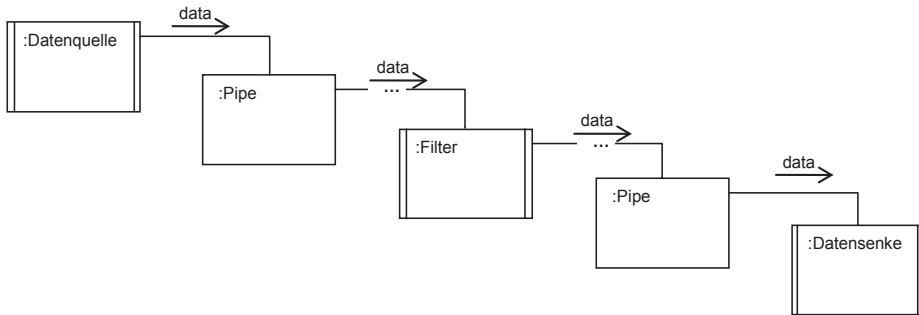


Bild 5-15 Kommunikationsdiagramm des Musters Pipes and Filters für aktive Filter

### 5.2.3.2 Teilnehmer

#### Datenquelle

Eine Datenquelle stellt die zu verarbeitenden Daten zur Verfügung. Sie kann sich gegenüber der Pipeline **aktiv** oder **passiv** verhalten. Aktiv bedeutet wie bei Filtern, dass die Datenquelle der Pipeline zyklisch oder ereignisbasiert Daten zur Verfügung stellt. Wenn eine Quelle sich passiv verhält, muss mindestens ein Element der Pipeline aktiv sein, damit nach dem Pull-Prinzip neue Daten zur Verarbeitung von der Quelle angefordert werden können.

#### Pipe

Pipes sind Kanäle und dienen als **passives** Verbindungsstück zwischen jeweils zwei Elementen vom Typ Quelle, Filter oder Senke. Es können dabei folgende Paarungen auftreten:

- Quelle – Pipe – Filter (am Anfang einer Pipeline),
- Filter – Pipe – Filter (in der Mitte der Pipeline) oder
- Filter – Pipe – Senke (am Ende der Pipeline).

Pipes funktionieren nach dem FIFO-Prinzip und dienen zur asynchronen Entkopplung in der Pipeline.

#### Filter

Filter verarbeiten ihre Eingabedaten. Es gibt zwei Gruppen von Filtern: **aktive** und **passive**. Aktive Filter sind typisch. Sie werden als parallele Einheit, z. B. als Thread

oder Betriebssystem-Prozess, gestartet und fordern zyklisch oder ereignisorientiert Eingabedaten an.

### Datensenke

Eine Datensenke ist der Empfänger von Daten. Senken können sich wie Filter und Quellen **aktiv** oder **passiv** verhalten. Aktiv verhält sich eine Senke, wenn sie zyklisch oder ereignisbasiert Daten anfordert (Pull-Prinzip).

#### 5.2.3.3 Dynamisches Verhalten

In diesem Kapitel werden verschiedene **Szenarien** des Musters Pipes and Filters aufgezeigt. Ein Filter  $x$  arbeitet mit seiner Funktion  $f_x()$  auf den Daten.  $x$  steht hierbei für die Nummer des Filters. In den folgenden Szenarien 1 bis 3 gibt es jeweils nur ein einziges aktives Element. Somit besteht keine Notwendigkeit, Elemente über einen Puffer asynchron zu entkoppeln. Auf eine Pipe kann also in diesen Szenarien verzichtet werden.

#### Szenario 1 – Pull-Prinzip

Das erste Szenario enthält eine **Pipeline**, die nach dem **Pull-Prinzip** arbeitet. Die **Senke** ist **aktiv**, **alle Filter und die Quelle** sind in diesem Szenario **passiv**. Die Senke startet die Abarbeitung und fragt dabei mit `read()` beim Filter vor ihr an. Da dieser keine Daten hat, fragt dieser seinen Vorgänger, dieser wieder bei seinem Vorgänger etc. Die Anfrage wird also von Vorgänger zu Vorgänger weitergereicht, bis die Anfrage ein Element der Pipeline erreicht, das Daten enthält, nämlich die Quelle. Dies ist in folgendem Bild dargestellt:

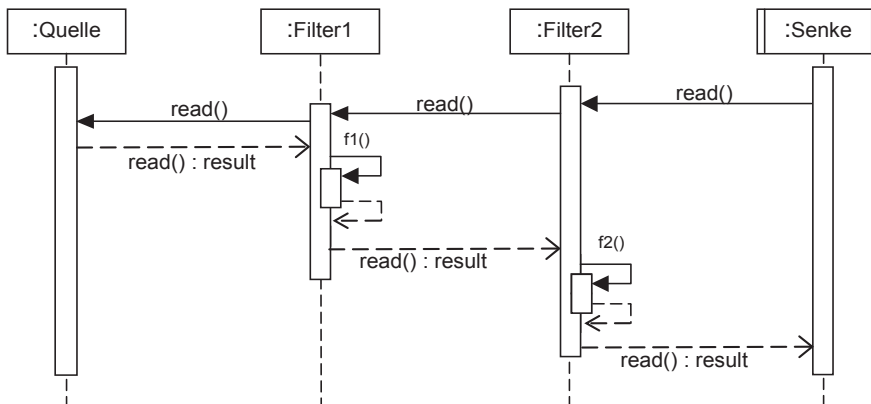


Bild 5-16 Sequenzdiagramm für das Pull-Prinzip

#### Szenario 2 – Push-Prinzip

Das Szenario 2 betrachtet eine **aktive Quelle**, zwei **passive Filter** und eine **passive Senke**. Die **aktive Quelle** sendet ereignisbasiert oder zyklisch Daten. Die Filter und die Senke sind passiv, d. h., sie fordern keine Daten an (siehe Bild 5-17). Das gezeigte Verhalten der Pipeline wird **Push-Prinzip** genannt.

Die Quelle startet die Abarbeitung und schreibt mit `write()` in den ersten passiven Filter. Nach der Verarbeitung schreibt der erste passive Filter in den zweiten und dieser in die Senke. Ein Vorteil dieses Szenarios besteht darin, dass ein Filter nur dann aktiv ist, wenn wirklich Daten zur Bearbeitung anstehen. Das folgende Bild zeigt dieses Szenario:

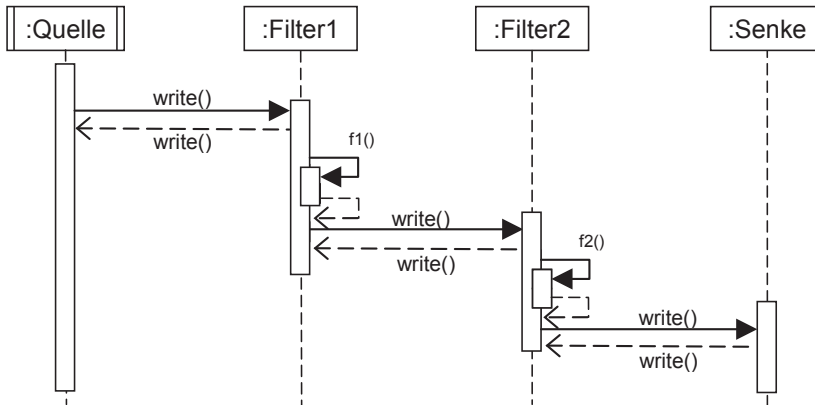


Bild 5-17 Sequenzdiagramm für das Push-Prinzip

### Szenario 3 – Pull- und Push-Prinzip gemischt

Szenario 3 behandelt exemplarisch eine Pipeline, die einen **aktiven Filter** (Filter 2) enthält. Die **Quelle**, die **Senke** und der **Filter 1** verhalten sich **passiv**. Dabei wird das **Push-** und das **Pull-Prinzip gemischt** angewandt, siehe Bild 5-18.

Der aktive Filter 2 fordert zyklisch Daten bei seinem Vorgänger an (**Pull-Prinzip**). Der passive Vorgänger wird von Filter 2 synchron mit `read()` aufgerufen. Die Anfrage wird mit `read()` weitergeleitet, bis sie ein Element der Pipeline erreicht, das Daten enthält, hier die Quelle. Diese Weiterleitung entspricht dem Pull-Prinzip. Die Quelle antwortet auf das `read()` und stößt mit ihrem Rückgabewert Filter 1 an. Dieser Filter arbeitet genau einmal und antwortet nach seiner Verarbeitung mit seinem Rückgabewert. Der Aufruf von `write()` durch Filter 2 entspricht dem **Push-Prinzip**. Hier das Sequenzdiagramm für dieses Szenario:

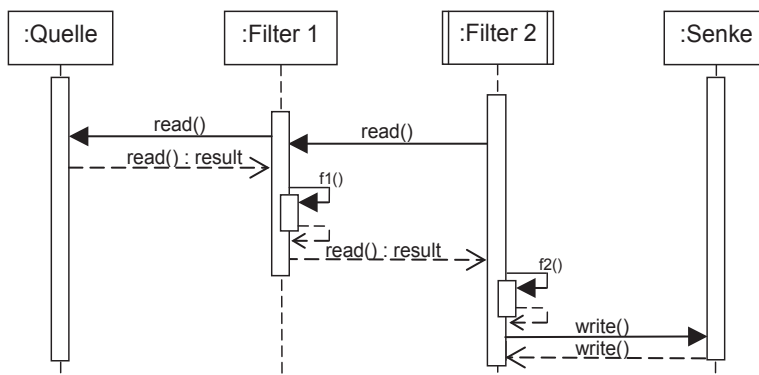


Bild 5-18 Sequenzdiagramm Pull- und Push-Prinzip gemischt

### Szenario 4 – Zwei aktive Filterelemente durch eine Pipe asynchron entkoppelt

Typischerweise gibt es bei Pipelines Szenarien, die zwei oder mehr aktive Filterelemente enthalten. In Szenario 4 wird eine Pipeline mit drei Filtern, einer Pipe, einer Quelle und einer Senke beschrieben. Dabei ist ein Filter passiv und 2 Filter sind aktiv. Eine Pipe dient als Entkopplungselement zwischen den beiden aktiven Filtern. Quelle, Filter 1 und die Senke sollen hier passive Elemente sein, die aufgerufen werden. Die **aktiven Filter 2 und 3** werden durch eine Pipe – wie bereits erwähnt – asynchron entkoppelt. Die anderen Pipes können weggelassen werden, weil hier nicht zwei Prozesse asynchron entkoppelt werden müssen.

Das Szenario 4 ist in Bild 5-19 zu sehen. Als erstes erfolgt ein Zugriff des aktiven Filters 3 auf die benachbarte Pipe mit `read()`. Da diese noch keine Daten enthält, muss Filter 3 so lange warten, bis neue Daten in der Pipe vorhanden sind. Asynchron zum genannten Vorgang fordert Filter 2 nach dem Pull-Prinzip mit `read()` Daten bei Filter 1 an. Dieser Filter hat keine Daten und leitet die Anfrage an die Quelle weiter. Daraufhin werden die Ausgabedaten der Quelle nacheinander von Filter 1 und Filter 2 bearbeitet und schließlich in der Pipe gespeichert. Filter 3 wartet bereits auf neue Daten, wie zu Beginn beschrieben. Die Pipe enthält nun Daten und damit ist das Lesen von Filter 3 abgeschlossen. Nachdem Filter 3 die so erhaltenen Daten abgearbeitet hat, werden sie an die Senke weitergegeben. Hier das entsprechende Sequenzdiagramm:

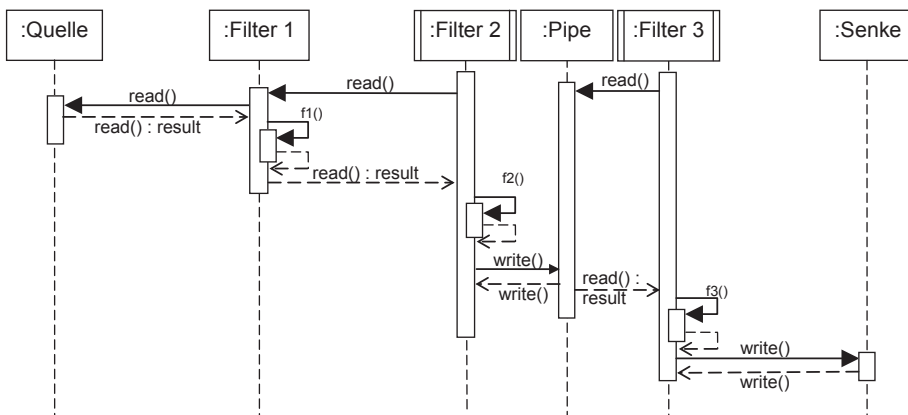


Bild 5-19 Asynchrone Entkopplung von parallelen Filtern

## 5.2.4 Bewertung

### 5.2.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Das Hauptmerkmal des Musters Pipes and Filters ist die Flexibilität gegenüber Änderungen oder Erweiterungen. Filter können oft auf einfache Weise innerhalb einer Pipeline ausgetauscht oder vertauscht werden. Wie einfach der Austausch eines

Filters jedoch ist, hängt jedoch von der Spezifikation des Datenformats der Datenkanäle ab. Quelle und Senke können bei entsprechendem Datenformat ebenfalls leicht ausgetauscht werden. Filter können auch leicht zusammengefasst werden.

- Im Hinblick auf Rapid Prototyping stellt sich die Verwendung dieses Musters als großer Vorteil heraus. Filter können von verschiedenen Personen bei Vorliegen der Schnittstellenspezifikation schnell entwickelt und integriert werden. Nach Fertigstellung der Filter kann ihre Funktion unabhängig voneinander überprüft werden.
- Nicht benachbarte Verarbeitungsstufen teilen keine Informationen und sind daher entkoppelt.
- Das Speichern von Zwischenergebnissen ist nicht notwendig, aber möglich. Das Zwischenspeichern von Daten in einer Pipeline ist jedoch fehlerträchtig. Ein solches Speichern von Zwischenergebnissen könnte beispielsweise genutzt werden, um Rohdaten für andere Systeme oder Zwecke zur Verfügung stellen.
- Die Aufgaben der Filter können bis zu einem gewissen Grad parallel abgearbeitet werden, wenn aktive Filter überlappend arbeiten. Dies setzt allerdings voraus, dass dabei ein Datenstrom vorliegt, der von den verschiedenen Filtern bearbeitet werden kann.
- Filter können sehr leicht in anderen Filterketten wiederverwendet werden.

#### 5.2.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

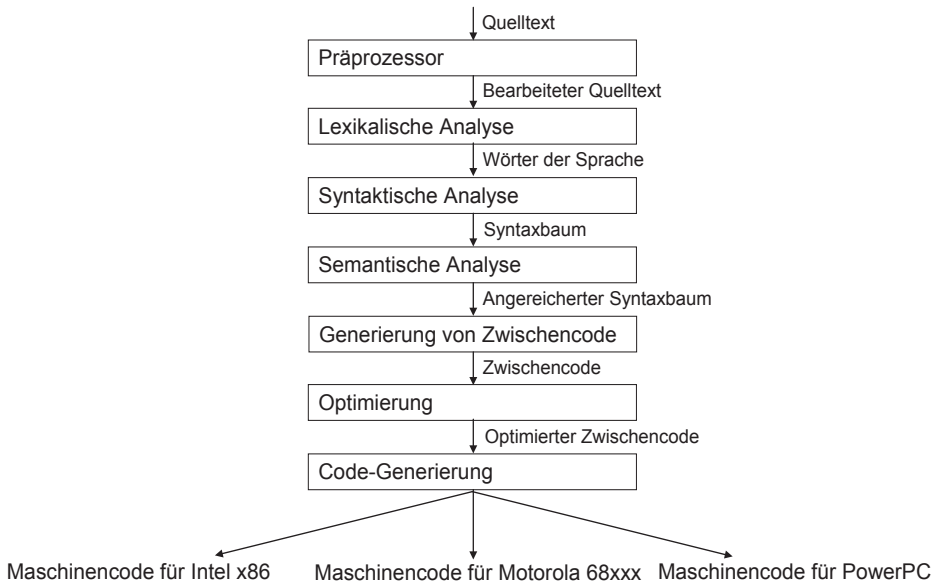
- Die Realisierung der Fehlerbehandlung ist schwierig, da im System kein gemeinsamer Zustand existiert.
- Der langsamste Filter in der Kette bestimmt die Verarbeitungsgeschwindigkeit. Es gibt keine vollständige Parallelisierung, da die Filter aufeinander warten müssen.
- Ist das Format der Datenkanäle nicht sorgfältig spezifiziert, können große Aufwände bei der Datenkonvertierung in einzelnen Filtern entstehen. Je nach Schnittstellenspezifikation der Filter können viele Konvertierungen notwendig sein.

#### 5.2.5 Einsatzgebiete

Die Einsatzgebiete von Pipes and Filters reichen von der Bildverarbeitung bis zu komplexen Simulationssystemen. Ein berühmtes Beispiel des Musters ist der Kommandozeileninterpreter von Unix-Systemen. Hier können Ausgaben von Programmen mittels Pipes an weitere Programme übergeben werden.

Zwei benachbarte aktive Filter, die über eine Pipe kommunizieren, kommen beim **Producer-Consumer-Muster** (siehe beispielsweise [Tan09]) vor. Ein Consumer holt sich nur dann Daten aus der Pipe, wenn er diese braucht. Ein Producer erzeugt Daten asynchron zu einem Consumer und übergibt sie der Pipe.

Ein Beispiel für eine Kette von Filtern ist ein Compiler mit seinen verschiedenen Phasen:



*Bild 5-20 Sequenzielle Phasen eines Compilers*

## 5.2.6 Ähnliche Muster

Sowohl beim Architekturmuster **Layers** als auch bei Pipes and Filters wird eine Anwendung funktionell in Komponenten oder Verarbeitungsschritte geschnitten. Das Architekturmuster Pipes and Filters ist datenstromorientiert, hat dabei eine Vorzugsrichtung und arbeitet einzelne Verarbeitungsschritte sequenziell ab. Ein Schichtenmodell hingegen ist serviceorientiert und stellt in Aufrufchnittstellen der jeweils höheren Schicht Dienste zur Verfügung. Ein Dienst gibt eine Antwort zurück, während bei Pipes and Filters ein Filter die Antwort an das nächste Element der Filterkette gibt.

Das Entwurfsmuster **Dekorierer** und das Architekturmuster Pipes and Filters können dazu benutzt werden, eine zusätzliche Funktionalität bereitzustellen. Beim Architekturmuster Pipes and Filters kann die Zusatzfunktionalität beispielsweise dadurch erzeugt werden, dass ein neuer Filter in die Pipeline eingeschoben wird oder ein Filter gegen einen anderen ausgetauscht wird. Filterketten sind sehr flexible und wiederverwendbare Mechanismen. Ein Dekorierer ist jedoch an die dekorierte Klasse gebunden und somit nicht ohne weiteres wiederverwendbar.

Das Muster Pipes and Filters basiert auf einer streng sequenziellen Reihe von Filtern, die durch Pipes miteinander verbunden sind. Filter können hierbei nur einen Eingabekanal und einen Ausgabekanal haben. Eine Variante des Musters Pipes and Filters wird in [Bus98] unter dem Namen **Tee-And-Join-Pipeline** beschrieben. Hier können Filter mehrere Eingabe- und Ausgabekanäle besitzen und somit Strukturen realisieren, bei denen Abzweigungen, parallel verlaufende Stränge von Filtern und das Zusammenführen von solchen Strängen möglich sind.



## 5.3 Das Architekturmuster Plug-in

Das Architekturmuster Plug-in stellt eine Architektur für ein **adaptives System** dar, bei der ein Drittanbieter eine lauffähige Software erweitern kann, ohne die Quellen der vorhandenen Software zu kennen.

### 5.3.1 Name/Alternative Namen

Plug-in (auch Plugin) kommt aus dem Englischen von "to plug in" und bedeutet "etwas einstecken". In der Literatur existieren zusätzlich die Begriffe Add-on (auch Addon) oder Add-in (auch Addin). Die Abgrenzung dieser Begriffe zu Plug-in ist nicht klar definiert. Im Folgenden wird der Begriff **Plug-in** benutzt.

### 5.3.2 Problem

Anwendungssoftware soll stabil sein. Sie soll möglichst lange eingesetzt und erweitert werden können. Dabei soll vorhandene lauffähige Software nicht mehr abgeändert werden, sie soll aber flexibel – auch durch dritte Parteien – erweitert werden können. Diese Forderungen sind vom Open-Closed-Prinzip her bekannt (siehe Kapitel 1.8).

Eine Applikation soll auch ohne die Anwesenheit der erweiternden Plug-ins lauffähig sein, es kann dann jedoch nicht die erweiterte Funktionalität abgedeckt werden. Die Erweiterungen sind oft für spezielle Benutzergruppen gedacht.



### 5.3.3 Lösung

Ein System, das im laufenden Betrieb flexibel erweiterbar sein soll, kann nach dem Architekturmuster Plug-in entworfen und realisiert werden.

Das Architekturmuster Plug-in strukturiert eine spezielle Anwendung so, dass diese über Erweiterungspunkte in Form von Schnittstellen verfügt, an denen dann die Plug-ins, die diese Schnittstellen implementieren, eingehängt werden können. Eine Anwendung ist aber bereits ohne diese zusätzlichen Erweiterungen lauffähig. Die Architektur der Anwendungssoftware muss **Schnittstellen** definieren, an deren Stelle **zur Laufzeit Komponenten, die diese Schnittstelle implementieren**, treten können.



Plug-ins sind speziell für die entsprechende Anwendung wie etwa Eclipse und nicht für andere Anwendungen vorgesehen<sup>82</sup>.

<sup>82</sup> Im Gegensatz zu Plug-ins sind EJBs so konzipiert, dass sie in jedem EJB-Container laufen können. EJBs werden beispielsweise in [Hei10] erläutert.

Ein Plug-in stellt einem System neue, zusätzliche Funktionalitäten zur Verfügung. Plug-ins erweitern dabei das bestehende System und sind in der Regel nicht eigenständig ausführbar. Dritte können unabhängig von den internen Abläufen der Anwendungssoftware erweiternde Funktionalitäten in Form von Plug-ins entwickeln, wenn die zu implementierenden Schnittstellen bzw. die Methodensignaturen einer Anwendungssoftware offengelegt werden.

Plug-ins stellen einem System eine erweiternde Funktionalität zur Verfügung. Dabei implementieren sie eine Schnittstelle und sind in der Regel nicht eigenständig ausführbar.



### Funktionsweise von Plug-ins

Ein Plug-in ist also eine Softwarekomponente, die eine vorhandene Applikation um eine neue, zusätzliche Funktionalität erweitert. Dabei implementiert ein Plug-in eine für solche **Erweiterungen vorgesehene Schnittstelle**. Fehlt eine solche Schnittstelle im System, kann dieses nicht durch Plug-ins erweitert werden. Plug-ins müssen also bereits in der Architektur einer Applikation berücksichtigt werden.

Die **Schnittstellendefinition** wird als Erweiterungspunkt (engl. **extension point**) bezeichnet. Ein Plug-in bietet einer Applikation an einem Erweiterungspunkt eine passende Implementierung der angebotenen Schnittstelle an.

Eine Applikation definiert **Erweiterungspunkte** in Form von Schnittstellen. Ein Plug-in, das eine dieser Schnittstellen implementiert, stellt eine **Erweiterung** für diesen Erweiterungspunkt dar.



Zum Einfügen von Plug-ins zur Laufzeit dient eine weitere Partei, der sogenannte **Plug-in-Manager**.

Ein Plug-in kann selbst auch weitere Schnittstellen zur Erweiterung definieren und kann damit durch eine weitere Ebene von Plug-ins erweitert werden. Auf diese Weise kann aus einer Applikation ein beliebig komplexes System nach dem Prinzip eines Baukastens aus einzelnen Plug-ins entstehen. Bild 5-21 zeigt den beispielhaften Aufbau einer Plug-in-Architektur:

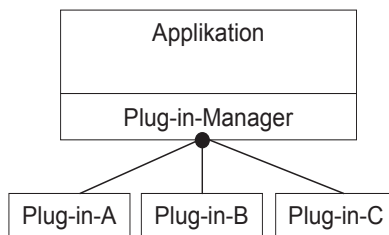


Bild 5-21 Konzept einer Plug-in-Architektur

Eine Applikation wird durch Plug-ins erweitert. Plug-ins können wiederum durch weitere Plug-ins erweitert werden (Kaskade von Plug-ins).



Die hohe Flexibilität beim Einsatz von Plug-ins führt zu einem höheren Verwaltungsaufwand sowohl bei der Implementierung als auch bei der Ausführung. Beispielsweise müssen die vorhandenen Plug-ins in Anzahl und Art verwaltet werden und der Plug-in-Manager muss zur Laufzeit entscheiden können, welches der verfügbaren Plug-ins zur Abarbeitung der aktuellen Aufgabe passt.

Dieser Verwaltungsmechanismus unterliegt in der Regel einem **Komponentenmodell**<sup>83</sup> und muss nicht implementiert werden, wenn ein geeignetes Framework<sup>84</sup> hierfür verwendet werden kann. Die Informationen über die Art und Anzahl der verfügbaren Plug-ins wird durch die Laufzeitumgebung dieses Frameworks zur Verfügung gestellt. Wird das Plug-in-Muster angewendet, können die Abhängigkeiten zwischen einer speziellen Applikation und den Plug-ins auf eine gemeinsame Schnittstelle reduziert werden. Die Bereitstellung der Plug-ins, d. h. die Instanziierung von Objekten der verfügbaren Plug-in-Klassen, übernimmt hier eine spezielle Komponente der Laufzeitumgebung, die über den Plug-in-Manager angesprochen werden kann (**Dependency Injection**, siehe Kapitel 1.10.2).

Der wesentliche Punkt bei einer Plug-in-Architektur ist die vollständige Entkopplung einer speziellen Applikation vom Prozess der Instanziierung von Objekten der Plug-in-Klassen. Plug-ins werden über den Plug-in-Manager instanziiert (unter Verwendung einer Komponente der Laufzeitumgebung).



Plug-ins bringen nicht zwangsläufig größere Einbußen in der Performance einer Anwendung mit sich. In der Regel werden zur Laufzeit nur diejenigen Plug-ins geladen, die auch wirklich zur Ausführung der aktuell gewünschten Aufgabe benötigt werden. Bei komplexen Systemen kann durch die Ladezeiten der Plug-ins jedoch die Performance des Programms negativ beeinflusst werden.

<sup>83</sup> Ein Komponentenmodell beschreibt in der Softwaretechnik einen Rahmen zur Ausführung von Systemen, welche sich aus einzelnen Komponenten zusammensetzen. Dabei definiert das Komponentenmodell neben einem Container, in dem die einzelnen Komponenten ausgeführt werden (Laufzeitumgebung), auch die Schnittstellen der Komponenten zum Container und zu anderen Komponenten. Zudem werden zahlreiche Mechanismen wie etwa zur Persistierung, zur Sicherheit oder zur Versionsverwaltung für Komponenten definiert.

<sup>84</sup> Ein Framework offeriert dem nutzenden System Klassen, von welchen es ableiten und somit Funktionslogik erben kann. Ein Framework bestimmt die Architektur der Anwendung, also die Struktur und Dynamik im Groben. Es definiert weiter die Unterteilung in Klassen und Objekte, die jeweiligen zentralen Zuständigkeiten, die Zusammenarbeit der Klassen und Objekte sowie deren Kontrollfluss [Gam95].

### 5.3.3.1 Klassendiagramm

Eine Applikation hat keinen direkten Bezug zu ihren Plug-ins, sie kennt lediglich den Plug-in-Manager. Mittels des Plug-in-Managers bekommt die Applikation Zugriff auf die Instanzen der verfügbaren Plug-ins, ohne diese selbst zu instanziiieren. Der Plug-in-Manager nutzt dabei von der Anwendungssoftware angebotenen Schnittstellen. Hier das Klassendiagramm einer Plug-in-Architektur:

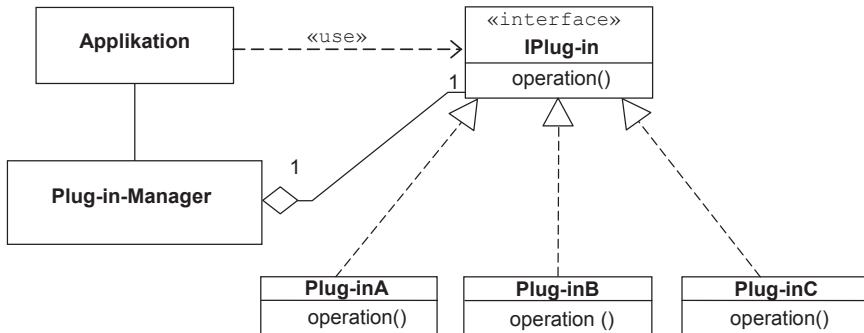


Bild 5-22 Klassendiagramm einer Plug-in-Architektur mit einem einzigen Interface

Der Plug-in-Manager ermittelt mit Hilfe der Laufzeitumgebung verfügbare Plug-ins für eine Schnittstelle und instanziiert die passenden Objekte. In einer ersten Variante ruft die Applikation die erzeugten Objekte vom Plug-in-Manager ab, aggregiert diese und benutzt die Objekte im weiteren Programmablauf. Alternativ können die instanziierten Objekte auch dauerhaft vom Plug-in-Manager aggregiert werden. Die Applikation leitet dann die Aufrufe an den Plug-in Manager weiter, der diese dann auf den entsprechenden Objekten ausführt.

Plug-ins können – wie bereits erwähnt – ihrerseits durch weitere Plug-ins ergänzt werden. In diesem Fall erhält ein Plug-in die Rolle des Clients und greift dann wieder über den Plug-in-Manager auf die Instanzen der gewünschten Plug-ins zu.

### 5.3.3.2 Teilnehmer

#### Applikation

Eine Applikation repräsentiert diejenige Komponente, die zur Erweiterung ihrer Funktionalität andere Komponenten nutzt. Sie stellt die Anwendung dar, die gestartet wird.

#### Plug-in-Manager

Ein Plug-in-Manager kennt alle verfügbaren Plug-ins anhand ihrer Schnittstellen, die sie implementieren. Die Applikation fragt beim Plug-in-Manager nach Plug-ins eines bestimmten Typs an.

#### IPlug-in

Das Interface **IPlug-in** spezifiziert in der Anwendung eine Schnittstelle als Stellvertreter für alle konkreten Implementierungen einer Reihe von Plug-ins. Hat die Applikation vom Plug-in-Manager das passende Plug-in erhalten, kann sie das Plug-in aufrufen.

### Plug-inX (X = A..Z)

Ein Plug-in erweitert die Applikation um Funktionalität. Dabei wird ein Plug-in vom Plug-in-Manager instanziiert und genutzt.

#### 5.3.3.3 Dynamisches Verhalten

Für die Implementierung von Plug-in-Architekturen gibt es bereits eine Vielzahl von Frameworks, die die Mechanismen von Plug-in-Architekturen implementieren. Hierzu zählen beispielsweise das Laden und Entladen der Plug-ins, sowie das Finden der passenden Plug-ins zur Laufzeit unter Wahrung einer Versionsnummer. Das Spring Framework oder die Eclipse Rich Client Plattform bieten u.a. auch Hilfsmittel für eine Plug-in-Architektur in Java.

In der Regel werden die Plug-ins dabei von einem Plug-in-Manager verwaltet. Die Applikation definiert Schnittstellen zur Erweiterung beispielsweise in einer Datei. Unter Verwendung einer Komponente der Laufzeitumgebung instanziiert und verwaltet der Plug-in-Manager die (anhand der Datei gefundenen<sup>85</sup>) Plug-ins. Das folgende Bild zeigt das Sequenzdiagramm eines typischen Systems mit einer Plug-in-Architektur<sup>86</sup>.

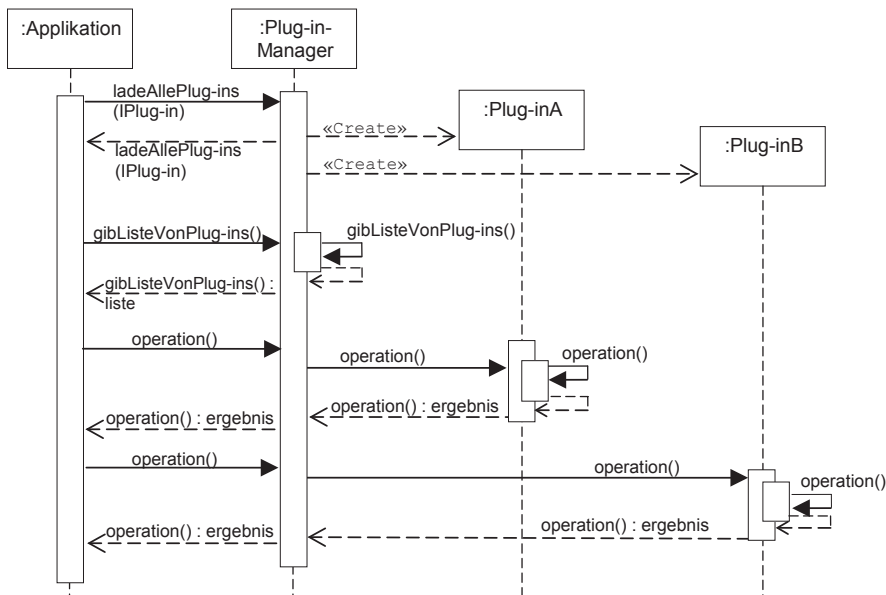


Bild 5-23 Sequenzdiagramm eines typischen Systems nach der Plug-in-Architektur

Die Applikation kann nun zur Laufzeit Plug-ins zu einer gewünschten Schnittstelle erhalten, indem sie beim Plug-in-Manager nach einem geeigneten Objekt nachfragt.

<sup>85</sup> Der Aufbau und Inhalt einer solchen Datei ist im Kapitel 5.3.7 gezeigt und beschrieben. Im dort gezeigten Beispiel ist der Name der Datei `Service.IWoerterbuch`.

<sup>86</sup> Die Datei der Plug-ins ist nicht dargestellt.

**Basisablauf:** Der Plug-in-Manager hat ein Plug-in gefunden, welches das von der Applikation gewünschte Interface implementiert. Der Plug-in-Manager erzeugt anschließend eine Instanz des Plug-ins. Die Applikation kann nun die in der Schnittstelle definierten Funktionen eines Plug-ins über den Plug-in-Manager aufrufen. Dieser leitet die Aufrufe an das gewünschte Plug-in weiter.

**Alternativablauf:** Der Plug-in-Manager hat kein Plug-in gefunden, das die gewünschte Schnittstelle implementiert. In der Regel wird dann eine Fehlerbehandlung ausgelöst und der gewünschte Anwendungsfall (z. B. die Wiedergabe einer Musikdatei) kann nicht durchgeführt werden. Die Applikation kann dann das Plug-in nicht ausführen, wird aber weiter ausgeführt.

### 5.3.4 Bewertung

#### 5.3.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Jedes Plug-in besitzt seine eigene Zuständigkeit (**separation of concerns**, siehe Kapitel 1.3).
- Plug-in-Architekturen zeigen in der Regel ein robustes Verhalten.
- Der Mechanismus von Plug-ins bietet im Gegensatz zu Bibliotheken und Frameworks den Vorteil, dass die Entwickler einer dritten Partei unabhängig Zusatzfunktionalität für eine vorhandene Software entwickeln können, ohne Kenntnis über den Programmcode der zu erweiternden Software haben zu müssen.
- Des Weiteren kann durch den modularen Aufbau eines Systems auf Basis von Plug-ins das System "schlank" gehalten werden, indem nur die wirklich genutzten Funktionen geladen werden.
- Der Wartungsaufwand sinkt, da die in Plug-ins ausgelagerte Funktionslogik für einen Anwendungsfall angepasst und als neue Version eines Plug-ins ausgeliefert werden kann.
- Die Entwicklung komplexer Software kann bequem aufgeteilt werden.
- Plug-ins können in mehreren verschiedenen Versionen vorhanden sein und auch gleichzeitig ausgeführt werden.
- Jedes Plug-in kann getrennt von den anderen Plug-ins getestet werden.

#### 5.3.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Der initiale Aufwand bei der Implementierung einer Anwendung ist höher, da die Architektur der Anwendung Schnittstellen bereitstellen muss, welche von den Plug-ins implementiert werden.
- Der Verwaltungsaufwand während der Ausführung einer speziellen Anwendung steigt.
- Für Erweiterungen muss eine gemeinsame Schnittstelle gefunden werden. Plug-ins haben zu dieser Schnittstelle eine Abhängigkeit.

### 5.3.5 Einsatzgebiete

Die Einsatzgebiete von Plug-ins liegen in erster Linie dort, wo ein **großer Kreis von Anwendern** ein System nutzt und jede Gruppe von Anwendern dabei eigene Anforderungen hat, die sich oft nur in Details von den Abläufen anderer Anwenderkreise unterscheiden. Für jeden dieser Anwenderkreise ein eigenes System zu entwickeln und durch Wartung am Leben zu erhalten, ist nicht kosteneffizient und wenig zweckmäßig.

Die Lösung liegt in einem einzigen System, das die Anforderungen aller Anwenderkreise abdeckt, wobei jedoch jedem Anwender nur die von ihm genutzten Funktionen zur Verfügung gestellt werden.

Ein zusätzliches Einsatzgebiet stellt die Möglichkeit der eigenen Erweiterung der Applikation dar, falls eine Anforderung nicht wie gewünscht abgedeckt wird. Durch die gezielte Definition von Schnittstellen zur Erweiterung einer Applikation kann diese – eventuell mit Einschränkungen – durch den Anwender selbst um Funktionalität erweitert werden.

Ein bekanntes Anwendungsbeispiel findet sich im Bereich von Webbrowsern. Hier kann eine Plug-in-Architektur bei der Wiedergabe von medialen Inhalten genutzt werden. Für jedes zu öffnende Medium (z. B. Video oder Bilder diverser Formate) lädt der Webbrowser das entsprechende Plug-in, das den medialen Inhalt am Bildschirm anzeigt.

### 5.3.6 Ähnliche Muster

Entwurfsmuster, die zur Erweiterung des Verhaltens bzw. der Funktionalität genutzt werden können, sind beispielsweise auch die Strukturmuster **Dekorierer** und **Strategie**. Im Gegensatz zu einem Plug-in gibt es bei diesen Mustern jedoch **keine** fest definierte **öffentliche** Schnittstelle, die von Dritten zur Erweiterung der Funktionalität zur Laufzeit genutzt werden kann. Ebenfalls existiert dort kein Mechanismus, um z. B. alle vorhandenen Erweiterungsmöglichkeiten aufzulisten. Beim Architekturmuster Plug-in übernimmt dies der Plug-in-Manager.

In [Bus98] findet sich das Architekturmuster **Microkernel**. Es hat eine ganz ähnliche Zielsetzung wie das Architekturmuster Plug-in: Ein Microkernel enthält Funktionen für die Kommunikation von Komponenten untereinander und bietet Schnittstellen an, mit denen Komponenten die Funktionen des Microkernel nutzen können. Im Unterschied zum Muster Plug-in ist ein Microkernel auch für die Verwaltung systemweiter Ressourcen wie beispielsweise Prozesse und Dateien verantwortlich und steuert und koordiniert den Zugriff auf diese Ressourcen. Daher wird das Muster Microkernel hauptsächlich im Zusammenhang mit dem Entwurf von Betriebssystemen erwähnt.

### 5.3.7 Programmbeispiel

An einem Anwendungsbeispiel Wörterbuch soll eine **Plug-in-Architektur** in der Programmiersprache Java veranschaulicht werden.

Eine Anwendung liest Wörter ein und übersetzt sie aufgrund der Informationen, die in Wörterbüchern enthalten ist. Wörterbücher sollen als Plug-ins geladen werden und die Schnittstelle `IWoerterbuch` implementieren. Beim Start der Anwendung kümmert sich die Klasse `WoerterbuchManager` um das Laden der verfügbaren Wörterbücher als Plug-ins. Hierfür wird die externe Komponente `ServiceLoaders` (seit Java 1.6 im JRE enthalten) eingesetzt.

Die Klasse `DeutschEnglischWoerterbuch` bietet die gewünschte Funktionalität für das Übersetzen von deutschen Begriffen in englische Begriffe. Für die französische Sprache ist die Klasse `DeutschFranzoesischWoerterbuch` zuständig.

Alle Wörterbücher müssen einen Standardkonstruktor haben. Dies ist erforderlich, damit der `ServiceLoader` eine Instanz dieser Klasse dynamisch erzeugen kann. Hier die Klasse `Woerterbuch`:

```
// Datei: Woerterbuch.java
import java.util.List;

public class Woerterbuch
{
    private WoerterbuchManager wm;

    // Konstruktor
    public Woerterbuch()
    {
        // erzeugt Instanz des WoerterbuchManagers als Singleton
        wm = WoerterbuchManager.gibInstanz();
        wm.ladeVorhandeneWoerterbuecher();
    }

    // Uebersetzt Eingabe in verschiedene Sprachen
    public String uebersetze (String wort, String sprache)
    {
        String uebersetztesWort = wm.gibUebersetzung (sprache, wort);
        if (!ueberpruefeWort(uebersetztesWort))
        {
            return "Fehler, Wort nicht im Woerterbuch vorhanden!";
        }
        return uebersetztesWort;
    }

    // Liste aller vorhandenen Woerterbuecher zurueckgeben
    public List<String> gibAlleWoerterbuecher()
    {
        return wm.gibAlleWoerterbuecher();
    }

    // Ueberprueft, ob das Wort im Woerterbuch vorhanden ist
    private boolean ueberpruefeWort (String wort)
    {
        boolean enthalten = true;
```



```
        if (wort == null) {
            enthalten = false;
        }
        return enthalten;
    }
}
```

Die Klasse `WoerterbuchManager` wurde als Singleton realisiert. Sie enthält den Aufruf an die externe JRE-Komponente `ServiceLoader`, die dynamisch weitere Wörterbücher (Plug-ins) einbinden kann. Voraussetzung hierfür ist, dass im Eclipse-Projektverzeichnis ein Ordner mit dem Namen `src/META-INF/services` erzeugt wird. In diesem Ordner muss eine Textdatei angelegt werden, die den Namen der Wörterbuch-Schnittstelle trägt (hier `Service.IWoerterbuch`). Diese Datei enthält Paket- und Klassennamen, in denen Wörterbücher für verschiedene Sprachen zu finden sind und kann auch zur Laufzeit geändert bzw. ergänzt werden. Diese Datei wird im Folgenden dargestellt:

```
// Datei: Service.IWoerterbuch (Zeile nicht in die Datei schreiben)
DeutschEnglischWoerterbuch
DeutschFranzoesischWoerterbuch
```

Es folgt nun der Quelltext der Klasse `WoerterbuchManager`:

```
// Datei: WoerterbuchManager.java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.ServiceLoader;

public final class WoerterbuchManager
{
    private static WoerterbuchManager woerterbuchManager;

    private Map<String, IWoerterbuch> worterbuecherMap;

    // Singleton
    private WoerterbuchManager()
    {
        worterbuecherMap = new HashMap<String, IWoerterbuch>();
    }

    // Aktuelle Instanz des Singletons Woerterbuchmanager zurueckgeben
    public static WoerterbuchManager gibInstanz()
    {
        if (woerterbuchManager != null)
        {
            return woerterbuchManager;
        }
    }
}
```

```

        else
        {
            woerterbuchManager = new WoerterbuchManager();
            return woerterbuchManager;
        }
    }

    // Laedt alle vorhandenen Woerterbuecher
    public void ladeVorhandeneWoerterbuecher()
    {
        ServiceLoader<IWoerterbuch> woerterbuecher =
            ServiceLoader.load (IWoerterbuch.class);
        // Statischer Teil des Interfaces "IWoerterbuch" an
        // ServiceLoader uebergeben, damit passende Plug-ins gefunden
        // werden
        for (IWoerterbuch aktuellesWoerterbuch : woerterbuecher)
        {
            worterbuecherMap.put (aktuellesWoerterbuch.gibSprache(),
                                   aktuellesWoerterbuch);
        }
    }

    // Uebersetzt das Wort in die gewuenschte Sprache
    public String gibUebersetzung (String sprache, String wort)
    {
        IWoerterbuch aktuellesWoerterbuch =
            worterbuecherMap.get (sprache);
        String uebersetztesWort =
            aktuellesWoerterbuch.gibUebersetzung (wort);

        return uebersetztesWort;
    }

    // Alle vorhandenen Woerterbuecher
    public List<String> gibAlleWoerterbuecher()
    {
        return new ArrayList<String>(worterbuecherMap.keySet());
    }
}

```

Es folgt die Schnittstelle `IWoerterbuch`, die für alle Wörterbücher die Methoden `gibUebersetzung(String wort)` und `gibSprache()` vorschreibt:

```

// Datei: IWoerterbuch.java
public interface IWoerterbuch
{
    // Uebersetzt das Wort in die Sprache
    public String gibUebersetzung (String wort);

    // Uebersetzungssprache
    public String gibSprache();
}

```

Die Klasse `DeutschEnglischWoerterbuch` implementiert das Interface `IWoerterbuch` und ist für die Übersetzung von deutschen Begriffen ins Englische zuständig:

```
// Datei: DeutschEnglischWoerterbuch.java
import java.util.HashMap;
import java.util.Map;

public class DeutschEnglischWoerterbuch implements IWoerterbuch
{
    // Woerterliste
    private static Map<String, String> woerterListe;
    // Initialisierung der Woerterliste

    static
    {
        woerterListe = new HashMap<String, String>();
        woerterListe.put ("Hallo", "Hello");
        woerterListe.put ("Auf Wiedersehen", "Goodbye");
    }

    // Uebersetzt in Englisch
    public String gibUebersetzung (String wort)
    {
        return woerterListe.get (wort);
    }

    // Uebersetzungssprache
    public String gibSprache()
    {
        return "Deutsch-Englisch";
    }
}
```

Das bestehende Wörterbuch wird um ein Deutsch-Französisch-Wörterbuch erweitert. Die Klasse `DeutschFranzoesischWoerterbuch` implementiert die Schnittstelle `IWoerterbuch`:

```
// Datei: DeutschFranzoesischWoerterbuch.java
import java.util.HashMap;
import java.util.Map;
public class DeutschFranzoesischWoerterbuch implements IWoerterbuch {
    // Woerterliste
    private static Map<String, String> woerterListe;

    // Initialisierung der Woerterliste
    static
    {
        woerterListe = new HashMap<String, String>();
        woerterListe.put ("Hallo", "Salut");
        woerterListe.put ("Auf Wiedersehen", "Au revoir");
    }
}
```

```

// Uebersetzungssprache
public String gibSprache()
{
    return "Deutsch-Franzoesisch";
}

// Uebersetzt ins Franzoesische
public String gibUebersetzung (String wort)
{
    return woerterListe.get (wort);
}
}

```

Alternativ könnten Klassen weiterer Wörterbücher auch als `jar`-Datei zum Build-Path hinzugefügt werden. Dadurch kann eine weitgehend unabhängige Entwicklung realisiert werden. Wichtig dabei ist, dass die `jar`-Datei ebenfalls einen Ordner `Meta-Inf` enthalten muss.

Die Klasse `Client` enthält die `main()`-Methode und stellt die Anwendung für Wörterbücher dar. Sie nimmt die Eingaben des Benutzers entgegen und übersetzt sie gemäß den installierten Wörterbüchern – also mit Hilfe der Plug-ins. Hier die Klasse `Client`:

```

// Datei: Client.java
import java.util.List;
public class Client
{
    public static void main (String[] args)
    {
        // Woerterbuch-Objekt erzeugen
        Woerterbuch allgemeinesWoerterbuch = new Woerterbuch();

        // Alle verfuegbaren Woerterbuecher
        System.out.println ("Alle vorhandenen Woerterbuecher:");
        List<String> alleWoerterbuecher = allgemeinesWoerterbuch
            .gibAlleWoerterbuecher();
        for (String aktuellesWoerterbuch : alleWoerterbuecher)
        {
            System.out.println (aktuellesWoerterbuch);
        }
        System.out.println();

        // Der Benutzer gibt ein zu uebersetzendes Wort ein
        String wort = "Hallo";
        System.out.println (
            "Es wird das Wort \"" + wort + "\" uebersetzt:");

        // Sprache festlegen und uebersetzen
        String sprache = "Deutsch-Englisch";
        String erg = allgemeinesWoerterbuch.uebersetze (wort, sprache);
        System.out.println (sprache + ": " + erg);
    }
}

```

```
// Sprache festlegen und uebersetzen
sprache = "Deutsch-Franzoesisch";
erg = allgemeinesWoerterbuch.uebersetze (wort, sprache);
System.out.println (sprache + ": " + erg + "\n");

// Der Benutzer gibt ein zu uebersetzendes Wort ein
wort = "Auf Wiedersehen";
System.out.println (
    "Es wird das Wort \"" + wort + "\" uebersetzt:");

// Sprache festlegen und uebersetzen
sprache = "Deutsch-Englisch";
erg = allgemeinesWoerterbuch.uebersetze (wort, sprache);
System.out.println (sprache + ": " + erg);

// Sprache festlegen und uebersetzen
sprache = "Deutsch-Franzoesisch";
erg = allgemeinesWoerterbuch.uebersetze (wort, sprache);
System.out.println (sprache + ": " + erg + "\n");
}
}
```



#### Zuletzt das Protokoll des Programmlaufs:

Alle vorhandenen Woerterbuecher:  
Deutsch-Englisch  
Deutsch-Franzoesisch

Es wird das Wort "Hallo" uebersetzt:  
Deutsch-Englisch: Hello  
Deutsch-Franzoesisch: Salut

Es wird das Wort "Auf Wiedersehen" uebersetzt:  
Deutsch-Englisch: Goodbye  
Deutsch-Franzoesisch: Au revoir

## 5.4 Das Architekturmuster Broker

Das Architekturmuster Broker strukturiert ein **verteiltes System** aus mehreren Clients und Servern in eine Architektur mit einem Broker als Anlaufstelle. Der Broker leitet eine Anfrage eines Clients an den passenden Server weiter und gibt dessen Antwort an den Client zurück. Client- und Server-Komponenten kommunizieren untereinander nur über einen Broker.

### 5.4.1 Name/Alternativer Name

Broker heißt "Makler" oder auch "Vermittler"<sup>87</sup>. Als Name für das Muster ist praktisch nur der englische Begriff "Broker" geläufig. In [Gr302] wird der Name "Object Request Broker" für das Muster benutzt – in Anlehnung an CORBA, einem wichtigen Einsatzgebiet für das Muster (siehe Kapitel 5.4.5).

### 5.4.2 Problem

Große und komplexe Softwaresysteme müssen "wachsen" können: die Anzahl der Nutzer eines Systems kann mit der Zeit zunehmen und daraus folgend wächst auch der Bedarf nach verfügbarer Rechenleistung und Speicherplatz. Die Grenzen eines Ein-Rechner-Systems sind schnell erreicht. Soll das System weiter wachsen können, müssen die Komponenten eines Systems auf mehrere Rechner verteilt werden können. Damit dabei kein kompletter Neuentwurf des Systems erforderlich wird, muss die Architektur des Systems so ausgelegt sein, dass das System skalierbar ist, d. h. leicht an veränderliche Leistungsanforderungen angepasst werden kann. Neben der Zerlegung des Systems in Komponenten (statisch) und neben ihrer Zusammenarbeit (dynamisch) muss bei der Architektur auch auf eine Entkopplung der Komponenten geachtet werden, damit eine Änderung nicht weitreichende Folgen hat. Da die Komponenten aber oftmals durch ihre Zusammenarbeit zu der Gesamtfunktionalität des Systems beitragen, werden sie nie komplett unabhängig voneinander sein, sondern werden stets miteinander kommunizieren müssen.

Wenn jede Komponente jede andere physisch kennt, entsteht eine große Abhängigkeit zwischen den Komponenten. Jede Komponente muss wissen, wie die andere Komponente physisch zu erreichen ist und außerdem muss sie das Kommunikationsprotokoll und das Nachrichtenformat des Partners voll verstehen. Viel besser wäre eine Lösung, bei der sich die Komponenten nur über logische Namen ansprechen würden und bei der jede Komponente senden bzw. empfangen könnte, ohne das spezielle Protokoll des Partners zu kennen. Damit würde die wechselseitige Abhängigkeit minimiert.

### 5.4.3 Lösung

In der folgenden Beschreibung des Musters werden die kommunizierenden Komponenten auf Grund ihrer Rolle in einer Kommunikationsbeziehung unterschieden:

---

<sup>87</sup> Das Architekturmuster Broker entspricht nicht dem Entwurfsmuster Vermittler, siehe dazu 5.4.6.

- **Server-Komponenten** stellen einen oder mehrere Dienste zur Verfügung.
- **Client-Komponenten** benötigen einen oder mehrere Dienste von Server-Komponenten.

Es sei noch erwähnt, dass im Allgemeinen diese Rollen nicht statisch sind, sondern dass eine Server-Komponente zur Laufzeit zur Erbringung ihres Dienstes wiederum die Dienste einer anderen Server-Komponente benötigen kann. Sie nimmt dann in dieser Kommunikation die Rolle eines Clients ein. Analoges gilt für eine Client-Komponente.

Das Broker-Muster entkoppelt in verteilten Softwarearchitekturen Client- und Server-Komponenten, indem zwischen diesen Komponenten ein **Broker als Zwischenschicht** eingeschoben wird und Client- und Server-Komponente untereinander nur über diesen Broker kommunizieren.



Server melden ihre Dienste am Broker an und warten auf eingehende Anfragen eines Clients. Clients, die einen Dienst benötigen, stellen ihre Anfrage an den Broker. Der Broker leitet eine solche Anfrage dann an den entsprechenden Server weiter, der diesen Dienst anbietet. Die Antwort des Servers geht wiederum über den Broker zurück an den Client. Damit ist der **Ort der Leistungserbringung transparent für den Client**. Die Verwendung des Broker-Musters bringt also den Vorteil, dass Server und Client sich nicht mehr gegenseitig physisch kennen, sondern lediglich logisch. Der Broker bildet die logischen Namen auf physische Adressen ab. Über den Broker kommunizieren Client und Server indirekt miteinander.

Ein Broker stellt die Verbindung zwischen Clients und Servern her. Er verbirgt den physischen Ort der Leistungserbringung eines Servers vor dem Client. Der Broker muss aber die Server bzw. deren Dienste kennen. Server müssen sich daher bei einem Broker anmelden, um ihre Dienste anbieten zu können. Clients können dann die angebotenen Dienste über den Broker nutzen.



Zunächst wird der Einfachheit halber nur die Kommunikation zwischen zwei Komponenten betrachtet, nämlich zwischen einem Client und einem Server. Im Folgenden soll eine Komponente eines Systems durch eine Klasse repräsentiert werden. Die Server-Klasse bietet ihren Dienst in Form einer öffentlichen Methode an und die Client-Klasse ruft die Dienstmethode der Server-Klasse auf.

Eine einfache Form der Kommunikation über Methoden kann aber natürlich nur auf einem einzigen Rechner und dort sogar nur innerhalb eines einzigen Betriebssystem-Prozesses (in Java: innerhalb einer Virtuellen Maschine) stattfinden.



Befinden sich die Client- und die Server-Klasse jeweils auf unterschiedlichen Rechnern, muss der Aufruf der Dienstmethode in eine serielle Nachricht umgesetzt werden, die über das Netzwerk zum Server transportiert werden kann. Auf der Seite des Servers entsteht das analoge Problem, dass eine ankommende serielle Nachricht in einen Methodenaufruf zurückgewandelt werden muss. Das Ergebnis des Methodenaufrufs muss auf ähnliche Weise wieder zum Client zurückkommen. Das Wandeln eines Methodenaufrufs in eine serielle Nachricht wird als **Serialisierung** oder **Marshalling** bezeichnet. Der umgekehrte Vorgang heißt **Deserialisierung** oder **Unmarshalling**. Im Rahmen dieser Vorgänge müssen Objekte, die als Argumente des Methodenaufrufs auftreten, ebenso gewandelt werden wie das Ergebnis des Methodenaufrufs – das beispielsweise auch eine geworfene Exception sein kann.

Prinzipiell könnte der Broker die Aufgabe des Serialisierens und Deserialisierens selbst übernehmen. Das würde aber dem Prinzip "Separation of Concerns" widersprechen. Die Aufgaben, Methodenaufrufe in Nachrichten zu wandeln und Nachrichten in Methodenaufrufe umzusetzen, werden deshalb an zwei zusätzliche Klassen übertragen, die eine neue Schicht zwischen dem Client und dem Broker bzw. zwischen dem Server und dem Broker bilden.

Auf der Clientseite wird zwischen Client und Broker ein sogenannter **Client-side Proxy** eingeschoben, der den Server gegenüber dem Client vertritt und die Aufgabe der Serialisierung übernimmt. In symmetrischer Weise vertritt ein **Server-side Proxy** den Client auf der Serverseite und nimmt wiederum die Deserialisierung vor.



Der Client wendet sich also über den Client-side Proxy an den Broker und dieser geht über den Server-side Proxy an den Server. In der umgekehrten Richtung funktioniert es bei der Übertragung des Methodenergebnisses genauso, wobei der Server-side Proxy dann serialisiert und der Client-side Proxy deserialisiert.

Dadurch, dass die Aufgabe der Serialisierung und Deserialisierung separaten Klassen übertragen wird, ergibt sich ein zusätzlicher Vorteil. Client und Server müssen nicht in der gleichen Programmiersprache erstellt werden. Der Methodenaufruf im Client kann über den Client-side Proxy aus dem Objekt- und Typmodell der Programmiersprache des Clients in eine Nachricht, die konform mit dem Objekt- und Typmodell der Programmiersprache des Servers ist, serialisiert werden.

#### 5.4.3.1 Klassendiagramm

Das folgende Klassendiagramm beschreibt die statische Struktur des Architekturmusters Broker:



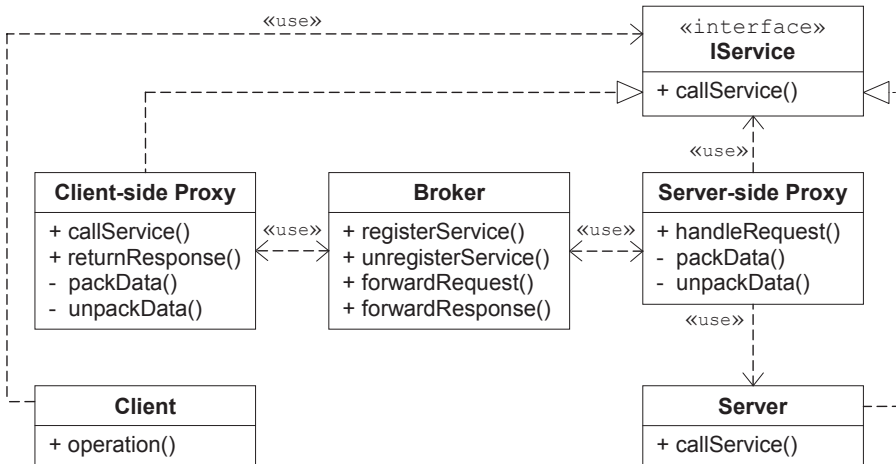


Bild 5-24 Klassendiagramm Broker-Muster

Die gegenseitigen Abhängigkeiten zwischen Broker und den beiden Proxys, die im Bild 5-24 zu sehen sind, sollten in der Praxis mit Hilfe von Interfaces aufgelöst werden.<sup>88</sup> Diese Vorgehensweise wird im Programmbeispiel in Kapitel 5.4.7 gezeigt.

Die Klasse `Server` repräsentiert eine Server-Komponente und stellt beispielhaft als Dienst die Methode `callService()` bereit. Die Aufrufschnittstelle dieser Dienstmethode wird im Interface `IService` definiert. Die Klasse `Client` stellt eine Anwendung dar, die den Dienst `callService()` des Servers benötigt. Die Methode `operation()` der Klasse `Client` steht stellvertretend für die Anwendungslogik des Clients und hat eigentlich für das Muster keine besondere Bedeutung. Es wird aber angenommen, dass im Rahmen der Methode `operation()` der Dienst des Servers benötigt wird – also die Methode `callService()` aufgerufen werden soll.

Die Klasse `Broker` stellt zum einen Methoden zur Verfügung, mit deren Hilfe Dienste dynamisch an- und abgemeldet werden können. Zum anderen wird in der Klasse `Broker` der Nachrichtenaustausch zwischen Client und Server implementiert: im Klassendiagramm sind dafür die Methoden `forwardRequest()` und `forwardResponse()` vorgesehen. Durch Aufruf der Methode `forwardRequest()` des Brokers wird eine Nachricht, die einen Dienstaufruf enthält, angenommen und bearbeitet. Dabei muss im ersten Schritt bei der Bearbeitung dieser Methode ein Broker den Aufrufer identifizieren und speichern, damit er eine spätere Antwort wieder an die richtige Adresse ausliefern kann. Im nächsten Schritt muss der Broker dann den Server ausfindig machen, der den gewünschten Dienst anbietet, und schließlich die Nachricht an den entsprechenden Server-side Proxy transportieren. Die Methode `forwardResponse()` des Brokers ist für den Rücktransport des Ergebnisses des Dienstaufrufs zuständig. Ein Server-side Proxy ruft diese Methode auf und übergibt dabei eine Nachricht, die das Ergebnis des Dienstaufrufs enthält. Der Broker reicht die Nachricht an den Aufrufer

<sup>88</sup> Im Klassendiagramm jedoch wurde aus Gründen der Übersichtlichkeit auf diese Interfaces verzichtet.

des Dienstes weiter, den sich der Broker beim Aufruf der Methode `forwardRequest()` gemerkt hatte.

Ein Broker kommuniziert nicht direkt mit Client und Server, sondern nur **indirekt** über die jeweiligen Proxys. Die Aufgabe der Proxys ist es, einen Methodenaufruf zu serialisieren bzw. zu deserialisieren und auf dem umgekehrten Weg das Ergebnis des Methodenaufrufs zu serialisieren bzw. zu deserialisieren.



Hierfür sind in beiden Proxy-Klassen die Methoden `packData()` und `unpackData()` vorgesehen – `packData()` zum Serialisieren und `unpackData()` zum Deserialisieren.

Wie in Bild 5-24 zu sehen ist, implementiert die Klasse `Client-side Proxy` das Interface `IService`. Dies entspricht dem **Entwurfsmuster Proxy** (siehe Kapitel 4.8). Ein Objekt der Klasse `Client-side Proxy` spielt dem Client gegenüber die Rolle eines Stellvertreters des Servers. Der Client ruft also die Methode `callService()` des `Client-side Proxys` genauso auf, wie er es bei einem direkten Aufruf des Servers auch tun würde. Der `Client-side Proxy` verpackt jedoch den Aufruf in eine Nachricht und schickt sie dem Broker, der die Nachricht weiterleitet. Erhält der Broker eine Antwortnachricht zu dem Dienstauftrag, dann ruft er die Methode `returnResponse()` des `Client-side Proxys` auf. Der `Client-side Proxy` deserialisiert die Nachricht und gibt sie an den Client als Ergebnis des ursprünglichen Dienstauftrags durch den Client zurück.

Auf der Serverseite ist die Klasse `Server-side Proxy` zu sehen. Der Name der Klasse ist etwas irreführend, denn bei dieser Klasse handelt es sich nicht um einen Proxy im Sinne des Proxy-Musters. Die Namensgebung des `Server-side Proxy` rührt daher, dass ein Objekt der Klasse `Server-side Proxy` in gewisser Weise der Stellvertreter des Clients auf der Seite des Servers ist. Ein `Server-side Proxy` erhält vom Broker die Nachricht mit einem Dienstauftrag (Methode `handleRequest()`). Der `Server-side Proxy` entpackt diese Nachricht, interpretiert sie und ruft die Dienstmethode – im Beispiel die Methode `callService()` – des Servers auf. Der `Server-Side Proxy` erhält das Ergebnis des Methodenaufrufs, packt das Ergebnis in eine Nachricht und schickt diese an den Broker mittels der Methode `forwardResponse()`. Für den Server macht es keinen Unterschied, ob die Methode `callService()` vom Client oder vom `Server-side Proxy` aufgerufen wurde.

Wenn die Ausführung einer Dienstmethode durch den Server mit einer Exception beendet wird, dann müssen entsprechende Vorkehrungen in den Proxys getroffen werden: der `Server-side Proxy` muss die Exception fangen und das Exception-Objekt in eine Nachricht serialisieren. Der `Client-side Proxy` muss die Exception-Nachricht deserialisieren und die Exception werfen, sodass der Client die Exception erhält.

Das Muster lässt viele Fragen offen, die erst durch eine Implementierung des Broker-Musters geklärt und beantwortet werden. Die Behandlung des Broker-Musters in diesem Buch kann auch nicht soweit ins Detail gehen, dass damit ein Broker problemlos implementiert werden könnte. In diesem Buch sollen nur die Grundzüge des Musters

erläutert werden. Eine tiefergehende Betrachtung des Broker-Musters ist beispielsweise in [Bus98] zu finden. Einige der offenen Fragen sind beispielsweise:

- Welches Nachrichtenformat soll benutzt werden?
- Woher kennen sich die Objekte in der Kommunikationskette?
- Ist die Kommunikation synchron oder asynchron?

Ein weiterer offener Punkt betrifft die Verteilung der Komponenten auf mehrere Rechner. Aus einem Klassendiagramm wird nicht ersichtlich, welche Komponenten zusammen auf einem Rechner installiert werden müssen. Ein Ziel des Musters ist es ja, dass die Komponenten auf mehrere Rechner verteilt werden können und sich die Verteilung sogar dynamisch ändern kann. Auf den Aspekt der Verteilung wird in der Beschreibung des dynamischen Ablaufs in Kapitel 5.4.3.3 genauer eingegangen.

#### 5.4.3.2 Teilnehmer

##### Client

Der Client enthält eine Anwendungslogik, zu deren Realisierung er die Dienste eines Servers benötigt. Er sendet eine Anfrage über den Client-side Proxy indirekt an einen Server. Der Client benötigt keine Kenntnis über den Ort eines Servers. Jedoch muss er die Schnittstelle des entsprechenden Servers kennen, um dessen lokalen Stellvertreter – also den Client-side Proxy – ansprechen zu können.

##### Client-side Proxy

Der Client-side Proxy ist für den Client der Stellvertreter des Servers. Anfragen eines Clients nach einem Dienst eines Servers werden vom entsprechenden Client-side Proxy entgegengenommen, serialisiert und an den Broker weitergeleitet. Der Client-side Proxy erscheint aus der Sicht des Clients wie der eigentliche Server und kapselt alle Funktionalitäten, die zur Kommunikation über den Broker notwendig sind. Dies umfasst das Ansprechen des Brokers sowie die Serialisierung des Aufrufs des Clients in ein Nachrichtenformat, welches der Broker sowie der Server-side Proxy verstehen. Ebenso ist der Client-side Proxy dafür verantwortlich, die Antwortnachricht, die er vom Broker erhält, wieder zu deserialisieren und als Ergebnis der Dienstanforderung dem Client zurückzugeben.

##### Broker

Der Broker ist für die Kommunikation zwischen Server und Client verantwortlich. Die Server melden ihre Dienste beim Broker an. Der Broker leitet einen Aufruf vom Client-side Proxy an den zugehörigen Server-side Proxy weiter. Hierfür muss er intern eine Liste der angebotenen Dienste mit den dazugehörigen Servern und Server-side Proxys verwalten. Natürlich muss er auch die Antwort des Servers an den Client weiterleiten und sich dafür jeweils den entsprechenden Client gespeichert haben.

##### Server-side Proxy

Der Server-side Proxy ist der Stellvertreter des Clients auf der Seite des Servers. Der Server-side Proxy als Stellvertreter des Clients ruft den Server so auf, dass der Aufruf für den Server so aussieht, als wäre er vom Client direkt aufgerufen worden. Der Server-side Proxy hat somit die Aufgabe, aus der Nachricht des Client-side Proxys durch Deserialisierung die Informationen für einen Methodenaufruf eines Dienstes des Ser-

vers zu erzeugen und dann den Aufruf auch durchzuführen. Der Rückgabewert des Dienstaufrufs wird wiederum vom Server-side Proxy serialisiert und an den Broker weitergeleitet.

### Server

Der Server ist diejenige Klasse bzw. Komponente, die den eigentlichen Dienst zur Verfügung stellt. Ein Dienst wird vom Client über den Client-side Proxy, den Broker und den Server-side Proxy aufgerufen. Das Ergebnis eines Dienstaufrufs geht wieder auf dem umgekehrten Weg zurück.

#### 5.4.3.3 Dynamisches Verhalten

Das folgende Sequenzdiagramm zeigt die Grundform des Broker-Musters, bei der die Komponenten zwar noch nicht verteilt sind, aber die Prinzipien des Broker-Musters auf die Komponenten angewandt wurden und die Infrastruktur zur Kommunikation zwischen den Komponenten bereit steht.

Die Komponenten sind durch das Broker-Muster soweit voneinander entkoppelt, dass sie in eigenen Betriebssystem-Prozessen ablaufen können. Im Sequenzdiagramm in Bild 5-25 wird durch senkrechte Striche angedeutet, dass ein Client und sein Client-side Proxy sowie ein Server und sein Server-side Proxy jeweils zusammen in einem getrennten Betriebssystem-Prozess ablaufen können. Der Broker kann ebenfalls in einem eigenständigen Betriebssystem-Prozess ablaufen.

Die Methodenaufrufe über die Prozessgrenzen hinweg werden zwar durch die Symbole in Bild 5-25 als "normale" synchrone Methodenaufrufe dargestellt, an diesen Stellen müssen aber Mechanismen der Interprozesskommunikation eingesetzt werden.



Das dynamische Verhalten der beteiligten Komponenten wird im Folgenden an Hand einer Client-Anfrage beschrieben. Hier nun das Sequenzdiagramm:

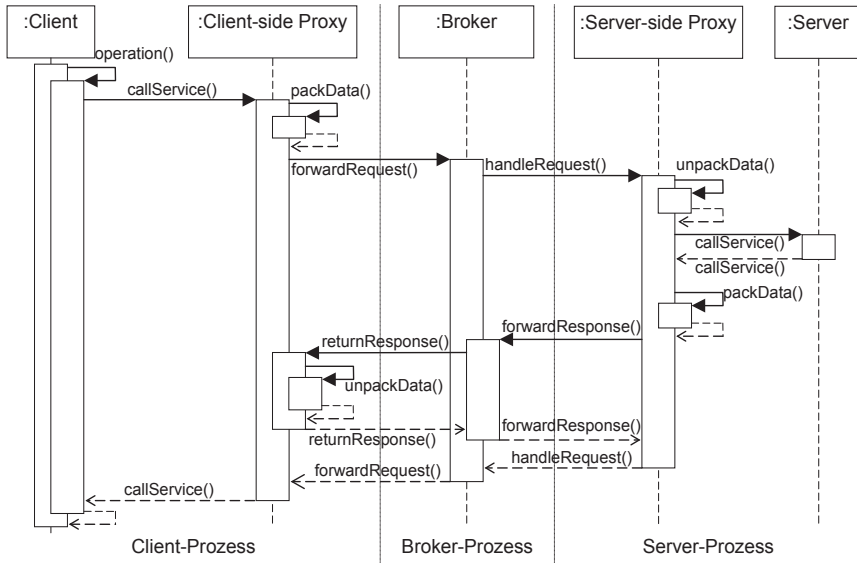


Bild 5-25 Sequenzdiagramm Broker-Muster auf einem Rechner

Der Client kennt nur den lokalen Stellvertreter des Servers – also den Client-side Proxy. Anfragen des Clients werden vom Client-side Proxy entgegengenommen und in eine Nachricht verpackt (`packData()`). Diese Nachricht wird dann zum Broker gesendet (`forwardRequest()`). Nachdem der Broker den passenden Server gefunden hat,<sup>89</sup> sendet er die Nachricht (`handleRequest()`) an den serverseitigen Stellvertreter des Clients, den Server-side Proxy. Der Server-side Proxy nimmt die Nachricht entgegen, entpackt sie (`unpackData()`) und ruft den gewünschten Service des Servers auf (`callService()`). Der Server verarbeitet anschließend den Aufruf und gibt das Ergebnis an den Aufrufer – den Server-side Proxy – zurück. Der Server-side Proxy verpackt das Ergebnis wieder in eine Nachricht (`packData()`) und sendet die Nachricht weiter an den Broker (`forwardResponse()`). Der Broker übergibt dann die Antwortnachricht an den clientseitigen Stellvertreter des Servers, den Client-side Proxy (`returnResponse()`). Der Client-side Proxy entpackt die Daten (`unpackData()`) und übergibt dem Client das Ergebnis.

### Verteilung der Komponenten auf mehrere Rechner

Wie bereits erwähnt wurde, zeigt das Bild 5-25 das Verhalten der Komponenten auf einem einzigen Rechner – also ohne dass die Komponenten auf mehrere Rechner verteilt sind. In einem ersten Ansatz für die Verteilung könnte man die eingezeichneten Grenzen zwischen den Betriebssystem-Prozessen als Rechnergrenzen sehen: Client mit Client-side Proxy, Server mit Server-side Proxy und Broker jeweils auf einem eigenen Rechner. Damit hätten jetzt aber die Proxys neben der Serialisierung bzw. Deserialisierung noch eine weitere Aufgabe, nämlich den Nachrichtentransport zum Broker. Außerdem müssten sie den physischen Ort des Brokers kennen. Damit könnte der

<sup>89</sup> Die Aktivitäten des Brokers zur Bestimmung des passenden Servers werden der Übersicht halber nicht in Bild 5-25 gezeigt.

Broker in einem Fehlerfall nicht so einfach auf einem anderen Rechner installiert werden.

Der Lösungsansatz für die Verteilung liegt darin, den Broker selber zu verteilen. Auf jedem Rechner, auf dem Client- oder Server-Komponenten installiert werden sollen, wird auch ein Broker installiert. Der Broker stellt die Middleware (hier zur Unterstützung der Kommunikation) dar. Client- bzw. Server-Komponenten besitzen also immer einen für sie jeweils **Rechner-lokalen Broker**, mit dem die Proxys kommunizieren können.



Benötigt ein Client einen Dienst von einem Server, der auf dem gleichen Rechner wie der Client installiert ist, kann der lokale Broker die Kommunikation übernehmen. Ist der Server eines angeforderten Dienstes dem lokalen Broker unbekannt, schickt der lokale Broker eines Rechners die Anforderung über das Netzwerk an die anderen **entfernten Broker**. Der Broker, bei dem der angeforderte Dienst registriert ist, übernimmt die Anforderung und beauftragt den Server-side Proxy seines Rechners mit dem Dienstauf-ruf. Das Ergebnis geht wiederum auf dem gleichen Weg zurück.

Das folgende Sequenzdiagramm zeigt diesen Ansatz mit zwei Rechnern:

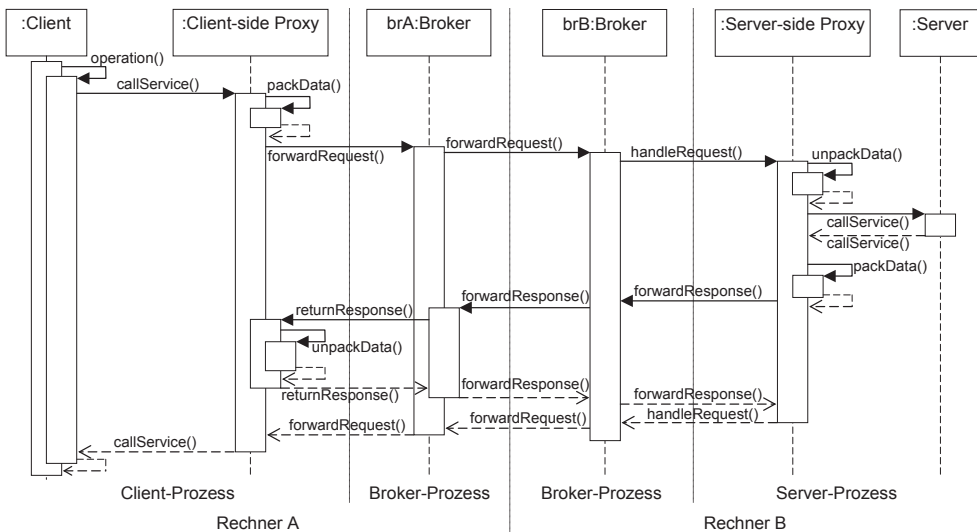


Bild 5-26 Dynamischer Ablauf mit mehreren Brokern

In einem verteilten Broker-System besitzt jede Client- bzw. Server-Komponente einen lokalen Broker, mit dem die Proxys kommunizieren können. Ein Client richtet eine Dienstanfrage immer an seinen lokalen Broker. Ist der Server eines angeforderten Dienstes dem lokalen Broker unbekannt, kommuniziert der lokale Broker über das Netzwerk mit den entfernten Brokern und schickt die Anforderung des Clients an den Broker, bei dem der angeforderte Dienst registriert ist. Eine Server-Komponente braucht daher ihre Dienste auch nur bei ihrem lokalen Broker zu registrieren.



## 5.4.4 Bewertung

### 5.4.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- Das Broker-Muster trennt die Kommunikation zwischen Client und Server von der Funktionalität eines Clients bzw. Servers (**Separation of Concerns**).
- Solange sich die Schnittstelle für die Dienstauftrufe eines Servers nicht ändert, kann die Implementierung eines Servers geändert werden, ohne dass die Clients eines Servers geändert werden müssen. Die Implementierung eines Servers kann sich sogar dynamisch während der Laufzeit eines Clients ändern, wenn gerade keine Dienstanforderung für den Server vorliegt.
- Ein Client muss den physischen Ort der Dienstleistung nicht kennen. Er muss nur über den Client-side Proxy den Rechner-lokalen Broker kennen und den logischen Namen des Servers, um Informationen anzufordern. Die Verteilung der Broker- und Server-Komponenten bleibt dem Client verborgen. Analoges gilt für den Server.
- Client und Server sind plattformunabhängig. Sie können auch in unterschiedlichen Programmiersprachen erstellt werden.
- Durch das Broker-Muster werden sehr große Softwaresysteme auf mehreren verteilten Rechnern möglich.

### 5.4.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Ein **lokaler Broker** stellt für die auf dem gleichen Rechner installierten Komponenten eine zentrale Anlaufstelle dar. Ein Fehler oder gar ein Ausfall eines lokalen Brokers betrifft alle auf dem gleichen Rechner installierten Client- und Server-Komponenten. Die Implementierung eines Brokers erfordert also geeignete Maßnahmen der Fehlertoleranz.
- Der Kommunikationsablauf wird durch indirekte Aufrufe über den Broker aufwendiger. Die Performance ist daher schlechter als bei einem Direktzugriff.

- Ein Broker kann zu einem Engpass werden, was geeignete Maßnahmen bezüglich des Durchsatzes erfordert.
- Die Proxys sind abhängig vom Broker. Wird der Broker ausgetauscht, müssen sie angepasst werden.

### 5.4.5 Einsatzgebiete

Das Broker-Muster ist in folgenden Fällen einzusetzen:

- Client- und Server-Komponenten sollen voneinander entkoppelt werden.
- Komponenten sollen auf Dienste anderer Komponenten über logische Namen zugreifen können, ohne zu wissen, wo – also auf welchem Rechner – sich diese gerade physisch befinden.
- Zur Laufzeit sollen sich Komponenten ändern dürfen, wenn gerade kein Dienst aufgerufen wird.
- Die Implementierungsdetails von Client-Komponenten und von Diensten sollen verborgen werden.

Das wohl bekannteste Beispiel für das Broker-Muster ist das **Internet**. Dort werden über eine Vielzahl von unabhängigen Servern Dienste angeboten. Für clientseitige Anwendungen ist es dabei unmöglich, alle Server und Dienste direkt zu kennen. Aus diesem Grund werden Vermittler eingesetzt, die mittels des Broker-Musters die Server und Clients zusammenführen. Ein Domain-Name-Server (DNS) ist dabei die zentrale Komponente des Vermittlers und ist für die Auflösung der Adressen von logischen in physische Adressen zuständig.

Ein weiterer bekannter Anwendungsfall für das Broker-Muster ist die **Common Object Broker Request Architecture (CORBA)**. CORBA wird von der Object Management Group (OMG) entwickelt [OMGCOS]<sup>90</sup> und ermöglicht den Zugriff auf entfernte Objekte, die in unterschiedlichen Programmiersprachen und auf unterschiedlichen Plattformen implementiert sein können. Dies wird durch den Einsatz einer sogenannten **Interface Definition Language (IDL)** und eines sogenannten **General Inter-ORB Protocol (GIOP)** erreicht. Die IDL dient dazu, eine Schnittstelle unabhängig von einer Programmiersprache zu beschreiben. Das GIOP legt fest, welche Daten in welchem Format zu übertragen sind. Mit Hilfe des GIOP können Broker, die für unterschiedliche Plattformen von unterschiedlichen Herstellern entwickelt wurden und auf unterschiedlichen Rechnern laufen, miteinander kommunizieren. Die eigentliche Umsetzung des GIOP kann für verschiedene Transportprotokolle unterschiedlich sein. Das bekannteste Beispiel einer Umsetzung ist das Internet Inter-ORB Protocol (IIOP), das ein GIOP auf Basis von TCP/IP als Transportprotokoll darstellt. Der Broker wird in CORBA als **Object Request Broker (ORB)** bezeichnet. Ähnlich wie beim Broker-Muster (vgl. Bild 5-26) ist ein ORB selbst eine verteilte Anwendung. Bild 5-27 zeigt die Struktur von CORBA:

---

<sup>90</sup> Ein Spezifikationsdokument ist in der Regel nicht geeignet als Einstieg in ein Thema. Eine sehr gute Übersicht über CORBA bietet [OMGCOB].



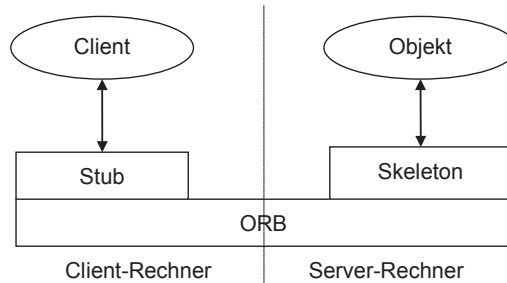


Bild 5-27 Struktur von CORBA

Damit ein Server seine Objekte anbieten kann, müssen zuerst die Schnittstellen der Objekte mit Hilfe der IDL definiert werden. Diese Schnittstellendefinition kann anschließend von einem IDL-Compiler verwendet werden, um den sogenannten **Skeleton** zu generieren. Der Skeleton entspricht hierbei dem **Server-side Proxy** des Broker-Musters und ist beim Aufruf des Servers für das sogenannte **Unmarshalling** zuständig, d. h., er wandelt die serialisierte Nachricht des Clients wieder in einen Methodenaufruf um. Liefert der Methodenaufruf ein Ergebnis, nimmt der Skeleton das Ergebnis entgegen, serialisiert es und leitet es auf dem umgekehrten Weg über den Broker an den Client zurück.

Auf der Seite des Clients wird ebenfalls ein Proxy mit Hilfe des IDL-Compilers erstellt. Dieser wird bei CORBA als **Stub** bezeichnet. Der Stub hat die Aufgabe, einen Methodenaufruf des Clients in eine Nachricht zu serialisieren (**Marshalling**), die Nachricht an den ORB weiterzuleiten und auf eine Antwortnachricht zu warten. Trifft eine Antwort ein, wird sie wieder deserialisiert und als Ergebnis des entsprechenden Methodenaufrufs an den Client zurückgegeben. Somit entspricht der Stub dem **Client-side Proxy**. Bild 5-28 stellt die Erzeugung von Stub und Skeleton dar:

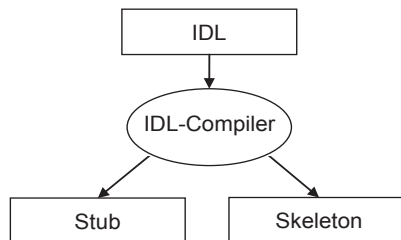


Bild 5-28 Erzeugung von Stub und Skeleton

### 5.4.6 Ähnliche Muster

Das Broker-Muster ähnelt in Aufbau und Verhalten dem Vermittler-Muster, dem Forwarder-Receiver-Muster und dem Client-Dispatcher-Server-Muster. Von der Zielsetzung her ist das Broker-Muster mit dem Ansatz einer serviceorientierten Architektur vergleichbar.

Ein Broker ist umgangssprachlich auch ein Vermittler. Das Broker-Muster hat auch eine gewisse Ähnlichkeit mit dem **Vermittler-Muster**, dadurch dass die Kommunikation zwischen Komponenten (Kollegen) zentralisiert über einen (verteilten) Broker (Vermittler) abläuft. Bei einer genaueren Betrachtung ergeben sich aber wesentliche Unterschiede:

- Beim Vermittler-Muster kommunizieren die Kollegen über den Vermittler, wobei ein Vermittler auf Grund von eingehenden Methodenaufrufen betroffene Kollegen informiert. Der Vermittler muss also die Information besitzen, welche Kollegen alle zu informieren sind. Alle Beteiligten befinden sich im selben System. Beim Broker-Muster spielt der Broker eine ähnliche Rolle wie ein Vermittler, jedoch können beim Broker-Muster Clients, Server und Broker verteilt auf verschiedenen Rechnern ablaufen und außerdem leitet der Broker eine Nachricht nur an den einen einzigen, gewünschten Empfänger weiter, der den entsprechenden Service zur Verfügung stellt.
- Eine Komponente teilt dem Broker den Empfänger einer Nachricht mit, der Broker ist für die Lokalisierung der Empfänger-Komponente im verteilten System, den Transport der Nachricht zum Empfänger und ggf. auch für den Rücktransport einer Antwort zuständig. Dieser letzte Aspekt der Antwort fehlt beim Vermittler-Muster vollständig. Dies liegt auch daran, dass es beim Vermittler-Muster keine 1-zu-1-Zuordnung zwischen Sender und Empfänger wie beim Broker-Muster gibt, sondern eine 1-zu-n-Zuordnung. Im Vermittler wird abgelegt, welche anderen Kollegen über eine Nachricht informiert werden sollen.

Beim Muster **Client-Dispatcher-Server** [Gal03] wird zwischen das klassische Client-Server Modell ähnlich wie beim Broker-Muster eine vermittelnde Schicht (Dispatcher) eingezeichnet. Der Dispatcher sorgt aber im Gegensatz zu einem Broker nur für einen direkten Kommunikationskanal zwischen Client und Server. Client und Server können dann nach der Bereitstellung des Kommunikationskanals direkt über den Kommunikationskanal miteinander kommunizieren.

Das Muster **Forwarder-Receiver** [Bie00] befasst sich mit der Interprozesskommunikation zwischen den Komponenten. Ein Forwarder entspricht in etwa einem Client-side Proxy des Broker-Musters. Auf der Serverseite hat ein Receiver eine ähnliche Aufgabe wie ein Server-side Proxy. Auf die Vermittlerkomponente wird aber im Forwarder-Receiver-Muster komplett verzichtet. Forwarder und Receiver kommunizieren direkt ohne Vermittler miteinander.

Eine **serviceorientierte Architektur (SOA)** sorgt für die Entkopplung von Client und Server. Dies ist auch das Ziel des Broker-Architekturmusters. Während beim Broker der physische Ort der Leistungserbringung verborgen wird, ist er bei einer SOA prinzipiell bekannt und z. B. in der Web Service-Beschreibung vermerkt. Sowohl bei einer SOA als auch beim Broker wird eine Zwischenschicht für die Kommunikation eingezeichnet. Im Gegensatz zum Broker wird bei einer SOA die Abbildung von Anwendungsfällen aus Verarbeitungssicht auf Komponenten berücksichtigt.

### 5.4.7 Programmbeispiel

In diesem Abschnitt wird eine Implementierung des Broker-Musters in Form einer Simulation in einem einzigen Betriebssystem-Prozess vorgestellt. In dieser Simulation

werden somit keine Interprozesskommunikations-Mechanismen benötigt, sondern es werden nur synchrone Methodenaufrufe verwendet. Die Verteilung der Klassen auf mehrere Rechner und ein asynchrones Verhalten spielen beim Nachrichtenaustausch in diesem Beispiel also keine Rolle.

Im vorliegenden Beispiel gibt es zwei Server: zum einen ein Objekt der Klasse `TemperaturDienst` und zum anderen ein Objekt der Klasse `WetterDienst`. Beide Klassen enthalten jeweils eine Methode, die als Dienst zur Verfügung gestellt werden soll: in der Klasse `TemperaturDienst` die Methode `erfrageTemperatur()` und in der Klasse `WetterDienst` die Methode `erfrageWetter()`. Die Dienste müssen mit einem Dienstenamen beim Broker registriert werden. Der Einfachheit halber wird als Dienstname der jeweilige Methodenname gewählt.

Intern arbeitet der Broker mit zwei Objekten der Klasse `HashMap`:

- In dem Objekt `serverMap` werden vom Broker die Dienste aller Server mit ihrem Namen registriert. Da der Broker aber nicht mit den Servern direkt kommunizieren kann, werden in dem Objekt `serverMap` die entsprechenden Server-side Proxys unter dem Dienstenamen eingetragen. Bei einer Anfrage eines Clients sucht der Broker in dieser Hashmap nach dem passenden Server-side Proxy und leitet die Anfrage des Clients weiter.
- In dem Objekt `clientMap` registriert der Broker die Client-side Proxys mit ihrem Namen für die Dauer eines Aufrufs. Dies ist notwendig, damit der Broker eine Antwortnachricht wieder an den richtigen Client-side Proxy zurückliefern kann. Als Name eines Client-side Proxys wird in dem Objekt `clientMap` der Klassenname des jeweiligen Client-side Proxys benutzt. Ein Client-side Proxy fügt den Namen seiner Klasse in die Anfragenachricht ein. Aus dieser Nachricht kann dann sowohl der Broker als auch später der Server-side Proxy den Namen des Client-side Proxys und damit indirekt auch den Namen des Clients ermitteln.

Als Format für Nachrichten wird ein Byte-Array verwendet. In dem Byte-Array, das die Nachricht mit der Anforderung eines Dienstaufrufs enthält, werden der Name des Client-side Proxys, der Name des angeforderten Dienstes und die Parameter für den Dienstaufwurf mit einem Semikolon getrennt übergeben. In dem Byte-Array für die Rückgabenachricht werden der Name des Client-side Proxys und der Rückgabewert übertragen.

Zuerst werden die jeweiligen Schnittstellen für die Dienste definiert. Dazu dienen die Interfaces `ITemperaturDienst` und `IWetterDienst`. Diese Interfaces werden dann von den Servern aber auch von den jeweiligen Client-Side Proxys implementiert. Es folgt zuerst das Interface `IWetterDienst` mit der Methode `erfrageWetter()`, mit der das Wetter von heute oder morgen abgefragt werden kann:

```
// Datei: IWetterDienst.java
public interface IWetterDienst
{
    public String erfrageWetter (Tag tag);
}
```

Das Interface `ITemperaturDienst` enthält die Methode `erfrageTemperatur()`, die als Ergebnis die Temperatur von heute oder morgen liefert:

```
// Datei: ITemperaturDienst.java
public interface ITemperaturDienst
{
    public Float erfrageTemperatur (Tag tag);
}
```

Die Enumeration `Tag` wird als Typ für den Übergabeparameter der beiden Dienstmethoden verwendet. Die Enumeration `Tag` erlaubt nur die Abfrage des Wetters für heute oder morgen:

```
// Datei: Tag.java
public enum Tag
{
    Heute,
    Morgen
}
```

Die Klassen `TemperaturDienst` und `WetterDienst` sind die eigentlichen Server in diesem Beispiel. Sie implementieren die in den jeweiligen Interfaces definierten Dienste. Zuerst wird die Klasse `TemperaturDienst` vorgestellt:

```
// Datei: TemperaturDienst.java
public class TemperaturDienst implements ITemperaturDienst
{
    public TemperaturDienst()
    {
        System.out.println ("TemperaturDienst: instanziiert");
    }

    // Diese Methode stellt den Dienst des Temperatur-Servers dar
    public Float erfrageTemperatur (Tag tag)
    {
        Float temperatur;
        switch (tag)
        {
            case Heute:
                temperatur = 25f;
                break;
            case Morgen:
                temperatur = 10f;
                break;
            default:
                temperatur = Float.NaN;
                break;
        }
        return temperatur;
    }
}
```

Die Klasse `WetterDienst` ist die zweite Server-Klasse in diesem Beispiel:

```
// Datei: WetterDienst.java
public class WetterDienst implements IWetterDienst
{
    public WetterDienst()
    {
        System.out.println ("WetterDienst: instanziiert");
    }

    // Diese Methode stellt den Dienst des Wetter-Servers dar.
    public String erfrageWetter (Tag tag)
    {
        String wetter;
        switch (tag)
        {
            case Heute:
                wetter = "Sonnenschein";
                break;
            case Morgen:
                wetter = "Regen";
                break;
            default:
                wetter = "Kein Wetter bekannt";
                break;
        }
        return wetter;
    }
}
```

Damit der Broker die Proxys auf der Server-Seite ansprechen kann, wird das Interface `IServersideProxy` definiert, das von den Server-side Proxys implementiert werden muss. Es enthält die Methode `bearbeiteAnfrage()`, die vom Broker aufgerufen wird:

```
// Datei: IServersideProxy.java
public interface IServersideProxy
{
    public void bearbeiteAnfrage (byte[] anfrage);
}
```

Ein Objekt der Klasse `ServersideTemperaturDienstProxy` ist der Server-side Proxy auf der Seite des Temperaturdienstes. Ein solches Objekt ist dafür zuständig, die von dem Broker erhaltenen Informationen in einen Aufruf zu übersetzen und den Aufruf dann so durchzuführen, dass es für die Klasse `TemperaturDienst` so aussieht, als würde sie vom Client direkt aufgerufen werden. Ebenso wandelt der Server-side Proxy die zurückgegebene Antwort, so dass der Broker und der zugehörige Proxy auf der Seite des Clients die Nachricht verstehen. Der Server-side Proxy ist ebenfalls dafür zuständig, sich beim Broker zu registrieren. Hierzu meldet er sich mit dem Namen des angebotenen Dienstes an. Im Folgenden die Klasse `ServersideTemperaturDienstProxy`:

```
// Datei: ServersideTemperaturDienstProxy.java
public class ServersideTemperaturDienstProxy
    implements IServersideProxy
{
    }
```

```

// Ein Server-side Proxy kennt den Broker und den
// eigentlichen Server.
private ITemperaturDienst temperaturDienst;
private Broker broker;

public ServersideTemperaturDienstProxy (Broker broker)
{
    System.out.println
        ("ServersideTemperaturDienstProxy: instanziiert");
    this.broker = broker;
    temperaturDienst = new TemperaturDienst();
    broker.anmelden ("erfrageTemperatur", this);
}

// Bearbeite eine Anfragenachricht, die vom Broker kommt.
// Aus der Anfragenachricht wird das Argument des Aufrufs geholt
// und anschliessend die Dienstmethode des Servers aufgerufen.
// Das Ergebnis des Servers wird in eine Antwortnachricht
// verpackt und an den Broker geschickt.
public void bearbeiteAnfrage (byte[] anfrage)
{
    Tag tag = auspacken (anfrage);
    Float temperatur = temperaturDienst.erfrageTemperatur (tag);
    byte[] antwort = verpacken (anfrage, temperatur);
    broker.antwortWeiterleiten (antwort);
}

// Packe aus einer Anfragenachricht das Argument des
// Dienstaufrufs aus.
private Tag auspacken (byte[] gepackteNachricht)
{
    String string = new String (gepackteNachricht);
    String[] entpackteNachricht = string.split (";");
    Tag tag = Tag.valueOf (entpackteNachricht[2]);
    return tag;
}

// Besorge aus der Anfragenachricht den Clientnamen und
// verpacke diesen zusammen mit dem Ergebniswert
// in eine Antwortnachricht.
private byte[] verpacken (byte[] anfrage, Float wert)
{
    String[] strings = (new String (anfrage)).split (";");
    String clientName = strings[0];
    return (clientName + ";" + wert).getBytes();
}
}

```

Die Klasse `ServersideWetterDienstProxy` hat die gleiche Rolle wie die Klasse `ServersideTemperaturDienstProxy` – sie ist allerdings für den Wetterdienst-Server zuständig:

```

// Datei: ServersideWetterDienstProxy.java
public class ServersideWetterDienstProxy implements IServersideProxy
{

```

```

// Ein Server-side Proxy kennt den Broker und den
// eigentlichen Server.
private IWetterDienst wetterDienst;
private Broker broker;

public ServersideWetterDienstProxy (Broker broker)
{
    System.out.println
        ("ServersideWetterDienstProxy: instanziiert");
    this.broker = broker;
    wetterDienst = new WetterDienst();
    broker.anmelden ("erfrageWetter", this);
}

// Bearbeite eine Anfragenachricht, die vom Broker kommt.
// Aus der Anfragenachricht wird das Argument des Aufrufs geholt
// und anschliessend die Dienstmethode des Servers aufgerufen.
// Das Ergebnis des Servers wird in eine Antwortnachricht
// verpackt und an den Broker geschickt.
public void bearbeiteAnfrage (byte[] anfrage)
{
    Tag tag = auspacken (anfrage);
    String wetter = wetterDienst.erfrageWetter (tag);
    byte[] antwort = verpacken (anfrage, wetter);
    broker.antwortWeiterleiten (antwort);
}

// Packe aus einer Anfragenachricht das Argument des
// Dienstaufrufs aus.
private Tag auspacken (byte[] gepackteNachricht)
{
    String string = new String (gepackteNachricht);
    String[] entpackteNachricht = string.split (";");
    Tag tag = Tag.valueOf (entpackteNachricht[2]);
    return tag;
}

// Besorge aus der Anfragenachricht den Clientnamen und
// verpacke diesen zusammen mit dem Ergebniswert
// in eine Antwortnachricht.
private byte[] verpacken (byte[] anfrage, String ergebnis)
{
    String[] strings = (new String (anfrage)).split (";");
    String clientName = strings[0];
    return (clientName + ";" + ergebnis).getBytes();
}
}

```

Ein Objekt der Klasse `Broker` leitet eine Anfragenachricht von einem Client-side Proxy weiter. Diese Nachrichten bestehen aus Byte-Arrays. Beim Empfang einer Anfrage legt der Broker eine Referenz auf den Client-side Proxy mit dessen Namen als Schlüssel in der Hashmap `clientMap` ab. Hierzu dient die private Methode `ermittleClientName()` der Klasse `Broker`, die den Namen des Clients aus der übergebenen Anfragenachricht ermittelt. Die private Methode `findServer()` der Klasse `Broker` hat die Aufgabe, den passenden Server zu dem angefragten Dienst zu finden.

Die Anfragenachricht wird dann vom Broker an den Server-side Proxy des gefundenen Servers geschickt.

Die Methode `findeService()` der Klasse `Broker` sucht in der `HashMap serverMap` nach dem passenden Server zu einem angefragten Dienst. Da der Broker nicht mit einem Server direkt kommunizieren kann sondern nur mit dessen Server-side Proxy, wird in der `HashMap serverMap` die Referenz auf den Server-Side Proxy eines Servers gespeichert. Über die Methoden `anmelden()` wird die Referenz auf einen Server-side Proxy unter dem angegebenen Dienstenamen in die `HashMap serverMap` eingetragen. Ein Eintrag kann unter Angabe des Namens mit Hilfe der Methode `abmelden()` aus dieser `HashMap` wieder ausgetragen werden.

Auf ähnliche Weise leitet der Broker die Antwortnachricht eines Servers wieder an den Client zurück. Dabei wird jedoch die `HashMap clientMap` in der privaten Methode `findeClient()` der Klasse `Broker` verwendet, um die Referenz des Client-side Proxys zu ermitteln, an den die Antwortnachricht übergeben werden soll. Hier die Klasse `Broker`:

```
// Datei: Broker.java
import java.util.HashMap;
public class Broker
{
    private HashMap<String, IClientsideProxy> clientMap;
    private HashMap<String, IServersideProxy> serverMap;

    public Broker()
    {
        clientMap = new HashMap<String, IClientsideProxy>();
        serverMap = new HashMap<String, IServersideProxy>();
        System.out.println ("Broker: instanziiert");
    }

    // Ein Server-side Proxy wird angemeldet. Anforderungen fuer den
    // angegebenen Dienst, koennen an ihn weitergeleitet werden.
    public void anmelden (String dienstName, IServersideProxy proxy)
    {
        serverMap.put (dienstName, proxy);
        System.out.println ("Broker: Dienst \"" + dienstName +
            "\" angemeldet");
    }

    // Ein Dienst wird wieder abgemeldet und aus der
    // serverMap entfernt.
    public void abmelden (String dienstName)
    {
        serverMap.remove (dienstName);
        System.out.println ("Broker: Dienst \"" + dienstName +
            "\" abgemeldet");
    }

    // Diese Methode ruft ein Client-side Proxy auf,
    // um einen Dienst anzufragen. Der Dienst ist in der
    // Anfragenachricht enthalten ebenso wie der Clientname.
```



```
public void anfrageWeiterleiten (byte[] anfrage,
                                IClientsideProxy proxy)
{
    String clientName = ermittleClientName (anfrage);
    String dienstName = ermittleDienstName (anfrage);
    clientMap.put (clientName, proxy);
    IServersideProxy serverProxy = findeServer (dienstName);
    serverProxy.bearbeiteAnfrage (anfrage);
}

// Diese Methode ruft ein Server-side Proxy auf, der eine
// Antwortnachricht als Ergebnis eines Dienstaufrufs
// an den Client schicken will. Der Clientname ist in der
// Antwortnachricht enthalten.
public void antwortWeiterleiten (byte[] antwort)
{
    String clientName = ermittleClientName (antwort);
    IClientsideProxy clientProxy = findeClient (clientName);
    clientProxy.bearbeiteAntwort (antwort);
    clientMap.remove (clientName);
}

// Diese Methode sucht in der serverMap nach dem Dienstnamen.
private IServersideProxy findeServer (String dienst)
{
    return serverMap.get (dienst);
}

// Diese Methode sucht in der clientMap nach dem Clientnamen.
private IClientsideProxy findeClient (String client)
{
    return clientMap.get (client);
}

// Ermittle Clientname aus Antwortnachricht
private String ermittleClientName (byte[] nachricht)
{
    String[] strings = (new String (nachricht)).split (";");
    return strings[0];
}

// Ermittelt Dienstname aus der Anfragenachricht.
private String ermittleDienstName (byte[] anfrage)
{
    String[] strings = (new String (anfrage)).split (";");
    return strings[1];
}
}
```

Damit ein Broker Antwortnachrichten wieder an einen Client-side Proxy zurückliefern kann, muss die entsprechende Klasse das folgende Interface `IClientsideProxy` implementieren:

```
// Datei: IClientsideProxy.java
public interface IClientsideProxy
{
    public void bearbeiteAntwort (byte[] antwort);
}
```

Die Klasse `ClientsideTemperaturDienstProxy` ist der Stellvertreter des Temperaturdienstes auf der Seite des Clients. Hierfür muss sie die gleiche Schnittstelle wie auch die Klasse `TemperaturDienst` implementieren. Damit der Broker die Antwortnachrichten wieder zurückliefern kann, muss die Klasse `ClientsideTemperaturDienstProxy` des Weiteren die Schnittstelle `IClientsideProxy` implementieren:

```
// Datei: ClientsideTemperaturDienstProxy.java
public class ClientsideTemperaturDienstProxy
    implements ITemperaturDienst, IClientsideProxy
{
    private Broker broker;
    private Float response;

    public ClientsideTemperaturDienstProxy (Broker broker)
    {
        this.broker = broker;
        System.out.println
            ("ClientSideTemperaturDienstProxy: instanziiert");
    }

    // Nimm Dienstaufwurf entgegen, erzeuge Anfragenachricht
    // und schicke sie an den Broker.
    // Die zurueckgelieferte Antwort wird als Ergebnis des Aufrufs
    // dem Client uebergeben.
    public Float erfrageTemperatur (Tag tag)
    {
        String dienstName = "erfrageTemperatur";
        byte[] anfrage = verpacken (dienstName, tag);
        broker.anfrageWeiterleiten (anfrage, this);
        return response;
    }

    // Hier kommt die Antwort des Servers ueber den Broker zurueck.
    // Die Nachricht wird ausgepackt und intern zwischengespeichert.
    public void bearbeiteAntwort (byte[] antwort)
    {
        response = auspacken (antwort);
    }

    // Erstelle eine Anfragenachricht in Form eines Byte-Arrays:
    // mit eigenem Namen, Namen der aufgerufenen Dienstmethode
    // und dem Argument des Dienstaufwurfs.
    private byte[] verpacken (String methode, Tag tag)
    {
        String meinName = "ClientsideTemperaturDienstProxy";
        String anfrage = meinName + ";" + methode + ";"
            + tag.toString();
    }
}
```

```

        return anfrage.getBytes();
    }

    // Packe die Antwortnachricht aus dem Byte-Array aus:
    // liefere das Ergebnis des Dienstaufrufs als Float zurueck.
    private Float auspacken (byte[] nachricht)
    {
        String[] strings = (new String (nachricht)).split (";");
        return Float.valueOf (strings[1]);
    }
}

```

Die Klasse `ClientsideWetterDienstProxy` ist der Stellvertreter des Wetterdienstes auf der Seite des Clients. Hierfür muss sie die gleiche Schnittstelle wie auch die Klasse `WetterDienst` implementieren. Auch hier gilt: damit der Broker die Antwortnachrichten wieder zurückliefern kann, muss die Klasse `ClientsideWetterDienstProxy` die Schnittstelle `IClientsideProxy` implementieren. Hier nun die Klasse `ClientsideWetterDienstProxy`:

```

// Datei: ClientsideWetterDienstProxy.java
public class ClientsideWetterDienstProxy
    implements IWetterDienst, IClientsideProxy
{
    private Broker broker;
    private String response;

    public ClientsideWetterDienstProxy (Broker broker)
    {
        this.broker = broker;
        System.out.println
            ("ClientSideWetterDienstProxy: instanziiert");
    }

    // Nimm Dienstaufwurf entgegen, erzeuge Anfragenachricht
    // und schicke sie an den Broker.
    // Die zurueckgelieferte Antwort wird als Ergebnis des Aufrufs
    // dem Client uebergeben.
    public String erfrageWetter (Tag tag)
    {
        String dienstName = "erfrageWetter";
        byte[] anfrage = verpacken (dienstName, tag);
        broker.anfrageWeiterleiten (anfrage, this);
        return response;
    }

    // Hier kommt die Antwort des Servers ueber den Broker zurueck.
    // Die Nachricht wird ausgepackt und intern zwischengespeichert.
    public void bearbeiteAntwort (byte[] antwort)
    {
        response = auspacken (antwort);
    }

    // Erstelle eine Anfragenachricht in Form eines Byte-Arrays:
    // mit eigenem Namen, Namen der aufgerufenen Dienstmethode
    // und dem Argument des Dienstaufrufs.
}

```

```

private byte[] verpacken (String methode, Tag tag)
{
    String meinName = "ClientsideWetterDienstProxy";
    String anfrage = meinName + ";" + methode + ";"
                    + tag.toString();
    return anfrage.getBytes();
}

// Packe die Antwortnachricht aus dem Byte-Array aus:
// liefere das Ergebnis des Dienstaufrufs als String zurueck.
private String auspacken (byte[] nachricht)
{
    String[] strings = (new String (nachricht)).split (";");
    return strings[1];
}
}

```

Die Klasse `Client` enthält die eigentliche Anwendungslogik in der Methode `druckeWetterdaten()`. Damit die Klasse `Client` mit lokalen wie mit entfernten Wetterservern arbeiten kann, werden als Typ für die Referenzen auf die Server nur Interfaces – nämlich `IWetterDienst` und `ITemperaturDienst` – benutzt. Diese Referenzen können von außen durch Aufruf der Methode `injectServices()` gesetzt werden. Im Beispiel erfolgt das Setzen der Referenzen im Hauptprogramm der Klasse `TestBroker`. Es folgt der Quellcode der Klasse `Client`:

```

// Datei: Client.java
public class Client
{
    // Diese Referenzen koennen sowohl auf echte Server-Objekte
    // zeigen, aber auch auf Client-side Proxys
    IWetterDienst wetterServer;
    ITemperaturDienst temperaturServer;

    public void injectServices (IWetterDienst wds,
                               ITemperaturDienst tds)
    {
        wetterServer = wds;
        temperaturServer = tds;
    }

    // Diese Methode stellt die eigentliche Client-Anwendung dar.
    // Es werden Wetter- und Temperaturdaten von den Servern
    // abgefragt und ausgegeben.
    public void druckeWetterdaten()
    {
        // Wetter und Temperatur fuer heute anfordern
        System.out.println();
        System.out.println ("Client: " +
                             "Wetter und Temperatur fuer heute anfragen:");
        String wetterHeute =
            wetterServer.erfrageWetter (Tag.Heute);
        Float temperaturHeute =
            temperaturServer.erfrageTemperatur (Tag.Heute);
        System.out.println ("Heute gibt es " + wetterHeute
                             + " bei " + temperaturHeute + " Grad.");
    }
}

```

```

// Wetter und Temperatur fuer morgen anfordern
System.out.println();
System.out.println ("Client: " +
    "Wetter und Temperatur fuer morgen anfragen:");
String wetterMorgen =
    wetterServer.erfrageWetter (Tag.Morgen);
Float temperaturMorgen =
    temperaturServer.erfrageTemperatur (Tag.Morgen);
System.out.println ("Morgen gibt es " + wetterMorgen
    + " bei " + temperaturMorgen+ " Grad.");
    }
}

```

Die Klasse `TestBroker` enthält das Hauptprogramm des Beispiels. Im Hauptprogramm wird ein Objekt der Klasse `Broker` instanziiert und die lokalen – d. h. Client-side Proxys und die Proxys auf der Seite der Server erzeugt. Nach diesen Vorbereitungen wird ein Objekt der Klasse `Client` erstellt und seine Referenzen so gesetzt, dass es die benötigten Dienste nutzen kann. Als letzter Schritt wird dann im Hauptprogramm die Clientanwendung – im Beispiel die Methode `druckeWetterdaten()` – aufgerufen, wie im folgenden Quellcode zu sehen ist:

```

// Datei: TestBroker.java
public class TestBroker
{
    public static void main (String[] args)
    {
        // Broker, Server und Server-side Proxys erzeugen
        System.out.println
            ("TestBroker: Broker und Dienste erzeugen");
        Broker broker = new Broker();
        new ServersideWetterDienstProxy (broker);
        new ServersideTemperaturDienstProxy (broker);

        // Client-side Proxys erzeugen
        System.out.println();
        System.out.println
            ("TestBroker: Client-side Proxys erzeugen");
        ClientsideWetterDienstProxy cwp =
            new ClientsideWetterDienstProxy (broker);
        ClientsideTemperaturDienstProxy ctp =
            new ClientsideTemperaturDienstProxy (broker);

        // Client-Objekt erstellen und mit Referenzen
        // auf die Dienste versorgen. Es werden die Referenzen
        // auf die Client-side Proxys uebergeben!
        Client anwendung = new Client();
        anwendung.injectServices (cwp, ctp);

        // Jetzt kann die Anwendung gestartet werden:
        anwendung.druckeWetterdaten();
    }
}

```



### Hier das Protokoll des Programmlaufs:

```
TestBroker: Broker und Dienste erzeugen
Broker: instanziiert
ServersideWetterDienstProxy: instanziiert
WetterDienst: instanziiert
Broker: Dienst "erfrageWetter" angemeldet
ServersideTemperaturDienstProxy: instanziiert
TemperaturDienst: instanziiert
Broker: Dienst "erfrageTemperatur" angemeldet
```

```
TestBroker: Client-side Proxys erzeugen
ClientSideWetterDienstProxy: instanziiert
ClientSideTemperaturDienstProxy: instanziiert
```

```
Client: Wetter und Temperatur fuer heute anfragen:
Heute gibt es Sonnenschein bei 25.0 Grad.
```

```
Client: Wetter und Temperatur fuer morgen anfragen:
Morgen gibt es Regen bei 10.0 Grad.
```

## 5.5 Das Architekturmuster Service-Oriented Architecture

Das Architekturmuster Service-Oriented Architecture bildet die geschäftsprozessorientierte Sicht der Verarbeitungsfunktionen eines Unternehmens direkt auf die Architektur eines **verteilten Systems** ab. Dienste (Anwendungsservices) in Form von **Komponenten entsprechen Teilen eines Geschäftsprozesses**. Durch den Einsatz von Diensten entstehen die Rollen des **Serviceanbieters** und **Servicesnutzers**.<sup>91</sup>

### 5.5.1 Name/Alternative Namen

Serviceorientierte Architektur (engl. Service-Oriented Architecture, abgekürzt SOA), dienstorientierte Architektur.

### 5.5.2 Problem

Auf Grundlage der Geschäftsprozesse werden die notwendigen Anwendungsfälle eines DV-Systems bestimmt. Ein Anwendungsfall stellt hierbei eine Leistung des Systems dar, die abgerufen werden kann und die ein Ergebnis hat. Natürlich darf ein Geschäftsprozess auch nur einen einzigen Anwendungsfall enthalten, in der Regel sind es aber mehrere. Daher ist ein Anwendungsfall ein Geschäftsprozess oder Teil eines Geschäftsprozesses, der auf einem DV-System läuft.

Änderungen der Geschäftsprozesse ziehen üblicherweise erhebliche Veränderungen der DV-Systeme nach sich, da sie meist die gesamte Architektur eines Systems beeinflussen. Als Folge davon sind viele Systeme sehr schlecht wartbar. Da viele Firmen gezwungen sind, ihre Geschäftsprozesse laufend an die sich rasch ändernden Marktbedingungen anzupassen, führen die erzwungenen Änderungen zu einer Instabilität der Programme und zu einer Kostenexplosion.

### 5.5.3 Lösung

Mit einer **serviceorientierten Architektur** soll die geschäftsprozessorientierte Sicht eines Unternehmens direkt auf die Architektur eines DV-Systems abgebildet werden, d. h., dass die Anwendungsfälle aus Sicht der Verarbeitung den Komponenten des Entwurfs zugeordnet werden. Ziel ist es, dadurch eine Anwendung änderungsfreundlich zu machen. Voraussetzung einer serviceorientierten Architektur ist, dass die Geschäftsprozesse vollständig und klar definiert sind.

Eine serviceorientierte Architektur kapselt die Verarbeitungsfunktionen einer Software, die Geschäftsprozessen oder Teilen von Geschäftsprozessen entsprechen, als Dienste in Komponenten bzw. Systemteilen.



<sup>91</sup> Ein Serviceverzeichnis ist optional.

Theoretisch kann eine SOA auch bei Einrechner-Systemen eingesetzt werden, aber in der Praxis wird sie vor allem bei verteilten Systemen angewendet. Die Kommunikation des verteilten Systems wird hier nur skizziert, aber nicht explizit betrachtet.

Zentraler Begriff in einer SOA ist ein **Dienst**. Dienste ergeben sich durch die Betrachtung von Geschäftsprozessen und deren Unterstützung durch ein DV-System. Nach [Gol12] ist ein **Anwendungsfall** ein Geschäftsprozess oder ein Teil davon, der durch ein DV-System unterstützt wird. Ziel eines Anwendungsfalls ist also die Erbringung eines Dienstes, der in einem **Geschäftsprozess** genutzt werden kann. Ein solcher Dienst wird daher auch als **Anwendungsservice** oder einfach kurz als **Service** bezeichnet. Eine serviceorientierte Architektur umfasst also in sich abgeschlossene Dienste eines DV-Systems, die Teilen eines Geschäftsprozesses aus Sicht der Verarbeitung entsprechen sollen. Die technischen Funktionen, die beim Entwurf eines Anwendungsfalls zur eigentlichen Verarbeitungsfunktion dazukommen, werden hier nicht betrachtet. Ein Beispiel für eine technische Funktion ist der Start-up/Shut-down.

Ein Anwendungsservice resultiert aus der 1-zu-1-Abbildung eines Geschäftsprozesses auf die Architektur eines DV-Systems. Auf dem Rechner wird nur die Verarbeitungsfunktionalität betrachtet, nicht jedoch die technischen Funktionen. Eine **Abbildung des Problembereichs auf Komponenten des Lösungsbereichs** soll verhindern, dass Geschäftsprozesse und Anwendungsfälle leicht auseinanderdriften, wenn die Geschäftsprozesse nicht direkt in der Architektur eines Systems sichtbar gemacht werden.



Die folgende Aufzählung gibt die Anforderungen an Services wieder. Diese Anforderungen sind nach Wichtigkeit sortiert:

- **Verteilung** (engl. **distribution**)  
Ein Dienst steht in einem Netzwerk zur Verfügung.
- **Geschlossenheit** (engl. **component appearance**)  
Jeder Dienst stellt eine abgeschlossene Einheit dar, die unabhängig aufgerufen werden kann. Aus der Sicht des Servicenutzers muss jeder Service als eine in sich geschlossene Funktion erscheinen.
- **Zustandslos** (engl. **stateless**)  
Ein Service startet bei jedem Aufruf eines Servicenutzers im gleichen Zustand. Das bedeutet, dass ein Service keine Informationen von einem früheren Aufruf zu einem späteren Aufruf weitergeben kann. Er hat also "kein Gedächtnis" für frühere Aufrufe.
- **Lose Kopplung** (engl. **loosely coupled**)  
Die Services sind untereinander entkoppelt. Dies bedeutet, dass ein beliebiger Service bei Bedarf dynamisch zur Laufzeit vom Servicenutzer gesucht und aufgerufen werden kann.
- **Austauschbarkeit** (engl. **exchangeability**)  
Ein Service muss stets **austauschbar** sein. Daher muss es standardisierte Schnittstellen geben. Solange die Schnittstelle stabil bleibt, kann ein Dienst problemlos optimiert werden.



- **Ortstransparenz** (engl. **location transparency**)  
Prinzipiell ist es für den Nutzer nicht von Belang, auf welchem Rechner ein Service platziert wird.
- **Plattformunabhängigkeit** (engl. **platform independence**)  
Servicenutzer und Serviceanbieter dürfen verschiedene Rechner- und Betriebssysteme verwenden. Insbesondere darf sich auch die eingesetzte Programmiersprache unterscheiden.
- **Zugriff auf einen Dienst über eine Schnittstelle** (engl. **interface**)  
Zu jedem Dienst wird eine zugehörige Schnittstelle publiziert. Die Kenntnis dieser Schnittstelle reicht zur Nutzung des Dienstes aus. Die Implementierung bleibt verborgen (Information Hiding).
- **Verzeichnisdienst** (engl. **service directory**, **service register** oder **service broker**)  
Es erfolgt eine Registrierung der Dienste in einem Verzeichnis<sup>92</sup>.

Ein Geschäftsprozess wird also auf einen oder mehrere Anwendungsfälle abgebildet. Die Verarbeitungsfunktionen der Anwendungsfälle werden als Komponenten realisiert. Ein Anwendungsfall kann aus Teilfunktionen bestehen. Jeder Anwendungsfall kann über eine Middleware einem Service oder mehreren elementaren Services zugeordnet werden. Hierbei können elementare Services und Services auch für verschiedene Anwendungsfälle genutzt werden. Services sind in der Regel aus mehreren **elementaren Services** (engl. **basic services**) zusammengesetzt, die weniger abstrakt sind. Die elementaren Services kapseln genau eine einfache Funktion der Applikation. Ein Service darf zur Erbringung eines Dienstes auch andere Services oder elementare Services aufrufen. Ein solcher Service wird **zusammengesetzter Service** (engl. **composite service**) genannt. Diese Zusammenhänge sind im folgenden Schichtendiagramm in Bild 5-29 gezeigt:

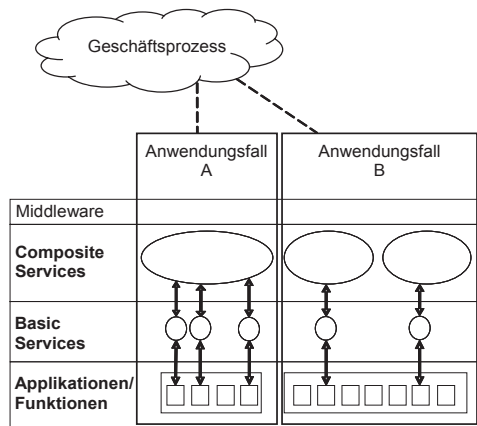


Bild 5-29 Schichtendiagramm für Services

Der sogenannte **Serviceanbieter** (engl. **service provider**) erbringt eine Dienstleistung, die von einem **Servicenutzer** (engl. **service consumer**) innerhalb oder außerhalb des anbietenden Unternehmens genutzt werden kann.<sup>93</sup> Die beiden Rollen Ser-

<sup>92</sup> Dies ist optional.

<sup>93</sup> Statt Servicenutzer ist auch der Begriff Servicekonsument gebräuchlich.

Serviceanbieter und Servicenutzer können durch Programme, ganze Geschäftsbereiche, Abteilungen, Teams oder einzelne Personen ausgeübt werden. Daneben gibt es eine dritte, optionale Rolle: häufig werden Informationen über einen Service in einem **Serviceverzeichnis** (engl. **service registry**) gespeichert, damit diese Informationen von Interessenten gesucht, gefunden und eingesetzt werden können. Das folgende Bild<sup>94</sup> visualisiert die Zusammenarbeit zwischen den erwähnten Rollen:

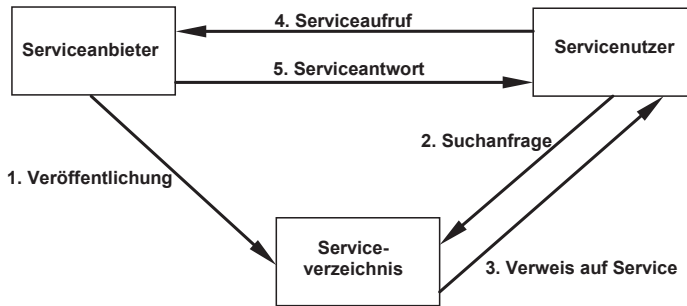


Bild 5-30 Zusammenarbeit in einer SOA

Als erstes muss ein Service vom Serviceanbieter im Serviceverzeichnis veröffentlicht werden. Dabei wird unter anderem auch eine Beschreibung des Service im Verzeichnis hinterlegt (1). Ein Servicenutzer kann Suchanfragen an das Serviceverzeichnis stellen, um einen gewünschten Service zu finden (2). Als Ergebnis der Anfrage bekommt der Servicenutzer die Adresse des Serviceanbieters, von dem der Service angeboten wird (3). Diese Adresse wird danach vom Servicenutzer genutzt, um einen direkten Serviceaufruf beim Serviceanbieter durchzuführen (4). Der Rückgabewert des Service wird anschließend zurück zum Servicenutzer übertragen (5).

### 5.5.3.1 Klassendiagramm

Die im Folgenden gezeigte Architektur ist nicht als Muster sondern als ein Beispiel zu sehen. Das Muster ist die Abbildung der Geschäftsprozesse auf Komponenten der Verarbeitungsfunktionalität der Anwendungsfälle und die Existenz der Rollen Serviceanbieter und Servicenutzer.

Eine serviceorientierte Architektur bezieht sich auf große Systeme, daher wird anstelle von einzelnen Klassen im beispielhaften Klassendiagramm ein Komponentendiagramm gezeigt, welches das Zusammenspiel der einzelnen Systemteile (Komponenten) einer beispielhaften SOA zeigt. Eine Servicekomponente kapselt einen Service oder mehrere elementare Services. Hier das bereits erwähnte Komponentendiagramm:

<sup>94</sup> Dieses Bild ist nicht UML-konform.

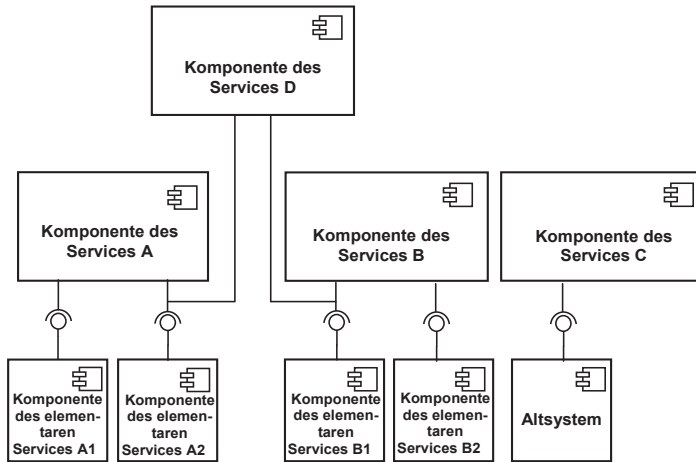


Bild 5-31 Komponentendiagramm SOA

### 5.5.3.2 Teilnehmer

#### Servicekomponente A

Zwei elementare Servicekomponenten *A1* und *A2* werden zur Servicekomponente *A* zusammengesetzt. Beispielsweise kann *A* für das Bestellen von Waren zuständig sein, wobei *A1* das Bezahlen und *A2* den Versand übernimmt.

#### Servicekomponente B

Diese Servicekomponente beinhaltet ebenfalls zwei elementare Komponenten, nämlich die elementaren Servicekomponenten *B1* und *B2*.

#### Servicekomponente C

Die Servicekomponente *C* kapselt in diesem Beispiel ein vorhandenes Altsystem (engl. legacy system), um es für Serviceaufrufe zugänglich zu machen.

#### Servicekomponente D

Mit der Servicekomponente *D* wird gezeigt, dass elementare Servicekomponenten – in diesem Falle *A2* und *B1* – mehrfach verwendet werden können.

In diesem Beispiel sind also *A*, *B* und *D* zusammengesetzte Servicekomponenten (composite services), während die Komponenten *A1*, *A2*, *B1* und *B2* elementare Servicekomponenten (basic services) darstellen.

### 5.5.3.3 Dynamisches Verhalten

Das folgende Bild zeigt ein beispielhaftes Zusammenspiel zwischen Servicenutzer und Servicekomponenten. Es beschränkt sich auf die Servicekomponente *A*.

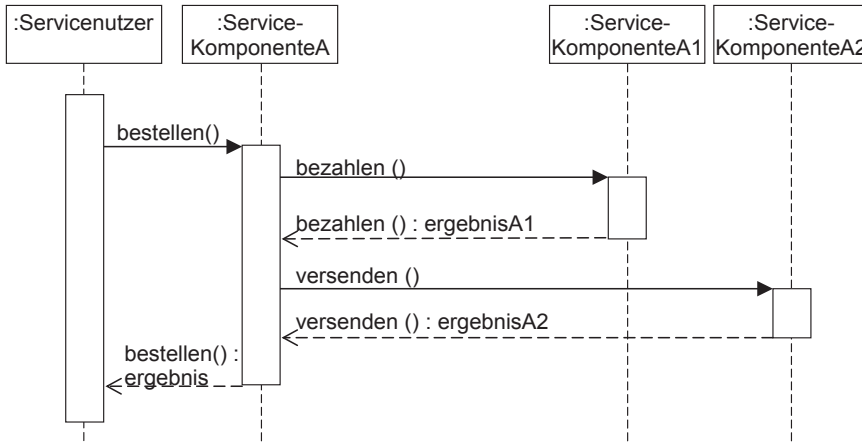


Bild 5-32 Beispielhaftes Sequenzdiagramm einer SOA

## 5.5.4 Bewertung

SOA ist ein Konzept für eine Architektur – also den Lösungsbereich – und ist unabhängig von der entsprechenden Lösungstechnologie. Im Problembereich leistet dieses Konzept keine Hilfe. Eine SOA ist prinzipiell nichts Neues, sondern bringt nur ein Architekturkonzept auf den Punkt. Bereits mit der Middleware CORBA konnte eine SOA implementiert werden, wenn das Ziel verfolgt wurde, Geschäftsprozesse auf Komponenten abzubilden.

### 5.5.4.1 Vorteile

Die folgende Liste zeigt die Vorteile einer SOA auf:

- Die Abbildung von Geschäftsprozessen auf Dienste in Komponenten schafft eine Übersicht über alle benötigten Dienste und deren Schnittstellen.
- Die Komplexität von verteilten Systemen wird reduziert durch die Aufteilung in Komponenten von Services bzw. elementaren Services.
- Services und elementare Services können mehrfach eingesetzt, d. h. wiederverwendet werden.
- Solange sich die Schnittstelle eines Service nicht ändert, kann die dahinter liegende Implementierung dieses Service dynamisch ausgetauscht werden.

### 5.5.4.2 Nachteile

Folgende Nachteile werden gesehen:

- Eine zu feine Granularität der Services erzeugt komplexe Strukturen.
- Es entsteht ein Mehraufwand durch die Kommunikation über mehrere Schichten hinweg.

- Die üblicherweise eingesetzten Protokolle stellen hohe Anforderungen an den Durchsatz der Netzwerkverbindungen.
- Eine SOA kann nur dann effektiv eingesetzt werden, wenn die Geschäftsprozesse klar definiert und dokumentiert sind.

### 5.5.5 Einsatzgebiete

Im Allgemeinen ist der Einsatz einer SOA bei allen Client/Server-Systemen sinnvoll. Besonders komplexe Systeme können durch eine SOA vereinfacht werden. Bestehende Altsysteme können in einer SOA auf einfache Art und Weise durch Kapselung plattformunabhängig zugänglich gemacht werden. Kunden bzw. Lieferanten einer Firma können über eine SOA in die Geschäftsprozesse der Firma eingebunden werden.

### 5.5.6 Ähnliche Muster

Eine SOA sorgt für die Entkopplung von Client und Server. Dies ist auch das Ziel des **Broker-Architekturmusters**. Während beim Broker der physische Ort der Leistungserbringung verborgen wird, ist er bei einer SOA prinzipiell bekannt und z. B. in der Web Service-Beschreibung vermerkt. Sowohl bei einer SOA als auch beim Broker wird eine Zwischenschicht für die Kommunikation eingezogen. Im Gegensatz zum Broker wird bei einer SOA die Abbildung von Anwendungsfällen aus Verarbeitungssicht auf Komponenten berücksichtigt.

### 5.5.7 Realisierung einer SOA

Die Architektur-Vorstellung von SOA kann unterschiedlich interpretiert und implementiert werden. Zur Realisierung einer SOA gibt es beispielsweise die folgenden Technologien:

- XML-basierte Web Services,
- RESTful (**R**epresentational **S**tate **T**ransfer) Web Services,
- CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture) sowie
- OSGi<sup>95</sup>.

Aus Platzgründen beschränkt sich dieses Kapitel ausschließlich auf die Technologien XML-basierte Web Services und RESTful Web Services. Auf die Realisierung einer SOA mit diesen Technologien in Java wird im Folgenden näher eingegangen. Bild 5-33 zeigt die für die einzelnen Schritte konkret eingesetzten Protokolle für XML-basierte Web Services.

---

<sup>95</sup> Die OSGi Serviceplattform wurde von der gemeinnützigen OSGi Alliance entwickelt. Früher stand OSGi für Open Services Gateway initiative, die Vorgängerorganisation der OSGi Alliance. Heute ist OSGi ein geschützter Markenname der OSGi Alliance.

### 5.5.7.1 JAX-WS (JavaAPI for XML Web Services)

Dieses Kapitel erläutert den Java-Standard XML Web Services (kurz JAX-WS) und zeigt den Einsatz dieses Standards anhand eines minimalen Implementierungsbeispiels.

Der Java-Standard **JAX-WS**<sup>96</sup> erlaubt das einfache Erstellen und Benutzen von XML-Web Services mithilfe von Annotationen. Die Beschreibung eines Web Service mittels der Web Services Description Language (**WSDL**) wird bei der Verwendung von JAX-WS zum Großteil automatisch generiert.

Die Kommunikation zwischen Serviceanbieter und Servicenutzer erfolgt im Falle von XML-basierten Web Services über das SOA-Protokoll (**SOAP**<sup>97</sup>). Ein Service kann durch die Registrierung in einem Verzeichnis veröffentlicht, anschließend dort gesucht und gefunden werden. Häufig wird hierfür Universal Discovery, Description, Integration (**UDDI**) als Protokoll eingesetzt. Diese Technologien werden im Folgenden kurz vorgestellt. Hier eine Visualisierung der typischen Protokollabläufe eines Web Service:

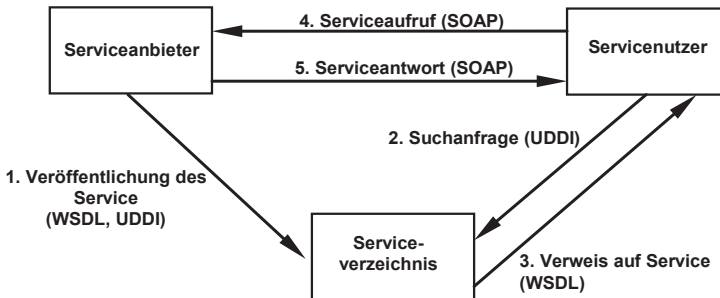


Bild 5-33 Protokollabläufe eines XML-basierten Web Service

JAX-WS ist eine Technologie zum Erstellen und Anwenden von Web Services. Damit ein Web Service angesprochen werden kann, muss ein Zugangspunkt (engl. Service-Endpoint-Interface, kurz SEI) definiert werden. In diesem Interface werden alle Methoden genannt, die von einem Servicenutzer aufgerufen werden können.

Das folgende Sequenzdiagramm zeigt einen beispielhaften Ablauf der Kommunikation zwischen Serviceanbieter und Servicenutzer:

<sup>96</sup> Java API for XML Web Services.

<sup>97</sup> Die Abkürzung SOAP stand ursprünglich für Simple Object Access Protocol.

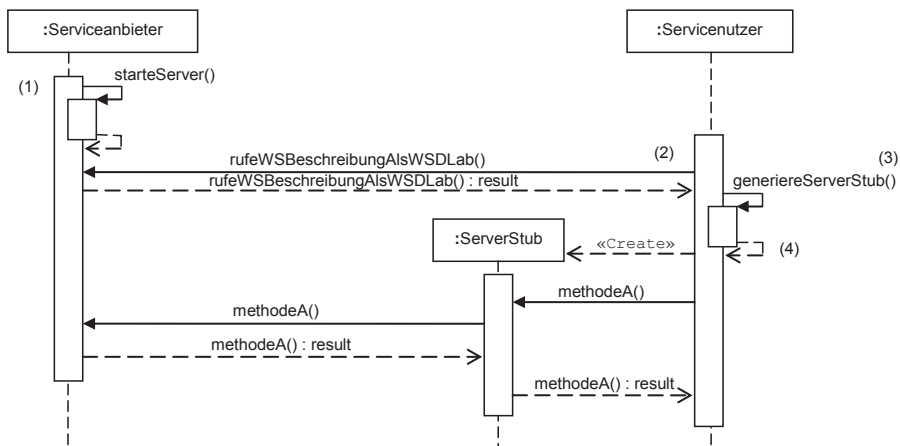


Bild 5-34 JAX-WS Sequenzdiagramm

Zuerst muss seitens des Serviceanbieters ein Web Service-Zugangspunkt definiert und der Serviceanbieter gestartet werden (1). Anschließend kann ein Servicenutzer die WSDL-Beschreibung eines Web Service vom Serviceanbieter abrufen (2). Die WSDL-Beschreibung kann vom Servicenutzer dazu verwendet werden, um Klassen, die die Kommunikation mit dem Serviceanbieter abstrahieren, automatisch zu generieren (sogenannte Stub-Klassen). Dieser Generierungsschritt muss nur einmal vor der ersten Nutzung des gewünschten Service durchgeführt werden (3). Die Anwendung eines Servicenutzers kann sich mit Hilfe der generierten Klassen leicht (es muss nur eine Klasse instanziiert werden) mit einem Serviceanbieter verbinden, ohne sich mit näheren Details zur Kommunikation beschäftigen zu müssen (4).<sup>98</sup>

In JAX-WS markiert die Annotation `@WebService` ein Interface oder eine Klasse. Eine so gekennzeichnete Klasse bzw. ein so gekennzeichnetes Interface stellt den Zugangspunkt für Aufrufe des Servicenutzers dar. In der Standardeinstellung sind anschließend alle Methoden des Interface bzw. der Klasse als öffentlich aufrufbar markiert. Mit den optionalen Argumenten `serviceName` und `targetNamespace` können aussagekräftige Namen gewählt werden, die in der Beschreibung des Service in der WSDL-Datei genannt werden und hilfreich sind, wenn ein Servicenutzer basierend auf dieser Beschreibung automatisiert Java-Klassen generiert. Das folgende Beispiel<sup>99</sup> zeigt eine Klasse als Web Service-Zugangspunkt:

```
// Datei Serviceanbieter.java
@WebService (targetNamespace="serviceanbieter")
public class Serviceanbieter
{
    @WebMethod
    public void methodeA (String para){
        System.out.println ("Server: Methode A wurde aufgerufen");
    }
}
```

<sup>98</sup> Instanziiert der Servicenutzer die ServerStub-Klassen, wird im Hintergrund eine Verbindung zum Serviceanbieter aufgebaut.

<sup>99</sup> In diesem Programmbeispielen wurde aus Gründen der Übersichtlichkeit auf die benötigten "import"-Anweisungen verzichtet. Der vollständige Quellcode ist über den begleitenden Webauftritt zugänglich.

```

        System.out.println ("Parameter: " + para);
    }

    @WebMethod
    public String methodeB(){
        System.out.println ("Server: Methode B wurde aufgerufen");
        return "Nachricht von Server an Client";
    }
}

```

Durch die optionale Annotation `@WebMethod` kann eine Methode als öffentlich aufrufbar markiert werden, aber nur wenn die entsprechende Schnittstelle bzw. Klasse bereits die Annotation `@WebService` trägt. Im Hintergrund werden für jede mit `@WebMethod` annotierte Methode zwei Klassen generiert, eine für die Anfrage vom Service-nutzer an den Serviceanbieter und eine für die Antwort vom Serviceanbieter an den Servicenutzer.

Jeder Übergabeparameter einer öffentlich zugänglichen Methode kann optional mit der Annotation `@WebParam(name="NeuerName")` versehen werden. Durch den Parameter `name` kann ein neuer Name für diesen Parameter vergeben werden.

Mithilfe der optionalen Annotation `@WebResult(name="EinName")` kann der Rückgabewert einer öffentlich zugänglichen Methode mit einem Namen versehen werden. Der Name "EinName" wird dann in der WSDL-Beschreibung des Services verwendet.

Die optionale Annotation `@SOAPBinding` bietet einige Detaileinstellungen zu den SOAP-Nachrichten, die zwischen Servicenutzer und Serviceanbieter ausgetauscht werden. Hier kann der Aufbau der SOAP-Nachrichten (durch den Parameter `parameterStyle`), die Kodierung (durch den Parameter `style`) und die Formatierung der Nachrichten (durch den Parameter `use`) beeinflusst werden.

Um den Serviceanbieter zu starten, genügt eine zweite Klasse mit wenigen Zeilen Quelltext, wie dieses Beispiel zeigt:

```

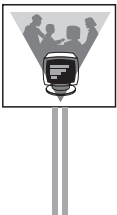
// Datei: ServiceanbieterMain.java
public class ServiceanbieterMain
{
    public static void main (String[] args) {
        String url = "http://localhost:8080/Serviceanbieter";
        Serviceanbieter server = new Serviceanbieter();
        try {
            Endpoint.publish (url,server);
            System.out.println ("Serviceanbieter ist nun aktiv "
                + "und wartet auf Anfragen...");
        } catch (Exception e){
            System.out.println ("Fehler beim Starten des Web Service.");
            e.printStackTrace();
        }
    }
}

```



Anschließend ist die WSDL-Beschreibung unter der folgenden Adresse aufrufbar: `http://localhost:8080/Serviceanbieter?wsdl`. Soll nun ein Servicenutzer zu diesem Web Service implementiert werden, kann diese Beschreibung seitens des Servicenutzers verwendet werden, um Java-Klassen zu generieren. Hierfür steht das Kommandozeilenprogramm `wsimport` bereit. Nach dem Ausführen des Kommandos `wsimport -s src -keep -d bin http://localhost:8080/Serviceanbieter?wsdl`<sup>100</sup> wurde u. a. eine Klasse `ServiceanbieterService` erzeugt, die nach dem Entwurfsmuster **Fabrikmethode** (siehe Kapitel 4.18) eine Instanz der Klasse `Serviceanbieter` zurückgibt. Hier nun das Programm, das den Web Service aufruft:

```
// Datei: Servicenutzer.java
public class Servicenutzer
{
    public static void main (String[] args) {
        ServiceanbieterService s = new ServiceanbieterServiceLocator();
        Serviceanbieter anbieter;
        try {
            anbieter = s.getServiceanbieterPort();
            anbieter.methodeA ("Nachricht an Serviceanbieter");
            anbieter.methodeB();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Hier das Protokoll des Programmlaufs des Serviceanbieters:

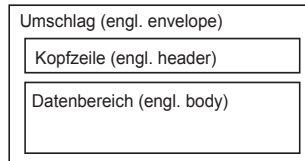
```
Serviceanbieter ist nun aktiv und wartet auf
Anfragen...
Server: Methode A wurde aufgerufen
Parameter: Nachricht an Serviceanbieter
Server: Methode B wurde aufgerufen
```

### 5.5.7.2 SOA-Protokoll – SOAP 1.2

SOAP wird vollständig in XML verfasst und wurde vom W3C spezifiziert [W3SOAP]. Der Aufbau einer SOAP-Nachricht wird dabei in drei Bereiche eingeteilt. Der sogenannte **Umschlag** oder **Rahmen** (engl. **envelope**) umhüllt die anderen beiden Bereiche der Nachricht. Diese Kapselung ist in Bild 5-35 dargestellt. Die **Kopfzeile** (engl. **header**) ist optional. Der Inhalt der Kopfzeile ist nicht genau festgelegt. SOAP bietet hier lediglich einen Bereich zur Übertragung von Metadaten an. Üblicherweise wird die Kopfzeile für die Übertragung von Sicherheitsinformationen genutzt. Nach der Kopfzeile folgt der **Datenbereich** (engl. **body**) der SOAP-Nachricht. In diesem Bereich be-

<sup>100</sup> Der Parameter `-d` gibt das Verzeichnis an, in dem die kompilierten `.class`-Dateien gespeichert werden sollen. Falls die Entwicklungsumgebung Eclipse verwendet wird, ist dies das Verzeichnis `bin`, im Falle von Netbeans ist es das Verzeichnis `build`.

finden sich die zu übertragenden Informationen. Eine SOAP-Nachricht muss als wohlgeformtes XML-Dokument formuliert sein, unter Ausschluss des sonst notwendigen XML-Prologs. Der Datenbereich wird auch zum Melden von Fehlern genutzt. Hierfür stehen in SOAP spezifizierte Elemente zur Verfügung. Das folgende Bild zeigt den Aufbau einer SOAP-Nachricht:



*Bild 5-35 Aufbau einer SOAP-Nachricht*

Im Folgenden wird der Quellcode einer SOAP-Nachricht gezeigt. Dieses Beispiel soll dem interessierten Leser einen ersten Eindruck über den Aufbau von SOAP vermitteln und zum grundlegenden Verständnis beitragen. Die Nachricht zeigt beide Bereiche: Kopfzeile und Datenbereich. In beiden Bereichen sind beispielhafte Informationen platziert.

#### Beispiel einer SOAP-Nachricht (für XML-Kenner):

```

<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <login:sec xmlns:login="http://firma.de"
      soap:mustUnderstand="true">
      <!-- Weitere Angaben wie Benutzername und Passwort ... -->
    </login:sec>
  </soap:Header>
  <soap:Body>
    <soap:message> Inhalt des Datenbereichs </soap:message>
  </soap:Body>
</soap:Envelope>
  
```

### 5.5.7.3 Web Services Description Language – WSDL 2.0

WSDL wird in XML verfasst und wurde vom W3C spezifiziert [W3WSDL]. Die Aufgabe von WSDL besteht darin, einen Web Service möglichst vollständig zu beschreiben. Dies erfolgt durch die Beschreibung der Nachrichten, die vom Web Service empfangen und gesendet werden. Dabei erfolgt die Beschreibung unabhängig vom verwendeten Transportprotokoll.

In WSDL wird die Beschreibung auf zwei Ebenen durchgeführt: abstrakt und konkret. Abstrakte Informationen beziehen sich auf die Funktionalität des Web Service. Folgende abstrakte Elemente stehen in der WSDL zur Beschreibung eines Service zur Verfügung<sup>101</sup>:

1. **Documentation:** Die Beschreibung der Funktionalität des Web Services in Prosa-text.

<sup>101</sup> Hinweis: Die fettgedruckten Begriffe spiegeln die Bezeichner der entsprechenden Tags in der WSDL-Beschreibung wieder.

2. **Types:** Eine Menge von Datentypen, die zum Austausch der dazugehörigen Nachrichten benötigt werden. Die Definition der Datentypen erfolgt in der Regel mit einem XML-Schema<sup>102</sup>. Ein- und Ausgabeparameter der einzelnen Web Servicemethoden können diese Datentypen annehmen.
3. **Message:** Zulässige Nachrichten, die zwischen Servicenutzer und Serviceanbieter ausgetauscht werden.
4. **Interface:** Beschreibung der Schnittstelle des Service. Die Schnittstelle ist die Menge aller Methoden des Serviceanbieters, die vom Servicenutzer aufgerufen werden können. Zu jeder Schnittstelle werden Ein- und Ausgabeparameter mit dem zugehörigen Datentyp hinterlegt. Zusätzlich wird jedem Parameter ein Message Exchange Pattern (MEP) zugeordnet, das die Richtung des Informationsflusses beschreibt (u. a. Eingabe oder Ausgabe). Außerdem kann bei der Schnittstellenbeschreibung ein Rückgabewert für Fehlerfälle hinterlegt werden.

Konkrete Komponenten der WSDL-Beschreibung beinhalten Informationen, die sich auf die technischen Details beziehen:

5. **Binding:** Definiert das Protokoll, das für den Nachrichtenaustausch verwendet wird. Im Allgemeinen wird hierfür SOAP verwendet.
6. **Service:** Name des Service sowie eine Menge von Zugangspunkten (URI), über die der Service aufgerufen werden kann.

Eine weitere wichtige Eigenschaft von WSDL ist, dass sich WSDL-Beschreibungen modularisieren lassen. Das bedeutet, dass sich eine umfangreiche WSDL-Beschreibung in mehrere Dateien aufteilen lässt. Häufig genutzte Teilbereiche können so einfach wiederverwendet werden. Hierzu stehen die WSDL-Elemente **include** oder **import** zur Verfügung [W3WSDL].

### Beispiel Quelltext für eine WSDL-Beschreibung:

Es folgt ein Beispiel zu einer WSDL-Beschreibung. Dabei handelt sich um einen unvollständigen Auszug der Beschreibung der in Kapitel 5.5.7.1 gezeigten Web Services. Hier nun das WSDL-Dokument:

```
<?xml version="1.0"?>
<definitions name="ServiceanbieterService"
  targetNamespace="serviceanbieter">

  <documentation> Beschreibung des Services in textueller Form.
</documentation>

  <!-- Datentypen definieren -->
  <types>
    <xs:complexType name="methodeA">
      <xs:sequence>
        <xs:element name="arg0" type="xs:string" minOccurs="0">
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </types>
```

<sup>102</sup> Standard spezifiziert vom W3C [W3XMLS].

```

...
<!-- Zulaessige Nachrichten aufstellen -->
<message name="methodeA">
  <part name="parameters" element="tns:methodeA" />
</message>
...
<!-- Schnittstellen des Services definieren -->
<interface name="SchnittstelleMethodeA">
  <operation name="MethodeA">
    <input messageLabel="In" message="tns:methodeA"/>
  </operation>
</interface>
...
<!-- Protokoll festlegen -->
<binding name="ProtokollMethodeA"
  interface="tns:SchnittstelleMethodeA"
  type="http://www.w3.org/2004/08/wsdl/soap12"
  protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
  <operation ref="tns:MethodeA" />
</binding>
...
<!-- Zugangspunkte benennen -->
<service name="ServiceanbieterService"
  interface="tns:SchnittstelleMethodeA">
  <endpoint name="ZugangspunktMethodeA"
    binding="tns:ProtokollMethodeA"
    address="http://localhost:8080/Serviceanbieter">
  </service>
...
</definitions>

```

#### 5.5.7.4 Universal Discovery, Description, Integration – UDDI

UDDI wurde vom Industriekonsortium OASIS<sup>103</sup> spezifiziert [OASUDD]. Die federführenden Firmen hinter dem Standard sind IBM, Microsoft und SAP. Ziel von UDDI ist es, ein Verzeichnis zu schaffen, das eine durchsuchbare Übersicht über eine (große) Menge von Web Services bietet. Langfristig sollen mit Hilfe von UDDI die Dienste einer beliebigen Firma gefunden werden. Ein UDDI-Verzeichnis kann mit der Funktionsweise von Telefonbüchern verglichen werden. Der Zugriff auf UDDI-Verzeichnisse kann sowohl durch menschliche Benutzer als auch durch Applikationen erfolgen. Der Inhalt eines UDDI-Verzeichnisses wird in vier Haupttabellen untergliedert:

- **White Pages:** Enthalten Informationen über die Unternehmen, die einen Web Service ins Verzeichnis einstellen. Der Servicenutzer kann so ein passendes Unternehmen durch die Eingabe eines Namens finden.
- **Yellow Pages:** Der Servicenutzer kann das gesuchte Unternehmen anhand einer kategorisierten Struktur finden, vergleichbar mit einem Branchenverzeichnis.
- **Green Pages:** Hier stehen WSDL-Beschreibungen und technische Details zum Web Service.

<sup>103</sup> Organization for the Advancement of Structured Information Standards.

- **Service Type Registration:** Bietet eine Tabelle der Servicebeschreibungen in Prosa-Text für die maschinelle Nutzung an.

Aus der technischen Sicht besteht ein UDDI-Verzeichnis im Wesentlichen aus zwei Komponenten:

- UDDI-API, die verschiedene Web Service-Methoden bereitstellt, die zum Suchen, Veröffentlichen und Verwalten eines Service im UDDI-Verzeichnis genutzt werden kann. Die Kommunikation erfolgt über SOAP-Nachrichten.
- Ein standardisiertes XML-Schema, welches das intern für UDDI genutzte Datenmodell beschreibt.

Als Alternative zu UDDI kann beispielsweise **die Web Services Inspection Language (WSIL)** [IBMWSI] eingesetzt werden. Bei WSIL wird der Ansatz von mehreren kleinen und dezentralen Verzeichnissen verfolgt.

Prinzipiell ist für die Realisierung eines Web Service nicht zwingend ein Verzeichnis nötig, es ist optional. Die notwendigen Informationen wie etwa die WSDL-Beschreibung können zwischen den Partnern auch auf andere Weise kommuniziert werden. Die Relevanz von Technologien wie UDDI ist deswegen nicht mehr so groß wie zu Beginn der Entwicklung. Globale UDDI-Verzeichnisse gibt es daher praktisch nicht (mehr). Hingegen werden UDDI-Verzeichnisse lokal in vielen Firmen eingesetzt, um damit die jeweiligen Entwicklerteams zu unterstützen und die Wiederverwendung von bereits existierenden Web Services zu fördern.

#### 5.5.7.5 Implementierung eines SOA-Beispiels mit Web Services in JAX-WS

In diesem Kapitel wird der Einsatz von JAX-WS anhand der Implementierung eines SOA-Beispiels verdeutlicht.<sup>104</sup> Hierzu soll eine Autovermietung als Beispiel betrachtet werden. Der wichtigste Geschäftsprozess im Unternehmen beschreibt den Ablauf, wie Fahrzeuge vermietet werden. Hierzu sind die folgenden Schritte notwendig:

1. Der Kunde lässt sich eine Übersicht der aktuell verfügbaren Fahrzeuge auf der Webseite des Unternehmens für einen Standort anzeigen. Angezeigt werden verschiedene Fahrzeugklassen wie z. B. Mittelklasse oder Kleinwagen. Auf eine Datumsangabe wird in diesem Beispiel verzichtet.
2. Der Kunde reserviert ein gewünschtes Fahrzeug für seinen Standort.
3. Das Fahrzeug wird vom Kunden abgeholt und genutzt.
4. Später bringt der Kunde das Fahrzeug wieder zu einem Standort des Unternehmens zurück und bezahlt es mit seiner Kreditkarte.

Im Unternehmen soll eine SOA realisiert werden, daher muss der Anwendungsfall "Fahrzeug vermieten" möglichst genau den oben beschriebenen Geschäftsprozess abbilden. Das folgende Bild zeigt ein Komponentendiagramm des Beispiels:

---

<sup>104</sup> In diesem Programmbeispiel wurde aus Gründen der Übersichtlichkeit auf die benötigten "import"-Anweisungen verzichtet. Der vollständige Quellcode ist über den begleitenden Webaufruf zugänglich.

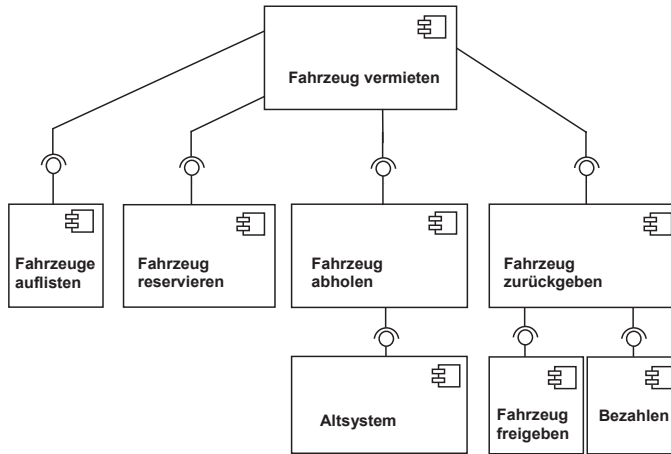


Bild 5-36 Komponentendiagramm der SOA-Beispielimplementierung

Aus Platzgründen werden die angebotenen Dienste in diesem Beispiel in eine einzige Schnittstelle zusammengefasst und in einer einzigen Komponente implementiert. Bei einer realen Implementierung würden die Dienste mit verschiedenen Schnittstellen getrennt definiert und implementiert.

Der Server besteht aus einer Schnittstelle (`IAutovermietung.java`), einer Implementierung der Geschäftslogik (`ServerImpl.java`) sowie einem Hauptprogramm (`ServerStartup.java`), das den Server über das Netzwerk aufrufbar macht. Hier zunächst die Schnittstelle, die die Dienste beschreibt, die zur Verfügung gestellt werden:<sup>105</sup>

```
// Datei: IAutovermietung.java
package autovermietung.server;

@WebService (targetNamespace="autovermietung.server")
public interface IAutovermietung{

    @WebResult (name="freieFahrzeuge")
    public String[] fahrzeugeAuflisten(
        @WebParam (name="standort") String standort);

    @WebResult (name="fahrzeugID")
    public String fahrzeugReservieren(
        @WebParam (name="standort") String standort,
        @WebParam (name="fahrzeugklasse") String fahrzeugklasse);

    @WebResult (name="abholungErfolgreich")
    public boolean fahrzeugAbholen(
        @WebParam (name="fahrzeugID") String fahrzeugID);
}
```

<sup>105</sup> Die Annotationen "WebResult", "WebService" und "WebParam" werden auf den Seiten 363 und 364 erklärt.

```

    @WebResult (name="rueckgabeErfolgreich")
    public boolean fahrzeugZurueckgeben (
        @WebParam (name="fahrzeugID") String fahrzeugID,
        @WebParam (name="creditcardNumber") String creditCardNumber);
}

```

Es folgt die Implementierung der Geschäftslogik des Servers. Hier werden z. B. freie Fahrzeuge ermittelt, Fahrzeuge reserviert oder die Preisberechnung implementiert:

```

// Datei: ServerImpl.java
package autovermietung.server;

@WebService (
    serviceName="AutovermietungService",
    targetNamespace="autovermietung.server",
    portName="FahrzeugMieten",
    endpointInterface="autovermietung.server.IAutovermietung")
public class ServerImpl implements IAutovermietung{

    // Diese Methode stellt Servicekomponente "Fahrzeuge auflisten"
    // dar. Alle verfuegbaren Fahrzeugklassen fuer einen Standort
    // auflisten
    public String[] fahrzeugeAuflisten (String standort) {

        String fahrzeuge[] = new String[] { "Oberklasse",
            "Mittelklasse", "Kleinwagen" };

        // Immer drei Fahrzeugklassen verfuegbar
        return fahrzeuge;
    }

    // Diese Methode stellt Servicekomponente "Fahrzeug reservieren"
    // dar. Fahrzeug an einem Standort reservieren
    public String fahrzeugReservieren (String standort,
        String fahrzeugklasse) {
        // Freies Fahrzeug ermitteln
        String beispielFahrzeug = "FZ_ID_4711";

        // Fahrzeug als belegt markieren
        System.out.println ("Das Fahrzeug mit der Kennung "
            + beispielFahrzeug + " wurde als belegt markiert.");

        // Fahrzeugkennung zurueckgeben
        return beispielFahrzeug;
    }

    // Diese Methode stellt die Servicekomponente "Fahrzeug abholen"
    // dar. Fahrzeug abholen
    public boolean fahrzeugAbholen (String fahrzeugID) {
        System.out.println ("Fahrzeug mit der Kennung "
            + fahrzeugID + " wurde abgeholt.");
    }
}

```

```

// Beispiel: Befehl an ein Altsystem uebergeben
Runtime.getRuntime().exec ("legacySystem.exe ...");
return true;
}

// Diese Methode stellt die Servicekomponente "Fahrzeug zurück-
// geben" dar. Fahrzeug zurueckgeben, freigeben und bezahlen
public boolean fahrzeugZurueckgeben (String fahrzeugID,
    String kreditkartenNr) {
    System.out.println ("Fahrzeug mit der Kennung " + fahrzeugID
        + " wurde zurueckgebracht und als frei markiert.");

    if (bezahlen (fahrzeugID, kreditkartenNr)){
        return true;
    } else {
        return false;
    }
}

// Fahrzeug bezahlen
private boolean bezahlen (String fahrzeugID,
    String kreditkartenNr) {
    // Preis bestimmen und Bezahlung durchfuehren
    System.out.println ("Bezahlung fuer das Fahrzeug mit der" +
        " Kennung " + fahrzeugID + " war erfolgreich.");
    return true;
}
}

```

Es folgt das Hauptprogramm, das den Server startet, den Dienst unter einer Adresse `http://localhost:8080/AutovermietungServer` im Netzwerk ansprechbar macht, sowie auf eingehende Anfragen von Servicenutzern wartet. Hier der Quelltext des Hauptprogramms:

```

// Datei: ServerStartup.java
package autovermietung.server;

// Hauptprogramm, das den Web Service im Netz ansprechbar macht
public class ServerStartup {
    public static void main (String[] args) {
        String url = "http://localhost:8080/AutovermietungServer";
        IAutovermietung server = new ServerImpl();
        Endpoint.publish (url,server);
        System.out.println ("Der Web Service ist nun im Netzwerk unter "
            + "der Adresse " + url + " aufrufbar.");
        System.out.println ("Serviceanbieter lauscht jetzt...");
    }
}

```

Nachdem der Server gestartet ist, kann eine Selbstbeschreibung der Schnittstelle des Dienstes unter der Adresse `http://localhost:8080/AutovermietungServer-`



?wsdl im WSDL-Format abgerufen werden. Nun soll ein Servicenutzer für diese Schnittstelle in einem neuen Projekt implementiert werden. Damit ein Servicenutzer mit dem Serviceanbieter kommunizieren kann, sind auf Seite des Servicenutzers Kommunikationsklassen erforderlich (Server-Stub). Diese Klassen können beispielsweise mit dem Kommandozeilenprogramm `wsimport`<sup>106</sup> oder direkt mit der Eclipse-Oberfläche<sup>107</sup> aus der WSDL-Beschreibung generiert werden. Anschließend existieren mehrere Dateien im Ordner `server.autovermietung`. Die folgende Implementierung zeigt, wie ein Servicenutzer die generierten Klassen nutzt:

```
// Datei: Servicenutzer.java
package autovermietung.servicenutzer;

public class Servicenutzer {

    @WebServiceRef (wsdlLocation =
        "http://localhost:8080/AutovermietungServer?wsdl")
    static IAutovermietung service;
    // Hauptprogramm Servicenutzer
    public static void main (String[] args) {
        System.out.println ("Servicenutzer gestartet.");
        try {
            // Zugangspunkt zum Server holen
            service = new AutovermietungService().getFahrzeugMieten();

            // Servicenutzer erzeugen und ein Fahrzeug mieten
            Servicenutzer client = new Servicenutzer();
            client.fahrzeugMieten();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println ("Servicenutzer beendet.");
    }

    //Servicenutzer mietet ein Fahrzeug
    public void fahrzeugMieten() {
        String reservierungsNr;
        String standort = "Stuttgart";
        String kreditkartenNr = "abc123456";

        try {
            // Alle verfuegbaren Fahrzeuge auflisten
            List<String> fahrzeuge =
                service.fahrzeugeAuflisten (standort);
            System.out.println ("Freie Fahrzeuge in " + standort + ":");
            for (int i = 0; i < fahrzeuge.size(); i++) {
                System.out.println ("-> " + fahrzeuge.get (i));
            }

            // Ein Fahrzeug reservieren
            System.out.println ("Reserviere einen Kleinwagen...");
        }
    }
}
```

<sup>106</sup> `wsimport -s src -d bin -keep http://localhost:8080/AutovermietungServer?wsdl`  
im Projektverzeichnis des Servicenutzers ausführen

<sup>107</sup> Im Menü Neu -> Andere -> Web Service Client.

```

        reservierungsNr = service.fahrzeugReservieren (
            standort, "Kleinwagen");
        System.out.println ("Reserviertes Fahrzeug hat "
            + "die Reservierungsnr. " + reservierungsNr + ".");

        // Fahrzeug abholen
        service.fahrzeugAbholen (reservierungsNr);
        System.out.println ("Fahrzeug abgeholt.");

        // Fahrzeug nutzen...

        // Spaeter Fahrzeug zurueckgeben
        service.fahrzeugZurueckgeben (reservierungsNr,
            kreditkartenNr);
        System.out.println ("Fahrzeug zurueckgegeben.");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```



#### Hier das Protokoll des Serviceanbieters (Server):

Der Web Service ist nun im Netzwerk unter der Adresse <http://localhost:8080/AutovermietungServer> aufrufbar.  
 Serviceanbieter lauscht jetzt...  
 Das Fahrzeug mit der Kennung FZ\_ID\_4711 wurde als belegt markiert.  
 Fahrzeug mit der Kennung FZ\_ID\_4711 wurde abgeholt.  
 Fahrzeug mit der Kennung FZ\_ID\_4711 wurde zurueckgebracht und als frei markiert.  
 Bezahlung fuer das Fahrzeug mit der Kennung FZ\_ID\_4711 war erfolgreich.



#### Hier das Protokoll des Servicenutzers (Client):

Servicenutzer gestartet  
 Freie Fahrzeuge in Stuttgart:  
 -> Oberklasse  
 -> Mittelklasse  
 -> Kleinwagen  
 Reserviere einen Kleinwagen...  
 Reserviertes Fahrzeug hat die Reservierungsnr. FZ\_ID\_4711.  
 Fahrzeug abgeholt.  
 Fahrzeug zurueckgeben.  
 Servicenutzer beendet.

### 5.5.7.6 JAX-RS (Java API for RESTful Web Services)

Dieses Kapitel erklärt den Web-Service Standard REST und zeigt die allgemeine Verwendung von REST an einem minimalen Implementierungsbeispiel.

Der Begriff REST (engl. **R**epresentational **S**tate **T**ransfer) stammt von Roy Fielding aus dem Jahr 2000 [Fie00]. In dieser Publikation schlug Roy Fielding vor, das hauptsächlich für die Übertragung von Webseiten eingesetzte HTTP (Hyper Text Transfer Protocol) für den Aufbau einer servicebasierten Architektur zu nutzen.

Der Java-Standard **JAX-RS** bietet Unterstützung beim Umsetzen von REST-Diensten in Java. Im Unterschied zu dem bereits oben vorgestellten JAX-WS handelt es sich bei JAX-RS aber nicht um Web Services auf Basis von XML bzw. SOAP sondern um Web Services, die direkt auf HTTP aufsetzen. Eine Referenzimplementierung ist als Open-Source-Projekt unter dem Namen Jersey<sup>108</sup> zu finden.

HTTP stellt verschiedene Anfragemethoden zur Verfügung. In einem Browser wird beispielsweise beim Aufruf einer Webseite die Methode HTTP-GET verwendet. Im Rahmen von REST werden den folgenden Anfragemethoden bestimmte Verwendungszwecke zugeordnet:

- **GET**: Eine Ressource wird vom Server angefordert bzw. gelesen.
- **POST**: Eine neue Ressource wird auf dem Server erstellt.
- **PUT**: Eine auf dem Server vorhandene Ressource wird aktualisiert.
- **DELETE**: Eine auf dem Server vorhandene Ressource wird gelöscht.

Eine Ressource ist eine adressierbare Information, z. B. ein Benutzerkonto.

Die folgende Klasse zeigt einen Serviceanbieter, der mit JAX-RS implementiert wurde. Durch die Annotation `@Path` wird der Basispfad des Serviceanbieters definiert. Für jede angebotene Methode kann dann die gewünschte HTTP-Anfragemethode (hier `@GET`) und die Adresse mit der Annotation `@Path` festgelegt werden (hier `methodeA` und `methodeB`). Im Beispiel erhält die Methode `methodeA()` einen Übergabeparameter mit dem Namen `nachricht`, der beim Aufruf über HTTP-GET gesetzt werden kann. Die Annotation `@Consumes` beschreibt die Codierung, die die Methode im übergebenen Argument erwartet (hier `MediaType.TEXT_PLAIN`). Die Annotation `@Produces` beschreibt die Codierung der Daten, welche die Methode zurückgibt. Hier die Klasse `ServiceanbieterImpl` mit den JAX-RS Annotationen<sup>109</sup>:

```
// Datei: Serviceanbieter.java
@Path ( "Serviceanbieter" )
public class ServiceanbieterImpl {
    @GET
    @Path ( "methodeA/{Nachricht}" )
    @Consumes ( MediaType.TEXT_PLAIN )
    public Response methodeA(@PathParam("Nachricht") String nachricht) {
        System.out.println (
            "Serviceanbieter: Methode A wurde aufgerufen");
        System.out.println ( "Parameter: " + nachricht);
        return Response.ok().build();
    }
}
```

<sup>108</sup> Die Webseite des Projekts ist: <http://jersey.java.net>.

<sup>109</sup> In diesem Programmbeispielen wurde aus Gründen der Übersichtlichkeit auf die benötigten "import"-Anweisungen verzichtet. Der vollständige Quellcode ist über den begleitenden Webauftritt zugänglich.

```

@GET
@Path ("methodeB")
@Produces (MediaType.TEXT_PLAIN)
public String methodeB() {
    String resultStr = "Nachricht von Server an Client";
    System.out.println (
        "Serviceanbieter: Methode B wurde aufgerufen");
    return resultStr;
}
}

```

Die nachfolgende Klasse zeigt, wie der Serviceanbieter gestartet werden kann:

```

// Datei: ServiceanbieterMain.java
public class ServiceanbieterMain {
    public static void main (String[] args) {
        HttpServer s;
        try {
            s = HttpServerFactory.create ("http://localhost:8080/rest/" );
            s.start();
            System.out.println ("Serviceanbieter ist nun aktiv "
                + "und wartet auf Anfragen...");
        } catch (IllegalArgumentException | IOException e) {
            e.printStackTrace();
        }
    }
}

```

Anschließend kann ein Servicenutzer implementiert werden, der die Methoden beim Serviceanbieter aufruft:

```

//Datei: Servicenutzer.java
public class Servicenutzer {
    public static void main (String[] args) {
        String restUrl = "http://localhost:8080/rest/Serviceanbieter";
        ClientConfig config = new DefaultClientConfig();
        Client client = Client.create (config);
        WebResource baseService = client.resource (restUrl);

        //MethodeA aufrufen
        WebResource serviceA = baseService.path ("methodeA")
            .path ("Nachricht an Serviceanbieter");
        serviceA.accept (MediaType.TEXT_PLAIN).get (String.class);

        //MethodeB aufrufen und Ergebnis zeigen
        WebResource serviceB = baseService.path ("methodeB");
        String result = serviceB.accept (MediaType.TEXT_PLAIN)
            .get (String.class);
        System.out.println ("Ergebnis: " + result);
    }
}

```



Hier das Protokoll des Programmlaufs des Serviceanbieters:

```
Serviceanbieter ist nun aktiv und wartet auf
Anfragen...
Serviceanbieter: Methode A wurde aufgerufen
Parameter: Nachricht an Serviceanbieter
Serviceanbieter: Methode B wurde aufgerufen
```



Hier das Protokoll des Programmlaufs des Servicenutzers:

```
Ergebnis: Nachricht von Server an Client
```

### 5.5.7.7 Implementierung eines SOA-Beispiels mit REST in JAX-RS

Dieses Kapitel fasst das SOA-Beispiel "Autovermietung" auf und erläutert die Implementierung dieses Beispiels unter Verwendung des Web Service Standards REST.<sup>110</sup> Anstelle der zuvor gezeigten XML-Web Services werden hier Methoden bereitgestellt, die direkt über HTTP-GET aufgerufen werden können. Als Beispiel soll die in Kapitel 5.5.7.5 gezeigte Autovermietung herangezogen werden.

Die Klasse `AutovermietungImpl` stellt die Umsetzung der Autovermietung mit REST dar. Es werden die Methoden `fahrzeugeAuflisten()`, `fahrzeugReservieren()`, `fahrzeugZurueckgeben()` und `bezahlen()` mit den entsprechenden Übergabe- und Rückgabewerten implementiert. Hier der Quelltext der Klasse `AutovermietungImpl`:

```
// Datei: AutovermietungImpl.java
@Path ("Autovermietung")
@XmlRootElement
public class AutovermietungImpl {
    // Alle verfügbare Fahrzeugklassen fuer einen Standort auflisten
    @GET
    @Path ("Fahrzeuge/{Standort}")
    @Produces (MediaType.TEXT_PLAIN)
    public String fahrzeugeAuflisten(
        @PathParam ("Standort") String Standort) {

        List<String> list = new ArrayList<String>();
        list.add ("Oberklasse");
        list.add ("Mittelklasse");
        list.add ("Kleinwagen");

        // Immer drei Fahrzeugklassen verfügbar
        return list.toString();
    }
}
```

<sup>110</sup> In diesem Programmbeispielen wurde aus Gründen der Übersichtlichkeit auf die benötigten "import"-Anweisungen verzichtet. Der vollständige Quellcode ist über den begleitenden Webauftritt zugänglich.

```

// Fahrzeug an einem Standort reservieren
@GET
@Path ("Reservieren/{Standort}/{Fahrzeugklasse}")
@Produces (MediaType.TEXT_PLAIN)
public String fahrzeugReservieren (
    @PathParam ("Standort") String standort,
    @PathParam ("Fahrzeugklasse") String fahrzeugklasse) {

    // Freies Fahrzeug ermitteln
    String beispielFahrzeug = "FZ_ID_4711";

    // Fahrzeug als belegt markieren
    System.out.println ("Das Fahrzeug mit der Kennung "
        + beispielFahrzeug + " wurde als belegt markiert.");

    // Fahrzeugkennung zurueckgeben
    return beispielFahrzeug;
}

// Fahrzeug abholen
@GET
@Path ("Abholen/{FahrzeugID}")
@Produces (MediaType.TEXT_PLAIN)
public String fahrzeugAbholen (
    @PathParam ("FahrzeugID") String fahrzeugID) {
    System.out.println ("Fahrzeug mit der Kennung "
        + fahrzeugID + " wurde abgeholt.");
    return "Abholung erfolgreich.";
}

// Fahrzeug zurueckgeben, freigeben und bezahlen
@GET
@Path ("Zurueckgeben/{FahrzeugID}/{KreditkartenNr}")
@Produces (MediaType.TEXT_PLAIN)
public String fahrzeugZurueckgeben (
    @PathParam ("FahrzeugID") String fahrzeugID,
    @PathParam ("KreditkartenNr") String kreditkartenNr) {
    System.out.println ("Fahrzeug mit der Kennung " + fahrzeugID
        + " wurde zurueckgebracht und als frei markiert.");

    if (bezahlen (fahrzeugID, kreditkartenNr)){
        return "Bezahlung erfolgreich.";
    } else {
        return "Bezahlung fehlgeschlagen.";
    }
}

// Fahrzeug bezahlen
private boolean bezahlen (String fahrzeugID, String kreditkartenNr){
    // Preis bestimmen und Bezahlung durchfuehren
    System.out.println("Bezahlung fuer das Fahrzeug mit der"
        + " Kennung " + fahrzeugID + " war erfolgreich.");
    return true;
}
}

```

In der Datei `AutovermietungMain.java` wird das Hauptprogramm definiert, das die Autovermietung als Serviceanbieter startet:

```
// Datei: AutovermietungMain.java
public class AutovermietungMain {
    public static void main (String[] args) {
        HttpServer s;
        String url = "http://localhost:8080/rest";
        try {
            s = HttpServerFactory.create (url);
            s.start();
            System.out.println ("Der Web Service ist nun im Netzwerk "
                                + "unter der Adresse " + url + " aufrufbar.");
            System.out.println ("Serviceanbieter lauscht jetzt...");
        } catch (IllegalArgumentException | IOException e) {
            e.printStackTrace();
        }
    }
}
```

Abschließend wird mit der Datei `AutovermietungClient.java` ein Nutzer für die Autovermietung implementiert:

```
// Datei: AutovermietungNutzer.java
public class AutovermietungNutzer {

    public static void main (String[] args) {
        String erg;
        WebResource res;
        String restUrl = "http://localhost:8080/rest/Autovermietung";

        String reservierungsNr;
        String standort = "Stuttgart";
        String kreditkartenNr = "abc123456";
        System.out.println ("Servicenutzer gestartet.");

        // Verbindung zum Serviceanbieter
        ClientConfig config = new DefaultClientConfig();
        Client client = Client.create (config);
        WebResource baseService = client.resource (restUrl);

        // Alle verfuegbaren Fahrzeuge auflisten
        res = baseService.path ("Fahrzeuge/" + standort);
        erg = res.accept (MediaType.TEXT_PLAIN).get (String.class);
        System.out.println ("Freie Fahrzeuge in " + standort + ":");
        System.out.println (erg);

        // Ein Fahrzeug reservieren
        System.out.println ("Reserviere einen Kleinwagen...");
        res = baseService.path ("Reservieren/" + standort
                                + "/Kleinwagen");
        reservierungsNr =
            res.accept (MediaType.TEXT_PLAIN).get (String.class);
        System.out.println ("Reserviertes Fahrzeug hat "
                            + "die Reservierungsnr. " + reservierungsNr + ".");
    }
}
```

```

// Fahrzeug abholen
res = baseService.path ("Abholen/" + reservierungsNr);
res.accept (MediaType.TEXT_PLAIN).get (String.class);
System.out.println ("Fahrzeug abgeholt.");

// Fahrzeug nutzen...

// Spaeter Fahrzeug zurueckgeben
res = baseService.path ("Zurueckgeben/"
    + reservierungsNr + "/" + kreditkartenNr);
res.accept (MediaType.TEXT_PLAIN).get (String.class);
System.out.println ("Fahrzeug zurueckgegeben.");
    }
}

```



### Hier das Protokoll des Serviceanbieters (Server):

Der Web Service ist nun im Netzwerk unter der Adresse  
<http://localhost:8080/rest> aufrufbar.  
 Serviceanbieter lauscht jetzt...  
 Das Fahrzeug mit der Kennung FZ\_ID\_4711 wurde als  
 belegt markiert.  
 Fahrzeug mit der Kennung FZ\_ID\_4711 wurde abgeholt.  
 Fahrzeug mit der Kennung FZ\_ID\_4711 wurde  
 zurueckgebracht und als frei markiert.  
 Bezahlung fuer das Fahrzeug mit der Kennung FZ\_ID\_4711  
 war erfolgreich.



### Hier das Protokoll des Servicenutzers (Client):

Servicenutzer gestartet.  
 Freie Fahrzeuge in Stuttgart:  
 [Oberklasse, Mittelklasse, Kleinwagen]  
 Reserviere einen Kleinwagen...  
 Reserviertes Fahrzeug hat die Reservierungsnr.  
 FZ\_ID\_4711.  
 Fahrzeug abgeholt.  
 Fahrzeug zurueckgegeben.



## 5.6 Das Architekturmuster Model-View-Controller

Das Architekturmuster Model-View-Controller trennt ein **interaktives System** in die Komponenten **Model** (Verarbeitung/Datenhaltung), **View** (Ausgabe) und **Controller** (Eingabe). Das Model hängt dabei nicht von der Ein- und Ausgabe ab.

### 5.6.1 Name/Alternative Namen

Model-View-Controller<sup>111</sup> (MVC).

### 5.6.2 Problem

Die Benutzerschnittstelle eines Systems wird besonders häufig geändert. Sie soll deshalb leicht – sogar zur Laufzeit – ausgetauscht werden können, ohne dass das restliche Programm dadurch geändert werden muss. Auf der Bedienoberfläche soll ein und dieselbe Information in verschiedener Form dargestellt werden können. Dabei sollen Änderungen an den Daten des Systems gleichzeitig in allen Darstellungen sichtbar sein.

### 5.6.3 Lösung

Eines der bekanntesten Architekturmuster ist das **Model-View-Controller-Architekturmuster** – kurz **MVC**. Es kann als "prägender Gedanke" bei der Erstellung von interaktiven Systemen angesehen werden. Ursprünglich wurde das MVC-Architekturmuster in den späten 70er Jahren von Trygve Reenskaug für die Smalltalk-Plattform entwickelt [Fow03]. Seitdem wurde es in vielen GUI-Frameworks verwendet.

Mit dem Architekturmuster MVC ist es möglich,

- die **Datenhaltung und Verarbeitung**,
- die **Präsentationslogik (Darstellung)** sowie
- die **Benutzereingaben**

in getrennten Komponenten zu realisieren. Das MVC-Architekturmuster beschreibt, wie die Daten und ihre Verarbeitung im **Model**, die Darstellung der Daten in einer **View** und die Interpretation der Eingaben des Benutzers im **Controller** voneinander getrennt werden, wobei die Komponenten Model, View und Controller dennoch einfach miteinander kommunizieren können. Views und Controller bilden zusammengenommen das User Interface (UI).

Die verschiedenen Funktionen eines Programms in verschiedenen Teilen einer Lösung zu realisieren, ist ein generelles Bestreben in der Programmentwicklung (**Trennung der Belange**, engl. **separation of concerns**, siehe Kapitel 1.3). Daher sollten die Datenhaltung/Verarbeitung, die Präsentationslogik sowie die Interpretation der Benutzereingaben in jeweils eigenen Komponenten getrennt programmiert und untereinander nicht vermischt werden. Dies verbessert die Struktur des Quelltextes und somit

<sup>111</sup> In diesem Kapitel werden die engl. Fachbegriffe "Model" und "View" anstelle der deutschen Begriffe "Modell" und "Ansicht" verwendet. Das Wort Controller steht aber bereits im deutschen Wörterbuch.

die Wartbarkeit der Anwendung. Da Model, View und Controller getrennte Komponenten darstellen, wobei ein Benachrichtigungsmechanismus die Kommunikation zwischen dem Model und der View ermöglicht, können ein und dieselben Daten des Model auf verschiedene Art und Weise dargestellt werden – man hat einfach verschiedene Views.

Das MVC-Muster ist sehr sinnvoll, wenn **mehrere Ansichten (Views)** auf ein Model zur gleichen Zeit benötigt werden, die die Daten verschieden darstellen.



### 5.6.3.1 Teilnehmer

Der Grundgedanke der MVC-Architektur ist die Zerlegung einer interaktiven Anwendung in die Komponenten **Model**, **View** und **Controller**. Dabei ist jeder View stets ein Controller zugeordnet. Ein Controller kann aber mehrere Views "bedienen".

Die Aufgaben dieser Komponenten sind im sogenannten Active Model<sup>112</sup>:

- Das **Model** umfasst die Kernfunktionalität und kapselt die Verarbeitung und die Daten des Systems.
- Eine **View** stellt die Daten für den Benutzer dar. Sie erhält die darzustellenden Daten vom Model.
- Ein **Controller** ist für die Entgegennahme der Eingaben des Benutzers und ihre Interpretation verantwortlich.

Änderungen am Model, die zur Laufzeit auftreten, werden im Active Model direkt vom Model an alle angemeldeten Views bzw. an die angemeldeten Controller gegeben.

Will eine View nicht länger über eine Änderung der Daten des Model informiert werden, weil sie zum Beispiel ausgeblendet wird, kann sie sich bei ihrem Model abmelden.

Das folgende Bild zeigt die Interaktionen zwischen den erwähnten drei Komponenten:

<sup>112</sup> Das Active Model selbst wird in diesem Kapitel noch detailliert vorgestellt und abgegrenzt. Hier werden die Aufgaben der verschiedenen Komponenten im Rahmen des Active Model kurz erläutert.

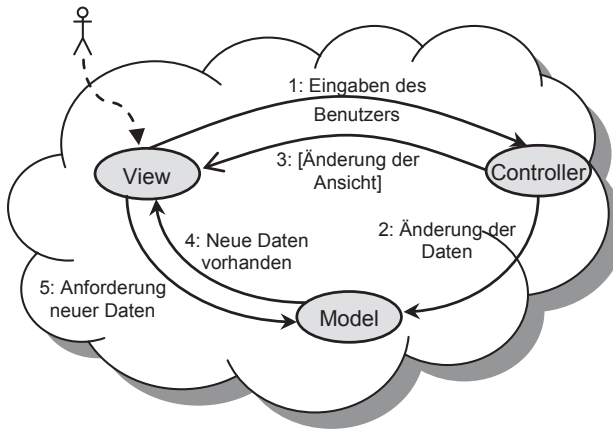


Bild 5-37 Interaktionen der Komponenten der MVC-Architektur

Die Interaktionen dieser Komponenten werden im Folgenden entsprechend der Nummerierung in Bild 5-37 genauer beschrieben:

- (1) Der Benutzer interagiert mit der View. Wenn der Benutzer eine Eingabe in der View macht (z. B. auf einen Button klickt), leitet die View dieses Ereignis an den zugeordneten Controller weiter.
- (2) Der Controller hat die Aufgabe, dieses Ereignis zu interpretieren und entsprechend zu handeln. Er veranlasst eine Änderung der Daten im Model.
- (3) Eventuell wird auch die View vom Controller veranlasst, ihren Zustand zu verändern (z. B. ein Eingabefeld auszublenden).
- (4) Das Model meldet nach einer Änderung seiner Daten an alle bei ihm zur Aktualisierung angemeldeten Views, dass sich die Daten des Model geändert haben.
- (5) Die Views holen die neuen Daten vom Model ab und aktualisieren ihre Darstellung. Eine View ist sozusagen ein Fenster zum Model.

Die beiden Komponenten View und Controller sind **abhängig** vom Model. Sie kennen den Aufbau der Daten des Model. Eine View muss Informationen über die Daten des Model besitzen, um diese darstellen zu können. Diese Informationen werden auch vom Controller benötigt, damit er die Daten des Model ändern kann.



Zwischen Controller und View besteht eine **wechselseitige Abhängigkeit**<sup>113</sup>, weil zum einen die View den Controller über Eingaben des Benutzers informieren muss und weil zum anderen der Controller den Zustand einer View bestimmt und die Ansicht einer View ändern kann.



<sup>113</sup> Diese wechselseitige Abhängigkeit ist auch der Grund dafür, dass Controller und View oftmals gemeinsam zu einer View-Controller-Komponente zusammengefasst werden. In Swing wird eine solche Komponente als **Delegate** bezeichnet.

Diese Abhängigkeiten sind im folgenden Bild symbolisiert:

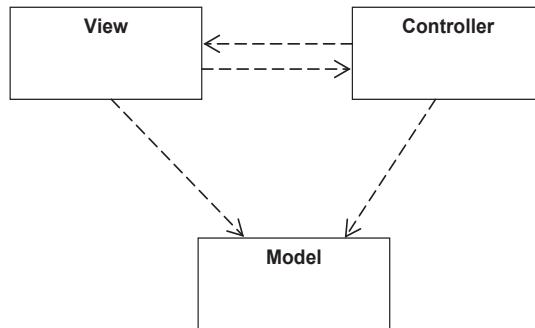


Bild 5-38 Komponenten der MVC-Architektur und ihre Abhängigkeiten

### Das Model

Die Aufgabe des Model ist das Durchführen der Geschäftsprozesse (ohne Ein- und Ausgabe) und das Speichern sämtlicher Geschäftsdaten. Das bedeutet, dass im Model alle Methoden implementiert werden, die nichts mit den Benutzereingaben und der Visualisierung direkt zu tun haben.

Bei der Implementierung des Model muss prinzipiell nicht auf die Implementierung der Views bzw. der Controller geachtet werden, da das Model nicht von einer View bzw. einem Controller abhängt.



Dies ist ein wichtiges Merkmal des Model, da die Unabhängigkeit des Model von View und Controller die Flexibilität der Architektur erhöht. In der Praxis ändern sich die Anforderungen an die Views und Controller häufiger, ihre Strukturänderungen wirken sich aber nicht auf das Model aus. Wird aber der Aufbau des Model geändert, so müssen auch die Controller und die Views angepasst werden.

Das Model kapselt die dem System zugrundeliegenden Daten. Es hat Operationen, mit deren Hilfe die gespeicherten Daten abgerufen, verarbeitet und geändert werden können. Ein Controller ruft bestimmte Methoden beim Model auf, um dessen Daten entsprechend der interaktiven Eingaben des Benutzers zu ändern. Welche Daten geändert werden können und welche Daten zur Ansicht bereitstehen, wird vom Model bestimmt.

In komplexen Systemen können verschiedene Geschäftsprozesse im Model "gleichzeitig" angestoßen werden. Dadurch können Daten des Model, die von einem Controller benutzt werden, um die Ansicht einer View zu steuern, von einem anderen Geschäftsprozess geändert werden. Ist dies der Fall, so muss der erwähnte Controller über eine Änderung des Model informiert werden, um entsprechend zu reagieren (z. B. um Kontrollelemente der View aus- oder einzuschalten). In einem solchen Fall muss

sich der Controller also ebenfalls beim Model anmelden, damit er bei einer Zustands- oder Datenänderung benachrichtigt wird.

Werden die Daten im Model geändert, gibt es zwei unterschiedliche Methoden, eine View über eine Änderung zu informieren:

- Die erste Möglichkeit ist die Verwendung eines sogenannten **Passive Model**. Dabei informiert der Controller die Views über eine erfolgte Änderung. Auf das Passive Model soll in diesem Buch nicht eingegangen werden.
- Bei der zweiten Möglichkeit, dem in diesem Kapitel verwendeten **Active Model**, ist es die Aufgabe des Model, seine Views über eine Änderung der Daten des Model zu informieren. Um diesen Kommunikationsfluss mit einer möglichst geringen Abhängigkeit zwischen dem Model und seinen Views bereitzustellen, kann das Beobachter-Muster (siehe Kapitel 4.11) verwendet werden. Im Active Model gibt es hierzu den Pull- und den Push-Betrieb:
  - Im **Push-Betrieb** sendet das Model die Daten an die View. Dabei werden die Daten als Übergabeparameter bei der Benachrichtigung der View durch das Model mitgegeben. Diese Lösung ist schlecht wiederverwendbar, da die entsprechenden Methoden zu speziell ausgeprägt werden müssen.
  - Im **Pull-Betrieb** informiert das Model die View nur, dass neue Daten vorliegen. Daraufhin holt die View die neuen Daten beim Model ab.

Im Folgenden soll der Pull-Betrieb des Active Model vertieft werden.

Die Aufgaben des **Model** im MVC-Muster sind im Pull-Betrieb des Active Model:

- Geschäftsprozesse (ohne Ein- und Ausgabe) und Speicherung der Daten durchführen.
- Die An- und Abmeldung von Views entgegennehmen.
- Angemeldete Views über geänderte Daten informieren.
- Eine Aufrufchnittstelle für das Abholen von Daten durch die Views bzw. Controller bereitstellen.
- In der View eingegebene Daten von Controllern entgegennehmen.



## Die View

Die View dient zur Darstellung der Daten des Model. Verschiedene Views können dem Benutzer dieselben Daten auf unterschiedliche Weise präsentieren. Wenn die Daten im Model geändert werden, so aktualisieren sich alle Views, die diese Daten anzeigen. Zum Beispiel können Messdaten in einer bestimmten View in einem Balkendiagramm angezeigt und in einer anderen View aber in einem Kreisdiagramm dargestellt werden. Dies ist im folgenden Bild symbolisiert:

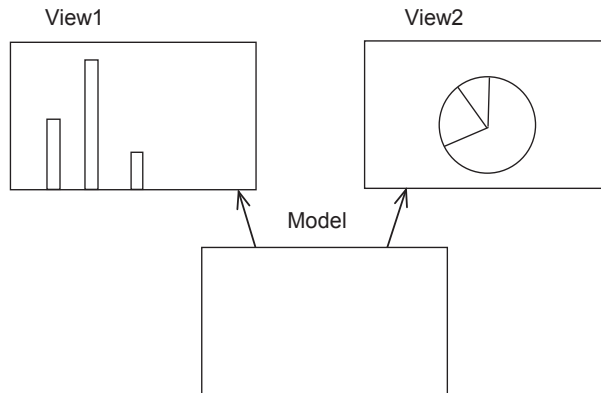


Bild 5-39 Aktualisierung zweier Views durch das Model

Damit eine View die Daten des Model anzeigen kann, muss sie den Aufbau der Daten des Model genau kennen. Somit existiert eine starke Abhängigkeit der View vom Model, in die andere Richtung – wie oben erwähnt – allerdings nicht. Folglich wirken sich Änderungen im Aufbau der Daten des Model auf seine Views aus und erfordern deren Anpassung an das Model.

Außer der Darstellung von Daten enthält eine View üblicherweise auch die typischen Kontrollelemente einer interaktiven Anwendung wie z. B. Buttons. Eine Interaktion des Benutzers mit den Kontrollelementen einer Oberfläche erzeugt dabei ein Ereignis. Dieses Ereignis wird von der View nicht selbst interpretiert, sondern wird – wie auch die in der View geänderten Daten – an den Controller zur Interpretation weitergeleitet.

Die Aufgaben einer **View** im MVC-Muster sind im Pull-Betrieb des Active Model:

- Darstellung der Daten des Model für einen Bediener.
- Abfrage der Daten beim Model nach einer Änderungsnachricht.
- Aktualisierung der Darstellung mit den neuen Daten.
- Übergabe von in der View eingegebenen Daten und der durch Betätigung der Kontrollelemente erzeugten Ereignisse an den Controller.
- Anpassung der Oberfläche infolge von Steuerbefehlen des Controllers.



## Der Controller

Der Controller steuert das Model und den Zustand einer View auf Grund von Eingaben des Bedieners.

Die Aufgabe des Controllers ist es dabei, die für das **Model** empfangenen Ereignisse und die Dateneingaben des Benutzers in Methodenaufrufe für das Model umzusetzen.

Der Controller interpretiert die Aktionen des Benutzers und entscheidet, welche Methoden des Model aufgerufen werden sollen. Er fordert das Model damit auf, eine Änderung seines Zustandes oder seiner Daten durchzuführen.

Des Weiteren kann die durch ein Ereignis ausgelöste Aktion es erfordern, dass die **View** angepasst werden muss, weil beispielsweise Kontrollelemente zu aktivieren oder zu deaktivieren sind. Die entsprechende Benachrichtigung einer View wird ebenfalls vom Controller nach der Interpretation eines Ereignisses vorgenommen. Wie bereits erwähnt, hat jede View genau einen Controller, aber ein Controller kann mehrere Views bedienen.

Die Aufgaben eines **Controllers** im MVC-Muster sind im Active Model:

- Bedienereingaben von der View entgegennehmen.
- Die View gemäß den Bedienereingaben steuern.
- Von der View erhaltene Daten dem Model übergeben.



### 5.6.3.2 Klassendiagramm

Ein Klassendiagramm für das MVC-Muster ist beispielhaft in Bild 5-44 zu sehen. Dem MVC-Muster liegen meist die Entwurfsmuster Beobachter, Kompositum und Strategie zugrunde. Es ist aber auch möglich, im MVC-Muster noch weitere Entwurfsmuster einzubauen. In seiner grundlegenden Form setzt sich das Klassendiagramm des MVC-Musters somit aus den Klassendiagrammen der Entwurfsmuster Beobachter, Kompositum und Strategie zusammen. Der Einsatz dieser Entwurfsmuster im MVC-Muster wird in Kapitel 5.6.3.4 ausführlich diskutiert.

### 5.6.3.3 Dynamisches Verhalten

Im MVC-Muster sind die Model-, View- und Controller-Objekte durch Protokolle entkoppelt. Ein View-Objekt muss sicherstellen, dass es den aktuellen Zustand seines Model-Objekts wiedergibt. Realisiert wird dies, indem das Model die von ihm abhängigen Views über eine Änderung seiner Daten unterrichtet. Die Views bringen sich daraufhin in einen konsistenten Zustand. Dieser Ansatz beruht auf dem **Beobachter-Muster** und ermöglicht es, einem Model mehrere Views zuzuordnen. Zudem können neue Views entwickelt werden, ohne das Model umschreiben zu müssen.

In Bild 5-40 soll das Zusammenspiel der Komponenten Controller, Model und View(s) von der Anmeldung der View(s) beim Model (`anmelden()`) bis zur Aktualisierung der Daten in der View bzw. den Views betrachtet werden. Bei einem Vergleich mit Bild 4-35 (Sequenzdiagramm des Beobachter-Musters) fällt auf, dass der in Bild 5-40 dargestellte Ablauf zwischen Model und View exakt dem des Beobachter-Musters entspricht. Darauf wird in Kapitel 5.6.3.4 noch genauer eingegangen. Hier das bereits erwähnte Sequenzdiagramm:





### 5.6.3.4 Entwurfsmuster im MVC

Es gibt beim MVC-Muster keine festen Vorgaben, welche Entwurfsmuster enthalten sein müssen. Es gibt mehrere Varianten und je nach den Anforderungen der Anwendung an die Architektur werden bestimmte Entwurfsmuster genutzt oder auch nicht.

Das Grundgerüst bildet aber – wie bereits in Kapitel 5.6.3.2 erwähnt – das **Beobachter-Muster** zwischen Model und View, das in einer MVC-Architektur praktisch immer vorzufinden ist. Zusätzlich werden in der Regel noch zwei weitere Muster bei der MVC-Architektur benutzt: Das **Strategie-Muster** (engl. **strategy pattern**) dient dem Zusammenspiel zwischen View und Controller. Das **Kompositum-Muster** (engl. **composite pattern**) kann für den Aufbau einer View aus den Elementen der grafischen Oberfläche eingesetzt werden.

### Das Beobachter-Muster im MVC

Das Beobachter-Muster wird im MVC-Architekturmuster verwendet, um eine möglichst lose Kopplung vom Model zur View herzustellen. In Kapitel 4.11 ist der prinzipielle Aufbau des Musters dargestellt. Hier ein an die Situation im MVC angepasstes Klassendiagramm dieses Musters:

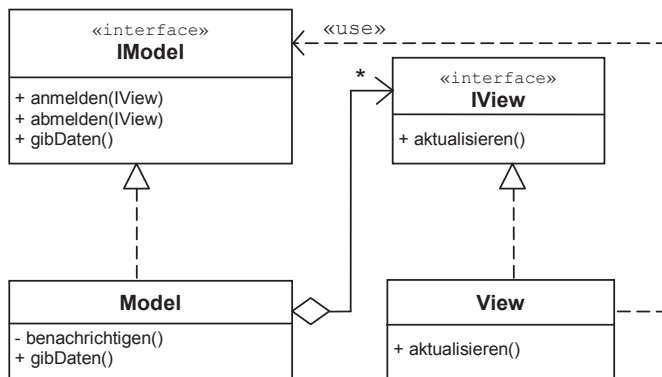


Bild 5-41 Das Beobachter-Muster zwischen Model und View im Pull-Betrieb des Active Model

Hierbei nimmt das Model die Rolle des Beobachtbaren ein und die View die Rolle des Beobachters. Wird eine View für ein bestimmtes Model instanziiert, so muss sie sich beim Model anmelden. Dies geschieht, indem die View beim Model die Methode `anmelden()` aufruft und eine Referenz auf sich selbst als Parameter übergibt. Dabei übergibt sich die View als ein Objekt vom Typ **IView**. Somit kann das Model nur die Methoden, die im Interface **IView** deklariert sind, bei der View aufrufen. Da das Model dieses Interface vorgibt, ist das Model nicht von ihm abhängig. Der Beobachter, sprich die View, darf das Interface nicht abändern. Das Model hat also keinerlei Informationen über den Aufbau der konkreten View und ist somit auch nicht abhängig von ihr. Alle registrierten Views werden in einer Liste im Model abgespeichert.

Wird nun das Model vom Controller verändert, ruft das Model die Methode `benachrichtigen()` zur Information der Views auf. In dieser Methode wird für jede View, die sich als Beobachter registriert hat, die Methode `aktualisieren()` aufgerufen. Die-

ser Aufruf wird auch als **Callback** bezeichnet, da der Beobachtbare einen Angemeldeten zurückruft. Durch den Callback weiß die View, dass die Daten, die sie darstellt, veraltet sind. Sie muss daraufhin die Daten neu vom Model auslesen. Dies tut sie, indem sie die Methode `gibDaten()` beim Model aufruft (Pull-Betrieb des Active Model, siehe Kapitel 5.6.3.1).

Views, die nicht mehr gebraucht werden, weil sie zum Beispiel ausgeblendet sind, sollten sich vom Model abmelden.



Wird dies nicht konsequent gemacht, ist die Anzahl der zu benachrichtigenden Views unnötig groß und verbraucht damit Rechenzeit. Folglich wird die Performance der Anwendung negativ beeinflusst.

### Das Kompositum-Muster im MVC

Beim Zusammensetzen von grafischen Oberflächen – wie es auch die Views im MVC-Muster sind –, ist das Kompositum-Muster<sup>115</sup> weit verbreitet. Eine View wird dabei aus nicht verschachtelten Komponenten wie z. B. Buttons oder Textfeldern und verschachtelten Komponenten wie z. B. Panels in Java zusammengesetzt. Dabei erben deren Klassen alle von derselben Basisklasse (von der Klasse `ViewComponent` in Bild 5-42). Die Klasse `ViewPanel` kann Elemente gruppieren und entspricht beim Kompositum-Muster einem Kompositum. Die Klasse `Button1` und `Button2` stellen Blatt-Klassen dar und sind hier nur stellvertretend für alle möglichen Oberflächen-Elemente zu sehen, die keine weiteren Elemente einschließen können. Nachfolgend ist ein beispielhaftes Klassendiagramm zum Kompositum-Muster im MVC abgebildet:

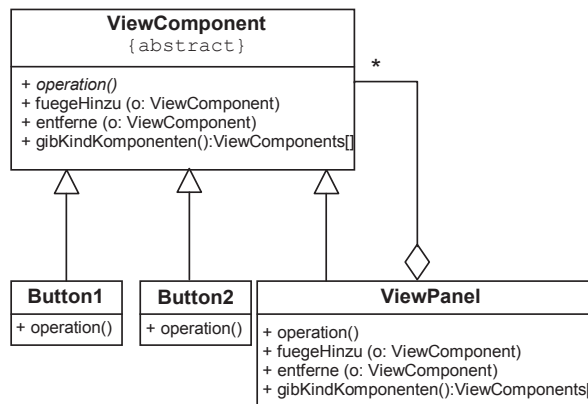


Bild 5-42 Klassendiagramm Kompositum-Muster im MVC

Jedes Element zeigt infolge der Vererbung dasselbe Verhalten wie die Elemente, aus denen es zusammengesetzt ist. Durch die Verschachtelung entsteht eine Baumstruktur aus Knoten und Blättern. Knoten sind dabei aus weiteren Elementen zusammengesetzt. Blätter – wie z. B. ein Button – tragen keine anderen Elemente in sich.

<sup>115</sup> Das Kompositum-Muster ohne Bezug zur MVC-Architektur ist in Kapitel 4.7 dargestellt.

Um einen Aufruf einer gemeinsamen Methode (z. B. `operation()`) in allen Bauelementen zu veranlassen, wird diese Methode im Wurzelement des Baumes aufgerufen. Von hier aus wird der Aufruf an die inneren Elemente delegiert (**Delegationsprinzip**).

Anstelle der Methode `operation()` kann man sich beispielsweise eine `resize()`-Methode vorstellen, mit deren Hilfe man eine Größenänderung eines Fensters an alle enthaltenen Elemente weiterleitet.

### Das Strategie-Muster im MVC

Durch das MVC-Muster ist es möglich, die Reaktion einer View auf Benutzereingaben zu ändern, indem einer View von außen ein neuer Controller zugeordnet wird. Dabei ist jeder View-Komponente zu jedem Zeitpunkt ein entsprechender Controller zugeordnet, so dass jede GUI-Komponente stets ein Paar aus Controller und View aufweist.

Die Beziehung zwischen View und Controller kann mit dem klassischen Strategie-Muster<sup>116</sup> realisiert werden. Der Controller stellt dabei die Strategie der View dar, also das Verhalten, das die Eingaben des Benutzers interpretiert. Die View delegiert die Informationen über erfolgte Eingaben an den Controller weiter. Die View kann aber auch Schnittstellenfunktionen bereitstellen, über die der Controller, wenn er von der View über eine Eingabe des Benutzers informiert wurde, die vom Benutzer veränderten Werte oder Parameter aus den View-Elementen auslesen kann (z. B. den Wert eines Textfeldes). Eine View ist dabei stets nur für den Empfang der Eingaben verantwortlich. Wie die Benutzereingaben interpretiert werden, obliegt allein dem Controller.

Bild 5-43 zeigt das Strategie-Muster im MVC-Kontext:

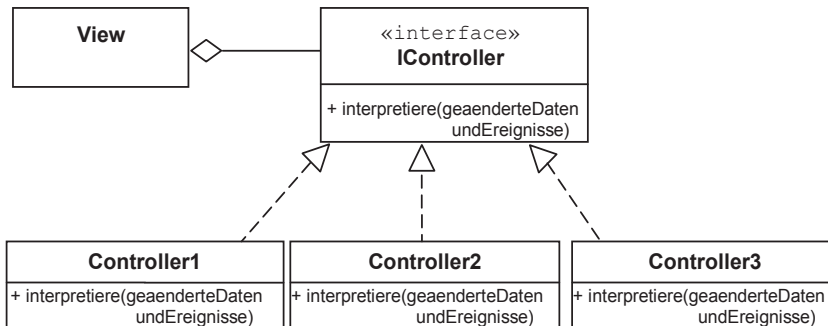


Bild 5-43 Klassendiagramm Strategie-Muster im MVC

Dabei ist `interpretiere()` eine Methode, an welche die View neu in der View eingegebene Daten und Ereignisse übergibt. Je nachdem, welcher konkrete Controller eingesetzt wird, kann die Methode `interpretiere()` eine andere Funktionalität aufweisen.

<sup>116</sup> Das Strategie-Muster ohne Bezug zur MVC-Architektur ist in Kapitel 4.12 dargestellt.



## 5.6.4 Bewertung

### 5.6.4.1 Vorteile

Vorteile des MVC-Musters sind:

- Die Model-Klassen können völlig unabhängig von der Gestaltung der Benutzeroberfläche entworfen werden.
- Die Benutzeroberfläche kann geändert werden, ohne eine Veränderung an den Model-Klassen vornehmen zu müssen.
- Das Model kann unabhängig von einer Oberfläche getestet werden.
- Es können verschiedene Benutzeroberflächen für dieselbe Anwendung entworfen werden.

### 5.6.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- Häufige Aktualisierungen können die Performance verschlechtern. Dies gilt insbesondere dann, wenn eine View mehrere Aufrufe benötigt, um die darzustellenden Daten vom Model zu erhalten.
- Der erhöhte Implementierungsaufwand durch die Unterteilung in viele Klassen ist bei kleinen Anwendungen nicht gerechtfertigt.

## 5.6.5 Einsatzgebiete

Das MVC-Muster kann bei jeder Art von interaktiver Software zur Strukturierung angewandt werden.

Das MVC-Architekturmuster wird in fast allen GUI-Frameworks verwendet. Beispielsweise wird es in der Grafikbibliothek Swing von Java erfolgreich eingesetzt. Damit ist der Entwickler gezwungen, die Daten von deren Darstellung zu trennen.

## 5.6.6 Ähnliche Muster

Die Übertragung des MVC-Musters auf Webanwendungen wird Model 2 genannt (siehe dazu beispielsweise [Lah09]). Das Muster Model 2 findet Anwendung, wenn die Darstellung – die Präsentation – vor jeder Auslieferung auf dem Server dynamisch erstellt wird. Bei Webanwendungen ist die gegenseitige Unabhängigkeit zwischen der Struktur, dem Aussehen und dem Inhalt notwendig. Dies ermöglicht beispielsweise eine Veränderung der Struktur, ohne das Aussehen (Design) mit verändern zu müssen.

Im Falle von Java wird Model 2 meistens in Kombination mit Servlets, JSPs und JavaBeans eingesetzt. Der Client, z. B. ein Web-Browser, sendet eine Anfrage nach einer Ressource an den Server, z. B. nach einer Webseite im HTML-Format über HTTP. Auf dem Server dient ein Servlet als Controller und empfängt zuerst die Anfrage des

Clients. Das Servlet entscheidet anschließend, welche View – hierbei eine JSP – als Antwort an den Client zurückgesendet werden soll. Nachdem die View vom Server ausgewählt worden ist, ruft diese die Methoden einer JavaBean auf, um den Inhalt zu erhalten. Die JavaBeans werden hier für die Haltung von Daten genutzt. Die View beinhaltet eine vordefinierte Struktur, beispielsweise ein HTML- oder ein XML-Gerüst. Dieses Gerüst wird mit dem erhaltenen Inhalt befüllt und als Antwort an den Client zurückgeliefert. Bei Web-Seiten wird das Aussehen über CSS (Cascading Style Sheet) definiert und in einer separaten Datei abgelegt. Diese wird vom Browser beim Auswerten der gelieferten Antwort nachgeladen.

### 5.6.7 Programmbeispiel

Im Folgenden wird ein Beispiel gezeigt und erklärt. In diesem Beispiel wird das MVC-Architekturmuster dazu verwendet, verschiedene Ansichten auf die Sitzplatzverteilung von Parteien nach einer Wahl zu präsentieren. Der Benutzer kann zur Laufzeit die Anzeige der Sitzplatzverteilung ändern, woraufhin sich die Views selbstständig aktualisieren.

Die Idee zu diesem Beispiel stammt aus [Bus98]. Es wurde hier zu einem lauffähigen Java-Programm ausgearbeitet. Der vollständige Quellcode ist allerdings zu umfangreich und ist im Begleitmaterial zu diesem Buch enthalten. In der nun folgenden Darstellung ist das Beispiel daher nur verkürzt wiedergegeben.

#### Model

Die beiden Schnittstellen `IObserverInterface` und `IObservableInterface` gehören zum Beobachter-Muster. Sie ermöglichen es dem Model, seine Views über Änderungen zu informieren:

```
// Datei: IObserverInterface.java
public interface IObserverInterface
{
    public void update();
}

// Datei: IObservableInterface.java
public interface IObservableInterface
{
    public void registerObserver (IObserverInterface o);
    public void removeObserver (IObserverInterface o);
    public void notifyObservers();
}
```

Die Schnittstelle `IModelInterface` definiert das Model, das die Sitzplatzanzahl von drei Parteien (rot, grün und blau) speichert. Zu beachten ist hier, dass die Schnittstelle `IModelInterface` von der Schnittstelle `IObservableInterface` ableitet, um für die View beobachtbar zu sein. Hier die Schnittstelle `IModelInterface`:

```
// Datei: IModelInterface.java
public interface IModelInterface extends IObservableInterface
{
    public double getRedPercentage();
}
```

```
public double getBluePercentage();  
public double getGreenPercentage();  
public void setRedValue (int value);  
public void setBlueValue (int value);  
public void setGreenValue (int value);  
}
```

Die Klasse `DataModel` implementiert die Schnittstelle `IModelInterface` und repräsentiert somit das Datenmodell. Es stellt die in den Schnittstellen definierten Methoden zum Lesen und Schreiben der Sitzplätze der Parteien sowie Methoden zum Verwalten und Benachrichtigen von Beobachtern (Views) zur Verfügung. Im Folgenden die Klasse `DataModel`:

**// Datei: DataModel.java**

```
import java.util.ArrayList;  
public class DataModel implements IModelInterface  
{  
    private int m_redValue = 0;  
    private int m_greenValue = 0;  
    private int m_blueValue = 0;  
    private ArrayList<IObserverInterface> m_observers  
        = new ArrayList<IObserverInterface>();  
  
    public double getBluePercentage()  
    {  
        double total = m_redValue + m_greenValue + m_blueValue;  
        if (total > 0)  
            return m_blueValue / total;  
        return 0;  
    }  
  
    public double getGreenPercentage()  
    {  
        double total = m_redValue + m_greenValue + m_blueValue;  
        if (total > 0)  
            return m_greenValue / total;  
        return 0;  
    }  
  
    public double getRedPercentage()  
    {  
        double total = m_redValue + m_greenValue + m_blueValue;  
        if (total > 0)  
            return m_redValue / total;  
        return 0;  
    }  
  
    public void setBlueValue (int value)  
    {  
        m_blueValue = value;  
        notifyObservers();  
    }  
  
    public void setGreenValue (int value)  
    {  
        m_greenValue = value;  
    }  
}
```

```

        notifyObservers();
    }

    public void setRedValue (int value)
    {
        m_redValue = value;
        notifyObservers();
    }

    public void notifyObservers()
    {
        for (int i = 0; i < m_observers.size(); ++i)
            m_observers.get (i).update();
    }

    public void registerObserver (IObserverInterface o)
    {
        m_observers.add (o);
    }

    public void removeObserver (IObserverInterface o)
    {
        if (m_observers.contains (o))
            m_observers.remove (o);
    }
}

```

## View

Die drei Klassen `TableView`, `PieChartView` und `BarChartView` stellen die unterschiedlichen Sichten auf das Model dar. Der Benutzer kann zwischen Tabellenform, Kreisdiagramm (engl. pie chart) und Balkendiagramm (engl. bar chart) wählen. Aus Gründen der Übersichtlichkeit wird die genaue Implementierung der Klassen in Java Swing hier weggelassen. Sie ist jedoch auf dem Webauftritt verfügbar. Die Views repräsentieren die Beobachter im Beobachter-Muster und implementieren deshalb die Schnittstelle `IObserverInterface`. Damit ist das Model in der Lage, die Views über Änderungen zu informieren. Voraussetzung ist natürlich, dass die Views – wie in den Konstruktoren gezeigt – das Model kennen und sich bei ihm als Beobachter anmelden. Die Klasse `TableView` stellt hierbei noch einen Spezialfall dar, da sie als Benutzerschnittstelle zur Eingabe der Sitzplätze dient. Um das Model über diese Benutzerschnittstelle modifizierbar zu machen, wird eine Instanz der Klasse `TableViewController` im Konstruktor erzeugt und bei Änderungen des Benutzers über die neu eingegebenen Werte informiert. Die Definition der Klasse `TableViewController` erfolgt später. Hier die Klassen der Views:

// Datei: `TableView.java`

```

public class TableView implements IObserverInterface
{
    private IControllerInterface m_controller;
    private IModelInterface m_model;
    . . .

    public TableView (IModelInterface model)
    {
        createComponents();
    }
}

```



```

        m_model = model;
        model.registerObserver (this);
        m_controller = new TableViewController (model, this);
    }

    public void createComponents()
    {
        . . .
        m_setButton = new JButton ("Set");
        m_setButton.addActionListener (new ActionListener()
        {
            public void actionPerformed (ActionEvent e)
            {
                m_controller.setValues (m_redTextField.getText(),
                                         m_greenTextField.getText(),
                                         m_blueTextField.getText());
            }
        });
        . . .
    }
    . . .
    public void update()
    {
        // Update-Methode wird nach erfolgter Datenaenderung vom
        // Model aufgerufen. Da die Datenaenderung nach Eingabe vom
        // Benutzer ueber die TableView-Klasse erfolgt, und - in
        // unserem Beispiel - von nirgendwo sonst, muessen die
        // angezeigten Daten hier nicht nochmal aktualisiert werden.
    }
}

```

**// Datei: PieChartView.java**

```

public class PieChartView implements IObservableInterface
{
    private IModelInterface m_model;
    private JPanel m_viewPanel;
    . . .
    public PieChartView (IModelInterface model)
    {
        createComponents();
        m_model = model;
        model.registerObserver (this);
    }

    public void createComponents()
    {
        . . .
    }

    public void update()
    {
        . . .
        double red = m_model.getRedPercentage();
        double green = m_model.getGreenPercentage();
        double blue = m_model.getBluePercentage();
    }
}

```

```

        m_viewPanel = createChartPanel (red, green, blue);
        . . .
    }

    private JPanel createChartPanel (final double red,
        final double green, final double blue)
    {
        . . .
    }
}

```

**// Datei: BarChartView.java**

```

public class BarChartView implements IObservableInterface
{
    IModelInterface m_model;
    JFrame m_viewFrame;
    JPanel m_viewPanel;
    public BarChartView (IModelInterface model)
    {
        createComponents();
        m_model = model;
        model.registerObserver (this);
    }

    public void createComponents()
    {
        . . .
    }

    public void update()
    {
        . . .
        double red = m_model.getRedPercentage();
        double green = m_model.getGreenPercentage();
        double blue = m_model.getBluePercentage();
        m_viewPanel = createChartPanel (red, green, blue);
        . . .
    }

    private JPanel createChartPanel (final double red,
        final double green, final double blue)
    {
        . . .
    }
}

```

## Controller

Die Schnittstelle `IControllerInterface` definiert einen Controller. Über die Methode `setValues()` kann der Controller von einer View über neue Werte für die Sitzplatzverteilung informiert werden. Hier die Schnittstelle `IControllerInterface`:

**// Datei: IControllerInterface.java**

```

public interface IControllerInterface
{

```

```

    public void setValues (String red, String green, String blue);
}

```

Die Klasse `TableViewController` implementiert die Schnittstelle `IController-Interface`. In der Methode `setValues()` werden die neuen Werte für die Sitzplatzverteilung entgegengenommen und anschließend das Model geändert. Da es sich bei den neuen Werten um eine Benutzereingabe handelt, sollten sie vor der Modifikation des Model unbedingt auf ihre Gültigkeit hin überprüft werden. Diese Überprüfungen werden hier – wieder aus Gründen der Übersichtlichkeit – nicht dargestellt. Hier die Klasse `TableViewController`:

```

// Datei: TableViewController.java
public class TableViewController implements IControllerInterface
{
    private IModelInterface m_model;
    private TableView m_tableView;
    public TableViewController (IModelInterface model,
                               TableView tableView)
    {
        m_tableView = tableView;
        m_model = model;
    }

    public void setValues (String red, String green, String blue)
    {
        int r = 0;
        int b = 0;
        int g = 0;
        . . .
        b = Integer.parseInt (blue);
        . . .
        g = Integer.parseInt (green);
        . . .
        r = Integer.parseInt (red);
        . . .
        m_model.setBlueValue (b);
        m_model.setGreenValue (g);
        m_model.setRedValue (r);
        . . .
    }
}

```

## Anwendung

Die Klasse `MVCTestDrive` dient lediglich dazu, ein Datenmodell und seine Views zu erzeugen:

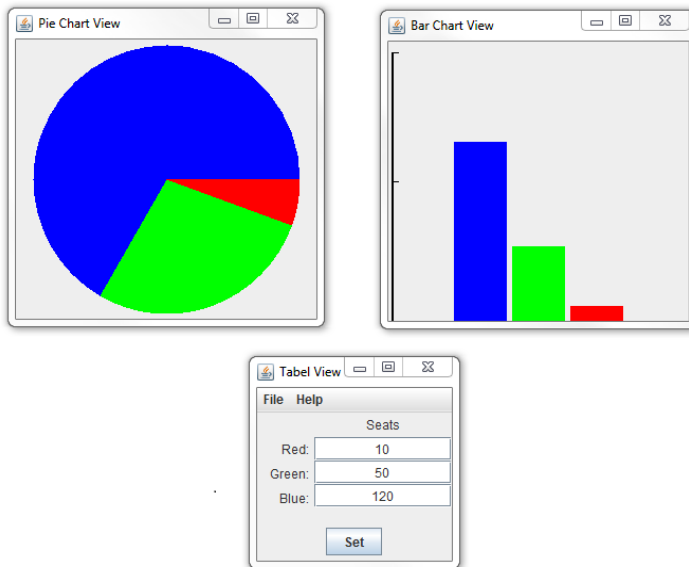
```

// Datei: MVCTestDrive.java
public class MVCTestDrive
{
    public static void main (String[] args)
    {
        DataModel model = new DataModel();
        TableView tableView = new TableView (model);
        PieChartView pieChartView = new PieChartView (model);
    }
}

```

```
BarChartView barChartView = new BarChartView (model);  
}  
}
```

Das fertige Programm erzeugt die in Bild 5-45 gezeigte Bildschirmausgabe:



*Bild 5-45 Beispielprogramm für das MVC-Architekturmuster*

## 5.7 Zusammenfassung

Mit Architekturmustern können Systeme in Systemkomponenten zerlegt werden. Im Gegensatz zu Entwurfsmustern sind Architekturmuster grobkörniger. Ein Architekturmuster kann mehrere verschiedene Entwurfsmuster enthalten, muss es aber nicht (siehe beispielsweise das Muster Layers). Die behandelten Architekturmuster sind im Folgenden kurz zusammengefasst:

- **Layers** (siehe Kapitel 5.1): Dieses Muster wird verwendet, wenn die hohe Komplexität eines Systems durch die Einführung mehrerer horizontaler Schichten vereinfacht werden soll. Dabei enthält eine Schicht die Abstraktion einer bestimmten Funktionalität. Eine höhere Schicht darf generell nur die Services der untergeordneten Schicht aufrufen. Zwischen den Schichten existieren feste Verträge/Schnittstellen.
- **Pipes and Filters** (siehe Kapitel 5.2): Ein System, das einen Datenstrom in einer Richtung verarbeitet, kann mit Hilfe dieses Musters in Komponenten strukturiert werden. Die Aufgabe des gesamten Systems ist in einzelne sequenzielle Verarbeitungsstufen zerlegt. Jeder Verarbeitungsschritt wird in Form einer Filter-Komponente implementiert. Jeweils zwei Filter werden durch eine Pipe verbunden, wobei die Ausgabe der einen Komponente die Eingabe der anderen Komponente ist.
- **Plug-in** (siehe Kapitel 5.3): Das Muster Plug-in bietet einen Lösungsansatz für Anwendungssoftware, die auch zur Laufzeit flexibel erweiterbar sein soll. Das Muster strukturiert eine Anwendungssoftware so, dass sie über Erweiterungspunkte (Schnittstellen) verfügt, an deren Stelle zur Laufzeit Komponenten (Plug-ins) treten können, die diese Schnittstelle implementieren. Ist die Schnittstelle offengelegt, können auch Dritte erweiterte Funktionalität für die Anwendungssoftware bereitstellen.
- **Broker** (siehe Kapitel 5.4): Das Broker-Muster entkoppelt in verteilten Softwaresystemarchitekturen Client- und Server-Komponenten, indem zwischen diesen Komponenten ein verteilter Broker als Zwischenschicht eingeschoben wird und Client- und Server-Komponenten untereinander nur über den Broker kommunizieren. Ein Broker verbirgt den physischen Ort der Leistungserbringung eines Servers vor dem Client. Der Broker muss aber die Server bzw. deren Dienste kennen. Server müssen sich daher bei dem Broker anmelden, um ihre Dienste anbieten zu können. Clients können dann die angebotenen Dienste über den Broker nutzen.
- **Service-Oriented Architecture** (siehe Kapitel 5.5): Häufig driften Geschäftsprozesse und IT-Lösungen auseinander. Mit einer serviceorientierten Architektur wird die geschäftsprozessorientierte Sicht der Verarbeitungsfunktionen eines Unternehmens direkt auf die Architektur eines DV-Systems abgebildet. In sich abgeschlossene Dienste (Anwendungsservices) in Form von Komponenten entsprechen Teilen eines Geschäftsprozesses.
- **Model-View-Controller** (siehe Kapitel 5.6): Das Muster MVC beschreibt, wie die Verarbeitung/Datenhaltung (Model), deren Darstellung (View) und die Eingaben des Benutzers (Controller) so weit wie möglich voneinander getrennt werden und dennoch einfach miteinander kommunizieren können. Dieses Muster erlaubt es, das Model beizubehalten und die Oberflächenkomponenten View und Controller auszutauschen.

## 5.8 Aufgaben

### Aufgaben 5.8.1: Layers

- 5.8.1.1 Erklären Sie die Grundzüge des Layers-Musters.
- 5.8.1.2 Kann eine Schicht  $n$  die Dienste einer Schicht  $n-2$  direkt nutzen?

### Aufgaben 5.8.2: Pipes and Filters

- 5.8.2.1 Erklären Sie die Grundzüge des Pipes and Filters-Musters.
- 5.8.2.2 Was ist der Unterschied zwischen einem aktiven und einem passiven Filter?

### Aufgaben 5.8.3: Plug-in

- 5.8.3.1 Erklären Sie die Grundzüge des Plug-in-Musters.
- 5.8.3.2 Welche Aufgaben hat ein Plug-in-Manager?

### Aufgaben 5.8.4: Broker

- 5.8.4.1 Erklären Sie die Grundzüge des Broker-Musters.
- 5.8.4.2 Welche Rollen spielen ein Client-side Proxy und ein Server-side Proxy im Rahmen des Broker-Musters?

### Aufgaben 5.8.5: Service-Oriented Architecture

- 5.8.5.1 Erklären Sie die Grundzüge einer Service-Oriented Architecture.
- 5.8.5.2 Erläutern Sie die Rollen von Serviceanbieter, Servicenutzer und Serviceverzeichnis.

### Aufgaben 5.8.6: Model-View-Controller

- 5.8.6.1 Erklären Sie die Grundzüge des Model-View-Controller-Musters.
- 5.8.6.2 Welche Abhängigkeiten bestehen zwischen View, Controller und Model?
- 5.8.6.3 Welche Schwachstellen hatten Systeme üblicherweise in der Zeit vor MVC?
- 5.8.6.4 Nennen Sie die Aufgaben des Model im Pull-Betrieb des Active Model.
- 5.8.6.5 Nennen Sie die Aufgaben der View im Pull-Betrieb des Active Model.
- 5.8.6.6 Nennen Sie die Aufgaben des Controllers im Pull-Betrieb des Active Model.

# Literaturverzeichnis

Abkürzungen erhalten bei Büchern 5 Zeichen. Die ersten drei werden aus dem ersten Namen der Autoren gebildet, wobei das erste Zeichen groß geschrieben wird. Ist das Erscheinungsjahr bekannt, so sind die Zeichen 4 und 5 die letzten beiden Ziffern des Erscheinungsjahrs. Gibt es von einem Autor mehrere Veröffentlichungen im selben Jahr, so wird sein Name in der 3. Stelle eindeutig abgeändert.

Bei privater Mitteilung oder Veröffentlichungen des Internets treten anstelle der beiden letzten Ziffern Buchstaben. Nationale oder internationale Standards mit einer Nummerierung werden mit der entsprechenden vollen Abkürzung genannt, auch wenn es Internetquellen sind. Der Name von Internetquellen, die keine Standards sind, ohne Jahreszahl besteht aus 6 klein geschriebenen Zeichen.

- Ale77      Alexander, Ch., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: "A Pattern Language: Towns, Buildings, Construction". Oxford University Press, 1977.
- Bäu97      Bäumer, D. et al.: "The Role Object Pattern". PLoP 97 (4th Conference on Pattern Languages of Programming, Washington University, 1997). <http://hillside.net/plop/plop97/Proceedings/riehle.pdf> (Stand: 17.05.2013). Zu finden auch auf der CD zu [Gam11].
- Bec08      Beck, K.: "Implementation Patterns: Der Weg zu einfacherer und kostengünstigerer Programmierung". Addison-Wesley, München, 2008.
- Bie00      Bienhaus, D.: "Muster-orientierter Ansatz zur einfacheren Realisierung Verteilter Systeme". Tectum Verlag, 2000.
- Boo95      Booch, G.: "Object Solutions: Managing the Object-Oriented Project". Addison-Wesley, Boston, 1995.
- Bus98      Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. und Stal, M.: "Pattern-orientierte Software-Architektur: Ein Pattern-System". Addison-Wesley-Longman, München, 1998.
- db2cop      IBM DB2 Infocenter zum Thema Connection Pooling: <http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/conn/c0006170.htm>. (Stand: 17.05.2013)
- DeM79      DeMarco, T.: "Structured Analysis and System Specification". Prentice Hall, Englewood Cliffs, 1979.
- Dij68      Dijkstra, E.: "The Structure of the "THE"-Multiprogramming System". Comm. ACM, Vol. 11, No. 5, May 1968.

- Dij82 Dijkstra, E.: "On the role of scientific thought", in: Dijkstra, E.: "Selected Writings on Computing: A Personal Perspective". Springer-Verlag, 1982. pp. 60-66. Siehe auch:  
<http://www.cs.utexas.edu/~EWD/ewd04xx/EWD447.PDF>.  
(Stand: 17.05.2013)
- Dou02 Douglass, B.: "Real Time Design Patterns: Robust Scalable Architecture for Real-time Systems". Addison-Wesley Longman, Amsterdam, 2002.
- Eil07 Eilebrecht, K., Starke, G.: "Patterns kompakt - Entwurfsmuster für effektive Software-Entwicklung". 2. Auflage, Spektrum Akademischer Verlag, Heidelberg, 2007.
- Fie00 Fielding R.: "Architectural Styles and Design of Network-based Software Architectures". Doctoral dissertation, University of California, Irvine, 2000.  
<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.  
(Stand: 17.05.2013)
- Fow03 Fowler, M.: "Patterns für Enterprise Application-Architekturen". mitp-Verlag, Heidelberg, 2003.
- Fow97 Fowler, M.: "Dealing with Roles". PLoP 97 (4th Conference on Pattern Languages of Programming, Washington University, 1997).  
<http://martinfowler.com/articles.html> (Stand: 17.05.2013)
- fowioc Fowler, M.: "Inversion of Control Containers and the Dependency Injection Pattern", <http://www.martinfowler.com/articles/injection.html>.  
(Stand: 17.05.2013)
- Gal03 Gall, H., Hauswirth, M., Dustdar, S.: "Software-Architekturen für Verteilte Systeme: Prinzipien, Bausteine und Standardarchitekturen für moderne Software". Springer, Berlin, 2003.
- Gam11 Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software". 6. Auflage, Addison-Wesley, 2011.
- Gam95 Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, Amsterdam, 1995.
- Gol11 Goll, J.: "Methoden und Architekturen der Softwaretechnik". Teubner, Wiesbaden, 2011.
- Gol12 Goll, J.: "Methoden des Software Engineering". Teubner, Wiesbaden, 2012.



- Gr102 Grand, M.: "Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML". 2. Auflage, John Wiley & Sons; 2002.
- Gr302 Grand M.: Patterns in Java, Volume 3: Java Enterprise Design Patterns. John Wiley & Sons, 2002.
- Hei10 Heinisch, C., Müller-Hofmann, F., Goll, J.: "Java als erste Programmiersprache – Vom Einsteiger zum Profi". 6. Auflage, Teubner, Wiesbaden, 2010.
- Hit05 Hitz, M., Kappel, G., Kapsammer, E., Retschitzegger, W.: "UML@Work – Objektorientierte Modellierung mit UML 2". 3. Auflage, dpunkt.verlag, Heidelberg, 2005.
- IBMWSI IBM, "An overview of the Web Services Inspection Language". <http://www.ibm.com/developerworks/library/ws-wslover/>. (Stand 17.05.2013)
- IEEE 1471 IEEE Std 1471-2000: "Recommended Practice for Architectural Description of Software-Intensive Systems". IEEE Computer Society, New York, 2000.
- Kin09 Kindel, O., Friedrich, M.: "Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis". 1. Auflage, dpunkt Verlag, 2009.
- Lah09 Lahres, B., Rayman, G.: "Objektorientierte Programmierung". 2. Auflage, Galileo Computing, 2009.
- Lis87 Liskov, B.: "Data Abstraction and Hierarchy". ACM SIGPLAN Notices, 23(5): pp. 17-34, May 1987.
- Mar03 Martin, R.C.: "Agile Software Development: Principles, Patterns and Practises". Pearson Education, 2003.
- mardip Martin, R.C.: "The Dependency Inversion Principle". <http://www.objectmentor.com/resources/articles/dip.pdf>. (Stand: 17.05.2013)
- marisp Martin, R.C.: "The Interface Segregation Principle". <http://www.objectmentor.com/resources/articles/isp.pdf>. (Stand: 17.05.2013)
- marsrp Martin, R.C.: "SRP: The Single Responsibility Principle". <http://www.objectmentor.com/resources/articles/srp.pdf>. (Stand: 17.05.2013)

- Mes07 Meszaros, G.: "xUnit Test Patterns: Refactoring Test Code". Addison-Wesley, Boston, 2007.
- Mey97 Meyer, B.: "Object-oriented Software Construction". 2. Auflage, Prentice-Hall International, Englewood Cliffs, 1997.
- OASUDD OASIS, "UDDI Specifications TC – Committee Specifications"  
<http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm>.  
(Stand: 17.05.2013)
- OMGCOB "CORBA Basics".  
<http://www.omg.org/gettingstarted/corbafaq.htm>.  
(Stand: 17.05.2013)
- OMGCOS "Common Object Request Broker Architecture: Core Specification".  
<http://www.omg.org/cgi-bin/doc?formal/04-03-12.pdf>.  
(Stand: 17.05.2013)
- OMGUSS "OMG Unified Modeling Language (OMG UML), Superstructure Version 2.4", OMG Document Number: ptc/2010-11-14.  
<http://www.omg.org/spec/UML/2.4/Superstructure/PDF>, 2011.  
(Stand: 17.05.2013)
- Ros90 Rose, M.T.: "The Open Book - A Practical Perspective on OSI". Prentice Hall International, Englewood Cliffs, 1990.
- Ros93 Rose, M.T.: "Verwaltung von TCP/IP-Netzen: Netzwerkverwaltung und das Simple Network Management Protocol (SNMP)". Carl Hanser Verlag, München, 1993.
- Sch00 Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: "Pattern-Oriented Software-Architecture Volume 2: Patterns for Concurrent and Networked Objects". 2. Auflage, John Wiley & Sons, New York, 2000.
- shespi Sherzad, R.: "Singleton Pattern in Java".  
<http://www.theserverside.de/singleton-pattern-in-java>.  
(Stand: 17.05.2013)
- Sta02 Starke, G.: "Effektive Software-Architekturen". Hanser Fachbuch, München, 2002.
- Sta11 Starke, G., Hruschka, P.: "Software-Architektur kompakt". 2. Auflage, Spektrum Akademischer Verlag, Heidelberg, 2011.
- Tan09 Tanenbaum, A.S.: "Modern Operating Systems". 3. Auflage, Pearson Education, 2009.

- W3SOAP "Simple Object Access Protocol (SOAP)".  
<http://www.w3.org/TR/soap/>.  
(Stand: 17.05.2013)
- W3WSDL "Web Services Description Language (WSDL) 1.1".  
<http://www.w3.org/TR/wsdl>.  
(Stand: 17.05.2013)
- W3XMLS "XML Schema 1.1".  
<http://www.w3.org/XML/Schema>.  
(Stand: 17.05.2013)

# Index

abstrakte Klasse/Schnittstelle	68	Erzeugung von Objekten	31
Abstraktion	7	Dependency Injection	37
Active Object	162	Dependency Inversion	146
Adapter	79, 97, 144	Dependency Look-Up	32
Architekt		Design by Contract	14
analytische Aufgaben	51	Double-Dispatch	215
konstruktive Aufgaben	52	Entwurf durch Verträge	15
Architektur	46	Entwurfsmuster	50, 68, 289
~muster	68, 288	klassenbasiert	75
Adaptive Systeme	288	Mikroarchitektur	69
Administration	54	Erzeugungsmuster	74
Ausnahmebehandlung	54	Fabrik	
Datenhaltungskonzept	53	Abstrakte~	123, 250, 251
Definition	46	Statische~	261
Dynamik	47	Fabrikmethode	243, 261, 280
Ein- und Ausgabekonzept	54	Fassade	87, 116, 260, 295
Entwurfsmuster	54	Forwarder-Receiver	338
IEEE 1471	47	Generalisierung	67
Interaktive Systeme	288	Idiom	68
Muster Architekturmuster	49	Information Hiding	7
Muster Entwurfsmuster	49	Interface Segregation-Prinzip	10
Muster Referenzarchitektur	49	Invariante	15, 17
Prototypen	56	Inversion of Control	28, 164
Prozesskonzept	53	Iterator	230, 231
Qualitäten	47	extern	232
Sicherheitskonzept	53	intern	232
Sichten	55	robust	240
Statik	47	JAX-RS	370
Struktur eines Systems	288	JAX-WS	358
Testkonzept	54	Kapselung	6
Verteilte Systeme	288	Klasse	
Assembly-Parts	136	Vertrag einer ~	15
Befehl	133, 153	Klasseninvariante	15
Beobachter	163, 191	Kommunikationssystem	
Besucher	114, 214	Software ISO/OSI-Modell	300
Broker	190, 326, 357	Komponente	
Brücke	87, 88	Architektur~	46
Callback	30	Kompositum	114, 124
Client-Dispatcher-Server	338	Konfigurationsdatei	39
Command Processor	162	Korrektheit	5
Composite-Message	133	Layer Bridging	291, 294
Container	33	Layers	290, 312
Controller	377, 382	liskovsches Substitutionsprinzip	12
Dekorierer	98, 135, 144, 180, 230, 312	Loose Coupling	11
Delegationsprinzip	66, 125, 387	Microkernel	319
Dependency		Model	377, 380

Model 2	389	Service-Verzeichnis	354
Model-View-Controller	133, 377	Single Responsibility-Prinzip	9
Beobachter	385	Singleton	262, 280
Kompositum	386	SOA	351
Strategie	387	Anwendungsservice	352
Muster		UDDI	364
feinkörnig	68	SOLID	4
grobkörnig	68	Spezialisierung	67
Nachbedingung	15, 16	Strategie	114, 152, 162, 174, 201, 319
Objektpool	271, 272	Strict Layering	291
Open-Closed-Prinzip	20	Strukturmuster	74
Pipes and Filters	115, 303, 304	Thread Pool	280
Plug-in	313	User Interface	377
Postcondition	<i>Siehe</i> Nachbedingung	Verhaltensmuster	74
Precondition	<i>Siehe</i> Vorbedingung	Vermittler	181, 326, 338
Properties-Datei	40	Verständlichkeit	6
Proxy	87, 114, 137	Vertrag	15
Registry	33	Invarianten	15
Rolle	202	Nachbedingungen	15
Schablonenmethode	145, 180	überschreibende Methode	18
Schichtenmodell		Vorbedingungen	15
Rechner	295	View	377, 381
Software Client/Server	298	Vorbedingung	15, 16
Software Drei-Schichten-Architektur	300	Web Services Description Language	362
Software Einrechner-System	298	Weiterentwickelbarkeit	5
Software Zwei-Schichten-Architektur	298	Whole-Part	135
vertikale Strukturen	303	Zusicherung	15
Schnittstelle/abstrakte Klasse	68	Invarianten	15, 17
Separation of Concerns	8	Nachbedingungen	15
Service-Oriented Architecture	351	Vorbedingungen	15
Service-Registry	354	Zustand	180, 192, 213