

Joachim Goll
Micha Koller
Michael Watzko

Architektur- und Entwurfsmuster der Softwaretechnik

Mit lauffähigen Beispielen in Java

3. Auflage



Springer Vieweg

Architektur- und Entwurfsmuster der Softwaretechnik

Joachim Goll · Micha Koller · Michael Watzko

Architektur- und Entwurfsmuster der Softwaretechnik

Mit lauffähigen Beispielen in Java

3. Auflage



Springer Vieweg

Joachim Goll
IT-Designers Gruppe
Esslingen, Deutschland

Micha Koller
Mercedes-Benz AG
Stuttgart, Deutschland

Michael Watzko
IT-Designers Gruppe
Esslingen, Deutschland

ISBN 978-3-658-42383-4 ISBN 978-3-658-42384-1 (eBook)
<https://doi.org/10.1007/978-3-658-42384-1>

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

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2013, 2014, 2023

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 allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Planung/Lektorat: Leonardo Milla
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Das Papier dieses Produkts ist recyclebar.

Vorwort

Entwurfsmuster haben seit dem Buch "Design Patterns: Elements of Reusable Object-Oriented Software" von Gamma, Helm, Johnson und Vlissides [Gam94] einen ungeahnten Aufschwung in der Softwaretechnik erfahren. Sie stellen eine Technik dar, die es erlaubt, dass Entwickler sich in einer einfachen, zeitsparenden und standardisierten Weise über die Architektur¹ eines Systems austauschen können. Dabei haben sich zahlreiche wertvolle Muster herausgebildet, die bei verschiedenartigen fachlichen Problemen eingesetzt werden können. Entwurfsmuster spezifizieren Abstraktionen zur Lösung spezieller Probleme, welche bei einer objektorientierten Vorgehensweise durch konkrete Klassen und Instanzen umsetzbar sind.²

Grady Booch schreibt in seinem Geleitwort zu dem Entwurfsmusterbuch von Gamma et al.:

"Konzentriert man sich bei der Systementwicklung auf die Zusammenarbeit zwischen den Objekten, so kann man zu einer Architektur gelangen, die einfacher, weniger umfangreich und sehr viel leichter ist als eine Architektur, bei der man die Muster ignoriert hat."

Entwurfsmuster lösen jeweils spezifische Entwurfsprobleme. Sie machen objekt-orientierte Entwürfe erst wieder verwendbar.

Oftmals erreicht man die Flexibilität der Muster durch die Einführung zusätzlicher Ebenen. Dies erhöht die Komplexität und kann dadurch die Performance verschlechtern. Ein solches Muster sollte nur dann verwendet werden, wenn dessen Flexibilität benötigt wird.

Im vorliegenden Buch werden in der Beschreibung der Muster nur objektorientierte Lösungen beschrieben. Die statischen Beziehungen zwischen den beteiligten Klassen werden in Form von Klassendiagrammen und das dynamische Verhalten bei der Zusammenarbeit der beteiligten Objekte in Form von Sequenzdiagrammen der Unified Modeling Language (UML) beschrieben. Lauffähige Beispiele in Java sollen das Verständnis der Muster erleichtern. Sind Grundkenntnisse in Java und UML vorhanden, so ist das vorliegende Buch für einen Leser besonders einfach zu verstehen. Aus Platzgründen kann nicht jedes Beispiel des Buchs vollständig abgedruckt werden. Auf dem begleitenden Webauftritt sind aber alle Beispiele vollständig dargestellt.

"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 Vorsichtssymbol () gekennzeichnet sind, zeigen Fallen und typische, gern begangene Fehler an, die in der Praxis oft zu einer langwierigen

¹ Eine Architektur beschreibt die Struktur eines Systems und seine Abläufe.

² Es gibt auch nicht objektorientierte Lösungen.

Fehlersuche führen – oder noch schlimmer – erst im Endprodukt beim Kunden erkannt werden.

Jedes Kapitel enthält einfache Kontrollfragen. 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.

Schreibweisen

In diesem Buch sind 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

Alle Beispielprogramme des Buches und die Lösungen der Aufgaben sind auf dem begleitenden Webauftritt unter link.springer.com zu finden, damit Sie diese bequem selbst bearbeiten können. Eine mögliche Errata-Liste mit Korrekturen zum Buch wird ebenfalls dort zu finden sein.

Danksagung

Herrn Prof. Dr. Manfred Dausmann – ehemaliger Mitautor – danken wir herzlich für seine tatkräftige, fundierte Unterstützung sowie Herrn Jordanis Andrianidis für die professionelle Durchführung des Konfigurationsmanagements.

Esslingen, im August 2023

Joachim Goll, Micha Koller und Michael Watzko

Wegweiser durch das Buch

Die nachfolgenden kurzen Inhaltsbeschreibungen der einzelnen Kapitel sollen dem Leser den Weg durch das Buch erleichtern:

- **Kapitel 1** erklärt den Begriff einer Softwarearchitektur und deren nicht funktionale Qualitäten. Analytische und konstruktive Aufgaben bei der Konzeption einer Softwarearchitektur, die Rolle eines Softwarearchitekten und Entscheidungen beim Bau einer Softwarearchitektur werden diskutiert.
- **Kapitel 2** betrachtet Muster beim Softwareentwurf und vergleicht die Eigenschaften von Architekturmustern, Entwurfsmustern und Idiomen³. Die Konzepte von klassenbasierten und objektorientierten Entwurfsmustern werden verglichen. Ein Schema zur Beschreibung von Entwurfs- und Architekturmustern schließt dieses Kapitel ab.
- **Kapitel 3** erläutert Entwurfs- und Konstruktionsprinzipien, die in den Mustern umgesetzt werden.
- **Kapitel 4** untersucht Entwurfsmuster und beschränkt sich dabei auf Strukturmuster, Verhaltensmuster und Erzeugungsmuster. Für den Einstieg in Entwurfsmuster werden leicht zu erlernende Entwurfsmuster vorgestellt.
- **Kapitel 5** behandelt die Strukturmuster Adapter, Brücke, Dekorierer, Fassade, Kompositum und Proxy.
- In **Kapitel 6** werden die Verhaltensmuster Befehl, Beobachter, Besucher, Iterator, Memento, Rolle, Schablonenmethode, Strategie, Vermittler und Zustand diskutiert.
- **Kapitel 7** stellt die Erzeugungsmuster Fabrikmethode, Abstrakte Fabrik, Singleton und Objektpool vor.
- **Kapitel 8** befasst sich mit Architekturmustern der folgenden Kategorien
 - Strukturierung eines Systems,
 - adaptierbare Systeme,
 - verteilte Systeme und
 - interaktive Systeme.
- **Kapitel 9** charakterisiert die Architekturmuster Layers sowie Pipes and Filters als Beispiele für die Kategorie "Strukturierung eines Systems".
- **Kapitel 10** beschreibt das Muster Plug-in für die Kategorie "adaptierbare Systeme".
- **Kapitel 11** studiert die Architekturmuster Broker und Service-orientierte Architektur für die Kategorie "verteilte Systeme".
- **Kapitel 12** analysiert das Architekturmuster Model-View-Controller als Beispiel für "interaktive Systeme".

³ Ein Idiom ist in ein Muster in einer bestimmten Programmiersprache wie z. B. ein ausprogrammiertes Entwurfsmuster.

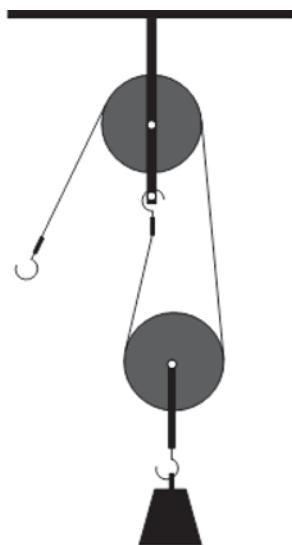
Inhaltsverzeichnis

1	Softwarearchitekturen.....	2
1.1	Ziele einer Softwarearchitektur	3
1.2	Definition des Begriffs "Softwarearchitektur"	4
1.3	Nicht funktionale Qualitäten einer Softwarearchitektur	6
1.4	Analytische und konstruktive Aufgaben bei der Konzeption einer Softwarearchitektur.....	7
1.5	Rolle eines Softwarearchitekten	13
1.6	Entscheidungen beim Bau einer Softwarearchitektur	16
1.7	Zusammenfassung	17
1.8	Kontrollfragen	18
2	Muster	20
2.1	Muster beim Entwurf als bewährte "Blaupausen"	21
2.2	Architektur-, Entwurfsmuster und Idiome.....	22
2.3	Klassenbasierte und objektbasierte Entwurfsmuster	26
2.4	Schema zur Beschreibung von Entwurfs- und Architekturmustern.....	27
2.5	Zusammenfassung	29
2.6	Kontrollfragen	30
3	Entwurfs- und Konstruktionsprinzipien zur Realisierung von Mustern	32
3.1	Wichtige Entwurfs- und Konstruktionsprinzipien.....	32
3.2	Wichtige Entwurfsprinzipien nach Gamma et al.	39
3.3	Zusammenfassung	48
3.4	Kontrollfragen	49
4	Übersicht über den behandelten Katalog von Entwurfsmustern	52
4.1	Strukturmuster	54
4.2	Verhaltensmuster	55
4.3	Erzeugungsmuster	56
4.4	Leicht zu erlernende Entwurfsmuster	57
4.5	Zusammenfassung	57
4.6	Kontrollfragen	58
5	Strukturmuster	60
5.1	Das Strukturmuster Adapter	60
5.2	Das Strukturmuster Brücke.....	70
5.3	Das Strukturmuster Dekorierer	82
5.4	Das Strukturmuster Fassade	101
5.5	Das Strukturmuster Kompositum.....	109
5.6	Das Strukturmuster Proxy.....	122
5.7	Zusammenfassung	131
5.8	Kontrollfragen	133
6	Verhaltensmuster	136
6.1	Das Verhaltensmuster Befehl	136
6.2	Das Verhaltensmuster Beobachter.....	149
6.3	Das Verhaltensmuster Besucher	160
6.4	Das Verhaltensmuster Iterator	178
6.5	Das Verhaltensmuster Memento	191

6.6	Das Verhaltensmuster Rolle	203
6.7	Das Verhaltensmuster Schablonenmethode	222
6.8	Das Verhaltensmuster Strategie	230
6.9	Das Verhaltensmuster Vermittler	238
6.10	Das Verhaltensmuster Zustand	248
6.11	Zusammenfassung	258
6.12	Kontrollfragen	260
7	Erzeugungsmuster	264
7.1	Das Erzeugungsmuster Fabrikmethode	264
7.2	Das Erzeugungsmuster Abstrakte Fabrik	273
7.3	Das Erzeugungsmuster Singleton	285
7.4	Das Erzeugungsmuster Objektpool	297
7.5	Zusammenfassung	307
7.6	Kontrollfragen	308
8	Übersicht über die behandelten Architekturmuster	310
8.1	Architekturmuster zur Strukturierung eines Systems	310
8.2	Architekturmuster für adaptierbare Systeme	311
8.3	Architekturmuster für verteilte Systeme	311
8.4	Architekturmuster für interaktive Systeme	312
8.5	Zusammenfassung	312
8.6	Kontrollfragen	313
9	Architekturmuster zur Strukturierung eines Systems	316
9.1	Das Architekturmuster Layers	316
9.2	Das Architekturmuster Pipes and Filters	331
9.3	Zusammenfassung	344
9.4	Kontrollfragen	345
10	Architekturmuster für adaptierbare Systeme	348
10.1	Das Architekturmuster Plug-in	348
10.2	Zusammenfassung	360
10.3	Kontrollfragen	361
11	Architekturmuster für verteilte Systeme	364
11.1	Das Architekturmuster Broker	365
11.2	Das Architekturmuster Service-Oriented Architecture	389
11.3	Zusammenfassung	418
11.4	Kontrollfragen	419
12	Architekturmuster für interaktive Systeme	422
12.1	Das Architekturmuster Model-View-Controller	422
12.2	Zusammenfassung	443
12.3	Kontrollfragen	444
	Literaturverzeichnis	445
	Abkürzungsverzeichnis	449
	Begriffsverzeichnis	451
	Index	471

Kapitel 1

Softwarearchitekturen



- 1.1 Ziele einer Softwarearchitektur
- 1.2 Definition des Begriffs "Softwarearchitektur"
- 1.3 Nicht funktionale Qualitäten einer Softwarearchitektur
- 1.4 Analytische und konstruktive Aufgaben bei der Konzeption einer Softwarearchitektur
- 1.5 Rolle eines Softwarearchitekten
- 1.6 Entscheidungen beim Bau einer Softwarearchitektur
- 1.7 Zusammenfassung
- 1.8 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_1.

1 Softwarearchitekturen

Das vorliegende Buch behandelt **objektorientierte Muster**⁴ für standardisierte **Softwarearchitekturen**.

Während eine **konkrete Architektur**

- neben der Erfüllung **nicht funktionaler** Eigenschaften auch
- **konkreten funktionalen** Anforderungen

genügt, stehen bei einem **Architekturmuster** bzw. **Entwurfsmuster**

- die **typischen funktionalen** und
- die nicht **funktionalen Anforderungen** wie Verständlichkeit, Ausbaufähigkeit und Bewährtheit

im Vordergrund.



Zu Beginn dieses Kapitels wird zuerst analysiert, was der Begriff "Softwarearchitektur" beinhaltet.

Objektorientierte Muster kommen beim Entwurf immer dann ins Spiel, wenn es bewährte Basisklassen für die Lösung bestimmter Probleme gibt, an deren Stelle im konkreten Fall abgeleitete Klassen unter Einhaltung des **Liskovschen Substitutionsprinzips** treten können.

Muster lösen durch ihre Zusammenarbeit eine bestimmte Problemstellung in

- einfacher und damit verständlicher,
- ausbaufähiger,
- stabiler sowie
- bewährter

Weise.

Kapitel 1.1 erklärt die Ziele einer Softwarearchitektur. Kapitel 1.2 gibt eine formale Definition für den Begriff einer Softwarearchitektur. Kapitel 1.3 beleuchtet die beim Entwurf einer Softwarearchitektur erstrebten nicht funktionalen Qualitätseigenschaften. Kapitel 1.4 befasst sich mit wesentlichen analytischen und konstruktiven Aufgaben bei der Konzeption einer Softwarearchitektur. Kapitel 1.5 erläutert die Rolle eines Softwarearchitekten und Kapitel 1.6 betrachtet Entscheidungen beim Bau einer Softwarearchitektur.

⁴ Muster sind Baupläne für Systeme bzw. Teilsysteme. Muster müssen nicht objektorientiert sein. Es ist möglich, Muster auch nicht objektorientiert zu formulieren. Solche Muster kommen insbesondere ohne Vererbung und Polymorphie aus [Bus98, S. 24]. Die in diesem Buch vorgestellten Entwurfsmuster werden allerdings alle in einer objektorientierten Fassung vorgestellt.

1.1 Ziele einer Softwarearchitektur

Eine **Softwarearchitektur** soll die Anwendungsfunktionen eines Systems modellieren und eine **Abstraktion** bereitstellen, wie die Anwendungsfunktionen des Systems aufgebaut sind und wie sie zusammenarbeiten sollen.



Eine **Architektur** bietet Nutzerfunktionen in einer High-Level-Sicht als **Bauplan** ("Blaupause", Schablone) des betreffenden Systems an. Dies betrifft die Struktur des Systems und die Abläufe seiner Funktionen.



Die **Architektur eines Systems** wird als vorläufiges Endergebnis in dem **Entwicklungsschritt Entwurf** konzipiert. Das Ergebnis des Entwurfs bzw. des Designs⁵ soll die Architektur des Systems sein.



Die Architektur modelliert die Funktionen des betrachteten Systems in **Struktur** und **Verhalten** und ist wesentlich verantwortlich für die nicht funktionalen Qualitäten eines Systems wie z. B. dessen Performance.



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



Das folgende Bild soll das Finden der Architektur im Lebenslauf der Softwareentwicklung symbolisieren:

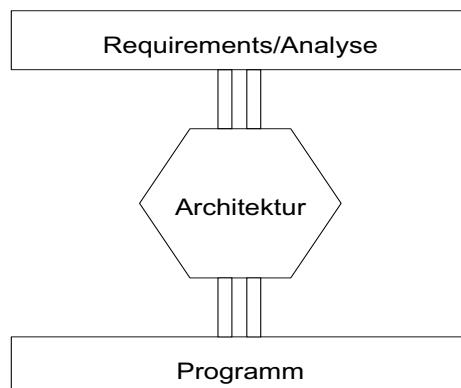


Bild 1-1 Architektur als Bindeglied zwischen Requirements/Analyse und Programm

⁵ In diesem Buch werden die Begriffe Entwurf und Design äquivalent verwendet.

Eine Architektur verknüpft die Anforderungen an das zu realisierende System und die Erkenntnisse der Analyse mit dem Programm.



Genau dieselbe Funktionalität eines Systems kann in unterschiedlichen Architekturen realisiert werden. Jede Architektur hat aber andere Eigenschaften.

Die **Architektur** ist der **Bauplan** für das Programm. Die Architektur einer Anwendung stellt sicher, dass

- **funktionale Anforderungen an das System** und
- **nicht funktionale Anforderungen an das System**, nämlich
 - Anforderungen an Qualitäten wie die Performance, die nicht auf einzelne Funktionen abbildungbar sind, oder
 - Einschränkungen des Lösungsraums⁶

erfüllt werden können.

Eine **Architektur** stellt einen **Bauplan** für die Programme eines Systems dar, welcher eine Abstraktion der Implementierung ist. Die Architektur 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 den Programmcode betrachtet.

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.



Das Finden der passenden Architektur erfolgt oft iterativ.



Die Architektur eines zu entwickelnden Systems beeinflusst wesentlich die Implementierung, den Test, die Integration und die Wartung des Systems.

1.2 Definition des Begriffs "Softwarearchitektur"

Was eine Softwarearchitektur ist, wird in verschiedenen Definitionen nicht einheitlich wiedergegeben. Im Folgenden die Definition, die diesem Buch zugrunde liegt:

⁶ Ein Beispiel hierfür ist die Vorgabe des zu verwendenden Betriebssystems.

Die **Architektur** eines Systems umfasst:

- die statische **Zerlegung** des Systems in seine physischen Bestandteile (Komponenten⁷) und bei verteilten Systemen die Verteilung (das Deployment) dieser Komponenten auf die einzelnen Rechner,
- die **Beschreibung des dynamischen Zusammenwirkens** aller Komponenten sowie mit der Umgebung und
- die Beschreibung der **Strategie für die Architektur**, d. h. wie die Architektur in Statik und Dynamik funktionieren soll⁸,



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

Die Zerlegung eines objektorientierten Systems umfasst seine **Struktur** bzw. seine **Statik** – also, aus welchen Komponenten das System aufgebaut ist –, das **Zusammenwirken** hingegen beinhaltet das **Verhalten** bzw. die **Dynamik** der Objekte.



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

Die Definition in diesem Buch ist ähnlich der von der IEEE in der "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems" festgelegten Definition [IEEE 1471]:

"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".

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

Die Komponenten einer Architektur sollen nach dem Prinzip **Loose Coupling and Strong Cohesion** eine **starke innere Kohäsion** haben, aber **wechselseitig möglichst schwach gekoppelt** sein, wobei die innere Struktur der Komponenten verborgen werden soll (**Information Hiding**).



Bei Vorliegen von Information Hiding muss bei der Änderung einer Komponente nicht das ganze System geändert werden, sondern nur die nach außen sichtbaren Teile der Komponenten. Loose Coupling und Information Hiding der Komponenten werden in Kapitel 3.1 behandelt.

⁷ 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.

⁸ Mit dieser Beschreibung können auch Projektneulinge die Prinzipien der Architektur verstehen.

Die Zusammenarbeit der Komponenten erlaubt es, ein kollektives Verhalten zu erzeugen, das über das Verhalten einer einzelnen Komponente hinausgeht.

1.3 Nicht funktionale Qualitäten einer Softwarearchitektur

Eine Architektur sollte bestimmte, nicht funktionale Anforderungen an die Qualität erfüllen. Zu den **nicht funktionalen Qualitäten** einer Architektur⁹ gehören:

- **eine gute Performance des Systems**

Die Performance eines Systems wird maßgeblich durch die Architektur des Systems und nicht durch einzelne Funktionen bestimmt.

- **Einfachheit und Verständlichkeit der Architektur**

Die Architektur eines Systems muss einfach sein, damit das Entwicklungsteam und der Kunde in der Lage sind, die Charakteristika einer Architektur schnell zu erfassen.¹⁰

- **eine leichte Änderbarkeit und Erweiterbarkeit des Systems**

Mögliche Erweiterungen eines Systems werden auf Basis der Architektur des bereits vorliegenden Systems bewertet und durchgeführt. Eine leichte Änderbarkeit und Erweiterbarkeit eines Systems liegt dann vor, wenn die Komponenten des Systems schwach gekoppelt und in sich selbst stark zusammenhängend sind. Die Änderbarkeit und Erweiterbarkeit eines Systems ist stark von der Architektur eines Systems abhängig.

- **eine leichte Testbarkeit des Systems**

Eine Architektur, die zu einem schlecht zu testenden System führt, ist zu verwerfen (**Design to Test**¹¹). Ein System ist schwer zu testen, wenn die Komponenten einer Architektur stark untereinander gekoppelt sind und deshalb nicht einzeln testbar sind.

- **eine möglichst große Unabhängigkeit der Komponenten**

Eine weitestmögliche Unabhängigkeit der Komponenten erleichtert Änderungen des Systems, da bei erforderlichen Änderungen von Komponenten nicht das ganze System infolge einer starren Kopplung der Komponenten geändert werden muss, sondern die Komponenten einzeln getestet werden können.

- **eine leichte Wartbarkeit des Systems**

Eine leichte Wartbarkeit des Systems erfordert Einfachheit, Verständlichkeit, Änderbarkeit und Erweiterbarkeit der Architektur.

⁹ Diese Qualitäten sind nicht unabhängig voneinander.

¹⁰ Durch Erweiterungen kann eine einfache Architektur aber auch komplexer werden.

¹¹ Siehe Kapitel 3.1.1.

- **eine hohe Stabilität**

Softwarearchitekturen sollen einerseits stabil sein. Dies erfordert, dass die Architektur aus weitgehend unabhängigen Komponenten besteht. Softwarearchitekturen sollen andererseits flexibel sein und Änderungen der Kundenwünsche in einem gewissen Rahmen erlauben. Dies wird vor allem durch die Verwendung von Basisklassen, die abgeleitet werden können, bzw. durch die Implementierung von Schnittstellen wie in Java erreicht.

- **eine einfache 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. Die Architektur eines Systems ist entscheidend für eine gute Skalierbarkeit.

- **Bewährtheit**

Die Verwendung bewährter Lösungen wie Muster verringert das Risiko eines Projekts.

Die genannten **nicht funktionalen Qualitäten** einer Architektur können nicht auf einzelne Anwendungsfunktionen abgebildet werden und sind auch nicht unabhängig voneinander. Es gibt aber auch **funktionale Qualitäten**, die an das Vorhandensein bestimmter Funktionen geknüpft werden. Dies betrifft beispielsweise die Sicherheit oder die Gewährleistung der Parallelität.

1.4 Analytische und konstruktive Aufgaben bei der Konzeption einer Softwarearchitektur

Zunächst werden in Kapitel 1.4.1 die Typen von Architekturen im Hinblick auf ihre Echtzeitfähigkeiten diskutiert. Die Konzeption einer **Systemarchitektur** erfordert zum einen Teil die Durchführung **analytischer Aufgaben** (siehe Kapitel 1.4.2), zum anderen Teil die Durchführung **konstruktiver Aufgaben** (siehe Kapitel 1.4.3). Kapitel 1.4.4 betrachtet den Einsatz von Prototypen beim Bau eines Systems.

1.4.1 Typen von Architekturen

Es gibt verschiedene **Typen von Systemarchitekturen** im Hinblick auf ihre Echtzeitfähigkeit. Der richtige Systemtyp einer Architektur muss zuallererst erkannt werden. Relevant sind:

- **Systeme mit harter Echtzeit¹² wie**

- Steuergeräte (Embedded Systems¹³) eines Autos oder
- die Steuerung einer Walzstraße in einem Stahlwerk.

¹² Harte Echtzeit ist praktisch deterministisch. Wenn die geforderte harte Echtzeit nicht erfüllt ist, tritt ein Schaden ein. Bei weicher Echtzeit kann es einzelne Ausreißer in einem gewissen Maße geben.

¹³ Bei Embedded Systems bringt die Software ein Gerät zum Laufen.

- **Systeme mit weicher Echtzeit wie**

- Multimedia-Geräte (Videoplayer, MP3-Player etc.),
- Videokonferenzsysteme oder
- das Embedded System eines Handy.

- **Systeme ohne Echtzeit wie**

- Buchungssysteme mit Transaktionen und weltweit verteilten Arbeitsplätzen.

Je nach Systemtyp stehen andere Features bei der Architektur im Mittelpunkt.



1.4.2 Analytische Aufgaben bei der Konzeption eines Systems

Analysiert werden müssen:

- **der Einsatzzweck eines neuen Systems und seine Hauptaufgaben**

Hierzu müssen die Anforderungen mit dem Kunden abgestimmt sein.

- **die Einbettung 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 beispielsweise "Drehstuhlschnittstelle"¹⁴ oder Datenschnittstelle (Daten als Schnittstelle zwischen Programmen). Bei den Nutzern interessiert die Art der Arbeitsplätze, die verschiedenen Rollen der Bediener wie beispielsweise Verkäufer, Buchhaltung oder Systemadministrator und die vorgesehenen Interaktionsmöglichkeiten.

- **die technische Sicht der Schnittstellen des Systems zu seiner Umgebung**

Bei der Analyse der Schnittstellen des Systems aus technischer Sicht wird das Innere des Systems selbst nicht betrachtet (Black-Box-Sicht).

- **die Struktur und die Abläufe des Systems aus logischer Sicht**

Dies stellt eine **White-Box-Sicht** des Systems dar. Hierbei werden in der Analyse in der Regel nur die Verarbeitungsfunktionen des Systems¹⁵ betrachtet.

¹⁴ Bei einer sogenannten "Drehstuhlschnittstelle" werden Daten zwischen Systemen manuell ausgetauscht, da die Systeme nicht automatisch miteinander reden können oder dürfen.

¹⁵ Im Lösungsbereich kommen technische Funktionen wie beispielsweise zur Datenhaltung hinzu.

1.4.3 Konstruktive Aufgaben bei der Konzeption eines Systems

Beim Bau eines Systems sind neben **analytischen** auch **konstruktive** Fähigkeiten gefragt.

Bei der **Analyse** wird aus **logischer Sicht** geklärt, was zu tun ist.



Bei der **Konstruktion** wird der Bau eines Systems aus **physischer Sicht** konzipiert und umgesetzt.



Zur Konstruktion gehört die Architektur eines Systems, d. h. 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, kommerzielle und Open-Source-Produkte eingesetzt werden.

Kapitel 1.4.3.1 betrachtet die Zerlegung von Systemen. In Kapitel 1.4.3.2 werden die beim Entwurf zu erstellenden Konzepte betrachtet. Kapitel 1.4.3.3 befasst sich mit der Integration der Software eines einzelnen Rechners und eines verteilten Systems aus einzelnen Rechnern.

1.4.3.1 Zerlegung eines Systems

Bei der Konstruktion eines Systems wird auf der obersten Ebene des Entwurfs zuerst das **Gesamtsystem** aus operationeller Sicht¹⁶ – d. h. aus Sicht der Anwender – betrachtet. Hierbei sind zu untersuchen:

- die **Hardwarearchitektur** des Gesamtsystems,
- die **Software** des Gesamtsystems,
- wie getestet werden soll¹⁷ und
- was ausgeliefert werden soll (**Bereitstellungssicht**)¹⁸.

Bei der **Software des Gesamtsystems** müssen betrachtet werden:

- die **Verteilung der Artefakte** der verschiedenen Funktionalitäten auf die Rechner,
- die **Schichtenstruktur der Software jedes einzelnen Rechners** sowie
- die **Strategie für die Dynamik in der Architektur des Gesamtsystems**.

¹⁶ Man unterscheidet die Software für die Anwender, die sogenannte operationelle Software, und die Software für die Administratoren des Systems, die administrative Software.

¹⁷ Eine Architektur, die nur schwer getestet werden kann, ist nach dem Prinzip Design to Test (siehe Kapitel 3.1.1) zu verwerfen.

¹⁸ Dies umfasst u. a. Rechner-Hardware, Software, Generier-Prozeduren und Dokumentation.

Die Struktur und das Verhalten des operationellen Systems müssen zunächst in groben Zügen festgelegt werden, können aber im weiteren Verlauf noch angepasst 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.



1.4.3.2 Zu erstellende Konzepte beim Entwurf

Die im Folgenden aufgeführten **technischen Konzepte**:

- Prozesskonzept,
- Datenhaltungskonzept,
- Ein- und Ausgabekonzept,
- Sicherheitskonzept,
- Konzept der Ausnahmebehandlung (Exceptions),
- Konzept für die einzusetzenden Entwurfs- und Architekturmuster,
- Konzept für das Management des Systems,
- Testkonzept,
- Integrationskonzept und
- Auslieferungskonzept

haben einen wesentlichen Einfluss auf die Konstruktion des Systems genauso wie die Verwendung von kommerziellen oder Open-Source-Produkten. Die hier genannten technischen Konzepte werden im Folgenden vorgestellt:

● **Prozesskonzept**

Zum Prozesskonzept gehört die Organisation des Systems in parallele Prozesse als Betriebssystem-Prozesse bzw. Threads und das Bereitstellen von Möglichkeiten, welche die vorgesehene Kommunikation/Interprozesskommunikation bzw. Middleware erlauben. Eine eventuell erforderlichen Synchronisation zwischen verschiedenen Prozessen muss ebenfalls betrachtet werden.

● **Datenhaltungskonzept**

Das Datenhaltungskonzept befasst sich mit Themen wie

- Persistenzhaltung von Daten des (verteilten) Systems,
- Caching,
- Transaktionen und
- Redundanz der Datenhaltung.

- **Ein- und Ausgabekonzept**

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

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

- **Sicherheitskonzept**

Zum Sicherheitskonzept gehören:

- eine Analyse der Bedrohungen,
- die Festlegung des erforderlichen Sicherheitsgrades und
- die anzuwendenden Sicherheitsmaßnahmen.

- **Konzept der Ausnahmebehandlung (Exceptions)**

Die Strategie, wie Ausnahmen behandelt werden sollen, ist systemweit festzulegen.

- **Konzept für die einzusetzenden Architektur- und Entwurfsmuster**

Einzusetzende Architektur- und Entwurfsmuster wie z. B. das Architekturmuster Model-View-Controller (siehe Kapitel 12.1) beeinflussen eine Architektur wesentlich und führen zur Wiederverwendbarkeit der Architektur.

- **Konzept für das Management des Systems**

Das Konzept für das Management des Systems befasst sich mit der Konzeption der Funktionen des Systemverwalters. Auch bei einem verteilten System müssen diese Funktionen 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.

- **Testkonzept**

Die Niederschrift des Testplanungsprozesses wird **Testkonzept** genannt. Dieses umfasst

- die Testvorgehensweise,
- die Prioritäten,
- die eingeplanten Ressourcen¹⁹ sowie
- den Zeitplan der beabsichtigten Aktivitäten.

¹⁹ Die Ressourcenplanung soll auf Basis der durchzuführenden Arbeitspakete erstellt werden und soll nicht nur die Mitarbeiter, sondern auch die verwendeten Hilfsmittel wie z. B. Werkzeuge und Testumgebungen umfassen.

- **Integrationskonzept**

Die Integration der Komponenten einschließlich der verwendeten kommerziellen bzw. Open-Source-Produkte muss betrachtet werden.

Kommerzielle und Open-Source-Produkte werden beispielsweise verwendet für

- das Betriebssystem,
- grafische Oberflächen,
- das Erreichen von Persistenz oder
- das Workflow Management.

- **Auslieferungskonzept**

Die Konfiguration und Wartung eines Systems soll mit geringem Aufwand erfolgen.

1.4.3.3 Realisierung der Anwendungssoftware eines einzelnen Rechners und Integration eines verteilten Systems aus mehreren Rechnern

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

- die **externen Schnittstellen** eines Rechners in organisatorischer und technischer Sicht,
- die **Zerlegung des Systems** eines Rechners in Software-Komponenten auf einem Rechner, deren **interne Schnittstellen** und die statischen **Beziehungen zwischen den Komponenten** sowie
- die Modellierung und Umsetzung der Abläufe, d. h. das **Verhalten** des Software-systems eines Rechners. Hierbei werden die Funktionalität der Komponenten und die Abläufe in Kollaborationen, d. h. die dynamische Zusammenarbeit der 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.



Jeder einzelne Rechner kann bottom-up, top-down, middle-out oder zufallsorientiert, d. h. nach der Fertigstellung der entsprechenden Komponenten, integriert werden. Es ist aber auch möglich, dass man einen Use Case eines Rechners nach dem anderen integriert. Nach der Fertigstellung der einzelnen Rechner eines verteilten Systems kann das verteilte Gesamtsystem aus mehreren Rechnern integriert werden.

1.4.4 Prototypen einer Softwarearchitektur

Prototypen können das Risiko auf dem Weg zum System sowohl für den Kunden als auch für die Entwickler mindern.

Es gibt verschiedene Prototypen:

- **horizontale Prototypen**

Horizontale Prototypen demonstrieren die Tauglichkeit einer Schicht wie z. B. des Man-Machine-Interface (abgekürzt zu MMI).

- **vertikale Prototypen**

Vertikale Prototypen sind ein Durchstich durch alle Schichten eines Systems.

- **Realisierbarkeitsprototypen**

Realisierbarkeitsprototypen untersuchen experimentell, ob bestimmte Systemteile realisierbar sind. Was man nicht theoretisch zeigen kann, muss man eben experimentell beweisen.

1.5 Rolle eines Softwarearchitekten

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 bzw. einem Generalunternehmer koordiniert, muss auch ein Softwarearchitekt zwischen Auftraggeber, dem Management und dem Entwicklungsteam des entwickelnden Softwarehauses vermitteln und die Entwickler des Entwicklungsteams fachlich führen können, obwohl er für die disziplinarische Führung des Entwicklungsteams in der Regel nicht zuständig ist.

Ob die Rolle eines Softwarearchitekten auf eine einzige Person projiziert wird oder ob es im Projekt mehrere kooperierende Architekten gibt, hängt letztendlich von der Größe eines Projektes sowie von dem Interesse und den Fähigkeiten der beteiligten Teammitglieder ab.



Als "Drehscheibe" eines neuen Systems hat ein Softwarearchitekt Umgang mit **Stakeholdern**²⁰ jeder Art. Dabei muss eine Architektur mit dem Entwicklungsteam, dem Management und dem Kunden abgestimmt sein und sollte von allen bejaht werden.



²⁰ Siehe Begriffsverzeichnis.

Ein Architekt eines Systems sollte durch Erfahrungen bei vergleichbaren Systemen beeindrucken können. Seine Meinung zu Projektrisiken sollte gefragt sein. Er sollte die **Aufwände** für die Realisierung der verschiedenen **Lösungsvorschläge** abschätzen können. Er sollte das Management und das Entwicklungsteam des Softwarehauses konzeptionell überzeugen können wie auch den Kunden. Er sollte außerdem ein Organisationstalent sein.



Die Entwicklung einer Architektur erfordert von den Architekten:

- **Kenntnis des Problembereichs**

Die fachlichen Aufgaben eines Systems und seines Umfelds müssen vertraut sein.

- **technische Fähigkeiten**

Dies betrifft vor allem, wie ein System konstruiert werden soll.

- **eine ausgeprägte Kommunikationsfähigkeit**

Die übrigen Stakeholder müssen einbezogen werden.

- **Sicherheit bei Schätzungen**

Erforderlich ist die Fähigkeit, das **Risiko, den Aufwand und die Zukunftssicherheit** der Realisierung von Lösungsvorschlägen und von Alternativen einigermaßen zuverlässig **abschätzen zu können**.

1.5.1 Kenntnisse des Problembereichs

Es ist wichtig, zu wissen, wie vergleichbare Systeme derselben **Fachdomäne** technisch umgesetzt werden, falls es solche Systeme überhaupt bereits gibt. Insbesondere müssen die Begriffe eines Fachgebiets verstanden werden.

1.5.2 Technische Fähigkeiten

Ein Architekt sollte über die Fachdomäne hinaus auch über gute **Erfahrungen im Stand der Technik** verfügen wie

- Erfahrungen mit der verwendeten Programmiersprache und dem Betriebssystem,
- Erfahrungen im Bau von Systemen bzw. Prototypen,
- Kenntnis von Altsystemen und fachlich ähnlichen Systemen,
- Erstellung von Modellen in methodischer und standardisierter Weise,
- Kenntnis von Entwurfs- und Architekturmustern,
- Überblick über kommerzielle und Open-Source-Produkte²¹ sowie
- Erfahrungen im Testen.

²¹ Dies betrifft beispielsweise Frameworks, Betriebssysteme oder Compiler.

1.5.3 Kommunikative Fähigkeiten eines Softwarearchitekten

Für den Entwurf einer Architektur erhält ein Softwarearchitekt Vorgaben beispielweise zu

- der erwarteten Funktionalität und zu nicht funktionalen Eigenschaften,
- den Kosten 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**

bei der Umsetzung des Systems und der Betrachtung der Konsequenzen von Anforderungen wie etwa dem Preis,

- **Management und Entwicklern**

bei der Zuweisung von Aufgaben an und zur Steuerung der Entwickler,

- **Entwicklern**

beim gemeinsamen Finden der Architektur und deren Umsetzung durch die Entwickler²²,

- **Systemmanagern/Hardwarelieferanten**

bei der Auswahl der für die Implementierung der beabsichtigten Architektur geeigneten Hardware,

- **dem Testteam**

bei Fragen der Testbarkeit der Architektur und der Funktionalität des Systems.

Eine positive "Bedienung" all der beteiligten Personen 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 die im Team gefundene Architektur betreiben, da er letztendlich alle Stakeholder von dem im Team gefundenen Konzept überzeugen muss, so dass ein jeder die vorgeschlagene Architektur für die beste Lösung hält.

²² Hierbei benötigt der Architekt unbedingt die Rückkopplung aller Mitglieder des Entwicklungsteams. Fachlich gesehen ist der Architekt der wichtigste Ansprechpartner für einen Entwickler.

1.6 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.



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

1.6.1 Technik

Entscheidungen in der **Technik** betreffen beispielsweise:

- **die Struktur des Systems**

Welche Komponenten des Systems mit welchen Verantwortlichkeiten gibt es?

- **die Schnittstellen der Komponenten**

Über welche Abstraktion einer Komponente kann man auf eine bestimmte Komponente zugreifen?

- **die Abläufe der Funktionen**

Wie sieht die Dynamik der Abläufe des Systems aus?

- **die eingesetzten Muster**

Welche Muster sollen verwendet werden?

- **die verwendeten Produkte**

Welche kommerziellen bzw. Open-Source-Produkte wie Betriebssystem, Datenbankmanagementsystem, Frameworks etc. sollen eingesetzt werden?

- **die eingesetzten Tools**

Welche Werkzeuge sollen den Entwicklungsprozess unterstützen?

- **die Strategie für das Testen**

Was soll wann und wie lange mit welchen Methoden und Werkzeugen getestet werden?

Bei der Verwendung von kommerziellen und Open-Source-Produkten sowie Tools muss insbesondere bedacht 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.

1.6.2 Organisation

In der Organisation sind ebenso Entscheidungen fällig wie zum Beispiel:

- **Einsatzplanung**

Ein Architekt muss die Fähigkeiten der Projektmitarbeiter mit den technischen Aufgaben des zu realisierenden Systems korrelieren.

- **Priorisierung der Use Cases**

Oftmals können aus Kostengründen nicht alle Kundenforderungen umgesetzt werden.

- **Kosteneffizienz**

Nicht jede Architektur hat dieselben Kosten bei ihrer Umsetzung.

1.6.3 Entscheidungsdokumentation

Alle relevanten Entscheidungen sind in einer **Entscheidungsdokumentation** einschließlich ihrer Voraussetzungen festzuhalten. Ändern sich die Voraussetzungen im Verlauf des Projekts, können ggf. auch gewisse Entscheidungen mit Hilfe der Entscheidungsdokumentation 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 zukunftssicher erweisen.



1.7 Zusammenfassung

Kapitel 1.1 erklärt die Ziele einer Softwarearchitektur.

Kapitel 1.2 definiert den Begriff einer Softwarearchitektur.

Kapitel 1.3 befasst sich vor allem mit verschiedenen erstrebenswerten, nicht funktionalen Qualitäten einer Architektur wie beispielsweise der Performance oder Testbarkeit.

Kapitel 1.4 diskutiert analytische und konstruktive Aufgaben beim Bau eines Systems und die Erstellung von Prototypen für eine Softwarearchitektur.

Kapitel 1.5 betrachtet die Rolle eines Softwarearchitekten und Kapitel 1.6 Entscheidungen beim Bau einer Softwarearchitektur.

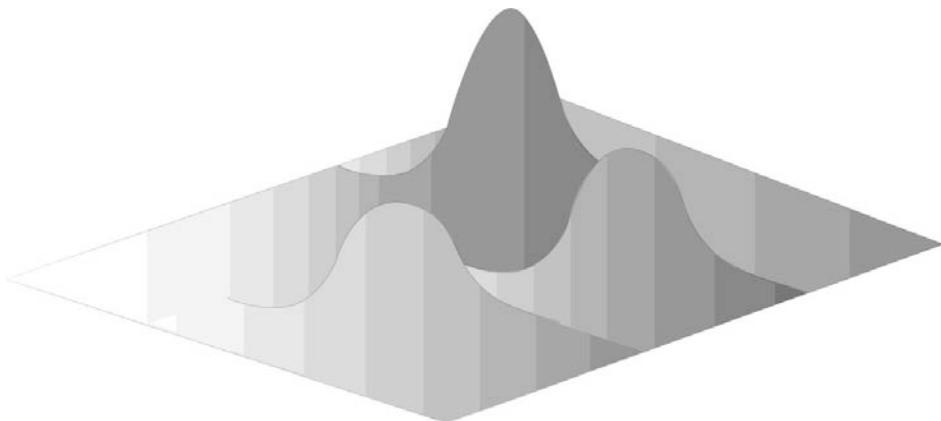
1.8 Kontrollfragen

Aufgaben 1.8.1: Softwarearchitektur allgemein

- 1.8.1.1 Was ist ein Stakeholder?
- 1.8.1.2 Wie lautet die Definition des Begriffs Architektur?
- 1.8.1.3 In welchem Entwicklungsschritt entsteht die Architektur eines Softwaresystems?
- 1.8.1.4 Was ist das Ergebnis des Entwurfs?
- 1.8.1.5 Welche Prototypen gibt es zur Verifikation einer Architektur und wozu werden die einzelnen Typen verwendet?
- 1.8.1.6 Beschreiben Sie das Verhältnis zwischen Programm, Anforderungen/Analyse und Architektur.

Kapitel 2

Muster



- 2.1 Muster beim Entwurf als bewährte "Blaupausen"
- 2.2 Architektur-, Entwurfsmuster und Idiome
- 2.3 Klassenbasierte und objektbasierte Entwurfsmuster
- 2.4 Schema zur Beschreibung von Entwurfs- und Architekturmustern
- 2.5 Zusammenfassung
- 2.6 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_2.

2 Muster

Muster erlauben es generell, mit der Komplexität der Realität durch das Bilden einfacher Abstraktionen fertig zu werden.

In der Softwareentwicklung gibt es beispielsweise

- Analysemuster oder
- Entwurfsmuster.

Die Namen der **Muster** erweitern die Fachsprache der Informatiker und erlauben es geschulten Entwicklern, sich auf einem **hohen Abstraktionsniveau** über ein Problem und seine Lösung verständigen zu können.



Muster für den objektorientierten Entwurf beschreiben die Zusammenarbeit von Klassen und Objekten für die Lösung bestimmter Problemstellungen, die **sich bereits in mehreren Systemen bewährt** haben.



Ein **objektorientiertes Muster** beim Softwareentwurf arbeitet oft mit Abstraktionen, sei es in der Notation einer **abstrakten Klasse** oder einer **Schnittstelle** im klassischen Sinne von Java²³. Die echten bzw. konkreten Klassen sind dann abgeleitete Klassen bzw. Implementierungen. Sie sind dem Muster nicht bekannt. Sie werden vom Nutzer des Musters definiert.



Dieses Kapitel konzentriert sich auf **Muster beim Softwareentwurf**. Dabei unterscheidet man zwischen:

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

Der **Ursprung der Muster beim Entwurf** geht auf den Architekten Christopher Alexander²⁴ zurück. Er hatte in den 70er Jahren des letzten Jahrhunderts 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 [Ale77]:

²³ Seit Java 8 ist es möglich, dass Schnittstellen in Java jetzt auch den Charakter einer abstrakten Klasse haben können und dedizierte Methoden vorgeben.

²⁴ 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.

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

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.

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 2.1 charakterisiert Muster als bewährte "Blaupausen". Kapitel 2.2 differenzierter Architektur-, Entwurfsmuster und Idiome. Kapitel 2.3 betrachtet die Kriterien "klassenbasiert" und "objektbasiert" für Entwurfsmuster. In Kapitel 2.4 wird die Gliederung für die Beschreibung von Entwurfs- und Architekturmustern vorgestellt, welche in diesem Buch verwendet wird, um diese Muster zu dokumentieren.

2.1 Muster beim Entwurf als bewährte "Blaupausen"

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



Objektorientierte **Muster** spezifizieren **Abstraktionen** zur Lösung spezieller Probleme oberhalb der Ausprägung konkreter Klassen oder Instanzen. Ihr Bauplan ist dadurch **für konkrete Fälle umsetzbar**.

Muster beim Entwurf stehen als "**Blaupausen**" (Schablonen) zum Kopieren zur Verfügung. Sie werden von Hand ausprogrammiert, es sei denn, sie werden bereits durch komfortable Entwicklungswerzeuge mit den erforderlichen Methodenköpfen zur Verfügung gestellt.

Muster sind kein wiederverwendbarer Quellcode, der in einer Anwendung einfach übernommen werden kann. Muster sind hingegen weitergereichte Erfahrungen im Entwurf.



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.

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.



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.

2.2 Architektur-, Entwurfsmuster und Idiome

Muster beim Entwurf betreffen Softwarearchitekturen. Sie stellen bewährte Entwurfs erfahrungen dar. Das Verwenden von Mustern kann dabei helfen, die **Entwicklungskosten** und das **Risiko von Projekten** zu senken, da die Muster bereits erprobt sind. Das Finden der passenden Muster stellt daher einen wichtigen Schritt bei der Konzeption der Architektur eines Systems dar.

Ziel von Architektur- und Entwurfsmustern ist es, gewonnene Erkenntnisse über die Lösung bestimmter Problemstellungen wiederverwendbar zu machen und durch ihre Anwendung die Flexibilität eines Entwurfs zu erhöhen.

Beim Entwurf eines Systems sind Architekturmuster, Entwurfsmuster und Idiome voneinander zu unterscheiden:

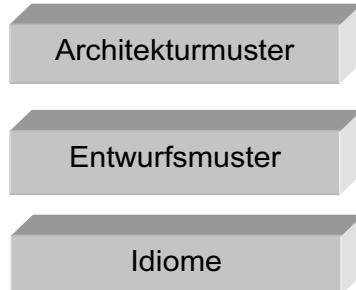


Bild 2-1 Muster bei der SW-Entwicklung

Diese Muster sind charakterisiert durch:

- Bei **Architekturmustern** betrachtet man die Architektur eines Systems.
- **Entwurfsmuster** betreffen in der Regel ein bestimmtes Problem innerhalb eines Subsystems. Ein Entwurfsmuster beeinflusst in der Regel nicht die grundsätzliche Architektur des Systems.
- Ein **Idiom²⁵** kann als Umsetzung eines abstrakten Musters in einer spezifischen Programmiersprache verstanden werden wie beispielsweise ein **ausprogrammiertes Entwurfsmuster**.

Letztendlich enthalten Architektur- und Entwurfsmuster Komponenten in Rollen.

2.2.1 Architekturmuster

Die Architektur eines Systems kann mehrere **Architekturmuster** enthalten, z. B. eine Schichtenarchitektur (engl. Layers)²⁶ für die Anwendungsssoftware verschiedener Rechentypen wie Client- und Server-Rechner oder das Architekturmuster Model-View-Controller²⁷ zur Trennung der Darstellung und der interaktiven Eingabe der Mensch-Maschine-Schnittstelle von dem Daten haltenden und verarbeitenden Model²⁸. Das Architekturmuster einer Schichtenarchitektur (engl. Layers) hingegen enthält keine weiteren Entwurfsmuster.

²⁵ Auf Idiome soll in diesem Buch nicht eingegangen werden.

²⁶ Siehe Kapitel 9.1.

²⁷ Abgekürzt als MVC.

²⁸ Das Model benachrichtigt alle Views, die von ihm abhängig sind, wenn sich seine Daten ändern.

Architekturmuster haben

- die Zerlegung des betrachteten Systems in **erweiterbare Subsysteme** nach einer bestimmten Strategie und
- die Zusammenarbeit der erweiterbaren Subsysteme zum Ziel.



Architekturmuster und Entwurfsmuster sind nicht nur programmiersprachenunabhängig und plattformunabhängig, sondern auch fachgebietsunabhängig.

2.2.2 Entwurfsmuster

Architekturmuster sind gröbere Einheiten als Entwurfsmuster.

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



Entwurfsmuster werden als bewährte Konstruktionsprinzipien im Kleinen eingesetzt.

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

Entwurfsmuster stellen **erweiterbare Klassen** eines Systems in bestimmten Rollen dar. Sie lösen oft ein bestimmtes Problem innerhalb eines Subsystems.



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



Die Idee der Verwendung von Entwurfsmustern in der Softwareentwicklung wurde erstmals 1987 von Kent Beck und Ward Cunningham [Bec87] 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 darauf beruhende Buch "*Design Patterns – Elements of Reusable Object-Oriented Software*" [Gam94] der sogenannten "Gang of Four"²⁹ (abgekürzt: GoF) führte zum flächendeckenden Einsatz der Entwurfsmuster in der Softwareentwicklung.

²⁹ Die "Gang of Four" sind Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. Diese Bezeichnung wurde den vier Autoren scherhaft zugewiesen und ist ohne großen Widerspruch von ihnen angenommen worden.

Nachfolgend zum GoF-Buch entstanden zahlreiche weitere Bücher, die auf diesem Werk aufbauen und dabei neue Entwurfsmuster behandelten. 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.

Arten von Entwurfsmustern

Entwurfsmuster betreffen zum einen Verarbeitungsfunktionen. Das gilt für die GoF-Muster.

Verarbeitungsfunktionen eines Systems gibt es im Problembereich und im Lösungsbereich. Bei Entwurfsmustern interessiert jedoch nur der Lösungsbereich.



Zum anderen betreffen Entwurfsmuster auch die sogenannten technischen Funktionen, die erst beim Entwurf betrachtet werden.

Technische Funktionen sind:

- Funktionen zur Datenhaltung,
- Funktionen zur Ein- und Ausgabe,
- Funktionen zur Rechner-Rechner-Kommunikation,
- Funktionen zur Sicherheit,
- Funktionen zur Parallelität/Interprozesskommunikation.

Natürlich muss es eine im Problembereich bzw. in der Analyse betrachtete Verarbeitungsfunktion in anderer Form auch im Lösungsbereich bzw. dem Entwurf geben, denn ansonsten würde ihre Funktionalität einfach unter den Tisch fallen.

Betrachtet man die Funktionen der **Datenhaltung** unter dem Aspekt der **Entwurfsmuster**, so kommen zum GoF-Buch die **Entwurfsmuster für eine objektrelationale Abbildung** wie z. B. das Data Access Object ins Spiel. **Kommunikationsmuster** werden beispielsweise von Frank Buschmann [Bus98] betrachtet. Bei der **Ein- und Ausgabe** kann etwa das allgemeine Entwurfsmuster Beobachtermuster in den Vordergrund rücken.

Ferner gibt es:

- **Muster zur funktionalen Qualität Sicherheit** wie die **Available System Patterns** oder **Security Patterns** sowie
- Muster zur **funktionalen Qualität Parallelität** wie das Producer/Consumer-Pattern.

2.2.3 Abgrenzung Architektur- und Entwurfsmuster

Einige Autoren unterscheiden nicht zwischen Entwurfsmustern und Architekturmustern.



Da die Architektur eines Systems beim Entwurf festgelegt wird, ist es durchaus berechtigt, auch Architekturmuster als Entwurfsmuster zu bezeichnen. Beispielsweise wird in [Eil07] das MVC-Muster zu den Entwurfsmustern gezählt. In dem vorliegenden Buch wird jedoch zwischen Architektur- und Entwurfsmustern unterschieden.

Kapitel 3 bis 7 in diesem Buch präsentieren Entwurfsmuster und Kapitel 8 bis 12 Architekturmuster verschiedener Kategorien.

2.3 Klassenbasierte und objektbasierte Entwurfsmuster

Generell unterscheidet man klassenbasierte und objektbasierte Entwurfsmuster.

2.3.1 Klassenbasierte Entwurfsmuster

Wer klassenbasiert denkt, versucht, durch **Generalisierung** die höchstmögliche Abstraktion – in anderen Worten die Basisklasse – zu finden, um daraus durch **Spezialisierung zur Kompilierzeit** mit Hilfe der statischen Vererbung die verschiedenen, konkreten Varianten zu gewinnen. Der Typ eines Objekts kann also zur Laufzeit nicht verändert werden.

Einige Beispiele der Muster sind mit abstrakten Basisklassen entworfen, andere jedoch mit Schnittstellen³⁰. 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, welche nur aus Methodenköpfen bestehen, übersichtlicher. Sie enthalten nur abstrakte Operationen.

Dass eine Schnittstelle nur aus Methodenköpfen 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.

2.3.2 Objektbasierte Entwurfsmuster

Bei objektbasierten Entwurfsmustern können Objekte **dynamisch zur Laufzeit** erzeugt werden. Dies erreicht man, indem man gegen Abstraktionen wie Schnittstellen³¹ von

³⁰ Hier meint man mit Schnittstellen Schnittstellen aus Methodenköpfen wie in Java vor Java 8.

³¹ An dieser Stelle ist es irrelevant, ob es sich um ein klassisches Interface von Java oder ein Interface von Java 8 handelt.

Java oder gegen abstrakte Basisklassen programmiert. Bei Einhaltung der Verträge einer Klasse gemäß dem **liskovschen Substitutionsprinzip** (siehe Kapitel 3.1.4) kann dann zur Laufzeit an die Stelle einer Referenz vom Typ einer abstrakten Basisklasse stets eine Referenz vom Typ einer abgeleiteten Klasse treten. Analoges gilt für ein Interface von Java.

Bei den objektbasierten Mustern spielt das **Delegationsprinzip**³² eine wichtige Rolle. Ein Objekt, das eine Nachricht empfängt, interpretiert die eingehende Nachricht und leitet sie an ein anderes Objekt zur Verarbeitung weiter. Das empfangende Objekt "delegiert" also die Verarbeitung der Nachricht an dieses ausführende Objekt. Zur Kompilierzeit kennt die Klasse des empfangenden Objekts nur eine Abstraktion (Schnittstelle bzw. abstrakte Klasse) des ausführenden Objekts. Zur Laufzeit wird dem empfangenden Objekt das ausführende Objekt, welches die Abstraktion implementieren muss, bekannt gemacht, beispielsweise über Dependency Injection (siehe Kapitel 3.1.3).

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.



2.4 Schema zur Beschreibung von Entwurfs- und Architekturmustern

In den folgenden Kapiteln werden Entwurfs- und Architekturmuster nach dem folgenden Schema³³ vorgestellt:

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.

Gute Namen zu finden, ist eine der schwierigsten Aufgaben in der Beschreibung von Entwurfsmustern. Der jeweilige Name sollte sehr sorgfältig und mit Weitblick gewählt werden. Der Name geht in das Vokabular von Software-Entwicklern 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.

³² Der Begriff der Delegation ist nicht eindeutig definiert.

³³ Bei Architekturmustern kann jedoch noch hinzukommen, welche Entwurfsmuster vom Architekturmuster verwendet werden und für welchen Zweck.

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:

- **Klassendiagramm**

Klassendiagramm mit Beschreibung der wechselseitigen statischen Beziehungen (Statik)

- **Teilnehmer**

Rollenbeschreibung der teilnehmenden Klassen

- **Dynamisches Verhalten der Objekte**

Sequenzdiagramm mit Beschreibung der einzelnen Schritte (Dynamik)

- **Programmbeispiel³⁴**

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 zwangsläufig 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 gegeneinander abwägen zu können. Vor- und Nachteile werden 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 werden konkrete Beispiele genannt. Diese Beispiele können optional in einem eigenen Abschnitt erklärt werden.

6. Ähnliche Muster

Hier werden Gemeinsamkeiten und Unterschiede im Vergleich mit ähnlichen Mustern beschrieben.

Alle Muster werden im Folgenden in dieser strukturierten Weise beschrieben. Damit soll sowohl die Verständlichkeit als auch die Vergleichbarkeit gefördert werden.

³⁴ Für Architekturmuster gibt es in diesem Buch aus Platzgründen nicht immer sinnvolle Beispiele, da sinnvolle Beispiele den Umfang des Buches sprengen würden. Manche Beispiele werden im Buch nur verkürzt präsentiert und sind dann vollständig auf dem begleitenden Webauftritt zu finden.

2.5 Zusammenfassung

Entwurfs- und Architekturmuster sollten beim Entwurf von Systemen beachtet werden, da sie Lösungsvorschläge für bestimmte Problemstellungen sind, die sich als Bauplan bereits in mehreren Systemen bewährt haben und dokumentiert sind. Die konkrete Implementierung von Mustern bleibt jedoch den Entwicklern selbst überlassen.

Kapitel **2.1** analysiert Muster beim Entwurf als bewährte "Blaupausen" (Schablonen).

Kapitel **2.2** betrachtet die Unterschiede beim Softwareentwurf zwischen Architekturmustern, Entwurfsmustern und Idiomen.

Kapitel **2.3** erläutert den Unterschied zwischen klassenbasierten und objektbasierten Entwurfsmustern.

Kapitel **2.4** stellt das in diesem Buch verwendete Schema zur Beschreibung von Entwurfs- und Architekturmustern vor.

2.6 Kontrollfragen

Aufgaben 2.6.1: Allgemeine Aufgaben

- 2.6.1.1 Erklären Sie den Zusammenhang zwischen Architekturmustern, Entwurfsmustern und Idiomen.
- 2.6.1.2 Wann wird eine abstrakte Basisklasse einer Schnittstelle aus Methodenköpfen vorgezogen?

Aufgaben 2.6.2: Architektur- und Entwurfsmuster

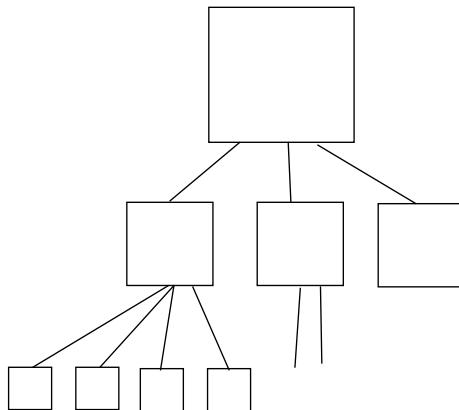
- 2.6.2.1 Was ist ein Architekturmuster?
- 2.6.2.2 Was ist ein Entwurfsmuster?
- 2.6.2.3 Ist ein Architekturmuster feiner als ein Entwurfsmuster?
- 2.6.2.4 Muss ein Architektur- oder Entwurfsmuster objektorientiert sein?

Aufgaben 2.6.3: Entwurfsmuster

- 2.6.3.1 Was ist das Ziel der Entwurfsmuster?
- 2.6.3.2 Wie entsteht ein Entwurfsmuster?
- 2.6.3.3 Kann man sich sicher sein, dass man eine gute Architektur erstellt hat, wenn sie viele Entwurfsmuster enthält?

Kapitel 3

Entwurfs- und Konstruktionsprinzipien zur Realisierung von Mustern



- 3.1 Wichtige Entwurfs- und Konstruktionsprinzipien
- 3.2 Wichtige Entwurfsprinzipien nach Gamma et. al.
- 3.3 Zusammenfassung
- 3.4 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_3.

3 Entwurfs- und Konstruktionsprinzipien zur Realisierung von Mustern

Kapitel 3.1 stellt in Kurzform wichtige Entwurfs- und Konstruktionsprinzipien der Softwaretechnik vor. Kapitel 3.2 befasst sich speziell mit dem von Gamma et al. [Gam94] vorgeschlagenen Entwurfsprinzip "Programmiere gegen Schnittstellen, nicht gegen Implementierungen", welches die Wiederverwendbarkeit von Mustern sicherstellen soll, und dem Prinzip "Ziehe Objektkomposition der Klassenvererbung vor", welches bei polymorphen Programmen zu weniger Abhängigkeiten als die Technik der Vererbung führt. Die Entwurfsprinzipien "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" und "Ziehe Objektkomposition der Klassenvererbung vor" sind eine zentrale Grundlage für das Buch von Gamma et al. [Gam94], der "Gang of Four" (abgekürzt als GoF). Die Bedeutung dieser beiden Prinzipien geht weit über die Entwurfsmuster von Gamma et al. hinaus.

3.1 Wichtige Entwurfs- und Konstruktionsprinzipien

Die im Folgenden aufgeführten, wichtigen Entwurfs- und Konstruktionsprinzipien können eingeteilt werden in³⁵:

- Entwurfsprinzipien zur modularen Struktur von Systemen (siehe Kapitel 3.1.1)
- Entwurfsprinzipien zur Reduktion der Komplexität (siehe Kapitel 3.1.2)
- Entwurfsprinzipien und Konzepte für schwach gekoppelte Teilsysteme (siehe Kapitel 3.1.3)
- Entwurfsprinzipien und Konzepte für korrekte Programme (siehe Kapitel 3.1.4)
- Prinzipien für die Stabilität und Erweiterbarkeit bei Programmänderungen (siehe Kapitel 3.1.5)
- Das Konzept Inversion of Control (siehe Kapitel 3.1.6)

3.1.1 Entwurfsmuster zur modularen Struktur von Systemen

Zu den Entwurfsprinzipien zur modularen Struktur von Systemen gehören nach [Gol19]:

- **Teile und Herrsche**

Nach dem Prinzip **Teile und Herrsche** soll generell ein schwer beherrschbares, komplexes Problem top-down in kleinere, möglichst unabhängige Teilprobleme zerlegt werden, die dann besser verständlich sind und einfacher gelöst werden.

Durch Zusammensetzen der Teillösungen ergibt sich die Lösung des Gesamtproblems.



Die Zerlegung kann rekursiv wiederholt werden, bis ein Teilproblem so klein ist, dass es gelöst werden kann.

³⁵ Eine ausführliche Beschreibung befindet sich in [Gol19].

- **Design to Test**

Nach dem Prinzip **Design to Test** wird eine Architektur so entworfen, dass sie leicht zu testen ist.



Sollte die entworfene Architektur wider Erwarten nicht leicht zu testen sein, so ist die gefundene Architektur zu verwerfen und eine gut testbare Architektur zu entwickeln.

3.1.2 Entwurfsprinzipien zur Reduktion der Komplexität

Im Folgenden werden wichtige Entwurfsprinzipien zur Vermeidung von Komplexität vorgestellt:

- **KISS**

Das Prinzip "**Keep it simple, stupid**" (abgekürzt als **KISS**) bedeutet, dass Systeme möglichst einfach und nicht komplex sein sollen.



- **YAGNI**

Das Prinzip "**You aren't gonna need it**" (abgekürzt als **YAGNI**) fordert, dass

- ein Over-Engineering³⁶ und
- spekulative, vom Kunden nicht geforderte Funktionen wie insbesondere eine spekulative Generalisierung



durch die Entwickler vermieden werden sollen.

Oftmals sind Generalisierungen durch die Entwickler zu voreilig, da sie später überhaupt nicht benötigt werden. Dies wird als **Premature Generalization** bezeichnet. Solche unnötigen Generalisierungen verzögern das ursprüngliche Projekt, erschweren die Ausbaubarkeit und kosten Geld. Auch Generalisierungen müssen vom Kunden gefordert werden. Es ist sinnvoll, vom Kunden nicht explizit geforderte Funktionen und Generalisierungen wegzulassen, falls diese nicht die benötigte Infrastruktur oder anerkannte Qualitätsziele wie Erweiter- und Änderbarkeit betreffen.

- **DRY**

Das Prinzip "**Don't Repeat Yourself**" (abgekürzt als **DRY**) aus dem Jahre 2003 [Tho03] fordert, dass nichts gedoppelt werden darf wegen der Gefahr, eine Kopie bei einer Änderung zu vergessen.



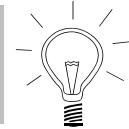
³⁶ Man spricht von Over-Engineering, wenn ein Produkt in höherer Qualität mit mehr Aufwand erstellt wird, als es der Kunde wünscht. Dabei wird oftmals die Zahlungsbereitschaft des Kunden überschritten.

Dieses Prinzip wurde im Jahre 1997 durch Kent Beck als Once and only once [Bec97] bezeichnet. Zuvor war dieses Prinzip als Single-Source-Prinzip bekannt.

- **Single Level of Abstraction Principle**

Das **Single Level of Abstraction Principle** will die Verständlichkeit von Programmen verbessern und deren Komplexität verringern.

Das Single Level of Abstraction Principle fordert deshalb, dass alle Anweisungen einer Methode dasselbe Abstraktionsniveau aufweisen sollen.



Anweisungen, deren Abstraktionsniveau von dem der anderen Anweisungen innerhalb einer Methode abweichen, sollten in eine eigene Methode ausgelagert werden. Damit hat ein Leser des Programms je nach Interesse die Möglichkeit, Details oder Wesentliches zu betrachten.

3.1.3 Entwurfsprinzipien und Konzepte für schwach gekoppelte Teilsysteme

Im Folgenden werden wichtige Entwurfsprinzipien für die Konstruktion schwach gekoppelter Teilsysteme diskutiert:

- **Loose Coupling and Strong Cohesion**

Ein gutes Softwaredesign sollte nach dem Prinzip **Loose Coupling and Strong Cohesion** eine schwache Kopplung (engl. loose coupling) zwischen den einzelnen Teilsystemen und eine starke Kohäsion (engl. strong cohesion) innerhalb eines einzelnen Teilsystems aufweisen.



Sonst kann man die verschiedenen Module nicht trennen.

- **Abstraktion und Information Hiding**

Abstraktion erlaubt es, sich auf das Wesentliche zu konzentrieren und das Unwesentliche wegzulassen.

Abstraktion erlaubt es, bei Modulen schmale Benutzungsschnittstellen der Module anzubieten (Benutzungsabstraktion).



Information Hiding von Modulen verbirgt modulinterne Daten sowie den Code eines Moduls – also die Implementierung – und erlaubt den Zugriff auf die Funktionalität dieses Moduls nur über eine schmale, minimale Schnittstelle.



Eine Schnittstelle soll so wenig wie möglich über das Innere der Module freigeben. Sie soll Designentscheidungen verbergen. Die Abhängigkeit von Implementierungen soll entfallen. Bleibt eine Schnittstelle unverändert, dann kann dennoch die Implementierung problemlos von Version zu Version geändert werden.

- **Separation of Concerns**

Separation of Concerns von Edsger W. Dijkstra ist ein allgemeines Denkprinzip, das besagt, dass ein concern³⁷ eines Systems sauber von den anderen concerns dieses Systems abgrenzen ist.



Separation of Concerns will die verschiedenen Belange eines Systems voneinander trennen und diese jeweils isoliert voneinander betrachten.

Separation of Concerns gilt nicht nur für Software, sondern generell für beliebige Systeme.

- **Law of Demeter**

Das **Law of Demeter** kann grundsätzlich als eine Projektion des Ansatzes von Loose Coupling and Strong Cohesion auf die Objekt-orientierung betrachtet werden.



Letztendlich kann das Law of Demeter als Richtlinie für die Verkettung objektorientierter Methodenaufrufe angesehen werden.

- **Dependency Inversion Principle**

In einem klassischen hierarchischen Entwurf ruft ein Modul einer höheren Ebene einen Dienst eines Moduls einer tieferen Ebene auf und wird so von der tieferen Ebene abhängig.

Das Ziel des Dependency Inversion Principle ist es, die starke Koppelung einer höheren Ebene an eine untergeordnete Ebene zu vermeiden und die **höheren Schichten wiederverwendbar** zu machen.



Folgt man beim Entwurf dem Dependency Inversion Principle, dann **definiert das Modul der höheren Ebene die Schnittstelle**, über welche das Modul auf Services zugreift. Ein Modul der niederen Ebene realisiert diese Schnittstelle. Auf diese Weise ist ein Modul der höheren Ebene unabhängig von einem Modul der tieferen Ebene und ist damit wiederverwendbar.

- **Interface Segregation Principle**

Clients sollten nach dem **Interface Segregation Principle** nicht gezwungen werden, von Methoden abhängig zu sein, die sie gar nicht brauchen.

³⁷ Ein concern kann mit Belang, Anliegen oder Verantwortung übersetzt werden.

Clients sollen nur Schnittstellen erhalten, deren Methoden sie auch tatsächlich nutzen (sogenannte **Rollen-Schnittstellen**).



Wenn mehrere Clients eine große, gemeinsame Schnittstelle hätten, könnte die Änderung eines Methodenkopfes für einen Client in unbeabsichtigter Weise einen anderen Client beschädigen.

- **Single Responsibility Principle**

Nach dem **Single Responsibility Principle** sollten Module, die sich aus verschiedenen Gründen ändern, voneinander getrennt werden:

Für die Veränderung eines Software-Moduls sollte es nur einen einzigen Grund und nicht mehrere Ursachen geben.



Mit dem einzigen Änderungsgrund einer Klasse liegt durch das Single Responsibility Principle eine neue Interpretation der starken Kohäsion vor. Mehrere Verantwortlichkeiten innerhalb eines Moduls würden zu einer Erhöhung des Fehlerrisikos führen, da bei einer Änderung einer Verantwortlichkeit eine andere Verantwortlichkeit beschädigt werden könnte.

- **Dependency Lookup sowie Dependency Injection**

Bei **Dependency Lookup** sucht ein Objekt, das ein anderes Objekt braucht, nach diesem Objekt etwa in einem Register (engl. registry), um die Verknüpfung mit diesem Objekt herzustellen.



Durch den Suchvorgang behält das suchende Objekt die Kontrolle, wann gesucht wird. Zur Suche braucht es beispielweise nur den Namen des gesuchten Objekts zu kennen und ist von diesem Objekt weitgehend entkoppelt. Aber natürlich hängt das suchende Objekt dann vom Register ab.

Bei **Dependency Injection** wird die Erzeugung von Objekten und die Zuordnung von Abhängigkeiten zwischen Objekten an eine eigens dafür vorgesehene Instanz, den Injektor, delegiert.



Für Dependency Lookup und Dependency Injection gilt:

Damit ein Objekt ein benötigtes Objekt zur Laufzeit verwenden kann, ist es erforderlich, dass eine Abstraktion der Klasse des benötigten Objekts zur Kompilierzeit im Programmcode des benutzenden Objekts enthalten ist. Ein benutzendes Objekt kann die Methoden des benutzten Objekts nur dann aufrufen, wenn es dessen Abstraktion kennt.

3.1.4 Entwurfsprinzipien und Konzepte für korrekte Programme

Durch den Einsatz von Entwurfsprinzipien bzw. Konzepten soll eine hohe innere Qualität der Software sichergestellt werden. Behandelt werden die Prinzipien und Konzepte:

- **Design by Contract**

Das Konzept Design by Contract fordert für die Beziehungen von Programmen eine formale Übereinkunft zwischen den beteiligten Partnern in Form von sogenannten Verträgen, in welchen präzise definiert wird, unter welchen Umständen ein korrekter Ablauf eines Programms erfolgt.

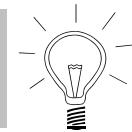


Verträge müssen auch bei polymorphen Erweiterungen von Basisklassen eingehalten werden. Eine polymorphe Erweiterung betrifft eine statische Ableitung oder die Realisierung einer aggregierten polymorphen Abstraktion, die sogenannte "Objektkomposition" (siehe Kapitel 3.2.2) zur Laufzeit.

- **Liskovsches Substitutionsprinzip**

Das liskovsche Substitutionsprinzip in Verbindung mit Design by Contract trägt wesentlich zur Korrektheit von Programmen mit polymorphen Objekten bei.

Das liskovsche Substitutionsprinzip postuliert, dass eine Referenz auf ein Objekt einer Basisklasse jederzeit auch auf ein Objekt einer abgeleiteten Klasse zeigen können soll.



Damit dies korrekt funktioniert, ist das Einhalten der Verträge einer Basisklasse nach dem Konzept Design by Contract in einer abgeleiteten Klasse erforderlich. Das Kundenprogramm darf es nicht merken, dass an der Stelle eines Objekts der Basisklasse plötzlich ein Objekt einer abgeleiteten Klasse steht. Daher muss sich die abgeleitete Klasse gleich verhalten wie die Basisklasse und die Verträge der Basisklasse einhalten. Analoges gilt für eine Schnittstelle und ihre Realisierung.

- **Principle of Least Astonishment**

Für das Vermeiden von Überraschungen ist das **Principle of Least Astonishment** von sehr großer Bedeutung, welches dafür sorgt, dass die Anwender bzw. die Programmierer sich auf das Einhalten gängiger Konventionen verlassen können und nicht getäuscht werden.

Nach dem Principle of Least Astonishment sollten Schnittstellen so entworfen werden, dass ihre Reaktion den Benutzer nicht überrascht.



Mit anderen Worten: Schnittstellen sollen sich so verhalten, wie sie es andeuten, und sollen die Nutzer der Schnittstellen nicht täuschen, da damit Fehler provoziert würden. Dieses Prinzip trägt damit zur Verständlichkeit und zur Korrektheit bei.

3.1.5 Prinzipien für die Stabilität und Erweiterbarkeit bei Programmänderungen

In diesem Kapitel wird das **Open-Closed Prinzip** und das Prinzip "**Programmiere gegen Schnittstellen, nicht gegen Implementierungen**" analysiert:

- Das Open-Closed Principle gilt für **Programme** und **Spezifikationen**.

Es fordert im Falle von Programmen, bei Änderungen eines Systems vorhandene, stabile Klassen wiederzuverwenden, um Fehlersituationen zu vermeiden. Dies erhöht die Stabilität von Programmen.

Das Open-Closed Principle verlangt, dass Module offen für Erweiterungen und gleichzeitig geschlossen für Veränderungen sind.



Offenheit bedeutet, dass stabile Module als unveränderte Bauteile wiederverwendet werden können, wobei die Änderungen sich auf Erweiterungen der stabilen Module beschränken. Eine Änderung soll im Falle von objektorientierten Programmen nicht im bereits bestehenden Code, sondern soll als objektorientierte Erweiterung bereits getesterter Programme erfolgen.

Geschlossenheit eines Moduls gegenüber Veränderungen heißt, dass ein Modul als stabiles Bauteil wiederverwendet werden können soll, ohne seinen Code anpassen zu müssen. Da ein solches Modul bereits getestet ist, kann es zur Stabilität³⁸ beitragen, wenn es wiederverwendet wird.

Erweiterungen des Quellcodes können beispielsweise durch

- eine statische Vererbung oder
- eine sogenannte Objektkomposition³⁹ nach dem Prinzip "Ziehe Objektkomposition der Klassenvererbung vor" (siehe Kapitel 3.2.2)

erfolgen.

Das Prinzip "Ziehe Objektkomposition der Klassenvererbung vor" ist eine Konkurrenztechnik zur statischen Vererbung, welche zu einer Verringerung von Abhängigkeiten gegenüber der Vererbung führt und statt der Vererbung die soge-

³⁸ Getesterter Code führt aber nur dann zur Stabilität, wenn er tatsächlich bereits mit den Daten der Anwendung getestet wurde.

³⁹ Zur Kompilierzeit wird eine Abstraktion von einem Objekt aggregiert. Zur Laufzeit tritt ein Objekt, dessen Klasse die Abstraktion implementiert, an die Stelle der Abstraktion. Siehe beispielsweise das Muster Strategie (siehe Kapitel 6.8).

nannte Objektkomposition verwendet. "Ziehe Objektkomposition der Klassenvererbung vor" kann für das Open-Closed-Prinzip verwendet werden und erhöht die Wandelbarkeit gegenüber einer statischen Vererbung.

- Gamma et al. [Gam94] konzentrieren sich mit dem Prinzip "**Programmiere gegen Schnittstellen, nicht gegen Implementierungen**" auf eine schnittstellenbasierte Programmierung.

Das Prinzip "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" von Gamma et al. [Gam94] fordert, dass Abstraktionen in Form von abstrakten Klassen bzw. Schnittstellen in Java anstelle konkreter Klassen als Datentypen verwendet werden und keine konkreten Klassen.



Es schafft dadurch einen wiederverwendbaren Code.

3.1.6 Das Konzept Inversion of Control

Das Konzept **Inversion of Control** führt zur Umkehr der Kontrolle zwischen zwei zusammenwirkenden Programmteilen. Während normalerweise ein Modul die Steuerung seines Kontrollflusses vorgibt, führt der Einsatz des Konzeptes Inversion of Control zu einer ereignisorientierten Programmierung. Viele Frameworks verwenden beispielsweise diese Technik und erzeugen Ereignisse, auf welche das eigentliche Programm reagieren muss. Das Auslösen eines Ereignisses und die Reaktion auf dieses Ereignis werden also getrennt. Ein Programm beschreibt nicht mehr, wann und in welcher Reihenfolge welche Befehle ausgeführt werden sollen, sondern nur noch, welche Reaktion bei Auftreten eines bestimmten Ereignisses erfolgen soll.

Durch Trennung des Auslösens eines Ereignisses und der Reaktion auf dieses Ereignis verbessert man die **Wandelbarkeit**, weil beide Seiten leichter geändert werden können, ohne dabei die jeweils andere Seite zu beschädigen. Aus dem gleichen Grund wird die **Wiederverwendbarkeit** beider Seiten erhöht.

3.2 Wichtige Entwurfsprinzipien nach Gamma et al.

3.2.1 Programmiere gegen Schnittstellen, nicht gegen Implementierungen

Bei diesem Prinzip geht es nicht darum, dass Schnittstellen für Module vorgeschrieben werden. Den Wert von Schnittstellen für Module hatte man damals schon längst erkannt. Hingegen geht es darum, dass man wiederverwendbar programmiert.

Das Prinzip "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" von Gamma et al. [Gam94] bewegt sich auf der Ebene der Programmierung und führt zur **Wiederverwendbarkeit des Programmcodes**.



Die gängigen Programmiersprachen verbieten es nicht, dass man beim Schreiben neuer Software bereits vorhandene konkrete Klassen als Datentypen für Variablen in neuem Code verwendet. Nur entsteht dadurch eine Abhängigkeit des neuen Codes zu diesen konkreten Klassen. Die Konsequenz ist, dass der neue Code nicht für andere konkrete Klassen direkt verwendet werden kann. Daher will das Prinzip "**Programmiere gegen Schnittstellen, nicht gegen Implementierungen**" die **Nutzung von konkreten Klassen unterbinden**.

Das Prinzip "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" fordert, dass nur abstrakte Datentypen bzw. Schnittstellen wie beispielsweise Interfaces in Java als Datentypen verwendet werden und keine konkreten Datentypen.



Alle von einer abstrakten Klasse abgeleiteten Klassen sind auch vom Typ der abstrakten Klasse. Denn eine solche Unterklasse kann zwar neue Methoden hinzufügen, muss aber auch die in der Basisklasse definierten abstrakten Methoden implementieren⁴⁰. Ebenso ist eine Klasse, die eine Java-Schnittstelle implementiert, auch vom Typ dieser Schnittstelle. Somit gilt:

Alle Unterklassen einer abstrakten Klasse können die in der Schnittstelle der abstrakten Klasse definierten Anfragen beantworten, da sie alle Subtypen der **abstrakten Klasse** sind. Analog gilt das in Java auch für **Schnittstellen** und deren Implementierungsklassen.



Die abgeleiteten Klassen bzw. die Implementierungsklassen müssen das liskovsche Substitutionsprinzip (siehe Kapitel 3.1.4) einhalten, damit diese Aussage gilt.

Durch die Verwendung **abstrakter** anstelle konkreter **Klassen** bzw. von **Schnittstellen in Java als Typen von Variablen** werden Implementierungsabhängigkeiten erheblich reduziert. Die entsprechende Anwendung wird durch die Verwendung abstrakter Klassen bzw. von Schnittstellen in Java wiederverwendbar.



3.2.1.1 Historie

Der Ansatz, Schnittstellen für Module zu verwenden, ist bereits durch das Prinzip "Loose Coupling and Strong Cohesion" implizit bekannt, denn ohne die Einführung von Schnittstellen gibt es keine schwache Kopplung von Modulen. Die Trennung von Schnittstelle und Implementierung der Module stellt die Voraussetzung für eine Benutzungsabstraktion (siehe Kapitel 3.1.3) und das Einhalten von Information Hiding (siehe ebenfalls Kapitel 3.1.3) dar. Die Trennung von Schnittstelle und Implementierung wird u. a. auch

⁴⁰ In Java können in diesem Falle die Methoden der Unterklasse mit der Annotation @override versehen werden. Der Compiler stellt dann sicher, dass eine so gekennzeichnete Methode die gleiche Signatur wie die abstrakte Methode der Basisklasse hat und diese implementiert, und gibt einen Fehler aus, wenn dies nicht der Fall ist. Viele Leute verwenden die Sprechweise "abstrakte Methoden werden überschrieben", obwohl im Falle einer abstrakten Methode noch gar keine Implementierung überschrieben werden kann. Diese Sprechweise leitet sich vom Namen und der Verwendung der Annotation @override her.

bei Schichtenmodellen (siehe Kapitel 9.1) oder dem Dependency Inversion Principle (siehe Kapitel 3.1.3) gefordert.

Gamma et al. gehen über die Verwendung von Schnittstellen für Module hinaus. Im Original [Gam94] lautet die Formulierung des Prinzips:

"Program to an interface, not an implementation."

Diese etwas abstrakte Formulierung wird im anschließenden Satz präzisiert:

"Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class."

3.2.1.2 Ziel

Die Menge aller öffentlichen Methodenköpfe einer Klasse beschreibt die sogenannte **Schnittstelle** einer Klasse.



Infofern hat eigentlich jede Klasse bereits eine Schnittstelle. Mit dem Begriff "Schnittstelle" im Namen des Prinzips ist aber eher eine Abstraktion der Schnittstelle einer Klasse gemeint, wie es beispielsweise ein Interface in Java darstellt. Das Ziel des Prinzips ist also die Einführung einer **neuen Schicht von Schnittstellen wie etwa abstrakte Klassen oder Interfaces in Java**, gegen die programmiert werden soll.

Programmieren gegen eine Schnittstelle bedeutet letztendlich, dass die eingeführten Schnittstellen wie etwa abstrakte Klassen oder Interfaces in Java als Datentypen verwendet werden, wo immer das möglich ist, sei es in den **Methodenköpfen**, bei **Attributen** oder bei **lokalen Variablen**.



Wird gegen eine Abstraktion beispielsweise in Form einer abstrakten Basisklasse programmiert, so kann eine Klasse bei Einhaltung des liskovschen Substitutionsprinzips problemlos gegen eine andere Klasse, welche dieselbe Abstraktion implementiert, ausgetauscht werden. Entsprechendes gilt für Schnittstellen in Java.

Durch das Prinzip "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" werden Implementierungsabhängigkeiten vermieden und auf diese Weise wird das **eigentliche Ziel** des Prinzips erreicht, nämlich die **Wiederverwendbarkeit des geschriebenen Programms**.



3.2.1.3 Bewertung

Das Prinzip "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" hat die folgenden **Vorteile**:

- **Abhangigkeit der Clients nur von abstrakten Klassen bzw. Schnittstellen**

Clients (Kunden) kennen nur die abstrakten Klassen bzw. Schnittstellen in Java und sind damit auch nur von diesen abhangig und nicht von konkreten Objekten bzw. deren Klassen. Clients wissen nichts uber die konkreten Klassen der von ihnen verwendeten Objekte, solange diese Objekte der von den Clients erwarteten Schnittstelle genugen. Die konkreten Objekte konnen beispielsweise durch Dependency Injection (siehe Kapitel 3.1.3) einem Client bekannt gemacht werden oder durch Dependency Lookup (siehe ebenfalls Kapitel 3.1.3) vom Client gefunden werden.

- **wiederverwendbarer Code**

Wird nach Gamma et al. gegen abstrakte Basisklassen bzw. gegen Schnittstellen in Java programmiert, so wird der Code wiederverwendbar.

- **breiter Anwendungsbereich**

Dieses Prinzip kann in statisch typisierten objektorientierten Programmiersprachen verwendet werden.

Ein **Nachteil** ist:

- **Einführung einer zusätzlichen Schicht**

Die Schicht der abstrakten Klassen bzw. Schnittstellen muss als zusätzliche Schicht im Entwurf eingefuhrt werden. Durch die neuen Elemente wird der Umfang von Entwurf und Implementierung vergroert.

3.2.2 Ziehe Objektkomposition der Klassenvererbung vor

Das Prinzip "Ziehe Objektkomposition der Klassenvererbung vor" verlangt, dass gegen abstrakte Klassen bzw. in Java gegen Schnittstellen programmiert wird, um einen wiederverwendbaren Code zu erzeugen.

3.2.3 Historie

Das Prinzip "Ziehe Objektkomposition der Klassenvererbung vor" wird im beruhmten Entwurfsmuster-Buch vom Gamma et al. [Gam94] besonders hervorgehoben. Auf Englisch heit dieses Prinzip "**Favour composition over inheritance**". Daher wird der Name dieses Prinzips auch abgekurzt zu **FCOI**.

Zusammen mit dem Prinzip "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" ist FCOI eine zentrale Grundlage fur die Entwurfsmuster von Gamma et al. FCOI dient zur Reduktion von Abhangigkeiten.

3.2.4 Ziel

Bei Verwendung des Prinzips "Ziehe Objektkomposition der Klassenvererbung vor" aggregiert nach UML zur Kompilierzeit eine Klasse eine Abstraktion. Zur Laufzeit tritt an die Stelle der Abstraktion ein Objekt, dessen Klasse die Abstraktion verfeinert.



Das Prinzip "Ziehe Objektkomposition der Klassenvererbung vor" ist eine **Konkurrenztechnik zur Vererbung** im Falle der Erzeugung von polymorphen Objekten.

Bei einer **Vererbung** kann eine Basisklasse ihre Elemente für eine abgeleitete Klasse sichtbar machen. Die abgeleitete Klasse wird dadurch von den geerbten Elementen stark abhängig (**White-Box-Verhalten**), wenn diese in ihr sichtbar sind, also nicht überschrieben bzw. verdeckt werden. Anders formuliert bedeutet das, dass in diesem Falle das Prinzip der Kapselung gebrochen wird. Bei FCOI gibt es substantiell weniger Implementierungsabhängigkeiten [Gam15, S. 51]. Der Inhalt der beteiligten Objekte ist bei FCOI nicht sichtbar (**Black-Box-Verhalten**). Was bleibt, ist die Abhängigkeit von der Abstraktion. Dabei ist es am besten, wenn die Abstraktion aus einer Schnittstelle alleine aus Methodenköpfen besteht.

3.2.5 Diskussion des Namens des Prinzips

Kommentar der Autoren:

Der Name dieses Prinzips ist etwas erklärungsbedürftig, denn es handelt sich nach UML um eine Aggregation und nicht um eine Komposition. Der Name "Objektkomposition" ist historisch zu sehen.



Gamma et al. [Gam15, S. 50] schreiben selbst: "Die Objektkomposition wird durch Zuweisung von Referenzen auf andere Objekte dynamisch zur Laufzeit definiert."

Aggregation bedeutet nach UML, dass die beteiligten Objekte unabhängig voneinander weiterbestehen können. Konträr hierzu funktioniert nach UML eine **Komposition**. Hierbei leben beide Objekte stets gleich lang.



Entscheidend für die Anwendung dieses Prinzips ist, dass es möglich ist, einen Teil der Klasse, der polymorph auftreten kann, zu lokalisieren und als Abstraktion auszuweisen. Dieser Teil ist dann aus der ursprünglichen Klasse "auszuschneiden" und getrennt vom restlichen Teil zu betrachten. Über die Aggregation seiner Abstraktion wird der ausgeschnittene Teil dann in der Klasse, aus welcher er ursprünglich stammt, referenziert. Dadurch kann die Abstraktion und somit der ausgeschnittene Teil polymorph realisiert werden.

3.2.6 Bekannte Beispiele für den Einsatz einer Aggregation

Im Folgenden werden zwei bekannte Entwurfsmuster, welche das Prinzip "Ziehe Objekt-komposition der Klassenvererbung vor" verwenden, als Beispiele diskutiert. Es handelt sich dabei um das Strategiemuster und das Dekorierermuster.

3.2.6.1 Strategiemuster

Ein bekanntes Beispiel für eine Objektkomposition ist das Strategiemuster. Es wird zur Kompilierzeit eine Schnittstelle als aggregierte Abstraktion verwendet.

Hier das entsprechende Klassendiagramm:

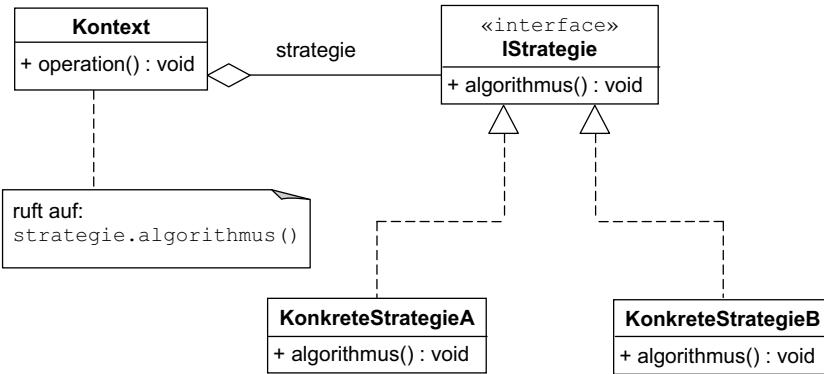


Bild 3-1 Klassendiagramm des Strategiemusters

Während im Kontext Beliebiges stecken kann, wird ein polymorph realisierbarer Anteil – die Strategie – als Abstraktion herausgetrennt.



In Bild 3-1 gibt der Kontext die Abstraktion (die Schnittstelle `IStrategie`) vor und nutzt sie mittels einer Aggregation. Bei Einhaltung des liskovschen Substitutionsprinzips kann zur Laufzeit an die Stelle der Schnittstelle in polymorpher Weise eine konkrete Strategie treten. Dabei ist der Vertrag der Schnittstelle einzuhalten.

Ein Programm, das den Kontext nutzt, setzt dabei die konkrete Strategie zur Laufzeit. Damit ist der Kontext unabhängig von der jeweiligen konkreten Strategie.

3.2.6.2 Dekorierer

Ein Dekorierer soll eine beliebige Komponente "dekorieren" können. Die folgende Abbildung zeigt eine der möglichen Formulierungen des bekannten Klassendiagramms des Dekorierermusters:

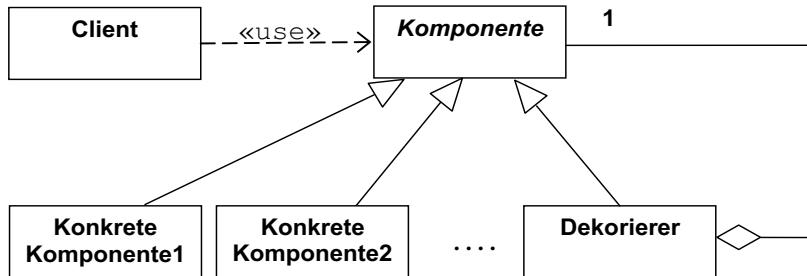


Bild 3-2 Klassendiagramm des Dekorierermusters

Der Client benutzt in Bild 3-2 eine Komponente in Form einer abstrakten Basisklasse als Abstraktion. An die Stelle der referenzierten Komponente kann nach Liskov auch eine konkrete Komponente bzw. ein Dekorierer treten.

Ein Dekorierer verwendet den von der Komponente geerbten Anteil nicht! Er verwendet den aggregierten Anteil. Damit kann auch eine konkrete Komponente dekoriert werden, da jederzeit bei Einhaltung der Verträge eine konkrete Komponente an die Stelle einer aggregierten Komponente treten kann.



3.2.7 Vergleich Vererbung und Objektkomposition

Zur Umsetzung des Open-Closed Principle kennt die Objektorientierung zwei generelle Mechanismen:

- die Vererbung (statische Klassenvererbung) und
- die Aggregation einer Abstraktion⁴¹, die sogenannte "Objektkomposition".

Objektkomposition ist eine Alternative zur Vererbung und ist der Vererbung nach Gamma et al. [Gam15, S. 51] vorzuziehen. Deshalb heißt dieses Prinzip auch "*Die Objektkomposition ist der Klassenvererbung vorzuziehen*".

Gamma et al. [Gam15, S. 51] sagen:

"Häufig lassen sich Designs durch eine verstärkte Anwendung der Objektkomposition einfacher und in höherem Maße wiederverwendbarer gestalten."

Eine Klasse referenziert bei der **Objektkomposition** eine Abstraktion, die implementiert werden muss. Da bei einer Objektkomposition keine internen Details der Objekte der referenzierenden Klasse sichtbar sind⁴² und die referenzierenden Objekte als Black-Boxes⁴³ erscheinen, wird dieser Stil der Wiederverwendung auch **Black-Box-Wieder-verwendung** genannt [Gam15, S. 50].

⁴¹ am besten einer Schnittstelle alleine aus Methodenköpfen

⁴² Nur die Methodenköpfe sind in Abstraktionen sichtbar.

⁴³ Das bedeutet, dass nur die Schnittstellenmethoden sichtbar sind.

Wiederverwendung durch Unterklassenbildung wird oft auch als **White-Box-Wiederverwendung** bezeichnet. Der Begriff "White-Box" bezieht sich auf die Sichtbarkeit, nämlich wenn infolge von **Vererbung** die Daten bzw. Methoden einer Oberklasse für eine Unterklasse (abgeleitete Klasse) sichtbar sind [Gam15, S. 49]. Bei der Vererbung wird zur Kompilierzeit festgelegt, dass ein Objekt einer abgeleiteten Klasse quasi ein Objekt der Basisklasse – nämlich den geerbten Anteil – enthält. Dies kann ebenfalls als eine Art der Komposition betrachtet werden, legt bei Sichtbarkeit jedoch die Implementierung von Elementen der Basisklasse offen. Die Beziehung zwischen den beiden Klassen ist statisch und kann zur Laufzeit nicht mehr geändert werden. Da bei der Vererbung eine abgeleitete Klasse die Daten und das Verhalten einer Basisklasse erbt, besteht durch die Ableitung eine starke Abhängigkeit einer abgeleiteten Klasse von ihrer zugehörigen Basisklasse, wenn die Elemente der Basisklasse in der abgeleiteten Klasse sichtbar sind. Vererbung als Technik der Wiederverwendung wird nach Gamma et al. in der Praxis überstrapaziert [Gam15, S. 51].

Auch wenn die **Vererbung** besonders leicht umgesetzt werden kann, da sie von den objektorientierten Programmiersprachen unterstützt wird, lohnt es sich auf jeden Fall, die **Objektkomposition** zu betrachten. Entscheidend für deren Anwendung ist, dass eine Abstraktion eindeutig identifiziert und separiert werden kann. Durch das Bereitstellen einer referenzierten Abstraktion wird ein Vertrag vorgegeben, der durch die Klasse des Objekts, welches zur Laufzeit an die Stelle der Abstraktion tritt, erfüllt werden muss.

Ein großer Vorteil der **Objektkomposition** ergibt sich auch dadurch, dass die aggregierende Klasse statisch **nur von der Abstraktion abhängig** ist und dass dynamisch erst zur Laufzeit entschieden werden kann, welches konkrete Objekt aggregiert wird. Natürlich muss die Klasse eines solchen Objekts den Vertrag der Abstraktion erfüllen, aber es kann flexibel entschieden werden, welches Objekt der entsprechenden Klasse von einem aggregierenden Objekt verwendet werden soll.

Auch bei der **Objektkomposition** gibt es eine Vererbung bzw. eine Implementierung, nämlich bei der Konkretisierung der Abstraktion. Entscheidend ist jedoch die **Black-Box-Sicht der beteiligten Klassen**.

3.2.8 Bewertung

Vorteile von "Ziehe Objektkomposition der Klassenvererbung vor" sind:

- **Einhaltung der Datenkapselung (engl. encapsulation) und Verringerung der Abhängigkeiten**

Es wird nicht wie bei der Vererbung die Kapselung gebrochen. Bei der Vererbung sind die Struktur bzw. das Verhalten einer Basisklasse auch in der abgeleiteten Klasse oft direkt sichtbar, nämlich dann, wenn die Daten bzw. Methoden der Basisklasse nicht mit einem Zugriffsmodifikator wie beispielsweise `private` in Java geschützt sind oder nicht überschrieben bzw. verdeckt werden. In diesem Fall schlägt jede Änderung der Implementierung einer Basisklasse auf die abgeleitete Klasse direkt durch. Damit hat eine abgeleitete Klasse eine starke Abhängigkeit von ihrer Basisklasse. Werden hingegen Objekte ausschließlich über die Implementierung aggregierter Abstraktionen verwendet wie bei "**Ziehe Objektkomposition der Klassenvererbung vor**", so wird die **Kapselung der ursprünglichen Implementierung**

nicht aufgebrochen. Bei der Objektkomposition besteht bei der Verwendung einer Schnittstelle als Abstraktion nur eine schwache Abhängigkeit der Klasse eines Objekts zu dem Vertrag der referenzierten Schnittstelle.

- **keine komplexen Klassenhierarchien**

Komplexe Klassenhierarchien erschweren die Übersicht. Bei der Verwendung von Objektkompositionen bleiben die Klassenhierarchien flach und übersichtlich, da jeweils die entsprechende Abstraktion implementiert wird. Dadurch entstehen keine komplexen Klassenhierarchien.

- **Simulation von Mehrfachvererbung**

Durch die Verwendung von "Objektkompositionen" zusätzlich zur Vererbung kann bei Programmiersprachen, die wie beispielsweise die Programmiersprache Java nur eine Einfachvererbung unterstützen, eine Mehrfachvererbung simuliert werden.

- **leichteres Testen**

Durch Verwendung einer Abstraktion wie beispielsweise einer Schnittstelle kann man beim Testen leichter mit Mock-Objekten arbeiten.

- **Flexibilität**

Bei einer Objektkomposition erfolgt zur Laufzeit der Austausch der Referenz auf die Abstraktion gegen eine Referenz auf ein konkretes Objekt, dessen Klasse die Abstraktion verfeinert. Dies erhöht die Flexibilität. Bei der Vererbung wird die Beziehung zwischen den Klassen bereits zur Kompilierzeit festgelegt.

- **einfacher Gebrauch von Dependency Injection**

Da bei der Objektkomposition die Objekte dynamisch zur Laufzeit festgelegt werden, können die einzelnen Objekte zur Laufzeit in einfacher Weise mithilfe von Dependency Injection an die aggregierte Abstraktion übergeben werden. Dies ist bei Vererbung nicht möglich, da die Bindung hier schon zur Kompilierzeit festgelegt wird.

Nachteile von "Ziehe Objektkomposition der Klassenvererbung vor" sind:

- **Prinzip oft unbekannt**

Das Verwenden einer aggregierten Abstraktion anstelle der Vererbung erfolgt in der Praxis zu selten, da in den Anfängervorlesungen an den Hochschulen oft nur die Vererbung gelehrt wird.

- **potenzielle Zunahme der Anzahl der Schnittstellen**

Die Objektkomposition verwendet vorzugsweise eine Schnittstelle als Abstraktion. Durch das Interface Segregation Principle, welches fordert, dass eine Schnittstelle nur Methodenköpfe haben sollte, die ein spezieller Client auch tatsächlich benötigt, kann die Anzahl solcher Schnittstellen allerdings sehr schnell zunehmen.

- **keine direkte Wiederverwendung von Code im Falle von Schnittstellen**

Vererbung führt zu einer direkten Wiederverwendung von Code, da geerbte Attribute und Methoden direkt Bestandteil der abgeleiteten Klasse werden – falls sie dort sichtbar und nicht verdeckt sind bzw. überschrieben werden. Bei der Objektkomposition mit Schnittstellen referenziert das aggregierende Objekt zur Kompilierzeit nur Methodenköpfe, deren Vertrag dann zur Laufzeit von dem entsprechenden Objekt eingehalten werden muss.

Vererbung ist sinnvoll, wenn:

- tatsächlich eine "is a"-Beziehung modelliert werden soll und es sich nicht nur um Code-Wiederverwendung handelt,
- sich der polymorphe Anteil nicht lokalisieren und herausziehen lässt, sondern Polymorphie unbeschränkt möglich ist, und
- man durch Änderung der Basisklasse alle abgeleiteten Klassen ändern möchte.

Wenn es nur um die Wiederverwendung von Code geht und nicht um die Modellierung einer "is a"-Beziehung, sollte die Vererbung nicht in Betracht gezogen werden.



3.3 Zusammenfassung

Kapitel 3.1 fasst Entwurfs- und Konstruktionsprinzipien, die auch für Entwurfsmuster eine hohe Bedeutung haben, in übersichtlicher Weise zusammen.

Kapitel 3.2 befasst sich speziell mit dem von Gamma et al. [Gam94] vorgeschlagenen Entwurfsprinzipien. Die Bedeutung dieser beiden Prinzipien geht weit über die Entwurfsmuster von Gamma et al. hinaus.

3.4 Kontrollfragen

Aufgaben 3.4.1: Entwurfsprinzipien zur Vermeidung von Komplexität

- 3.4.1.1 Was ist der Ansatz von KISS?
- 3.4.1.2 Was will YAGNI? Warum macht YAGNI Sinn?
- 3.4.1.3 Nennen Sie andere Namen für DRY.
- 3.4.1.4 Was soll mit dem Single Level of Abstraction Principle erreicht werden?

Aufgaben 3.4.2: Entwurfsprinzipien für die Konstruktion schwach gekoppelter Teilsysteme

- 3.4.2.1 Erklären Sie Loose Coupling and Strong Cohesion.
- 3.4.2.2 Was ist der Vorteil von Information Hiding?
- 3.4.2.3 Welches Ziel hat Separation of Concerns?
- 3.4.2.4 Was will das Law of Demeter erreichen?
- 3.4.2.5 Erklären Sie das Dependency Inversion Principle.
- 3.4.2.6 Was bedeutet das Interface Segregation Principle?
- 3.4.2.7 Was ist das Ziel des Single Responsibility Principle?
- 3.4.2.8 Was ist Dependency Look-up? Was ist Dependency Injection?

Aufgaben 3.4.3: Entwurfsprinzipien für die Stabilität bei Programmierweiterungen

- 3.4.3.1 Erklären Sie das Open-Closed Principle.
- 3.4.3.2 Was ist das Ziel von "Ziehe Objektkomposition der Klassenvererbung vor"?
- 3.4.3.3 Warum redet man bei "Ziehe Objektkomposition der Klassenvererbung vor" von einer Black-Box-Verwendung?
- 3.4.3.4 Erläutern Sie die White-Box-Verwendung bei der Vererbung.
- 3.4.3.5 Nennen Sie drei Vorteile der Objektkomposition.
- 3.4.3.6 Was ist das Ziel von "Programmiere gegen Schnittstellen, nicht gegen Implementierungen"?
- 3.4.3.7 Beschreiben Sie die Rolle von konkreten und abstrakten Klassen bei "Programmiere gegen Schnittstellen, nicht gegen Implementierungen"

Aufgaben 3.4.4: Das Konzept Inversion of Control

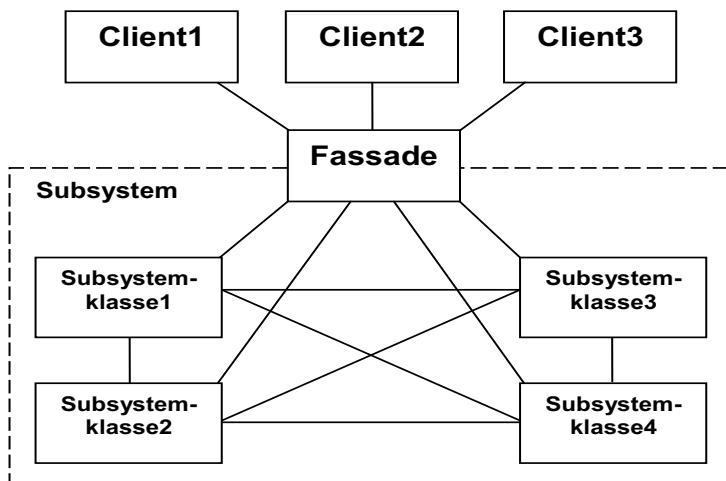
- 3.4.4.1 Erklären Sie das Konzept Inversion of Control.

Aufgaben 3.4.5: Entwurfsprinzipien auf Systemebene

- 3.4.5.1 Welche Idee steckt hinter Teile und Herrsche?
- 3.4.5.2 Was ist das Ziel von Design to Test?

Kapitel 4

Übersicht über den behandelten Katalog von Entwurfsmustern



- 4.1 Strukturmuster
- 4.2 Verhaltensmuster
- 4.3 Erzeugungsmuster
- 4.4 Leicht zu erlernende Entwurfsmuster
- 4.5 Zusammenfassung
- 4.6 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_4.

4 Übersicht über den behandelten Katalog von Entwurfsmustern

Kapitel 4.1 gibt eine Übersicht über die gezeigten Strukturmuster, Kapitel 4.2 über die behandelten Verhaltensmuster und Kapitel 4.3 über die dargestellten Erzeugungsmuster. Kapitel 4.4 nennt zur Einführung in Entwurfsmuster einige einfache Entwurfsmuster.

Gamma et al. [Gam94] differenzieren Entwurfsmuster zum einen in die Sichtweisen

- Strukturmuster,
- Verhaltensmuster und
- Erzeugungsmuster,

zum anderen in "klassenbasierte" und "objektbasierte" Muster (siehe Kapitel 2.3).

Strukturmuster befassen sich mit der Zusammensetzung von Klassen und Objekten, um größere Strukturen von Klassen bzw. Objekten zu bilden.



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



Erzeugungsmuster kapseln die Objekterzeugung, wobei zur Komplizierzeit nur die Schnittstellen der Klassen der zu erzeugenden Objekte definiert werden müssen. Die konkrete zur Instanziierung verwendete Klasse muss erst zur Laufzeit festgelegt werden.



Strukturmuster, Verhaltensmuster und Erzeugungsmuster betreffen die Modellierung sogenannter Verarbeitungsfunktionen⁴⁴ der Softwaretechnik beim Entwurf. Anzumerken ist, dass es auch Entwurfsmuster für sogenannte technische Funktionen⁴⁵ gibt wie z. B. **Kommunikationsmuster** oder für Systemeigenschaften wie **Security-Muster**.

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

⁴⁴ Verarbeitungsfunktionen modellieren bereits den Problembereich, betreffen dabei das jeweilige Fachkonzept und werden im Rahmen der Analyse analysiert. Es gibt Verarbeitungsfunktionen auch beim Entwurf. Hier sind sie im Falle der Entwurfsmuster zu sehen. Siehe Begriffsverzeichnis.

⁴⁵ Sogenannte technische Funktionen treten beim Entwurf zu den Verarbeitungsfunktionen der Analyse hinzu. Die technischen Funktionen ergänzen die Verarbeitungsfunktionen zu einem lauffähigen System. Siehe Begriffsverzeichnis.

- **Strukturmuster**

Strukturmuster werden in zwei Kriterien aufgeteilt. Die erste Kategorie sind **klassenbasierte** Strukturmuster, die über die Eigenschaft der Vererbung statisch die Klassen der betrachteten Objekte gewinnen.

Das einzige **klassenbasierte Strukturmuster** von Gamma et al. ist der **klassenbasierte Adapter**.



Die zweite Kategorie bilden die **objektbasierten** Strukturmuster. Das Proxy-Muster ist ein Beispiel für ein objektbasiertes Strukturmuster. Ein Proxy dient hierbei als ein Stellvertreter für ein anderes Objekt.

- **Verhaltensmuster**

Auch bei **Verhaltensmustern** wird zwischen **klassenbasierten** und **objektbasierten** Verhaltensmustern unterschieden. Klassenbasierte Verhaltensmuster verwenden Vererbung, um statisch das gewünschte Verhalten zu gewinnen.

Die einzigen **klassenbasierte Verhaltensmuster** von Gamma et al. sind die **Schablonenmethode** und der **Iterator**.



Objektbasierte Verhaltensmuster können das liskovsche Substitutionsprinzip verwenden, um durch Austausch von Objekten zur Laufzeit das Verhalten abzuändern. Ein Beispiel für den dynamischen Austausch des Verhaltens zur Laufzeit ist das Strategie-Muster.

- **Erzeugungsmuster**

Erzeugungsmuster verbergen die Erzeugung von Objekten in einer Operation bzw. einer Klasse.

Das einzige **klassenbasierte Erzeugungsmuster** von Gamma et al. ist die **Fabrikmethode**.



Die Fabrikmethode verwendet Operationen, deren Klasse durch Vererbung statisch zur Kompilierzeit gewonnen wird.



Objektbasierte Erzeugungsmuster wie die Abstrakte Fabrik verwenden zum dynamischen Austausch der zu erzeugenden Objekte das liskovsche Substitutionsprinzip.



Hingegen kommt das Singleton-Muster komplett ohne Vererbung aus.

4.1 Strukturmuster

In Kapitel 5 werden gezeigt:

- **Muster Adapter**

Das Strukturmuster **Adapter** 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. Durch die Vererbung 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".

- **Muster Brücke**

Das **objektbasierte** Strukturmuster **Brücke** trennt eine Implementierung und ihre Schnittstelle zum Client (Abstraktion) weitestgehend voneinander. Die Klassenhierarchie der Abstraktion und der Implementierung sind durch die sogenannte "Brücke" verbunden. Dadurch wird erreicht, dass beide Teile getrennt verändert und erweitert werden können.

- **Dekorierermuster**

Das **objektbasierte** Strukturmuster **Dekorierer** ermöglicht es, zur Laufzeit eine zusätzliche Funktionalität zu einem Objekt bzw. den Objekten seiner Kinder in dynamischer Weise hinzuzufügen.

- **Fassadenmuster**

Das **objektbasierte** Strukturmuster **Fassade** stellt eine vereinfachte Schnittstelle zum gekoppelten Ablauf mehrerer zusammengehöriger Folgen von Aufrufen der Objekte hinter der Fassade bereit. Ein Fassadenobjekt delegiert die Aufrufe zusammengehöriger Sequenzen von Methoden an Objekte der Klassen dieses Subsystems.

- **Kompositum-Muster**

Das **objektbasierte** Strukturmuster **Kompositum** erlaubt es, dass bei der Verarbeitung von Elementen in einer Baumstruktur einfache und zusammengesetzte Objekte gleich behandelt werden.

- **Proxy-Muster**

Das **objektbasierte** Strukturmuster **Proxy** verbirgt die Existenz eines echten 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 eine Zusatzfunktionalität hinzufügen.

4.2 Verhaltensmuster

Kapitel 6 zeigt die folgenden Verhaltensmuster:

- **Schablonenmethode**

Das **klassenbasierte** Verhaltensmuster **Schablonenmethode** legt bereits in einer Basisklasse die Struktur eines Algorithmus fest. Realisiert werden variante Teile des Algorithmus in verschiedenen Unterklassen. Dadurch ist eine Klasse, welche die Schablonenmethode enthält, nicht abhängig von der Ausprägung des Algorithmus in einer tieferen Ebene (Dependency Inversion, siehe Kapitel 3.1.3).

- **Befehlsmuster**

Beim **objektbasierten** Verhaltensmuster **Befehl** wird ein Befehl als Objekt gekapselt. Dieses kann versandt werden. 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.

- **Beobachter-Muster**

Das **objektbasierte** Verhaltensmuster Beobachter erlaubt es, dass ein Objekt von ihm abhängige Objekte von einer Änderung seines Zustands informiert. Damit muss ein abhängiges Objekt nicht pollen (Inversion of Control, siehe Kapitel 3.1.6).

- **Besucher-Muster**

Das **objektbasierte** Verhaltensmuster **Besucher** ermöglicht es, eine neue Operation zu einer bestehenden Datenstruktur hinzuzufügen, welche auf den Objekten der Datenstruktur arbeitet, um ihre Funktion zu erbringen. Eine solche Operation wird in einem Besucher-Objekt gekapselt. Beim Besucher-Muster müssen die zu besuchenden Objekte auf den Besuch "vorbereitet" sein, dadurch dass sie eine entsprechende Methode dem Besucher zur Verfügung stellen.

- **Iterator-Muster**

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

- **Memento-Muster**

Das **objektbasierte** Verhaltensmuster **Memento** beschreibt, wie man den Zustand eines Objekts kapselt und speichert, um das Objekt später dann erneut in einen dieser Zustände versetzen zu können. Es müssen zwei Schnittstellen definiert werden, eine für die Verwaltung der gespeicherten Zustände (Mementos) und eine Schnittstelle für das ursprüngliche Objekt, damit es seinen Zustand als Memento speichern oder daraus auslesen kann.

- **Rollenmuster**

Das **objektbasierte** Verhaltensmuster **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.

- **Strategie-Muster**

Mit Hilfe des **objektbasierten** Verhaltensmusters **Strategie** 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 eine Schnittstelle und kapselt sie in einem Objekt.

- **Vermittler-Muster**

Das **objektbasierte** Verhaltensmuster **Vermittler** soll es erlauben, dass Objekte miteinander über einen Vermittler reden, der bei einer bei ihm eingehenden Nachricht alle 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.

- **Zustandsmuster**

Im **objektbasierten** Verhaltensmuster **Zustand** werden die einzelnen Zustände durch eigene Klassen gekapselt, die von einer gemeinsamen abstrakten Basisklasse ableiten bzw. eine gemeinsame Schnittstelle implementieren. Ein zustandsabhängiges Objekt referenziert in der Grundform des Musters den jeweils aktuellen Zustand und führt Zustandsänderungen durch.

4.3 Erzeugungsmuster

Die folgenden kurzen Zusammenfassungen beschreiben die Erzeugungsmuster, die in Kapitel 7 vorgestellt werden:

- **Muster Fabrikmethode**

Im **klassenbasierten** Erzeugungsmuster **Fabrikmethode** soll eine erzeugende Klasse zur Kompilierzeit nur die Basisklasse des zu erzeugenden Objekts kennen, soll aber zur Kompilierzeit nicht wissen, von welcher Unterklasse das entsprechende Objekt zur Laufzeit im lauffähigen Programm später sein soll. Die Erzeugung von Objekten wird in Unterklassen verlagert. Unterklassen legen bei diesem Muster die Struktur und das Verhalten der zu erzeugenden Objekte statisch fest. Eine Klasse einer höheren Ebene ist damit nicht mehr abhängig von der Ausprägung einer Klasse einer tieferen Ebene (Dependency Inversion, siehe Kapitel 3.1.3).

- **Muster Abstrakte Fabrik**

Beim **objektbasierten** Erzeugungsmuster **Abstrakte Fabrik** kann durch Wahl der entsprechenden konkreten Fabrik zur Laufzeit entschieden werden, welche Produktfamilie⁴⁶ erzeugt werden soll.

- **Singleton-Muster**

Das **objektbasierte** Erzeugungsmuster **Singleton** stellt sicher, dass von einer Klasse nur ein einziges Objekt erzeugt werden kann. Dieses Muster arbeitet komplett ohne Vererbung.

- **Muster Objektpool**

Das **objektbasierte** Erzeugungsmuster **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.

4.4 Leicht zu erlernende Entwurfsmuster

Dieses Kapitel nennt einige einfache Entwurfsmuster, die leicht zu verstehen sind und daher gut als Einstieg in das Gebiet der Entwurfsmuster dienen können:

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

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

4.5 Zusammenfassung

Kapitel 4.1 bespricht die in Kapitel 5 gezeigten Strukturmuster.

Kapitel 4.2 gibt einen Überblick über die in Kapitel 6 behandelten Verhaltensmuster.

Kapitel 4.3 skizziert die in Kapitel 7 dargestellten Erzeugungsmuster.

Kapitel 4.4 nennt zum Einstieg in das Gebiet der Entwurfsmuster einige leicht verständliche Entwurfsmuster.

⁴⁶ Das ist eine Gruppe von zusammengehörigen Objekten, die z. B. für ein bestimmtes Betriebssystem erzeugt werden.

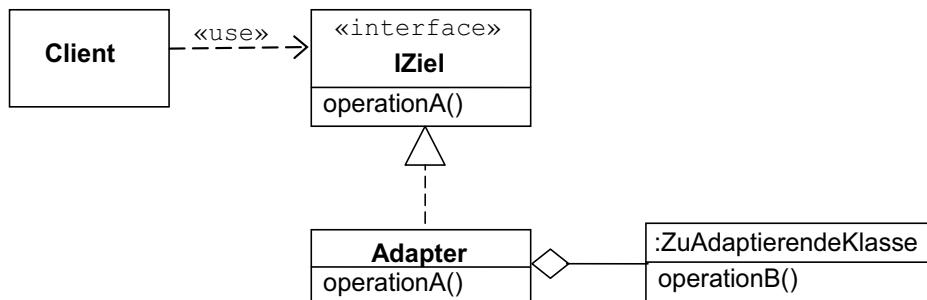
4.6 Kontrollfragen

Aufgaben 4.6.1: Kriterien für Entwurfsmuster

- 4.6.1.1 Charakterisieren Sie alle Kriterien für die Einteilung der Entwurfsmuster von Gamma et al.
- 4.6.1.2 Was ist der Unterschied zwischen klassenbasierten und objektbasierten Entwurfsmustern?
- 4.6.1.3 Nennen Sie einige klassenbasierte Entwurfsmuster.

Kapitel 5

Strukturmuster



- 5.1 Das Strukturmuster Adapter
- 5.2 Das Strukturmuster Brücke
- 5.3 Das Strukturmuster Dekorierer
- 5.4 Das Strukturmuster Fassade
- 5.5 Das Strukturmuster Kompositum
- 5.6 Das Strukturmuster Proxy
- 5.7 Zusammenfassung
- 5.8 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_5.

5 Strukturmuster

Der Begriff "Strukturmuster" wurde von Gamma et al. als eine Kategorie der Entwurfsmuster geprägt [Gam94].

Strukturmuster untersuchen, auf welche Art und Weise Klassen bzw. Objekte zu größeren Strukturen zusammengefasst werden können.



Klassenbasierte Strukturmuster verwenden hierzu die statische Vererbung. Objektbasierten Strukturmustern liegt das Prinzip der Objektkomposition (siehe Kapitel 3.2.2) zugrunde.

Wie in Kapitel 2.2.2 erwähnt betreffen Entwurfsmuster Verarbeitungsfunktionen. Diese existieren bereits im Problembereich, werden im Rahmen von Entwurfsmustern jedoch nur im Lösungsbereich betrachtet. Dies gilt auch für die im Folgenden betrachteten Strukturmuster.

Kapitel 5.1 untersucht das Strukturmuster Adapter und Kapitel 5.2 die Brücke. Der Dekorierer wird in Kapitel 5.3 behandelt, die Fassade in Kapitel 5.4, das Kompositum in Kapitel 5.5 und der Proxy in Kapitel 5.6.

5.1 Das Strukturmuster Adapter

5.1.1 Name/Alternative Namen

Adapter (engl. adapter), UMWICKLER (engl. wrapper), UMWANDLER.

5.1.2 Problem

Eine vorhandene Klasse soll in einem anderen Umfeld wiederverwendet werden. Die wiederzuverwendende Klasse bietet zwar die richtigen Daten an, ihre Schnittstelle passt jedoch nicht für den Zugriff eines neuen Clients auf diese Daten⁴⁷.

Das **Adapter-Muster** hat zum Ziel, eine existierende "falsche" Schnittstelle einer bereits vorhandenen Klasse, welche nicht der vom Client benötigten Schnittstelle entspricht, **an die vom Client gewünschte Form anzupassen**.



Das Adapter-Muster wird oft angewandt, wenn Klassen von Dritten, die nicht geändert werden sollen bzw. können – wie die Klassen einer Klassenbibliothek –, an eine andere Schnittstelle angepasst werden sollen.



⁴⁷ Als Beispiel aus dem täglichen Leben sei ein Adapter genannt, mit dem man einen deutschen Rasierapparat an einer amerikanischen Steckdose verwenden kann. Um den Rasierapparat auch bei einer andersartigen Steckdose wiederverwenden zu können, bedarf es eines Adapters.

Im Fall des Adapter-Musters gibt es nach Gamma et al. [Gam94] zwei Möglichkeiten für einen Adapter:

- einen **klassenbasierten Adapter**⁴⁸ und
- einen **objektbasierten Adapter**⁴⁹.

Der **Klassen-Adapter** funktioniert aber dann **nicht**, wenn man nicht nur eine vorgegebene Klasse, sondern auch deren **Unterklassen** anpassen will. Der **Objekt-Adapter** kann zusätzlich zu einer zu adaptierenden Klasse **auch deren Unterklassen anpassen**.



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

5.1.3 Lösung

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



Das Adapter-Muster ist ein Entwurfsmuster, welches dieses Anpassungsproblem behandelt. Der Adapter ergänzt die zu adaptierende Klasse um ein "Übersetzungssprogramm" für deren Methoden. Der Adapter bietet dem Client die von diesem benötigten Methoden an, übersetzt sie in Methodenaufrufe der anzupassenden Klasse und ruft diese auf.

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

5.1.3.1 Klassendiagramm

Variante 1: Klassen-Adapter (Adapter mit Vererbung)

Das folgende Bild zeigt das Klassendiagramm des **klassenbasierten Adapter-Musters** in symbolischer Weise:

⁴⁸ Im Folgenden kurz "Klassen-Adapter" genannt.

⁴⁹ Im Folgenden kurz "Objekt-Adapter" genannt.

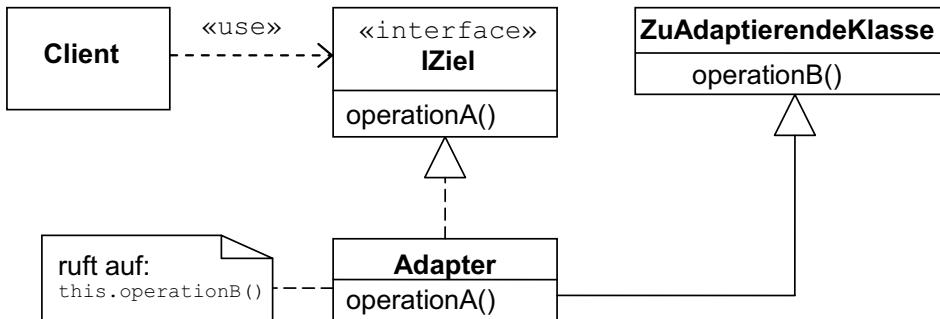


Bild 5-1 Klassendiagramm Klassen-Adapter

Der **Klassen-Adapter** basiert auf **statischer Vererbung der anzupassenden Klasse**. Der Client benutzt die in der Schnittstelle `IZiel` aufgelisteten Köpfe der Operationen. Diese Operationen werden vom Client vorgegeben. Die Klasse `Adapter` implementiert diese Schnittstelle 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 fordert in diesem Fall also indirekt die Operation `operationB()` an, wenn er selbst direkt beim Adapter die Operation `operationA()` aufruft.

Die Funktionalität eines Adapters wird in den Methoden zur Verfügung gestellt, welche die Schnittstelle `IZiel` realisieren, hier in der Methode `operationA()`.

Innerhalb der Operation `operationA()` können Konvertierungen der Parameter oder des Rückgabewerts, aber auch komplexere Anpassungen vorgenommen werden.



Im strengen Sinn handelt es sich wegen der übrigen geerbten Methoden der zu adaptierenden Klasse beim Klassen-Adapter um keinen echten Adapter.



In der Lösung des Klassen-Adapters bietet der Adapter alle Methoden – auch die der zu adaptierenden Klasse – an, falls die verwendete Programmiersprache wie im Falle von Java keine private Vererbung zur Verfügung stellt oder die nicht benötigten Methoden nicht überschrieben werden.



Variante 2: Objekt-Adapter (Adapter mit Objektkomposition)

Das Klassendiagramm des **objektbasierten Adapter-Musters** in folgendem Bild veranschaulicht, wie der Aufruf einer Operation durch den Client an eine Operation des aggregierten Objekts mit anderem Namen weitergereicht wird:

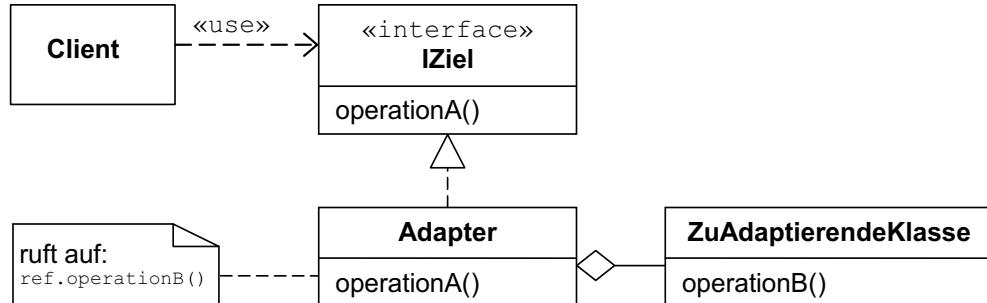


Bild 5-2 Klassendiagramm Objekt-Adapter

Der Objekt-Adapter greift selbst auf die zu adaptierende Klasse über eine Referenz zu.

Auch in dieser Variante stellt die Schnittstelle vom Typ **IZiel** die von der Client-Anwendung vorgegebene Aufrufsschnittstelle dar. Die zu adaptierende Klasse wird mittels Aggregation dem Adapter bekannt gemacht und wird zur Laufzeit nach dem Delegationsprinzip aufgerufen.

Die Klasse **Adapter** implementiert die Aufrufsschnittstelle **IZiel** und delegiert den Aufruf an das aggregierte Objekt der zu adaptierenden Klasse weiter.



Wird die Operation **operationA()** beim Adapter (siehe Bild 5-2) aufgerufen, dann ruft der Adapter die Operation **operationB()** beim aggregierten Objekt auf.

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

Die weitere Beschreibung des Adapter-Musters basiert auf der gezeigten Variante 2, dem Objekt-Adapter.



5.1.3.2 Teilnehmer

Im Folgenden werden die Teilnehmer am Adapter-Muster aufgeführt:

- **ZuAdaptierendeKlasse**

Die Klasse `ZuAdaptierendeKlasse` bietet wiederzuverwendende Operationen als Dienstleistungen an, deren Schnittstellen 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**

Die Schnittstelle `IZiel` definiert die Schnittstelle, welche der Client verwenden möchte. Diese Schnittstelle muss implementiert werden. Da die Schnittstelle `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.

5.1.3.3 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 eben eine falsche Schnittstelle hat. Das Ergebnis wird von dem Objekt der zu adaptierenden Klasse über das Adapter-Objekt an das Client-Objekt zurückgegeben. Dies wird im folgenden Sequenzdiagramm dargestellt:

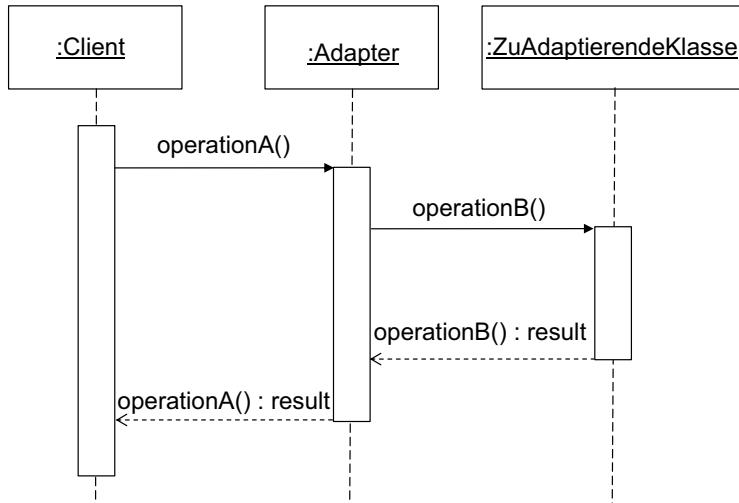


Bild 5-3 Sequenzdiagramm des Adapter-Musters

5.1.3.4 Programmbeispiel

Als Beispiel soll eine kleine Anwendung dienen, welche Personendaten aus einer CSV-Datei⁵⁰ 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 Aufrufschwittstelle 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 die vom Kunden geforderte Schnittstelle IPersonenLeser implementieren.

Zur Speicherung von Vor- und Nachname dient die Klasse Person. Die Klasse Person ist sozusagen eine Hilfsklasse und spielt im Adaptermuster selbst keine Rolle.

Das folgende Bild zeigt das Klassendiagramm dieses Beispiels:

⁵⁰ 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 hierbei nicht unbedingt ein Komma sein.

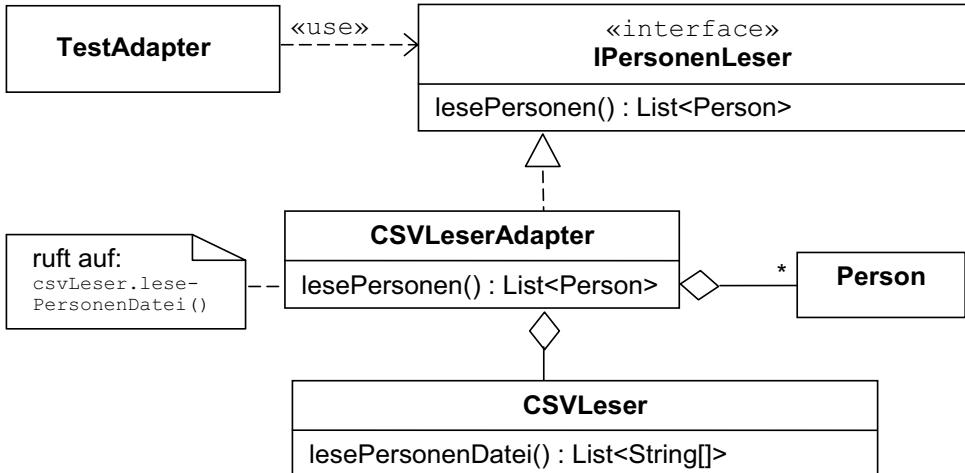


Bild 5-4 Klassendiagramm für das Beispiel zum Lesen aus einer CSV-Datei

Es folgt der Programmcode. Hier die Klasse `Person`:

```

// Datei: Person.java

public class Person {

    private final String nachname;
    private final String vorname;

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

    @Override
    public String toString() {
        return String.format("%s %s", 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 zur geforderten Schnittstelle `IPersonenLeser` hat die bereits realisierte Methode zum Einlesen der Personen einen anderen Namen und gibt keine Liste von Personen zurück, sondern eine Liste von Strings, in welcher Vor- und Nachname jeweils einer Person gespeichert sind. Da diese Implementierung nicht kompatibel zur Schnittstelle `IPersonenLeser` ist, muss sie adaptiert werden. Hier die Klasse `CSVLeser`:

```

// Datei: CSVLeser.java

import java.io.*;
  
```

```
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
import java.util.stream.Collectors;

public class CSVLeser {

    // hat als Rueckgabewert eine Liste vom Typ eines String-Arrays
    public List<String[]> lesePersonenDatei(String file) {
        try (BufferedReader input = Files
            .newBufferedReader(Paths.get(file), StandardCharsets.UTF_8)) {
            return input.lines()
                .map(s -> s.split(","))
                .filter(s -> s.length >= 2)
                .map(s -> new String[] { s[0], s[1] })
                .collect(Collectors.toList());
        } catch (IOException ex) {
            throw new IllegalStateException(ex);
        }
    }
}
```

Die geforderte Schnittstelle `IPersonenLeser` dient als Schnittstelle für alle Klassen, die Personendaten einlesen können. Sie definiert die Methode `lesePersonen()`, die eine Liste von Personen zurückgibt. Hier die Schnittstelle `IPersonenLeser`:

```
// Datei: IPersonenLeser.java

import java.util.List;

public interface IPersonenLeser {

    List<Person> lesePersonen();
}
```

Die Klasse `CSVLeserAdapter` dient als Adapter für die Klasse `CSVLeser`. Sie implementiert die Schnittstelle `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 geforderten Schnittstelle genügen. Hier die Klasse `CSVLeserAdapter`:

```
// Datei: CSVLeserAdapter.java

import java.util.List;
import java.util.stream.Collectors;

public class CSVLeserAdapter implements IPersonenLeser {

    private String fileName;
    private CSVLeser csvLeser = new CSVLeser();
```

```

public CSVLeserAdapter(String fileName) {
    this.fileName = fileName;
}

@Override
public List<Person> lesePersonen() {
    return csvLeser.lesePersonenDatei(fileName).stream()
        .map(p -> new Person(p[0], p[1]))
        .collect(Collectors.toList());
}
}

```

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

public class TestAdapter {

    public static void main(String[] args) {
        IPersonenLeser leser = new CSVLeserAdapter("Personen.csv");
        leser.lesePersonen().forEach(System.out::println);
    }
}

```

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



Hier das Protokoll des Programmablaufs:

```

Heinz Mueller
Volker Schmied
Hannah Schneider

```

5.1.4 Bewertung

5.1.4.1 Vorteile

Die folgenden **Vorteile** werden gesehen:

- **Kommunikation zwischen zwei unabhängigen Softwarekomponenten**

Das Adapter-Muster ermöglicht die Kommunikation zwischen zwei unabhängigen Softwarekomponenten, nämlich zwischen dem Client und einem Objekt der zu adaptierenden Klasse.

- **Erweiterung beliebig möglich**

Adapter können beliebig anpassen.

- **Optimierung möglich, da individuell**

Adapter sind individuell an die jeweilige Lösung angepasst und können daher optimiert werden.

- **Austausch von Clients**

Clients können ausgetauscht werden. Gegebenenfalls muss nur ein neuer Adapter zur Verfügung gestellt werden.

- **anwendbar auf ein Objekt einer Unterklasse beim Objekt-Adapter**

Ein Objekt-Adapter kann auf Grund des liskovschen Substitutionsprinzips auch Unterklassen anpassen, da er eine Referenz auf ein Objekt der zu adaptierenden (Basis-)Klasse besitzt. Er kann eine neue Funktionalität allen adaptierten Klassen in einem Zug hinzufügen. Allerdings kann er dabei keine erweiternden Methoden einer Unterklasse nutzen.

5.1.4.2 Nachteile

Die folgenden **Nachteile** werden gesehen:

- **zusätzliche Schicht verschlechtert die Performance**

Durch das Adapter-Muster wird beim Aufruf einer Operation eine zusätzliche Zwischenschicht eingeführt. Dies führt zu zeitlichen Verzögerungen gegenüber einem Direktzugriff.

- **schlechte Wiederverwendbarkeit**

Durch die individuelle Anpassung der Adapter an die jeweilige Client-Anforderung weisen die Adapter eine schlechte Wiederverwendbarkeit auf.

- **Klassen-Adapter gilt nicht für Unterklassen**

Der Klassen-Adapter basiert auf Vererbung. Er kann somit nur diejenige Klasse adaptieren, von der er abgeleitet ist, nicht aber deren Unterklassen. Er passt also nur eine einzige Zielklasse an.

- **Voraussetzungen an die Programmiersprache bei einem Klassen-Adapter**

Klassen-Adapter erfordern, dass die verwendete Programmiersprache Schnittstellen oder Mehrfachvererbung unterstützt.

5.1.5 Einsatzgebiete

Das Adapter-Muster ermöglicht die Zusammenarbeit von Klassen mit inkompatiblen Schnittstellen. Es wird in der Regel dazu verwendet, um unabhängig voneinander 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,
- eine Klasse wiederverwendbar sein soll, aber zum Zeitpunkt der Entwicklung nicht klar ist, mit welchen weiteren Klassen sie zusammenarbeiten soll, oder
- eine Klasse wie die Klassen einer Klassenbibliothek nicht geändert werden darf.

Das Adapter-Muster ist praktisch in jedem Application Programming Interface (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 stimmen die Schnittstellen der bereits implementierten Anwendung und die Schnittstellen der "neuen" Plattform nicht überein, so dass ein Adapter dazwischen benötigt wird.

5.1.6 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 Muster **Fassade** und beim Muster **Adapter** werden vorhandene Programme gekapselt. Fassade und Adapter sind Wrapper, welche die vorhandenen Programme kapseln. 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. Beim Adapter hingegen gibt ein Client die zu implementieren Schnittstelle vor.

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

5.2 Das Strukturmuster Brücke

5.2.1 Name/Alternative Namen

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

5.2.2 Problem

Die Schnittstelle einer Klasse wird in diesem Muster als Abstraktion der Klasse bezeichnet. Die Abstraktion einer Klasse und ihre Implementierung sollen getrennt werden, damit beide unabhängig voneinander weiterentwickelt werden können. Clients sollen nur die Abstraktion kennen und nur über diese auf Objekte der Klasse zugreifen. Die Implementierung ist dadurch vor dem Client verborgen und kann ausgetauscht werden.

Gamma et al. [Gam15, S. 200] sagen:

"Entkopple eine Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können."



Man will die **Abstraktion und die Implementierung** einer Klasse **unabhängig voneinander** durch Unterklassenbildung **entwickeln** können.



Bleibt beispielsweise die Abstraktionsschnittstelle dieselbe und wird nur die Implementierung grundlegend verändert, so sind keine Änderungen am Client erforderlich.

5.2.3 Lösung

Vererbung oder die Realisierung einer Schnittstelle wie in Java kann für die Beziehung zwischen Abstraktion und Implementierung nicht verwendet werden, da in den genannten Fällen die Methodenköpfe der Abstraktion durch die Implementierung realisiert würden und damit eine starke Abhängigkeit zwischen Abstraktion und Implementierung bestehen würde.

Es soll möglich sein, Methoden der Abstraktion neue Methoden der Implementierung zuzuordnen.



Eingesetzt wird beim Muster Brücke eine **lose Kopplung zwischen Abstraktion und Implementierung in Form einer Referenz**.



Damit gibt es im Gegensatz zu einer statischen Kopplung durch Vererbung bzw. durch Realisierung einer Schnittstelle nur eine schwache Kopplung zwischen Abstraktion und Implementierung. Abstraktion und Implementierung befinden sich dabei jeweils in einer eigenen Klassenhierarchie.

Beim Muster **Brücke** realisiert die "Implementierung" Methoden der "Abstraktion". Im Gegensatz zu einer statischen Ableitung kann die Schnittstelle der Implementierung verschieden zu der Schnittstelle der Abstraktion sein.



Würde hingegen eine Abstraktion die Implementierung direkt ohne eine Schnittstelle für die Implementierung aufrufen, würden Änderungen der Implementierung voll auf die Abstraktion durchschlagen. Die Abstraktion wäre dann von der Implementierung abhängig.

Durch die Einführung einer Schnittstelle für die Implementierung ist eine Abstraktion von ihrer konkreten Implementierung unabhängig.



Ein konkreter Implementierer ist vollständig vor dem Aufrufer (Client) versteckt.



5.2.3.1 Klassendiagramm

Das folgende Bild zeigt das Klassendiagramm des Musters Brücke:

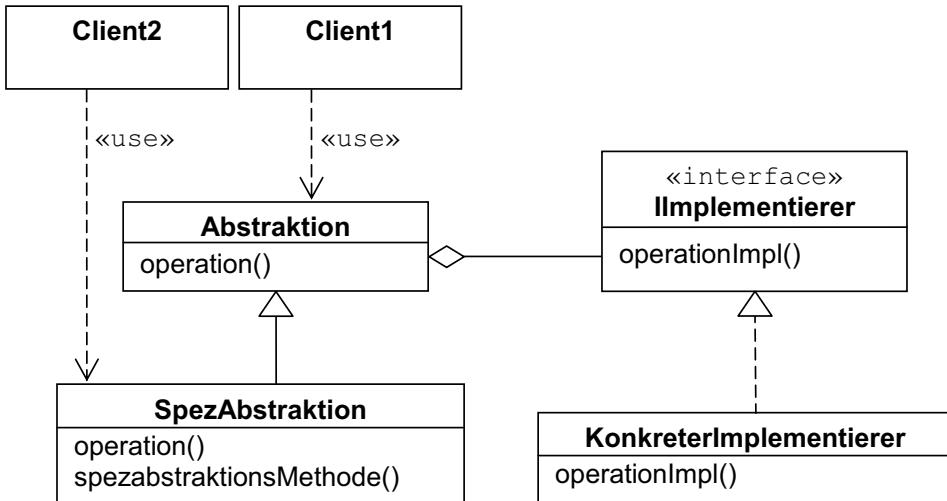


Bild 5-5 Klassendiagramm der Brücke

Alle Operationen einer Abstraktionsklasse werden auf der Basis der abstrakten Operationen der Schnittstelle `IImplementierer` realisiert [Gam94]. Damit sind die Abstraktionen nicht von der konkreten Implementierung abhängig. Das Ändern der Implementierungsklasse erfordert keine Neuübersetzung der Abstraktionsklasse und der Clienten.

Die konkrete Implementierung der Schnittstelle kann **zur Laufzeit** mit Hilfe von **Dependency Injection** von einem Injektor gesetzt werden.



Typischerweise bietet die Schnittstelle des **Implementierers nur primitive Operationen**, während die **Abstraktion komplizierte Operationen** auf Basis dieser primitiven Operationen definiert [Gam15, S. 204].

Aus der Sicht des Clients ist die Schnittstelle `IImplementierer` vollständig verborgen, lediglich die Klasse `Abstraktion` bzw. `SpezAbstraktion` ist dem Client bekannt. Es sei noch angemerkt, dass an die Stelle einer Schnittstelle `IImplementierer` auch eine abstrakte Klasse treten kann. Ebenso kann die Klasse `Abstraktion` abstrakt sein. Beides ist im Rahmen des Musters Brücke möglich.

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

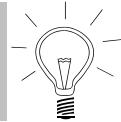


Die Brücke verbindet beide Seiten, d. h. beide Klassenhierarchien, miteinander. Die Methoden einer Abstraktion werden unter Verwendung der Methoden der Schnittstelle `IImplementierer` (siehe Bild 5-5) realisiert.

Das Muster Brücke legt nicht fest, wie die Implementierungsobjekte erzeugt werden. Beispielsweise kann eine abstrakte Fabrik⁵¹ 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 5-5 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.

Die Schnittstelle `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 zunichten machen.

5.2.3.2 Teilnehmer

Im Folgenden werden die Teilnehmer des Musters Brücke beschrieben:

- **Client**

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

⁵¹ Siehe Kapitel 7.2.

- **Abstraktion**

Die Klasse **Abstraktion** definiert die Schnittstelle, über welche ein Client auf die Funktionalität der Klasse **Abstraktion** zugreifen kann. Zusätzlich enthält die Klasse **Abstraktion** eine Referenz auf die von ihr selbst vorgegebene Schnittstelle **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 nach dem lis-kovschen Substitutionsprinzip an die Stelle eines Objekts der Klasse **Abstraktion**, so kann der Client dennoch nur die Methoden der Klasse **Abstraktion** benutzen.

- **IImplementierer**

Die Schnittstelle **IImplementierer** definiert die Schnittstelle einer konkreten Implementierung. Die Methoden der Schnittstelle **IImplementierer** müssen also die Erwartungen der Methoden der Klasse **Abstraktion** erfüllen. Die Schnittstelle **IImplementierer** bietet Operationen an, die zur Realisierung von Operationen der Klasse **Abstraktion** dienen.

- **KonkreterImplementierer**

Letztendlich muss die Schnittstelle **IImplementierer** für den jeweils vorliegenden konkreten Fall implementiert werden. Diese Aufgabe übernimmt die Klasse **KonkreterImplementierer**.

5.2.3.3 Dynamisches Verhalten

Das folgende Sequenzdiagramm veranschaulicht den Mechanismus einer Brücke:

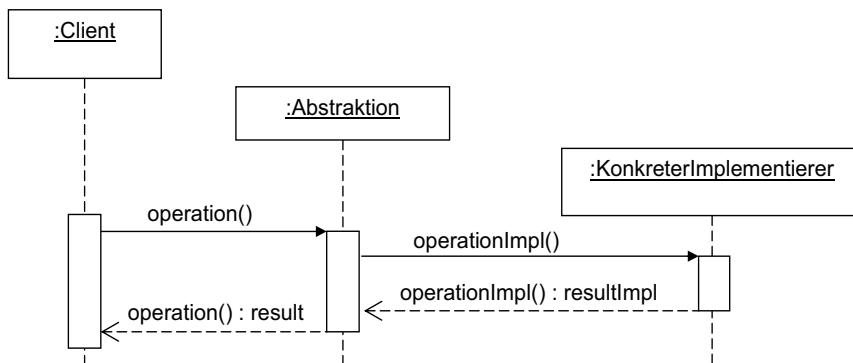


Bild 5-6 Sequenzdiagramm des Musters Brücke

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.

5.2.3.4 Programmbeispiel

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

Bild 5-7 zeigt das Klassendiagramm dieses Beispiels, aus dem auch die Rollen der beteiligten Klassen und Schnittstellen im Rahmen des Brücke-Musters ersichtlich werden:

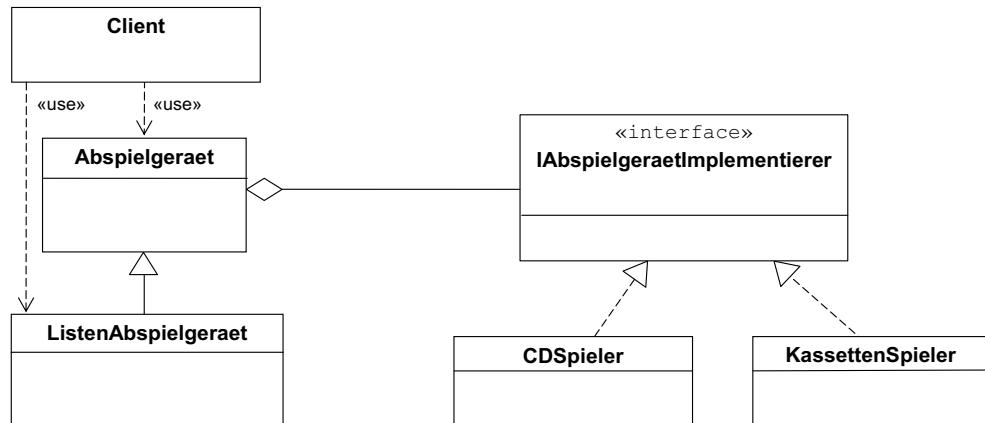


Bild 5-7 Klassendiagramm für das Beispiel Musikanlage

Die Klasse **AbspielDaten** ist eine Hilfsklasse, welche die abzuspielenden Daten enthält. Sie gehört nicht zum Muster Brücke und wird deswegen in Bild 5-7 nicht gezeigt. Hier die Klasse **AbspielDaten**:

```

// Datei: AbspielDaten.java

public class AbspielDaten {

    private final String daten;

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

```

@Override
public String toString() {
    return daten;
}
}
}

```

Die Klasse `Abspielgeraet` dient zur Abstraktion gegenüber dem Client. Sie enthält die "Brücke" zur Implementierung – in diesem Beispiel wird ein Objekt aggregiert, dessen Klasse die Schnittstelle `IAbspielgeraetImplementierer` implementiert. Hier nun der Quellcode der Klasse `Abspielgerät`:

```

// Datei: Abspielgeraet.java

// Abstraktion
public class Abspielgeraet {

    protected final IAbspielgeraetImplementierer impl;

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

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

    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.*;

// Spezialisierte Abstraktion
public class ListenAbspielgeraet extends Abspielgeraet {

    private final List<Integer> liederNummern;

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

```

```
public void abspielenUndAusschalten() { // neue Funktion
    for (int eachLiedNummer : liederNummern) {
        spieleAb(eachLiedNummer);
    }
    impl.ausschalten();
}
```

Die Schnittstelle `IAbspielgeraetImplementierer` definiert Methoden zum Abspielen und Ausschalten:

```
// Datei: IAbspielgeraetImplementierer.java

// Implementierer als Schnittstelle
public interface IAbspielgeraetImplementierer {
    // springt zu Track liedNummer
    void springeZuTrack(int liedNummer);
    // liest die Daten zum Abspielen
    AbspielDaten leseDaten();
    // schaltet das Gerät aus
    void ausschalten();

}
```

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

```
// Datei: CDSpieler.java

// Konkreter Implementierer
public class CDSpieler implements IAbspielgeraetImplementierer {

    @Override
    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.printf("Springe zu Lied %d durch Positionierung"
            + " des Lasers.%n", liedNummer);
    }

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

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

Zum 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. Diese Funktionalität wird im Beispielprogramm der Einfachheit halber nur in Grundzügen realisiert. Hier die Klasse `KassettenSpieler`:

```
// Datei: KassettenSpieler.java

// Konkreter Implementierer
public class KassettenSpieler
    implements IAbspielgeraetImplementierer {

    private int currentTrack;

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

    @Override
    public void springeZuTrack(int liedNummer) {
        int diff = liedNummer - currentTrack;
        if (diff > 0) {
            System.out.printf("Spule um %d Tracks vor.%n", diff);
        } else {
            System.out.printf("Spule um %d Tracks zurück.%n", -diff + 1);
        }
        System.out.printf("Nun sind wir an der richtigen Stelle"
            + " (Lied: %d)%n", liedNummer);
        currentTrack = liedNummer;
    }

    @Override
    public AbspielDaten leseDaten() {
        return new AbspielDaten("Kassettendaten");
    }

    @Override
    public void ausschalten() {
        System.out.println("Kassettenspieler ausgeschaltet.");
    }
}
```

Der Client nutzt nur die Funktionen der Abstraktionsklassen `Abspielgeraet` und `ListenAbspielgeraet` zur Steuerung eines Abspielvorgangs. Er kennt insofern keine Implementierungsobjekte.

Das Muster sagt nichts über die Erzeugung der Implementierungsobjekte aus. Um das Beispiel übersichtlich zu halten, wurde ein einfacher Weg gewählt: Das Hauptprogramm in der Klasse `TestAbspielgeraete` 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: TestAbspielgeraete.java

import java.util.*;

public class TestAbspielgeraete {

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

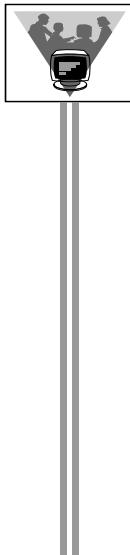
        Abspielgeraet b = new Abspielgeraet(new KassettenSpieler());
        b.spieleAb(5);

        List<Integer> abspielliste = Arrays.asList(1, 9, 3);

        ListenAbspielgeraet listenAbspielgeraet = new ListenAbspielgeraet(
            new CDSpieler(),
            abspielliste
        );

        // Dieser Client nutzt sowohl die Abstraktion als auch
        // die spezielle Abstraktion.
        listenAbspielgeraet.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 den 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.
```

5.2.4 Bewertung

5.2.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Unabhängigkeit der Abstraktion von der konkreten Implementierung**

Eine Abstraktion und eine konkrete Implementierung sind nur über die von der Abstraktion vorgegebene Schnittstelle einer konkreten Implementierung gekoppelt. Damit ist eine Abstraktion nicht von der konkreten Implementierung abhängig. Die konkrete Implementierung kann beispielsweise mit Hilfe von Dependency Injection der Abstraktion zugeordnet werden.

- **Möglichkeit der unabhängigen Weiterentwicklung von Abstraktion und Implementierung**

Abstraktionen und Implementierungen können unabhängig voneinander in neuen Unterklassen weiterentwickelt werden. Neue Abstraktionen und neue Implementierungen können sehr einfach hinzugefügt werden. Die Erweiterbarkeit von Abstraktion und Implementierung ist damit verbessert.

- **Unabhängigkeit eines Clients von der Implementierung einer Klasse**

Eine konkrete Implementierung ist vor dem Client vollständig 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.

- **dynamische Änderbarkeit**

Die Zuordnung einer Implementierung kann zur Laufzeit erfolgen und abgeändert werden. Sie ist nicht fest.

- **Erleichterung einer Schichtenbildung**

Die oberen Schichten eines Systems müssen nur die Abstraktion und ggf. deren Spezialisierung, nicht aber die Implementierung kennen.

- **gemeinsame Nutzung von Implementierungen durch mehrere Abstraktionen**

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

5.2.4.2 Nachteile

Der folgende Nachteil wird gesehen:

- **Erschweren der Übersicht**

Beim Muster Brücke 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.

5.2.5 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 Muster Brücke wird beispielsweise bei der Abstraktion verschiedener Datenbankschnittstellen wie etwa für Oracle, MySQL etc. angewandt oder bei der Erstellung plattformübergreifender Benutzeroberflächen.

In C++ kann das Muster Brücke verwendet werden, um die Implementierung einer Klasse physisch komplett zu verbergen. In C++ können zwar die Methodenrümpfe physisch aus einer Klassendefinition ausgelagert werden, aber nicht die Definition der Attribute. Folglich sind die von einer Klasse benutzten Datenstrukturen – auch die als privat deklarierten – für den Leser⁵² der Definition dieser Klasse immer ersichtlich. Ebenso müssen bei Änderungen von privaten Attributen (eigentlich ein Implementierungsdetail) alle abhängigen Klassen übersetzt werden. D. h., die nutzenden Klassen hängen von einem Implementierungsdetail der genutzten Klasse ab. Durch das Muster Brücke können auch die Attribute in eine Implementierungsklasse ausgelagert werden und in der ursprünglichen Klassendefinition ist nur die Brücke zu sehen, also die Aggregation eines Objekts einer Implementierungsklasse.

5.2.6 Ähnliche Entwurfsmuster

Sowohl bei der **Brücke** als auch beim **Adapter** soll die konkrete Implementierung versteckt werden. Der Adapter macht nachträglich eine Anpassung einer bereits vorhandenen Schnittstelle an die von einem bestimmten Client verlangte Form der Schnittstelle. Bei der Brücke ist es das Ziel, dass sich eine Schnittstelle – die sogenannte Abstraktion – und ihre Implementierung unabhängig voneinander entwickeln können. Dies muss bereits bei der Entwicklung eingepflegt werden.

Das **Proxy**-Muster definiert einen Stellvertreter für ein anderes Objekt, wobei es dessen Schnittstelle nicht ändert.

⁵² Für einen menschlichen Leser sind die Datenstrukturen sichtbar. Auf der Ebene der Programmierung stellt jedoch ein C++-Compiler den Zugriffsschutz (Information Hiding) sicher, sodass als privat deklarierte Attribute außerhalb einer Klasse nicht sichtbar sind und nicht benutzt werden dürfen.

Der **Dekorierer** erweitert ein anderes Objekt, wobei der Dekorierer dessen Schnittstelle nicht verändert.

5.3 Das Strukturmuster Dekorierer

5.3.1 Name/Alternative Namen

Dekorierer (engl. decorator).

5.3.1.1 Problem

Mit Vererbung kann man die Funktionalität einer Klasse erweitern. Dies erfolgt zur Kompilierzeit und damit ist man zur Laufzeit total inflexibel.

Das **Dekorierer-Muster** soll es erlauben, **zur Laufzeit** eine **zusätzliche Funktionalität** zu einem vorhandenen Objekt einer Klassenhierarchie in dynamischer Weise hinzuzufügen.

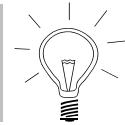


5.3.2 Lösung

Das Dekorierer-Muster ist ein **objektbasiertes Strukturmuster**. Die zu erweiternden **Objekte** gehören alle einer **Klassenhierarchie** an, an deren Spitze eine Basisklasse steht. Diese **Basisklasse** wird im Folgenden auch als "**Komponente**" bezeichnet.

Der sogenannte Dekorierer referenziert über eine Aggregation ein Objekt der zu verzierenden (dekorierenden) Klasse.

Der **Dekorierer** leitet Aufrufe von Methoden an das referenzierte Objekt weiter und führt eine zusätzliche Funktionalität aus, wenn er selbst aufgerufen wird und Methoden dieses Objekts aufruft.



Die zusätzliche Funktionalität "verziert" den ursprünglichen Methodenaufruf.

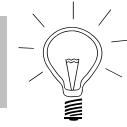
Damit der Client nichts merkt, wenn ein durch den Dekorierer dekoriertes Objekt an die Stelle eines undekorierten Objekts tritt, muss die Schnittstelle des Dekorierers die Schnittstelle des referenzierten Objekts einhalten. Um dieselbe Schnittstelle wie das ursprüngliche Objekt zu erhalten, leitet man die Klasse **Dekorierer** von der Klasse der referenzierten Komponente ab.

Nach dem **liskovschen Substitutionsprinzip** kann dann im Programm ein Objekt der Klasse **Dekorierer** an die Stelle eines Objekts der referenzierten Komponente treten, solange vom Dekorierer der Vertrag der referenzierten Klasse mit dem Client nicht gebrochen wird.



Dadurch kann der Client sowohl mit einem durch den Dekorierer dekorierten Objekt arbeiten als auch mit einem undekorierten Objekt.

Der **Dekorierer überschreibt** alle von der Klasse Komponente geerbten Methoden.



Die **überschreibende Methode** enthält

- ggf. das Anbringen der Verzierung und
- den Aufruf der ursprünglichen Methode des aggregierten Objekts nach dem Delegationsprinzip, um die ursprüngliche Funktionalität sicherzustellen.

Mit dem Verzierungsanteil darf die Klasse Dekorierer aber – wie schon erwähnt – den Vertrag der überschriebenen Methode nicht brechen.

Das zu dekorierende Objekt merkt nicht, dass es dekoriert wird. An ihm muss keine Veränderung vorgenommen werden.



5.3.2.1 Klassendiagramm

Die Klasse Komponente ist die Basisklasse aller zu dekorierenden Objekte und legt deren Schnittstelle fest. Im folgenden Klassendiagramm sollen auch Objekte der Klassen KonkreteKomponente1 oder KonkreteKomponente2, die von der Klasse Komponente abgeleitet sind, dekoriert werden können. Hier das Klassendiagramm:

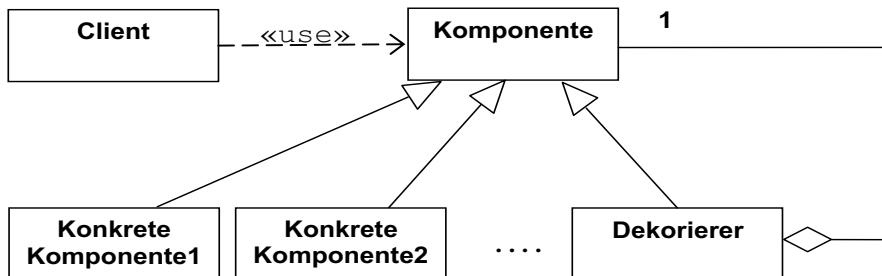


Bild 5-8 Klassendiagramm des Dekorierer-Musters

Die Klasse Dekorierer ist von der Klasse Komponente abgeleitet. Ferner aggregiert der Dekorierer gleichzeitig genau ein Objekt vom Typ Komponente.

Bei Einhaltung des liskovschen Substitutionsprinzips kann zur Laufzeit auch ein Objekt **jeder beliebigen Subklasse** (KonkreteKomponente1, KonkreteKomponente2, ...) der Klasse Komponente aggregiert und dekoriert werden.



Da eine aggregierte konkrete Komponente selbst zur Laufzeit bereits Methoden der Basisklasse Komponente überschrieben haben könnte, muss der Dekorierer die von der Basisklasse Komponente geerbten Methoden auf jeden Fall überschreiben.



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 gewünschte Zusatzfunktionalität ausführen.

Die Tatsache, dass die Klasse Dekorierer zur Kompilierzeit von der Klasse Komponente abgeleitet ist und der Dekorierer 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 vom Typ der Komponente. Diese werden vom Dekorierer aufgerufen, 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 weggecasted. Somit verbleiben bei den Methoden für den Dekorierer zwei Möglichkeiten für das Überschreiben. Der Dekorierer kann beim Überschreiben

- geerbte Methoden einfach weiterdelegieren oder
- Zusatzfunktionalität – also die Dekoration – hinzufügen.

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 seiner überschreibenden Methode die überschriebene Methode des aggregierten Objekts auf.



Ein Dekorierer muss das liskovsche Substitutionsprinzip einhalten.



Der Dekorierer darf nicht einfach die Methoden der Basisklasse Komponente aufrufen. Die vom Dekorierer geerbten Methoden der Klasse Komponente würden nicht die gewünschte Funktionalität liefern, wenn eine konkrete Komponente die Methoden der Klasse Komponente erweitert haben sollte.

Der Dekorierer muss eine geerbte Methode auf jeden Fall überschreiben, auch wenn er sie gar nicht erweitern will.



In der Praxis wird daher häufig ein sogenannter "**abstrakter Dekorierer**"⁵³ 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 abgeleitet werden.

Der Entwickler eines **konkreten Dekorierers** muss bei Vorliegen eines abstrakten Dekorierers nur noch die für ihn als Dekorierer interessanten Methoden betrachten und überschreiben. Er braucht sich nicht um alle von der Klasse Komponente geerbten Methoden zu kümmern, die unverändert bleiben. Diese werden bereits vom abstrakten Dekorierer überschrieben.



Diese Vorgehensweise hat insbesondere dann einen Vorteil, wenn mehrere konkrete Dekorierer zu entwickeln sind (siehe Kapitel 5.3.3.4).

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 5-9 dargestellt:

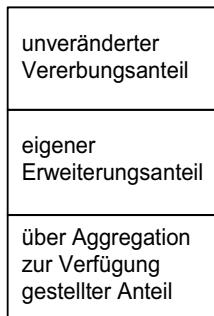


Bild 5-9 Aufbau eines Dekorierers

⁵³ Die hier als "abstrakter Dekorierer" bezeichnete Klasse ist nicht abstrakt im Sinne einer abstrakten Klasse. Er besitzt keine abstrakte Methode, sondern überschreibt alle geerbten Methoden und delegiert an das aggregierte Objekt weiter. Er ist "abstrakt" in dem Sinne, dass er keine konkrete Dekorationsfunktion implementiert.

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.

Die beiden anderen Anteile sind **nur 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, nicht aber über den statischen Anteil.

Der Dekorierer darf die **von der Klasse Komponente geerbten Daten nicht benutzen**.



Der von der Klasse Komponente geerbte Anteil ist im Dekorierer überflüssig.



Die Aggregation zeigt erst dann 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 des Dekorierers mit einer Schnittstelle für die Klassenhierarchie

Steht an der Spitze der ursprünglichen Klassenhierarchie eine **Schnittstelle** statt einer Basisklasse, so kann das Dekorierer-Muster ebenfalls angewandt werden. In dieser Variante muss die Klasse Dekorierer die Schnittstelle **IKomponente** implementieren und ein Objekt vom Typ dieser Schnittstelle aggregieren, wie es das folgende Bild zeigt:

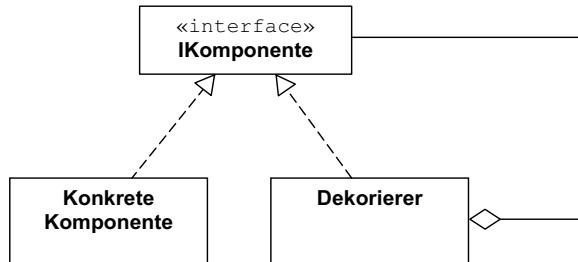


Bild 5-10 Dekorierer als Realisierung einer Schnittstelle

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 einer Schnittstelle hat aber einen entscheidenden Vorteil: Die Probleme mit den geerbten Anteilen existieren hier nicht.



Auf Grund dieses Vorteils wird im Programmbeispiel in Kapitel 5.3.3.5 mit dieser Lösungsvariante gearbeitet.

5.3.2.2 Teilnehmer

Im Folgenden werden die Teilnehmer des Musters Dekorierer in Bild 5-8 beschrieben:

- **Komponente**

Mit **Komponente** wird 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** unter Einhaltung des liskovschen Substitutionsprinzips abgeleiteten Klasse zugreifen. Der Dekorierer überschreibt alle geerbten Methoden und leitet die Aufrufe per Delegation an das aggregierte Objekt weiter. Enthält die überschreibende Methode des Dekorierers eine zusätzliche Funktionalität, so wird die zusätzliche

Funktionalität 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.

5.3.2.3 Dynamisches Verhalten

Der Client ruft eine Methode des Dekorierers auf.

Enthält der Dekorierer keine Zusatzfunktion, so leitet er den Methodeaufruf direkt an das aggregierte Objekt der Klasse Komponente weiter, welches diesen Aufruf abarbeitet. Das Objekt der Klasse Komponente gibt das Ergebnis über das Objekt der Klasse Dekorierer an das Client-Programm zurück.

Enthält die überschreibende Methode des Dekorierers eine Zusatzfunktion des Dekorierers, so führt der Dekorierer vor oder nach bzw. vor und nach der Delegation des Methodenauftrags diesen zusätzlichen Code selbst aus.

Das folgende Bild visualisiert den Ablauf für die Dekoration einer Komponente:

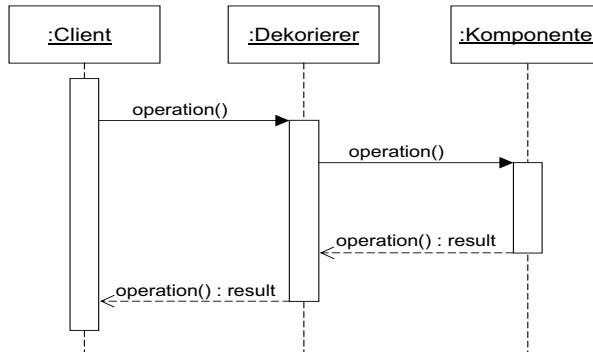


Bild 5-11 Sequenzdiagramm für die Dekoration einer Komponente

Wegen des liskovschen Substitutionsprinzips kann in Bild 5-11 auch eine konkrete Komponente bzw. Subklasse an die Stelle einer Komponente treten. Genau das ist in dem in Kapitel 5.3.3.5 folgenden Programmbeispiel der Fall.

5.3.2.4 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 – insbesondere, wenn

viele geerbte Methoden nur delegiert werden müssen – ein sogenannter "**abstrakter Dekorierer**" sinnvoll. Wie bereits in Kapitel 5.3.2.1 beschrieben wurde, ruft ein abstrakter Dekorierer für jede seiner Methoden nur die entsprechende Methode der aggregierten Komponente auf (Delegationsprinzip).

Konkrete Dekorierer werden von dem abstrakten Dekorierer abgeleitet und müssen nur noch diejenigen Methoden erweitern, die für die Erbringung ihrer zusätzlichen Funktionalität notwendig sind. Das folgende Klassendiagramm zeigt einen abstrakten Dekorierer und konkrete Dekorierer:

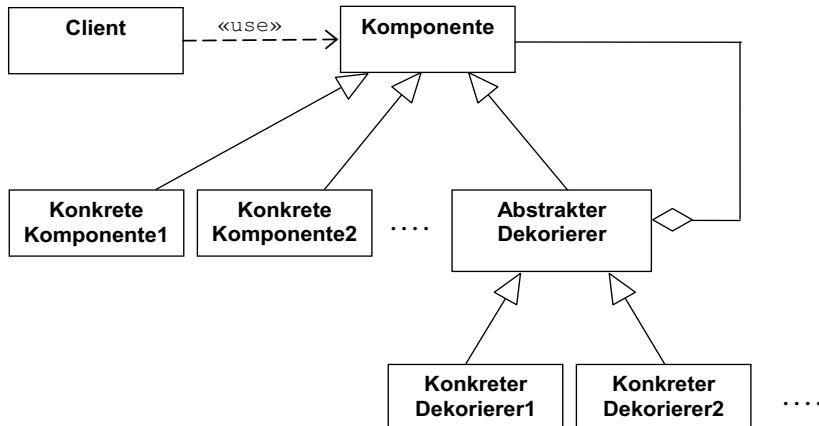


Bild 5-12 Klassendiagramm mit mehreren konkreten Dekorierern

Bild 5-12 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.

5.3.2.5 Programmbeispiel

Das folgende Bild zeigt das Klassendiagramm dieses Beispiels:

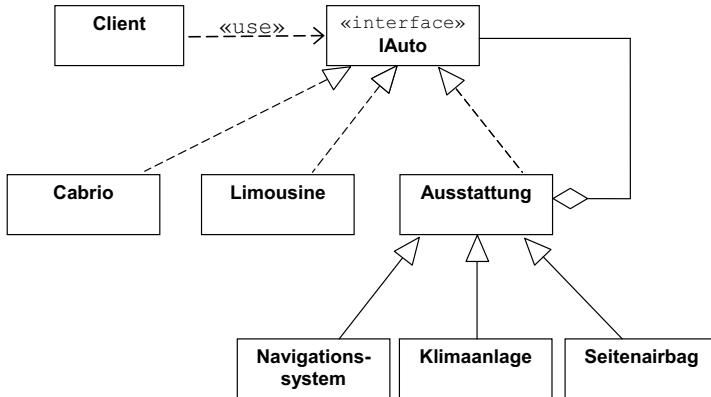


Bild 5-13 Klassendiagramm des folgenden Programmbeispiels zum Dekorierer

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 5.3.2.1 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 Unterklasse (z. B. CabrioKlimaAirbags) geben muss.

Die Methoden gibKosten() und zeigeDetails() dienen der letztendlichen Ausgabe in diesem Programmbeispiel. Sie zeigen auf, dass sich durch die Dekorierung eine Schachtelung der Methodenaufrufe ergibt, durch die beispielsweise ein Aufsummieren der Kosten möglich ist.

Hier die Schnittstelle IAuto:

```

// Datei: IAuto.java

public interface IAuto {

    void zeigeDetails();

    int gibKosten();

}
  
```

Die Klasse Limousine stellt eine zu dekorierende konkrete Komponente dar. Die Methoden geben die Art des Autos sowie dessen Grundkosten aus bzw. zurück. Hier die Klasse Limousine:

```
// Datei: Limousine.java

public class Limousine implements IAuto {

    @Override
    public void zeigeDetails() {
        System.out.print("Limousine");
    }

    @Override
    public int gibKosten() {
        return 35000;
    }
}
```

Die Klasse `Cabrio` stellt eine weitere zu dekorierende konkrete Komponente dar, welche die gleichen Methoden mit jedoch anderen Werten als die entsprechenden Methoden der Klasse `Limousine` besitzt:

```
// Datei: Cabrio.java

public class Cabrio implements IAuto {

    @Override
    public void zeigeDetails() {
        System.out.print("Cabrio");
    }

    @Override
    public int gibKosten() {
        return 50000;
    }
}
```

Die Klasse `Ausstattung` stellt einen **abstrakten Dekorierer** dar, der gemeinsamen 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 in einem abstrakten Dekorierer gemäß Kapitel 5.3.2.4 wurde aus Platzgründen 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 final IAuto auto;

    protected Ausstattung(IAuto auto) {
        this.auto = auto;
    }
}
```

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 der Kosten der Ausstattung. Es folgen die Klassen Klimaanlage, Navigationssystem und Seitenairbags:

```
// Datei: Klimaanlage.java

public class Klimaanlage extends Ausstattung {

    public Klimaanlage(IAuto auto) {
        super(auto);
    }

    @Override
    public void zeigeDetails() { // "dekiert" die Details
        auto.zeigeDetails();
        System.out.print(", Klimaanlage");
    }

    @Override
    public int gibKosten() { // "dekiert" die Kosten
        return auto.gibKosten() + 1500;
    }
}

// Datei: Navigationssystem.java

public class Navigationssystem extends Ausstattung {

    public Navigationssystem(IAuto auto) {
        super(auto);
    }

    @Override
    public void zeigeDetails() { // "dekiert" die Details
        auto.zeigeDetails();
        System.out.print(", Navigationssystem");
    }

    @Override
    public int gibKosten() { // "dekiert" die Kosten
        return auto.gibKosten() + 2500;
    }
}
```

```
// Datei: Seitenairbags.java

public class Seitenairbags extends Ausstattung {

    public Seitenairbags(IAuto auto) {
        super(auto);
    }

    @Override
    public void zeigeDetails() { // "dekoriert" die Details
        auto.zeigeDetails();
        System.out.print(", Seitenairbags");
    }

    @Override
    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:

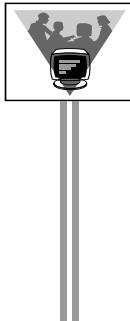
```
// Datei: Client.java

public class Client {

    public static void main(String[] args) {
        // Limousine mit Klimaanlage
        IAuto auto = new Klimaanlage(new Limousine());
        auto.zeigeDetails();
        System.out.printf("%nfür %,d Euro.%n%n", auto.gibKosten());

        // Dynamische Erweiterung der Limousine mit Ausstattungen
        auto = new Navigationssystem(new Seitenairbags(auto));
        auto.zeigeDetails();
        System.out.printf("%nfür %,d Euro.%n%n", auto.gibKosten());

        // Cabrio Variante
        auto = new Navigationssystem(new Seitenairbags(new Cabrio()));
        auto.zeigeDetails();
        System.out.printf("%nfür %,d Euro.%n%n", auto.gibKosten());
    }
}
```



Hier das Protokoll des Programmlaufs:

Limousine, Klimaanlage
für 36,500 Euro.

Limousine, Klimaanlage, Seitenairbags,
Navigationssystem
für 40,000 Euro.

Cabrio, Seitenairbags, Navigationssystem
für 53,500 Euro.

5.3.3 Bewertung

5.3.3.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Dekorierer ist einer Komponente nicht bekannt**

Eine Komponente kennt ihren Dekorierer nicht.

- **dynamische Erweiterung**

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 ein Dekorierer seine zusätzliche Funktionalität in dynamischer Weise zu einem Objekt hinzufügen und auch wieder entfernen. Wird hingegen eine Klasse statisch durch die Bildung von Unterklassen erweitert, so wird bereits zum Zeitpunkt des Kompilierens die Erweiterung einer Klasse statisch festgelegt und kann auch nicht mehr geändert werden.

- **übersichtliche Programmierung**

Es kann einfacher sein, mit einem Dekorierer inkrementell als mit einer sehr komplexen, parametrisierbaren Klasse zu arbeiten.

- **gleichzeitige Dekoration mehrerer Klassen**

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.

- **Kombination mehrerer Dekorierer**

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.

- **flachere Vererbungshierarchie**

Das Dekorierer-Muster erzeugt eine nur unwesentlich höhere Vererbungshierarchie im Vergleich zu der Hierarchie der zu dekorierenden Klassen. Fügt man jedoch die Dekorationsfunktion über die Bildung von Unterklassen hinzu, kann die Zahl der Ebenen der Vererbungshierarchie stark zunehmen, wie beispielsweise in Bild 5-15 im Kapitel 5.3.4.1 zu sehen ist.

- **Vermeiden von Code-Duplikaten**

Ebenfalls in Bild 5-15 ist zu sehen, dass die Dekorationsfunktion in mehreren Klassen implementiert werden muss, wenn man die "klassische" Lösung per Ableitung wählt. Das Dekorierer-Muster vermeidet Code-Duplikate dadurch, dass die Dekorationsfunktion in einer einzigen Klasse implementiert wird.

5.3.3.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **Verzögerung durch Delegation**

Dekoriert ein Dekorierer eine Klasse mit sehr vielen Methoden, erweitert aber nur sehr wenige Methoden, so besteht der Dekorierer zum größten Teil nur aus Delegationsmethoden. Eine Delegation führt aber zu einer Verzögerung.

- **schwierige Fehlersuche in kombinierten Dekorierern**

Fehler in kombinierten Dekorierern sind oft sehr schwer zu finden.

- **Nichtverwendung des geerbten Anteils**

Man hat einen geerbten Anteil, den man gar nicht nutzt. Dieser Nachteil entfällt, wenn an Stelle der Klasse Komponente eine Schnittstelle verwendet wird, die der Dekorierer realisiert (siehe Lösungsvariante mit einer Schnittstelle in Kapitel 5.3.2.1).

5.3.4 Einsatzgebiete

Die Möglichkeiten des Dekorierer-Musters sollen im Folgenden anhand von zwei Beispielen gezeigt werden. Das eine Beispiel (siehe Kapitel 5.3.4.1) ist in den Stream-Klassen von Java zu finden, beim anderen Beispiel (siehe Kapitel 5.3.4.2) handelt es

sich um die Erweiterung von grafischen Elementen. Dabei sollen Elemente von grafischen Oberflächen wie beispielsweise Textfelder oder Schaltflächen durch Dekorierer mit grafischen Zusätzen wie etwa einem Rahmen erweitert werden können.

5.3.4.1 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 nicht in dieser Form 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:

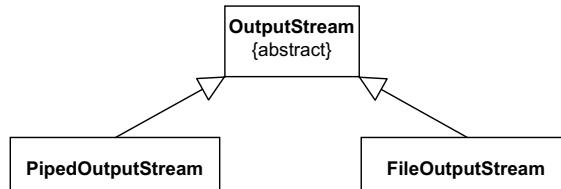


Bild 5-14 Vererbungshierarchie der `OutputStream`-Klassen

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:

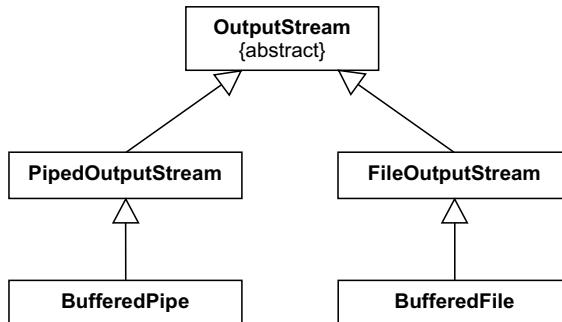


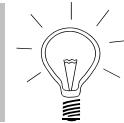
Bild 5-15 Erweiterung der Funktionalität durch Ableiten

Die Realisierung einer in einer Klassenhierarchie mehrfach auftretenden Zusatzfunktionalität durch eine Ableitung ist schon deshalb unbefriedigend, weil dann der gleiche Code in der Klassenhierarchie mehrmals auftritt.



Wie so oft, liegt die bessere Lösung nicht direkt auf der Hand.

Die Fähigkeit der Pufferung muss zentral so ausgelagert werden, dass sie für alle Klassen, welche diese Funktionalität zusätzlich nutzen wollen, nur ein einziges Mal implementiert wird.



Das folgende Bild zeigt die bessere Lösung:

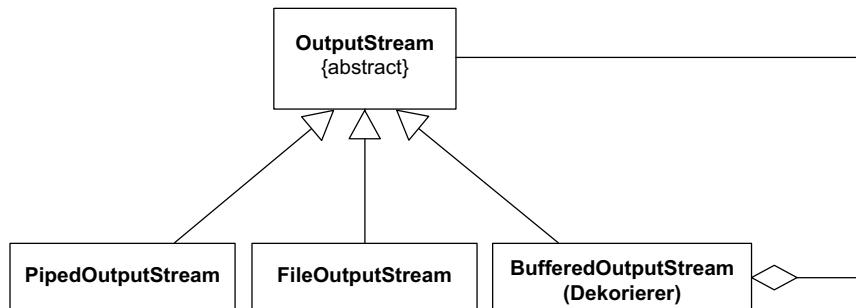


Bild 5-16 Beispiel für den Dekorierer

Die Klasse `BufferedOutputStream` wird als Dekorierer angelegt. Der Dekorierer implementiert genau dieselbe Schnittstelle wie die anderen Klassen vom Typ `OutputStream`, 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:

```

// Datei: BufferedOutputStream.java

import java.io.*;

public class BufferedOutputStream extends OutputStream {

    private final OutputStream out; // dekoriertes Objekt
    private final byte[] buffer = new byte[1024]; // Puffergröße
    private int counter = 0;

    public BufferedOutputStream(OutputStream s) {
        out = s;
    }

    public void write(int b) {
        if (counter == buffer.length) {
            flush();
            counter = 0;
        }
        buffer[counter] = (byte) b;
        counter++;
    }

    public void flush() {
        try {
            out.write(buffer);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
  
```

```
@Override
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
}

@Override
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
}

@Override
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;
    }
}

@Override
public void write(byte[] b) throws IOException {
    for (int 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;
        }
    }
}

@Override
public void write(byte[] b, int off, int len) throws IOException {
    for (int 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 – wie bereits erwähnt – 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.

5.3.4.2 Anwendungsbeispiel 2: grafische Komponenten

Im folgenden Beispiel dekoriert der Dekorierer im wahrsten Sinne des Wortes, nämlich er verziert grafische Komponenten mit einem Rahmen:

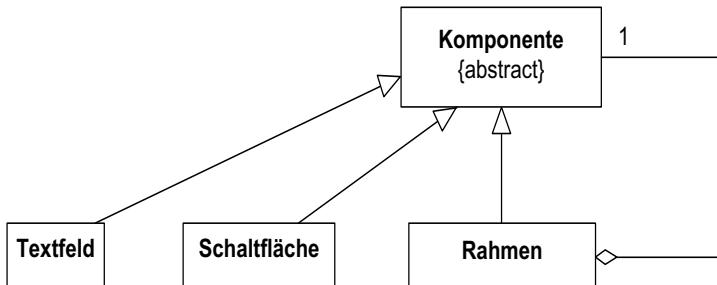


Bild 5-17 Weiteres Beispiel für den Dekorierer

Auf lauffähigen Code wird an dieser Stelle verzichtet.

Ein ganz ähnliches Beispiel findet sich in Kapitel 5.5.5 beim Kompositum-Muster (vgl. Bild 5-25). Der Unterschied zwischen diesen beiden Beispielen liegt aber in der Funktionalität: Durch die Dekoriererkelasste `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, da beim Muster Dekorierer nur ein einziges Objekt aggregiert wird. Das Zusammenfassen leistet beispielsweise die Klasse `JPanel` in Bild 5-25 beim Kompositum-Muster.

Da die Klasse `JPanel` aus Bild 5-25 wiederum eine grafische Komponente darstellt, könnte sie durch die Dekoriererkelasste mit einem Rahmen verziert werden.

5.3.5 Ähnliche Entwurfsmuster

Vom Aufbau des Klassendiagramms her ist das Entwurfsmuster **Kompositum** sehr ähnlich zu dem **Dekorierer-Muster**. Wenn man die beiden Klassendiagramme in Bild 5-8 und in Bild 5-21 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 jedoch mit nur einer einzigen aggregierten Komponente angesehen werden. Es zeigen sich jedoch erhebliche Unterschiede in der Verwendung des Musters. Während mit Hilfe des Dekorierer-

Musters eine zusätzliche 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. 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 die Implementierung einer von einem Kontextobjekt aggregierten Schnittstelle zur Laufzeit austauschen. 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 und dabei nicht die Realisierung einer Schnittstelle ausgetauscht. 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 konkreten Strategie versehen (kompletter Austausch der Realisierung eines durch eine Schnittstelle vorgegebenen Algorithmus). Damit wird keine Zusatzfunktionalität erzeugt, sondern es findet ein Austausch des gesamten Algorithmus der Strategie für das Kontextobjekt statt.

Ebenso wie ein **Besucher** realisiert ein **Dekorierer** eine neue Funktionalität. Das Besucher-Muster erlaubt es, allen Objekten, die in einer Datenstruktur aus Objekten verschiedener Klassen zusammengefasst sind, eine neue Funktion hinzuzufügen, die alle Objekte der Datenstruktur "besucht". Mit dem Dekorierer-Muster können jedoch nur einzelne Objekte erweitert werden. Die dekorierbaren Objekte müssen beim Dekorierer-Muster aus einer Klassenhierarchie mit einer gemeinsamen Basisklasse stammen, während die Klassen der besuchbaren Objekte keinerlei Beziehung untereinander besitzen müssen. 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.

5.4 Das Strukturmuster Fassade

5.4.1 Name/Alternative Namen

Fassade (engl. facade oder auch façade).

5.4.2 Problem

Das Muster Fassade bzw. das Fassadenmuster stellt ein ganzes Subsystem⁵⁴ nach außen dar, wie wenn dieses Subsystem ein einziges Objekt wäre. Dabei wird für dieses Subsystem eine einzige Schnittstelle konstruiert.

Die Fassade soll die einzelnen Instanzen des betreffenden Subsystems verbergen, sodass ein Client über das komplexe Subsystem nichts wissen muss, sondern nur die Fassade kennen muss. Der Client muss beispielsweise die Klassen des Subsystems, ihre Beziehungen und Abhängigkeiten nicht kennen.

Der Trick beim Fassadenmuster ist, dass man mit der Fassade eine **einfache Schnittstelle** erfindet, welche die Steuerung mehrerer, aufeinanderfolgender Aufrufe im Subsystem durch einen einzigen Aufruf einer Methode der Fassade zur Verfügung stellt.



Auf diese Weise vereinfacht die Fassade die Arbeit mit dem Subsystem für den Programmierer.

Die Fassade delegiert den Aufruf ihrer Methoden an die hinter der Fassade versteckten Objekte. Die Fassade soll also keine neue Funktionalität definieren, sondern sie soll nur wissen, welche Objekte hinter der Fassade für die Bearbeitung eines Aufrufs über die Fassade zuständig sind und soll deren Methoden bei Bedarf aufrufen.

Die Fassade soll die Details der internen Struktur eines Subsystems verbergen. Dadurch soll die interne Architektur eines Subsystems geändert werden können, ohne dass die Kundenobjekte abgeändert werden müssen.



Die Schnittstelle einer Fassade kann frei entworfen werden.

Die von der Fassade angebotene Schnittstelle umfasst aber **meist nur die häufig verwendeten Funktionen** des entsprechenden Subsystems und stellt in diesem Fall nur eine Teilmenge der vom Subsystem insgesamt implementierten Funktionen zur Verfügung.



⁵⁴ 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.

Kunden, denen eine Fassadenklasse nicht genügt, können ggf. die Module – objektorientiert: die Objekte – hinter der Fassade direkt aufrufen. Dies muss individuell festgelegt werden. Das Muster macht hierzu keine Aussage.



Kunden, welche direkt auf Objekte des Subsystems zugreifen, können die Vorteile des Information Hiding durch die Fassade nicht nutzen.



5.4.3 Lösung

Große Softwaresysteme bestehen meist aus mehreren Subsystemen, die miteinander interagieren. Diese Modularität dient nicht nur der Übersichtlichkeit und der Wartbarkeit, sondern führt auch zu einer einfacheren Verwaltung der Zugriffsrechte.

Damit sich Programmierer nicht mit den Interna der Subsysteme eines Systems auseinandersetzen müssen, erfolgt der Zugang eines Programmierers zu einem Subsystem beim Fassadenmuster über eine vereinfachte Schnittstelle, welche in der Lage ist, eine ganze Folge von Aufrufen eines Subsystems zu kapseln.



Diese Schnittstelle wird in einer Fassadenklasse bereitgestellt und kapselt ein Subsystem gegenüber den aufrufenden Kunden (Clients).

Die Aufrufe von Methoden der Fassade werden von der Fassade an die Klassen des Subsystems weitergeleitet (delegiert).



In den meisten Fällen greift eine Methode einer Fassadenklasse auf mehrere Objekte eines Subsystems zu, wobei diese Objekte jeweils einen Teil der entsprechenden Aufgabe ausführen.

5.4.3.1 Klassendiagramm

Beim Muster Fassade wird die Schnittstelle zwischen den Clients und den Objekten eines Subsystems durch eine eigene Klasse zur Verfügung gestellt, die sogenannte Fassadenklasse. Diese ist im folgenden Bild dargestellt:

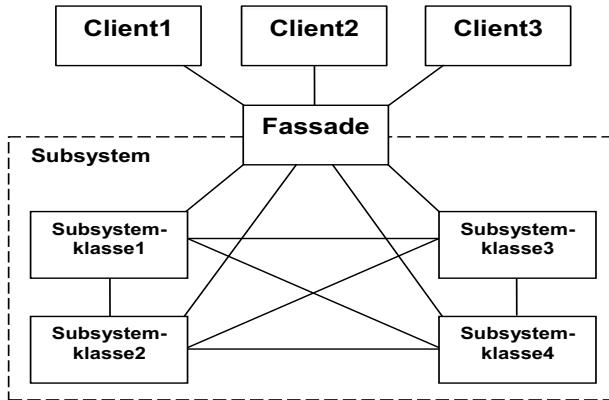


Bild 5-18 Beispiel für ein Klassendiagramm des Musters Fassade

Die Hüllkurve um das Subsystem in Bild 5-18 ist hierbei frei und nicht UML-konform gezeichnet.

5.4.3.2 Teilnehmer

Die Teilnehmer des Musters Fassade sind:

- **ClientX**

Ein Objekt der Klasse ClientX greift über ein Objekt der Klasse Fassade auf die Objekte eines Subsystems zu.

- **Fassade**

Die Klasse Fassade bietet eine vereinfachte Schnittstelle zum Zugriff auf aufeinanderfolgende Aufrufe von Objekten der Klassen eines Subsystems an.

- **SubsystemklasseX**

Auf die Klasse SubsystemklasseX eines Subsystems wird von einem Client nach dem Fassadenmuster über die zugeordnete Fassade zugegriffen.

5.4.3.3 Dynamisches Verhalten

Ein Client ruft eine Methode des Objekts einer Fassadenklasse auf. Das Objekt der Klasse Fassade leitet diesen Aufruf an Objekte des Subsystems weiter. Eventuelle Ergebnisse der Methodenaufrufe einer Subsystemklasse werden auf dem umgekehrten Weg an den Client zurückgeliefert.

Prinzipiell können viele Methoden von Objekten aus unterschiedlichen Subsystemklassen von einer Fassade aufgerufen werden, um die Aufgaben der von der jeweiligen Fassade angebotenen Methoden zu erledigen.

Das folgende Sequenzdiagramm zeigt der Übersichtlichkeit wegen nur den Zugriff auf einziges Objekt einer Subsystemklasse über eine Fassade:

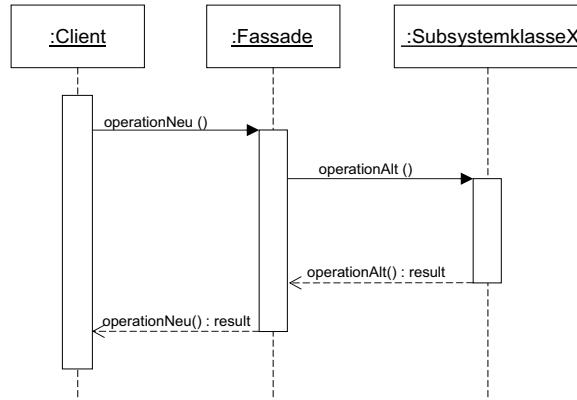


Bild 5-19 Prinzipielles Sequenzdiagramm des Fassadenmusters

In diesem Beispiel ruft der Client die Methode `operationNeu()` der Fassade auf, welche den Aufruf an die Methode `operationAlt()` des Objekts der Klasse SubsystemklasseX weiterleitet.

5.4.3.4 Programmbeispiel

In dem folgenden Beispiel wird das Fassadenmuster dazu verwendet, das Subsystem Lagerverwaltung zu vereinfachen. Im Lager werden Motoren, Getriebe und Fahrwerke verwaltet.

Beim Bau eines Autos soll über eine Methode der Fassade jeweils ein Objekt aus dem Fahrwerk-Lager, dem Getriebe-Lager und dem Motoren-Lager entnommen werden. Die Fassade soll es erlauben, dass alle 3 Teile aus den verschiedenen Lagern stets in einfacher Weise gemeinsam entnommen werden. Ebenso sollen die entsprechenden Lager über eine Methode der Fassade gemeinsam um jeweils 1 Stück nachgefüllt werden.

Es gibt drei Lagerklassen, die Klasse `FahrwerkLager`, die Klasse `GetriebeLager` und die Klasse `MotorenLager`. Die Objekte dieser 3 Klassen sollen nur über die Fassade zugänglich sein.

Im Folgenden werden diese 3 Lagerklassen beschrieben.

Die Klasse `FahrwerkLager` repräsentiert die erste Subsystemklasse:

```
// Datei: FahrwerkLager.java

public class FahrwerkLager {

    private int fahrwerkeAnzahl;

    public void lagerFuellen(int anzahl) {
        fahrwerkeAnzahl += anzahl;
    }
}
```

```
System.out.printf("Lager wurde um %d Fahrwerke auf insgesamt"
    + " %d Fahrwerke aufgestockt.%n", anzahl, fahrwerkeAnzahl);
}

public void fahrwerkeEntnehmen(int anzahl) {
    fahrwerkeAnzahl -= anzahl;
    System.out.printf("Für die Produktion wurden %d Fahrwerke "
        + "entnommen.%n", anzahl);
}
}
```

Die folgende Klasse **GetriebeLager** stellt die zweite Subsystemklasse dar:

```
// Datei: GetriebeLager.java

public class GetriebeLager {

    private int getriebeAnzahl;

    public void lagerFuellen(int anzahl) {
        getriebeAnzahl += anzahl;
        System.out.printf("Lager wurde um %d Getriebe auf insgesamt"
            + " %d Getriebe aufgestockt.%n", anzahl, getriebeAnzahl);
    }

    public void getriebeEntnehmen(int anzahl) {
        getriebeAnzahl -= anzahl;
        System.out.printf("Für die Produktion wurden %d Getriebe "
            + "entnommen.%n", anzahl);
    }
}
```

Die Klasse **MotorenLager** bildet die dritte Subsystemklasse:

```
// Datei: MotorenLager.java

public class MotorenLager {

    private int motorenAnzahl;

    public void lagerFuellen(int anzahl) {
        motorenAnzahl += anzahl;
        System.out.printf("Lager wurde um %d Motoren auf insgesamt"
            + " %d Motoren aufgestockt.%n", anzahl, motorenAnzahl);
    }

    public void motorenEntnehmen(int anzahl) {
        motorenAnzahl -= anzahl;
        System.out.printf("Für die Produktion wurden %d Motoren "
            + "entnommen.%n", anzahl);
    }
}
```

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()`. Diese Methoden der Fassade müssen dabei sicherstellen, dass aus jedem der drei Lager stets gleich viele Teile entnommen bzw. ihm hinzugefügt werden.

In diesem Beispiel erstellt die Fassade die Objekte des Subsystems. Generell ist es nicht die Aufgabe der Fassade, die Objekte eines Subsystems zu erstellen. Das ist eine Besonderheit dieser Lösung.



Hier die Klasse `Lagerfassade`:

```
// Datei: LagerFassade.java

public class LagerFassade {

    private final FahrwerkLager fahrwerkLager;
    private final GetriebeLager getriebeLager;
    private final MotorenLager motorenLager;

    public LagerFassade() {
        fahrwerkLager = new FahrwerkLager();
        getriebeLager = new GetriebeLager();
        motorenLager = new MotorenLager();
    }

    public void alleLagerFuellen(int anzahl) {
        System.out.println("Bestände aller Lager werden gefüllt.");
        fahrwerkLager.lagerFuellen(anzahl);
        getriebeLager.lagerFuellen(anzahl);
        motorenLager.lagerFuellen(anzahl);
        System.out.println();
    }

    public void produktionsteileEntnehmen(int anzahl) {
        System.out.println("Alle für die Produktion benötigten Teile"
            + " werden entnommen.");
        fahrwerkLager.fahrwerkeEntnehmen(anzahl);
        getriebeLager.getriebeEntnehmen(anzahl);
        motorenLager.motorenEntnehmen(anzahl);
        System.out.println();
    }
}
```

Die im Folgenden gezeigte Klasse `TestFassade` erstellt ein Objekt der Klasse `LagerFassade` und kann somit alle drei Lager mit den von der Fassade bereitgestellten Methoden verwalten:

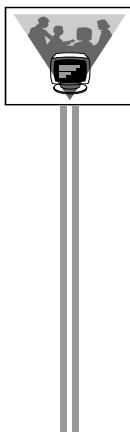
```
// Datei: TestFassade.java

public class TestFassade {

    public static void main(String[] args) {
        LagerFassade fassade = new LagerFassade();

        // Lager werden durch Lagerverwaltung gefüllt.
        fassade.alleLagerFuellen(10);

        // Teile werden durch die Produktion entnommen.
        fassade.produktionsteileEntnehmen(5);
    }
}
```



Hier das Protokoll des Programmlaufs:

Bestände aller Lager werden gefüllt.
Lager wurde um 10 Fahrwerke auf insgesamt 10 Fahrwerke aufgestockt.
Lager wurde um 10 Getriebe auf insgesamt 10 Getriebe aufgestockt.
Lager wurde um 10 Motoren auf insgesamt 10 Motoren aufgestockt.

Alle für die Produktion benötigten Teile werden entnommen
Für die Produktion wurden 5 Fahrwerke entnommen.
Für die Produktion wurden 5 Getriebe entnommen.
Für die Produktion wurden 5 Motoren entnommen.

5.4.4 Bewertung

5.4.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Subsystemklassen sind unabhängig von der Fassade**

Subsystemklassen kennen ihre Fassade nicht. Dies führt zu einer gewissen Entkopplung.

- **Subsysteme können einfacher verwendet werden**

Der Zugang zu einem Subsystem erfolgt beim Fassadenmuster über eine vereinfachte Schnittstelle, welche in der Lage ist, eine ganze Folge von Aufrufen des Subsystems zu kapseln.

- **lose Kopplung von Client und gekapselten Funktionen**

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 Implementierung der zugeordneten Fassade anpassen, nicht jedoch die Clients, die diese Fassade benutzen.

- **einfachere Schichtenbildung**

Das Verwenden einer Fassade erleichtert es, ein System und die Abhängigkeiten zwischen den Objekten in Schichten zu unterteilen [Gam15, S. 241].

- **direkter Zugriff wird nicht verbaut**

Das Fassadenmuster bietet einfache Schnittstellen zum Zugriff auf Subsysteme, es lässt aber offen, ob direkte Zugriffe auf die Klassen eines 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 eines Subsystems. Solange der Direktzugriff erlaubt ist, kann man im betreffenden Subsystem nicht die Vorteile des Information Hiding nutzen.

5.4.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **Zwischenschicht kostet Performance**

Man führt einen zusätzlichen Methodenaufruf ein und schafft dadurch zusätzlichen Ballast.

- **Abhängigkeit der Fassade von den gekapselten Komponenten**

Wenn sich die Schnittstellen der gekapselten Komponenten ändern, muss die Fassade eventuell ebenfalls geändert werden.

5.4.5 Einsatzgebiete

Das Fassadenmuster ist generell bei Subsystemen anwendbar, wenn eine Schnittstelle zum Zugriff auf das Subsystem frei entworfen werden kann. Das Muster kann eingesetzt werden, um die Komplexität eines Subsystems zu verbergen. Eine Fassade kann aber auch mehrere Subsysteme kapseln.

Hängen die Komponenten eines Subsystems voneinander ab, kann man deren Abhängigkeiten vereinfachen, indem man auch sie über die Fassade kommunizieren lässt [Gam15, S. 239].

Bei einem Schichtenmodell (siehe Architekturmuster **Layers**) kann eine neue Schicht mit Hilfe des Musters Fassade eingeführt werden.

Wenn ein System in Microservices [Lew14] zerlegt wird, wird häufig eine Fassade – in diesem Zusammenhang API-Gateway genannt – zur Verfügung gestellt, damit Clients die einzelnen Microservices und deren Lokalisierung nicht kennen müssen.

Außerdem kann das Muster Fassade zur Kapselung eines Legacy-Systems verwendet werden. Die Fassade dient dann oft zur **objektorientierten Einhüllung von Legacy-Systemen**⁵⁵. 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.

5.4.6 Ähnliche Entwurfsmuster

Sowohl das Muster **Fassade** als auch das Muster **Adapter** kapseln vorhandene Programme. Fassade und Adapter sind Wrapper. Im Gegensatz zur Fassadenschnittstelle, die von den Objekten eines Subsystems definiert wird, wird die Adapterschnittstelle vom Client vorgegeben. Hinter der Fassade verbergen sich die Objekte von Subsystemklassen, hinter dem Adapter verbirgt sich nur eine einzige Klasse. Für eine Fassadenschnittstelle gibt es keine Vorgaben von außerhalb des Subsystems. Die Schnittstelle einer Fassade kann vom Subsystem frei festgelegt werden. Der Zugriff auf die Objekte eines Subsystems bei gekoppelten Folgen von Aufrufen wird 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 gekoppelten Sequenzen von Objektaufrufen zu vereinfachen. Alle miteinander kommunizierenden Objekte kennen ihren Vermittler und der Vermittler kennt sie. 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 nur 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 können muss, sondern auch einen Direktzugriff auf die Objekte der konkreten Produktklassen benötigt, um die von der Fabrik erzeugten Produkte nutzen zu können.

5.5 Das Strukturmuster Kompositum

5.5.1 Name/Alternative Namen

Kompositum (engl. composite), Kompositionsmuster.

⁵⁵ Ein Legacy-System ist in der Informatik ein historisch gewachsenes System, oft in einer veralteten Technologie. Es stellt deshalb sozusagen oftmals eine "Altlast" dar.

5.5.2 Problem

Das Muster Kompositum soll es erlauben, dass Clienten bei der Verarbeitung von Objekten in einer **Baumstruktur** einfache und aus einfachen Objekten zusammengesetzte Objekte einheitlich behandeln können.



Ein Client soll also die Unterschiede zwischen primitiven Objekten, den **Blättern**, und zusammengesetzten Knoten, den **Komposita**, ignorieren können.

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 Objekte in einer Baumstruktur werden auch als **Knoten** bezeichnet.

Eine für das Kompositum-Muster typische Teil-Ganze-Hierarchie – sowohl aus primitiven Objekten, als auch aus Kompositionen von primitiven Objekten – in einer baumartigen Struktur könnte wie folgt aussehen:

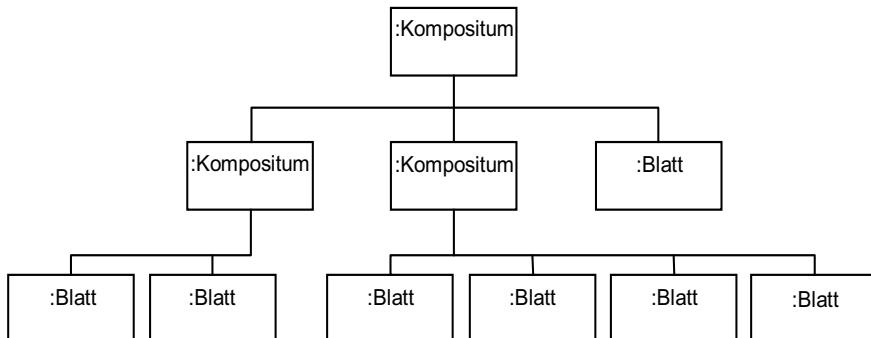


Bild 5-20 Beispielhafte Struktur einer Teil-Ganze-Hierarchie

Wie in Bild 5-20 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.

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 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.

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 zwischen Blättern und Komposita erforderlich werden. Ein Client soll nur die abstrakte Schnittstelle eines `Knotens` 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.



5.5.3 Lösung

Das Kompositum-Muster ist ein **objektbasiertes Strukturmuster**⁵⁶. Durch die rekursive Komposition des Kompositum-Musters müssen Clienten keine Unterscheidung zwischen primitiven Elementen und aus primitiven Elementen zusammengesetzten Komposita in einer Baumstruktur treffen.

Dadurch wird der Aufwand im Client für die Verwaltung der resultierenden Baumstruktur verringert und es ist leicht, geschachtelte Objektstrukturen (Bäume) zu bilden.

Die Unterscheidung, ob es sich bei einer Klasse der Baumstruktur um eine Blatt-Klasse oder um 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 nur beim Blatt ausgeführt, da das Blatt keine Kinder hat.



5.5.3.1 Klassendiagramm

Das Klassendiagramm in Bild 5-21 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:

⁵⁶ Kompositum ist zugleich auch der Name für die Klasse, deren Objekte als `Knoten` in einem Baum auftreten können und deren Objekte selber auch weitere `Knoten` aggregieren können, um so die Baumstruktur aufzubauen.

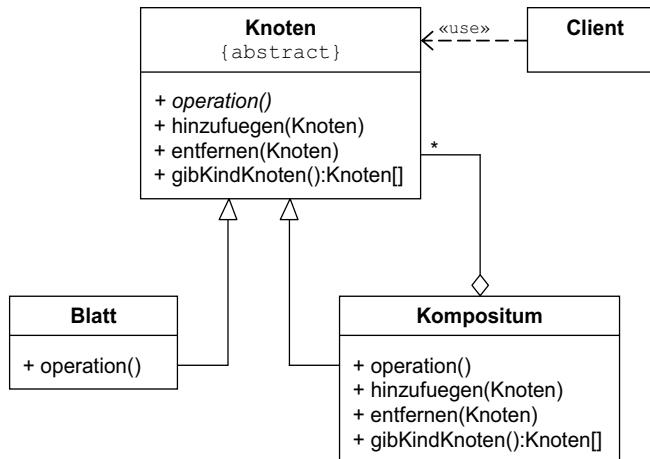


Bild 5-21 Klassendiagramm Kompositum

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 auf Kindknoten bezogene Operation wie z. B. `gibKindKnoten()` aufrufen. Er soll ja nicht wissen, um was es sich bei einem Element handelt. Deshalb müssen die Kindknoten bezogene Operationen `hinzufuegen()`, `entfernen()` und `gibKindKnoten()` in der Wurzel der Klassenhierarchie bei der Komponente definiert werden. Es ist sinnvoll, in der Wurzelklasse `Knoten` ein Default-Verhalten für diese Kindoperationen zu implementieren, welches für `Blatt`-Klassen geeignet ist und das von einer Kompositum-Klasse überschrieben wird.

Eine gängige Lösung ist, dass die **abstrakte Klasse** `Knoten` die auf **Kindknoten bezogene Operationen** so implementiert, dass die Methode `gibKindKnoten()` eine leere Liste zurückgibt und die Methoden `hinzufügen()` und `entfernen()` eine Exception werfen, wenn sie aufgerufen werden.



Die `Blatt`-Klasse überschreibt dieses Verhalten nicht. Die `Kompositum`-Klasse hingegen überschreibt die auf Kindknoten bezogenen Operationen in sinnvoller Weise und wirft dabei keine Exception. Dies bedeutet, dass die Nachbedingung einer auf Kindknoten bezogenen Operation in der `Kompositum`-Klasse verschärft wird, was den Vertrag der überschriebenen Methode der Basisklasse nicht bricht.

Die Klassen `Kompositum` und `Blatt` sind in Bild 5-20 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 Bild 5-25 zu sehen.

5.5.3.2 Teilnehmer

Das Muster Kompositum hat die folgenden Teilnehmer:

- **Knoten**

Die abstrakte Klasse `Knoten` legt die Schnittstelle und ein Default-Verhalten für die Kind-Operationen der abgeleiteten Klassen `Kompositum` und `Blatt` fest.

- **Blatt**

Die Klasse `Blatt` repräsentiert ein Abschlusselement in der Baumstruktur. Ein Blatt besitzt keine Kindobjekte und kann selbst immer nur Kind-Knoten sein. Ein Blatt definiert das Verhalten der primitiven Objekte.

- **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 Default-Verhalten, das in der Klasse `Knoten` implementiert ist.

- **Client**

Ein Client bearbeitet die Objekte der Teil-Ganzes-Hierarchie über die Schnittstelle der Klasse `Knoten`.

5.5.3.3 Dynamisches Verhalten

Ist ein Blatt der Empfänger, so wird der Auftrag direkt bearbeitet. Ist ein Kompositum der Empfänger, bearbeitet es selbst den Auftrag und gibt ihn an seine Kinder weiter. Eventuell werden vor, nach oder vor und nach der Weiterleitung zusätzliche Operationen eingeführt.

Das dynamische Verhalten des Kompositum-Musters wird an der Beispiel-Baumstruktur in folgendem Bild verdeutlicht. In diesem Beispiel fügt der Client zuerst 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 dann folgendermaßen aus:

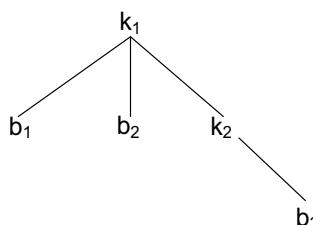


Bild 5-22 Baumstruktur des Beispiels

Nach dem Aufbau der Baumstruktur soll die Methode `operation()` des Objekts `k1` von einem Client aufgerufen werden. Der gesamte Ablauf dieses Aufrufs ist im folgenden Sequenzdiagramm dargestellt:

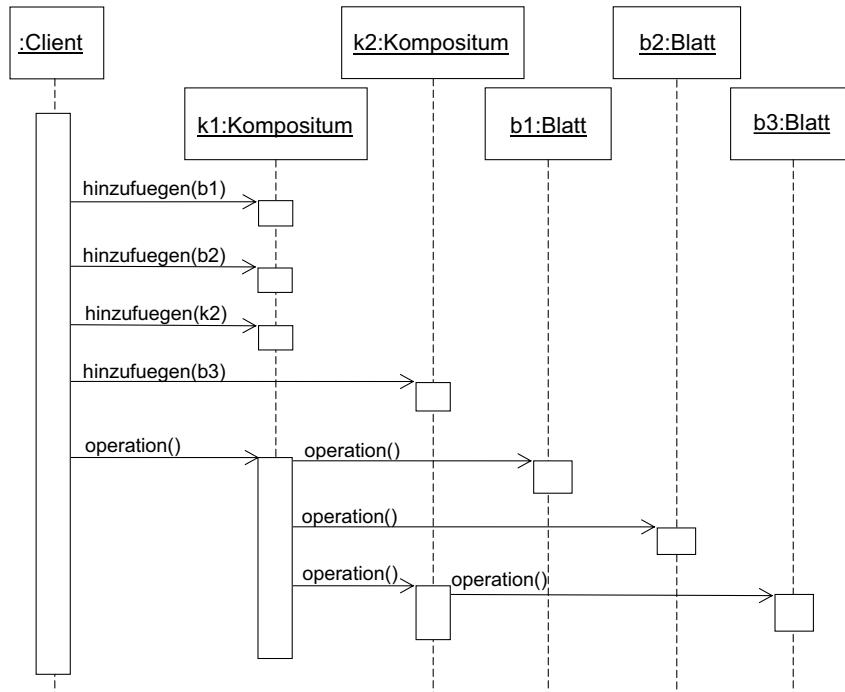


Bild 5-23 Sequenzdiagramm Kompositum-Muster

Wie in Bild 5-23 zu sehen ist, wird beim Aufruf der Methode `operation()` des Objekts `k1` die Anfrage rekursiv an alle untergeordneten Blätter und Kompositum-Instanzen weitergeleitet.

5.5.3.4 Programmbeispiel

Die Klasse `Knoten` definiert eine abstrakte Basisklasse, von welcher die Klassen `Kompositum` und `Blatt` abgeleitet werden. Die Klasse `Knoten` definiert die abstrakte Methode `operation()`. Der Name dieser Methode wurde in Analogie zu den bisherigen Ausführungen zum Muster Kompositum gewählt. Im konkreten Programmbeispiel soll die Methode `operation()` die in einem Knoten gespeicherte Information ausgeben. Damit in der Ausgabe die Baumstruktur der Kompositum-Struktur ersichtlich wird, enthält die Klasse `Knoten` eine Hilfsvariable und eine Hilfsmethode, um die jeweilige Einrückungsebene zu berechnen. Es folgt der Quellcode der Klasse `Knoten`:

```

// Datei: Knoten.java

import java.util.*;

public abstract class Knoten {

```

```
private final String name;

public Knoten(String name) {
    this.name = name;
}

public abstract void operation();

public List<Knoten> getKinder() {
    return Collections.emptyList();
}

public void hinzufuegen(Knoten kind) {
    throw new UnsupportedOperationException();
}

public void entfernen(Knoten kind) {
    throw new UnsupportedOperationException();
}

public String getName() {
    return name;
}

// Hilfsvariable für die Ausgabe.
protected static int ebene;

// Hilfsmethode für die Ausgabe.
protected static String getIndentation() {
    char[] indent = new char[ebene * 2];
    Arrays.fill(indent, ' ');
    return new String(indent);
}
```

Die Klasse Kompositum ist von der Klasse Knoten abgeleitet. Sie überschreibt die kindbezogenen Methoden und implementiert die Operation operation().

Hier die Klasse Kompositum:

```
// Datei: Kompositum.java

import java.util.*;

public class Kompositum extends Knoten {

    private final List<Knoten> kinder;

    public Kompositum(String name) {
        super(name);
        kinder = new ArrayList<>();
    }
```

```

@Override
public void operation() {
    // Erhöhen der Ausgabe-Ebene.
    ++ebene;
    System.out.printf("%s+ %s%n", getIndentation(), getName());
    // Aufrufen der Operation für alle Kindknoten.
    getKinder().forEach(Knoten::operation);
    // Zurücksetzen der Ausgabe-Ebene.
    --ebene;
}

@Override
public List<Knoten> getKinder() {
    // Kopie erstellen, um die Kontrolle über die Elemente
    // zu behalten.
    return new ArrayList<>(kinder);
}

@Override
public void hinzufuegen(Knoten kind) {
    kinder.add(Objects.requireNonNull(kind));
}

@Override
public void entfernen(Knoten kind) {
    for (Knoten knoten : kinder) {
        if (knoten instanceof Kompositum) {
            knoten.entfernen(kind);
        }
    }
    kinder.remove(kind);
}
}

```

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);
    }

    @Override
    public void operation() {
        System.out.printf("%s - %s%n", getIndentation(), getName());
    }
}

```

Die Klasse `TestKompositum` legt insgesamt vier Kompositum-Objekte an. Um die Möglichkeit der Rekursion aufzuzeigen, wird dabei die Referenz `komp121` dem Kompositum `komp12` hinzugefügt. Anschließend werden sechs Instanzen der Klasse `Blatt` erstellt. Diese werden dann den Kompositum-Objekten hinzugefügt, wie das Programm zeigt. Im nächsten Schritt wird der Inhalt der Baumstruktur ausgegeben. Dazu genügt es, dass die Methode `operation()` bei dem Kompositum-Objekt der obersten Ebene, in diesem Falle das Objekt `komp1`, aufgerufen wird. Danach werden ein Blatt sowie ein Kompositum-Objekt entfernt und somit die Baumstruktur geändert. Zum Schluss wird die Methode `operation()` erneut bei dem Objekt `komp1` aufgerufen, um die Änderungen sichtbar zu machen.

Hier nun der Quellcode der Klasse `TestKompositum`:

```
// Datei: TestKompositum.java

public class TestKompositum {

    public static void main(String[] args) {
        // Aufbau der Objektstruktur
        Kompositum komp1 = 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");
        komp1.hinzufuegen(komp11);
        komp1.hinzufuegen(komp12);
        komp12.hinzufuegen(komp121);

        // Erstellen von 6 Objekten der Klasse Blatt
        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");

        // Hinzufuegen der Blätter an Kompositum-Objekte
        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\n");
        komp1.operation();

        // Modifikation der Objektstruktur
        komp12.entfernen(blatt122);
        komp12.entfernen(komp121);
```

```
// erneuter Aufruf der Operation fuer das Kompositum
System.out.println("\nZweite Ausgabe der Kompositum-Operation\n");
kompl.operation();
}
```



Hier das Protokoll des Programmlaufs, in dem durch die Einrückungen die Baumstruktur angedeutet wird:

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
```

5.5.4 Bewertung

5.5.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **einheitliche Behandlung von Kompositum- und Blatt-Objekten**

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.

- **einfache Verschachtelung**

Das Strukturmuster Kompositum erlaubt es, verschachtelte Strukturen auf einfache Weise zu erzeugen bzw. um neue Blatt- oder Kompositum-Objekte zu erweitern. Blatt- und Kompositum-Klassen verwenden dieselbe Schnittstelle. Hierbei kann ein zusammengesetztes Objekt mehrfach verschachtelt sein.

5.5.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **Schwierigkeiten bei Einschränkungen**

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.

- **Änderungen der Basisschnittstelle bedeuten Änderungen bei abgeleiteten Klassen**

Das Entwurfsmuster Kompositum ist sehr mächtig. Sobald jedoch Änderungen an der Basisschnittstelle der Klasse `Knoten` durchgeführt werden wie z. B. das Hinzufügen einer neuen Methode, bedeutet dies, dass alle von der Klasse `Knoten` abgeleiteten Klassen ebenfalls geändert werden müssen⁵⁷.

5.5.5 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 9.1) 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. Die Darstellung einer View wird dabei in einem GUI-Framework aus Komponenten zusammengesetzt, welche verschachtelte Komponenten enthalten. Durch die Verschachtelung entsteht eine

⁵⁷ Dies gilt nicht für abgeleitete Klassen, die eine von der Klasse `Knoten` zur Verfügung gestellte Default-Implementierung unverändert übernehmen und nicht überschreiben wollen.

Baumstruktur aus Komponenten, die aus weiteren Komponenten zusammengesetzt sind (in Swing: Panels⁵⁸), und aus Blättern, die keine anderen Komponenten in sich tragen (z. B. Schaltflächen). Die Anwendung des Kompositum-Musters bei grafischen Oberflächen wird im folgenden Beispiel im Detail ausgeführt.

Anwendungsbeispiel: Grafische Oberflächen mit Swing

Es soll die in folgendem Bild dargestellte Oberfläche erzeugt werden:

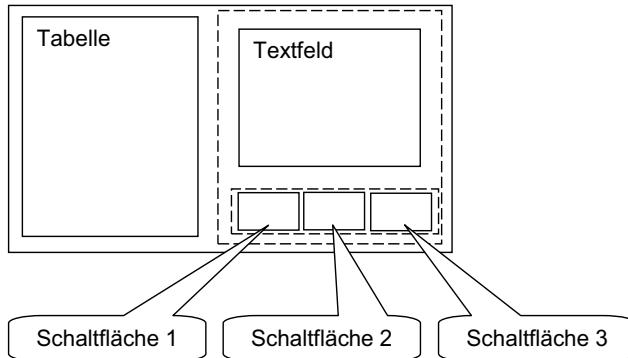


Bild 5-24 Zu realisierende grafische Oberfläche

Dabei repräsentieren die gestrichelten Linien unsichtbare Behälterobjekte (Panels), die andere grafische Komponenten gruppieren können. Der äußere Rahmen – auch "Frame" genannt – stellt quasi das Wurzelement der verschachtelten Struktur dar.

In Grafikpaket Swing von Java gibt es zur Realisierung dieser Beispielloberfläche die folgenden Klassen: `JTextField` für Textfelder, `JButton` für Schaltflächen, und `JTable` für Tabellen. Diese und andere grafischen Komponenten können in einem Objekt der Klasse `JPanel` gruppiert werden, was es erlaubt, die in einem Panel-Objekt enthaltenen Objekte beispielsweise gemeinsam zu vergrößern, zu verkleinern oder zu verschieben. 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 5-25 nicht gezeigt ist, damit die Analogie zum Klassendiagramm des Kompositum-Musters in Bild 5-21 deutlicher wird:

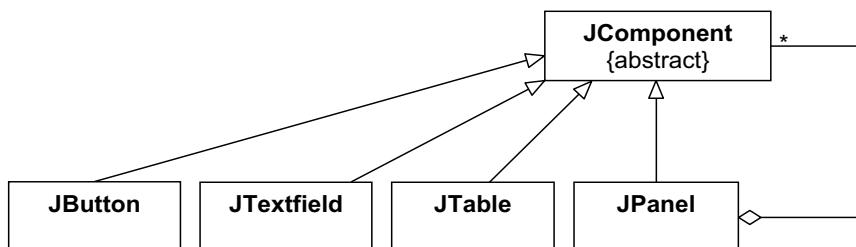


Bild 5-25 Klassendiagramm für eine Anwendung des Kompositum-Musters

⁵⁸ Panels stellen unsichtbare Objektbehälter dar, mit deren Hilfe es möglich ist, grafische Komponenten zu Gruppen zusammenzufassen.

Die Klasse `JComponent` selbst ist abgeleitet von der Klasse `Container` aus dem Paket der grafischen Benutzerschnittstelle AWT (**Abstract Window Toolkit**). Damit haben alle in Bild 5-25 gezeigten Klassen die Eigenschaften der Klasse `Container` und können beliebig viele grafische Elemente aufnehmen. Diese Fähigkeiten werden aber im Bild 5-25 nur von der Klasse `JPanel` genutzt, was durch die Aggregationsbeziehung 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.

Damit Swing-Komponenten auf dem Bildschirm erscheinen, müssen sie in einen top-level Container eingebettet werden. Dazu wird im folgenden Programmbeispiel die Klasse `JFrame` benutzt, die ebenfalls von der Klasse `Container` abgeleitet ist. Die Klasse `JFrame` kann aber nur als Wurzelement für den äußeren Rahmen einer grafischen Oberfläche genutzt werden und kann daher nicht als Kompositum-Klasse angesehen werden.

Das folgende Codefragment zeigt, wie man in Swing die in Bild 5-24 skizzierte Oberfläche mit Objekten der genannten Klassen zusammensetzen kann:

```
// 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 durch das obige Codefragment erzeugte Klassenhierarchie zugrunde legt und den äußeren Rahmen (`frame`) vergrößert? Das "Frame"-Objekt vergrößert sich selbst und leitet den Aufruf an alle Objekte, die in ihm enthalten sind, weiter (Delegationsprinzip). Durch die Weiterleitung geht der Aufruf an die Objekte `table` und `panelRechts`. Das Objekt `panelRechts` ist selber ein Kompositum-Objekt und leitet wiederum den Aufruf an alle enthaltenen Objekte (`textfield`, `panelUnten`) weiter. Dies geht rekursiv weiter, da beispielsweise `panelUnten` ein Kompositum-Objekt ist, bis alle Elemente vergrößert sind.

5.5.6 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 statt vielen Objekten aggregieren kann. Es zeigen sich jedoch deutliche 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 dass 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 im Falle des Kompositum-Musters.

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 5-21) vorgegeben ist.

5.6 Das Strukturmuster Proxy

5.6.1 Name/Alternative Namen

Proxy, Stellvertreter (engl. proxy), Surrogat⁵⁹ (engl. surrogate).

5.6.2 Problem

Der Zugriff auf ein Objekt soll mit Hilfe eines vorgelagerten Stellvertreter-Objekts kontrolliert werden.

Das Proxy-Muster soll eine Ebene der Indirektion beim Zugriff auf ein Objekt einführen. Das **Proxy-Muster** soll die Existenz eines echten Objekts hinter einem Stellvertreter (Proxy) mit derselben Schnittstelle verbergen.



⁵⁹ Surrogat bedeutet Ersatz.

Auf die Anfrage des Clients soll der Proxy selbst antworten können, wenn er die Anfrage beantworten kann. Er soll die Anfrage des Clients aber auch an das echte Objekt weiterleiten können, wenn dies erforderlich ist. Wer tatsächlich antwortet, soll für den Client verborgen sein.

5.6.3 Lösung

Das **Proxy-Muster** ist ein objektbasiertes Entwurfsmuster, bei dem das Proxy-Objekt den Zugriff auf das echte Objekt kontrolliert.



Ein Stellvertreter-Objekt erhält dieselbe Schnittstelle wie ein echtes Objekt. Der Zugang eines Client-Programms zum echten Objekt erfolgt über den Proxy. Die gesamte Kommunikation des Proxys mit dem echten Objekt wird im Stellvertreter-Objekt gekapselt. Das echte Objekt ist dem Client verborgen.



Das Client-Programm greift direkt auf den entsprechenden Proxy als **Stellvertreter des echten Objekts** zu.

Der Proxy kann als zusätzliche Funktionen (siehe Kapitel 5.6.5) 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 oder, wenn das echte Objekt nur über das Netzwerk erreichbar ist, muss sich der Proxy erst mit dem echten Objekt verbinden.

5.6.3.1 Klassendiagramm

In Bild 5-26 wird das Proxy-Muster dargestellt:

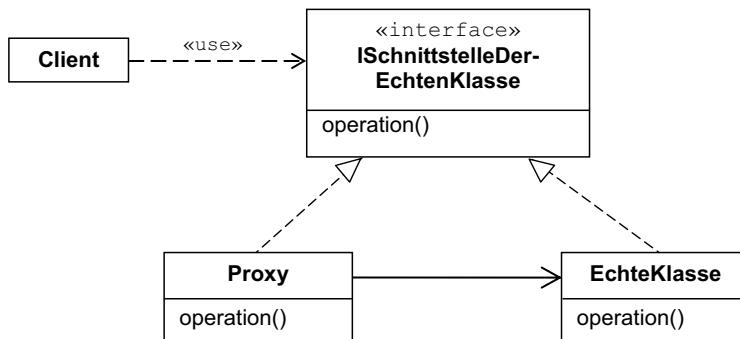


Bild 5-26 Klassendiagramm Proxy-Muster

Die Klasse `EchteKlasse` und die Klasse `Proxy` implementieren beide dieselbe Schnittstelle `ISchnittstelleDerEchtenKlasse`. Dadurch kann ein Proxy-Objekt als Stellvertreter eines echten Objekts fungieren. Über die Schnittstelle 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 der Schnittstelle bei einem Objekt aufruft, so bedeutet das, dass der Methodenaufruf bei einem Objekt einer Klasse stattfindet, welche diese Schnittstelle implementiert. Das kann ein echtes Objekt sein, aber auch ein Stellvertreter-Objekt, das den Methodenaufruf des Clients entgegennimmt und ihn ggf. 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 Methodenaufruf 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 kennt. Beispiele für mögliche Varianten des Proxy-Musters sind in Kapitel 5.6.5 zu finden.

Ein Client hat eine Referenz vom Typ der Schnittstelle `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 Methodenaufrufe, die er an das referenzierte Objekt stellt, in der Folge weiterverarbeitet bzw. weiterdelegiert werden.

5.6.3.2 Teilnehmer

Im Folgenden werden die Teilnehmer des Muster Proxy beschrieben:

- **ISchnittstelleDerEchtenKlasse**

Die Schnittstelle `ISchnittstelleDerEchtenKlasse` definiert die gemeinsame Schnittstelle von Stellvertreter und echter Klasse. Sie stellt die Methodenköpfe einer echten Klasse und des zugehörigen Proxys zur Verfügung, die vom Client verwendet werden.

- **Client**

Der Client ruft die Methoden eines Objekts über eine Referenz auf die Schnittstelle `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 eine Schnittstelle als Typ für die Referenz nutzt, ist er nicht abhängig von der Ausprägung der Klassen, die diese Schnittstelle implementieren.

Zum Setzen der Referenz können beispielsweise Techniken der Dependency Injection⁶⁰ eingesetzt werden.

- **EchteKlasse**

Die Klasse `EchteKlasse` definiert die Klasse des echten Objekts, welches durch ein Stellvertreter-Objekt der Klasse `Proxy` repräsentiert werden soll. Sie implementiert die Methoden der Schnittstelle `ISchnittstelleDerEchtenKlasse`.

- **Proxy**

Die Klasse `Proxy` verwaltet die Referenz auf ein echtes Objekt. Sie implementiert die Methoden der Schnittstelle `ISchnittstelleDerEchtenKlasse`. Das Objekt der Klasse `Proxy` delegiert bei Bedarf 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 5.6.5).

5.6.3.3 Dynamisches Verhalten

Der Client ruft in diesem Beispiel eine Methode des Proxy-Objekts auf. Der Aufruf wird an ein bereits vorhandenes Objekt der Klasse `EchteKlasse` weitergeleitet, welches die Methode abarbeitet und das Ergebnis zurückliefert an das Proxy-Objekt. Das Ergebnis wird danach vom Proxy-Objekt an den Client zurückgegeben. Das folgende Bild zeigt das Sequenzdiagramm dieses Beispieles:

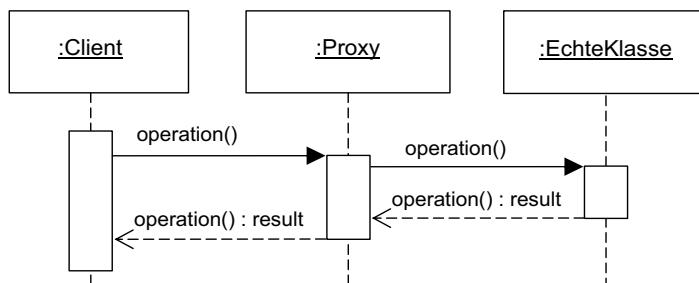


Bild 5-27 Sequenzdiagramm des Proxy-Musters

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 wird im folgenden Programmbeispiel so ausgeführt.

5.6.3.4 Programmbeispiel

Das Proxy-Muster soll an einem Beispiel zum Zugriff auf eine Datei vorgestellt werden. Die Schnittstelle `IDateiZugriff` definiert zwei Methoden `getName()` und `getInhalt()` zur Ausgabe des Dateinamens und des Inhalts der Datei. Um das Programm

⁶⁰ Siehe Kapitel 3.1.3.

möglichst einfach zu halten, beschränkt sich der Dateizugriff auf eine simulierte Leseoperation. Hier die Schnittstelle `IDateiZugriff`:

```
// Datei: IDateiZugriff.java

public interface IDateiZugriff {

    String getName();

    String getInhalt();

}
```

Die echte Klasse `DateiZugriff` implementiert diese Schnittstelle, 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 {

    private final String name;

    public DateiZugriff(String name) {
        this.name = name;
        System.out.println("Echtes Objekt instanziert.");
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getInhalt() {
        // Simulation einer Datei-Leseoperation
        return String.format("Daten von %s", name);
    }
}
```

Ein Proxy speichert den Namen seiner zugehörigen Datei. Solange ein Client nur die Methode `getName()` aufruft, ist es für den Proxy 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 eines Zugriffs 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

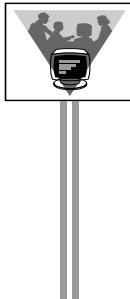
public class ProxyDateiZugriff implements IDateiZugriff {

    private final String name;
    private IDateiZugriff realeDatei;
```

```
public ProxyDateizugriff(String name) {  
    this.name = name;  
    System.out.println("Stellvertretendes Objekt instanziert.");  
}  
  
@Override  
public String getName() {  
    return name;  
}  
  
@Override  
public String getInhalt() {  
    return getRealeDatei().getInhalt();  
}  
  
private IDateizugriff getRealeDatei() {  
    if (realeDatei == null) {  
        System.out.println("Reale Datei liegt lokal noch nicht vor.");  
        // Das echte Datei-Objekt wird erzeugt.  
        realeDatei = new Dateizugriff(name);  
    }  
    return realeDatei;  
}  
}
```

Der TestProxy erstellt ein Objekt der Klasse ProxyDateizugriff, welche die Schnittstelle IDateizugriff implementiert und ruft dann die Methoden getName() und getInhalt() der Schnittstelle 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.printf("Name: %s%n", pDatei.getName());  
        System.out.printf("Inhalt: %s%n", pDatei.getInhalt());  
        System.out.println();  
        System.out.printf("Inhalt: %s%n", pDatei.getInhalt());  
    }  
}
```



Die Ausgabe des Programms ist:

Stellvertretendes Objekt instanziert.

Name: TestDatei.dat

Reale Datei liegt lokal noch nicht vor.

Echtes Objekt instanziert.

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 5.6.5). Solange ein Client nur den Namen wissen will, werden somit "teure" Dateioperationen vermieden. Erst wenn ein Client auf den Inhalt der Datei zugreifen will, wird das eigentliche Dateiobjekt erzeugt.

5.6.4 Bewertung

5.6.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Erweiterung bestehender Anwendungen**

Bereits bestehende Anwendungen können mit dem Proxy-Muster erweitert werden (siehe Kapitel 5.6.5).

- **individuelle Gestaltung eines Proxys**

Der Funktionsumfang eines Proxys kann individuell gestaltet werden.

- **Transparenz der Implementierung**

Der Proxy verdeckt das echte Objekt. Ein Client muss nur die gemeinsame Schnittstelle des echten Objekts und des Proxys kennen. Er weiß aber beim Zugriff nicht, welches der beiden Objekte er benutzt.

- **Proxy kann echtes Objekt oft vertreten**

Es wird ein Stellvertreter (Proxy) des echten Objektes erzeugt. Erst wenn bestimmte Operationen durchgeführt werden sollen, für welche der Zugriff auf den Proxy nicht mehr ausreicht, wird auf das echte Objekt zugegriffen.

- **Performance-Gewinn durch Proxy**

Durch die zusätzliche Zwischenschicht eines Proxys kann bei Operationen, die kein Original benötigen – z. B. beim Caching der Anfragen – ein Performance-Gewinn entstehen.

5.6.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **Fehlersuche erschwert**

Die Fehlersuche wird beim Zwischenschalten eines Proxys erschwert.

- **Performance-Verlust**

Durch die zusätzliche Zwischenschicht eines Proxys entsteht beim Zugriff auf das Originalobjekt ein Performance-Verlust, da eine Weiterleitung (Delegation) erfolgt und dies mit Aufwand verbunden ist.

5.6.5 Einsatzgebiete

In folgenden **Situationen** kann das Proxy-Muster eingesetzt werden:

- **Der Ort des echten Objekts soll verborgen werden**

Der entsprechende Proxy wird als **Remote-Proxy** bezeichnet.

Ein Remote-Proxy befindet sich lokal auf dem gleichen Rechner wie das Client-Programm und steht diesem als Stellvertreter der eigentlichen Komponente zur Verfügung, welche auf einem entfernten (remote) Rechner läuft. Das Client-Programm kennt nur das Stellvertreter-Objekt und nicht die eigentliche Komponente (Transparenz des echten Objekts). Das Client-Programm weiß nicht, wie Methodenaufrufe an das Stellvertreter-Objekt in der Folge weiterverarbeitet werden. Mit dem Remote-Proxy können Anwendungen gekapselt, ausgetauscht und auf weitere Rechner ausgelagert werden, ohne dass das Client-Programm verändert werden muss.

Ein Remote-Proxy ist beispielsweise ein zentraler Bestandteil des **Broker-Musters** (siehe Kapitel 12.1).

- **Methode soll durch Proxy statisch erweitert werden**

Dies ist der Fall, wenn das echte Objekt nicht verändert werden kann bzw. soll, aber eine Methode eines echten Objekts 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, Synchronisation und Einführung einer Rechteverwaltung beziehen.

Aus den angeführten Gründen 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**

Ein virtueller Stellvertreter verzögert den "teuren" Zugriff auf ein Objekt einer echten Klasse, solange der Proxy selbst ausreichend ist.

Der sogenannte 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 instanziert und initialisiert werden. Beim Virtuellen Proxy werden an Stelle der rechenintensiven echten Objekte Stellvertreter-Objekte instanziert, die stellvertretend für die eigentlichen Objekte verwendet werden. Der Virtuelle Proxy schiebt meist die Erzeugung des echten Objekts so lange hinaus, bis dieses tatsächlich gebraucht wird. Beim Aufruf einer Methode, für die das echte Objekt benötigt wird, wird das eigentliche Objekt zur Laufzeit instanziert. 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.

- **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 Browern eingesetzt, um Webseiten zwischenzuspeichern. 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 Anforde rungsrate der zu cachenden Daten hat.

Das Proxy-Muster kann auch beim Testen⁶¹ von Software eingesetzt werden. Die Stell vertreter-Objekte werden dann als Mock⁶²-Objekte bezeichnet. Sie dienen dazu, eine noch nicht oder nicht vollständig vorhandene Umgebung für eine zu testende Kom ponente 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.

5.6.6 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 ein⁶³, 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.

5.7 Zusammenfassung

Strukturmuster befassen sich mit dem Aufbau von Strukturen aus Klassen und Ob jekten.

⁶¹ Für Muster, die in der Testphase angewendet werden können, hat sich der Begriff Test Pattern etab liert, zu denen beispielsweise das Mock Object Pattern [Mes07] gezählt wird.

⁶² To mock (engl.) bedeutet nachahmen.

⁶³ Beim Adapter-Muster wird hier die objektbasierte Variante betrachtet.

Das Adapter-Muster in Kapitel **5.1** passt eine vorhandene "falsche" Schnittstelle an die gewünschte Form an.

Das Muster Brücke in Kapitel **5.2** trennt die Klassenhierarchie einer Abstraktion und ihrer Implementierung. Dadurch wird erreicht, dass die Abstraktion und die Implementierung unabhängig voneinander verändert und erweitert werden können. Beide Teile werden durch eine "Brücke" verbunden. Die Implementierung ist vor dem Client verborgen.

Das Dekorierer-Muster in Kapitel **5.3** erlaubt es, dass zur Laufzeit eine zusätzliche Funktionalität zu einem vorhandenen Objekt einer Klasse oder Subklasse in dynamischer Weise hinzuzufügt wird.

Das Muster Fassade in Kapitel **5.4** stellt ein ganzes Subsystem nach außen dar, wie wenn dieses Subsystem ein einziges Objekt wäre. Dabei wird für dieses Subsystem eine neue Schnittstelle, die Fassade, konstruiert. Die Methoden der Fassade enthalten meist Folgen von Methodenaufrufen auf den Objekten des Subsystems.

Mit dem Kompositum-Muster in Kapitel **5.5** können bei der Verarbeitung von Elementen einer Baumstruktur einfache und zusammengesetzte Objekte gleich behandelt werden. Ein Programmierer braucht sich also um die Unterscheidung einfacher und zusammengesetzter Objekte nicht zu kümmern.

Das Proxy-Muster in Kapitel **5.6** 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.

5.8 Kontrollfragen

Aufgaben 5.8.1: Adapter

- 5.8.1.1 Welches Ziel hat das Adaptermuster?
- 5.8.1.2 Was ist der Unterschied zwischen dem Proxy- und dem Adaptermuster?

Aufgaben 5.8.2: Brücke

- 5.8.2.1 Wie funktioniert das Muster Brücke?
- 5.8.2.2 Wo ist die Brücke zu sehen?

Aufgaben 5.8.3: Dekorierer

- 5.8.3.1 Welches Entwurfsmuster hat bis auf eine andere Multiplizität das gleiche Klassendiagramm wie das Dekorierer-Muster?
- 5.8.3.2 Wozu dient das Dekorierer-Muster?

Aufgaben 5.8.4: Fassade

- 5.8.4.1 Wozu dient das Fassadenmuster?
- 5.8.4.2 Kann das Fassadenmuster den Zugriff auf die Subsystemklassen verhindern? Begründen Sie Ihre Antwort.

Aufgaben 5.8.5: Kompositum

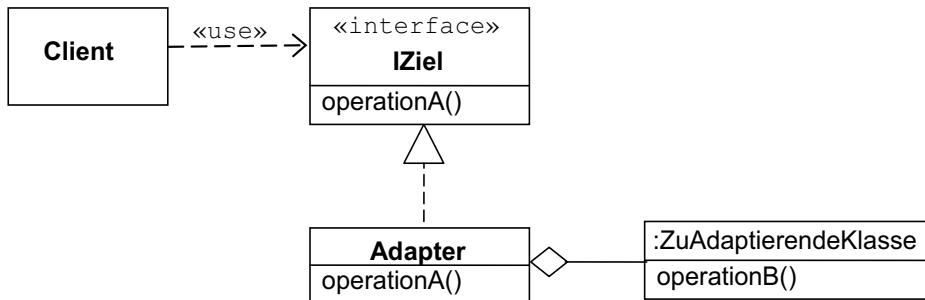
- 5.8.5.1 Welche Rollen spielen die Klassen Knoten, Kompositum und Blatt im Rahmen des Kompositum-Patterns.
- 5.8.5.2 Eine Blatt-Klasse erbt von der Basisklasse Knoten die Kindfunktionen, hat aber keine Kinder. Wie geht man üblicherweise vor?

Aufgaben 5.8.6: Proxy

- 5.8.6.1 Was ist das Ziel des Proxy-Musters?
- 5.8.6.2 Was ist ein Schutz-Proxy?

Kapitel 6

Verhaltensmuster



- 6.1 Das Verhaltensmuster Befehl
- 6.2 Das Verhaltensmuster Beobachter
- 6.3 Das Verhaltensmuster Besucher
- 6.4 Das Verhaltensmuster Iterator
- 6.5 Das Verhaltensmuster Memento
- 6.6 Das Verhaltensmuster Rolle
- 6.7 Das Verhaltensmuster Schablonenmethode
- 6.8 Das Verhaltensmuster Strategie
- 6.9 Das Verhaltensmuster Vermittler
- 6.10 Das Verhaltensmuster Zustand
- 6.11 Zusammenfassung
- 6.12 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_6.

6 Verhaltensmuster

Der Begriff "Verhaltensmuster" wurde von Gamma et al. [Gam94] geprägt. Verhaltensmuster sind Entwurfsmuster und gehören damit zum Lösungsbereich.

Verhaltensmuster beschreiben, wie Objekte durch ihr Zusammenwirken ein bestimmtes Verhalten erzeugen.



Wie in Kapitel 2.2.2 erwähnt betreffen Entwurfsmuster Verarbeitungsfunktionen. Diese existieren bereits im Problembereich, werden im Rahmen von Entwurfsmustern jedoch nur im Lösungsbereich betrachtet. Dies gilt auch für die im Folgenden vorgestellten Verhaltensmuster.

Kapitel 6.1 betrachtet das Verhaltensmuster Befehl, Kapitel 6.2 das Beobachtermuster und Kapitel 6.3 das Besuchermuster. Es folgen die Verhaltensmuster Iterator (siehe Kapitel 6.4), Memento (siehe Kapitel 6.5), Rolle (siehe Kapitel 6.6), Schablonenmethode (siehe Kapitel 6.7), Strategie (siehe Kapitel 6.8), Vermittler (siehe Kapitel 6.9) und Zustand (siehe Kapitel 6.10).

6.1 Das Verhaltensmuster Befehl

6.1.1 Name/Alternative Namen

Befehl, Kommando (engl. command), Aktion (engl. action).

6.1.2 Problem

Das Verhaltensmuster Befehl soll einen Befehl, eine durchzuführende Aktion, eine Anfrage bzw. einen Methodenaufruf – im Folgenden kurz "Befehl" genannt – in einem Objekt kapseln. Ziel ist es, dieses **Befehlsobjekt** weiterzuleiten.

Das **Verhaltensmuster Befehl** soll die Details eines Befehls vor dem Aufrufer des Befehls verbergen. Das Muster Befehl soll es erlauben, dass die **Erzeugungszeit** und die **Ausführungszeit** des Befehls getrennt werden können.



Das bedeutet, dass das Erzeugen eines Befehls und seine Ausführung zu verschiedenen Zeiten stattfinden können.

Eine **Anwendung** (hier **Client** genannt) erstellt **konkrete Befehle** und weist ihnen einen Empfänger zu.



Der entsprechende Empfänger und die auszuführenden Methoden müssen dem konkreten Befehl jeweils bekannt sein. Die Referenz auf den Empfänger ist in einem **konkreten Befehlsobjekt** gespeichert.



Das Muster lässt offen, wer für die Auslösung der Ausführung eines Befehls zuständig ist.



In der im Folgenden beschriebenen **Grundform des Musters** ist es die Aufgabe des **Aufrufers**⁶⁴, einen Befehl vom Client entgegenzunehmen, zu speichern und auf Anforderung auszuführen.



Der Aufrufer **entkoppelt** die Erzeugung eines Befehls von seiner Ausführung. Ein Client kann einen Befehl erzeugen und dem Aufrufer eine Referenz auf den Befehl übergeben. **Der Aufrufer führt den Befehl aus.**



Der **Aufrufer** erhält vom Client eine Referenz auf den Befehl und hat die Aufgabe, die Ausführung eines Befehls zu initiieren.



Der **Aufrufer** eines Befehls 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. Außerdem soll der Aufrufer den Empfänger des Befehls nicht kennen.

Der **Empfänger** soll im Befehlsobjekt referenziert werden.



6.1.3 Lösung

Das Verhaltensmuster Befehl ist ein **objektbasiertes Verhaltensmuster**. 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 der Schnittstelle `IBefehl` basierenden Objekt **gekapselt**. Die Referenz auf den Befehl wird **von einem Client dem Aufrufer übergeben**. Mit anderen Worten, der Client kennt die Ausprägung des konkreten Befehls. Der Aufrufer hingegen hängt

⁶⁴ Ein Beispiel für einen Aufrufer ist das Drücken eines Buttons (einer Schaltfläche) einer grafischen Oberfläche. Der Client muss natürlich die Referenz auf den Befehl beim Aufrufer hinterlegt haben.

nicht von den Details des Befehls ab, sondern nur von der Schnittstelle `IBefehl`. Gibt der Aufrufer die Schnittstelle selbst vor, ist er von ihr natürlich nicht abhängig.

Das Muster macht keine Aussage darüber, wer die Schnittstelle vorgibt.



Der entsprechende Empfänger und die auszuführenden Methoden müssen dem konkreten Befehl jeweils bekannt sein. Die Referenz auf den Empfänger ist im Objekt der Klasse `KonkreterBefehl` gespeichert.



Der erzeugende Client oder aber auch ein anderer kann zu einem späteren Zeitpunkt die Ausführung des Befehls veranlassen.

Die Anforderung zu einer Befehlausführung kann beispielsweise durch das Drücken einer Schaltfläche erfolgen, die von einem Client interpretiert wird und der daraufhin die Ausführung des der Schaltfläche entsprechenden Befehls veranlasst.

Abhängig vom Kontext, in dem das Befehlsmuster angewandt werden soll, kann ein Aufrufer wesentlich komplexer werden als hier in der Grundform dargestellt. Beispielsweise enthält das Programmbeispiel in Kapitel 6.1.3.4 einen Aufrufer mit einem größeren Funktionsumfang.

6.1.3.1 Klassendiagramm

Hier das Klassendiagramm:

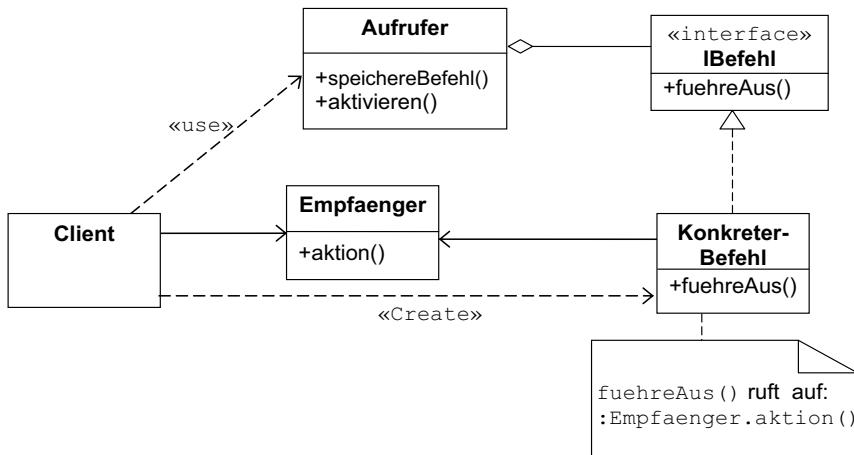


Bild 6-1 Klassendiagramm des Befehlsmusters

Der **Aufrufer** stellt in der Grundform des Musters eine Methode `speichereBefehl()` bereit, durch deren Aufruf ein Client einen Befehl über gibt. Der Aufrufer besitzt eine Methode `aktivieren()`. In dieser Methode wird die Methode `fuehreAus()` des Befehlsobjekts aufgerufen.



Ein **konkreter Befehl** implementiert die Schnittstelle `IBefehl` und definiert dadurch die Methode `fuehreAus()`. Bei Gültigkeit des liskovschen Substitutionsprinzips ist ein Aufrufer in der Lage, auch Befehlsobjekte von anderen konkreten Befehlsklassen, welche die Schnittstelle `IBefehl` implementieren, entgegenzunehmen, zu speichern und auszuführen. Durch den Aufrufer können dann die Referenzen auf solche Befehlsobjekte in einer Liste aufgereiht werden, um beispielsweise die Historie ihrer Ausführung zu speichern.

Die Methode `aktion()` des **Empfängers** symbolisiert eine Methode des Empfängerobjekts zur Ausführung des Befehls. Die Methode `fuehreAus()` des Befehlobjekts sorgt dafür, dass die Methode `aktion()`, d. h. die Ausführung des Befehls, beim Empfängerobjekt aufgerufen wird.



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, welche 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.

Auslösung eines Befehls

Ein **Client** erzeugt einen konkreten Befehl (siehe «Create» im Klassendiagramm in Bild 6-1) und speichert im erzeugten Befehlsobjekt eine Referenz auf den konkreten Empfänger. Der Client über gibt einem Aufrufer mit Hilfe dessen 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.

Das Muster legt nicht fest, wer die Ausführung eines Befehls wann auslöst. In Bild 6-2 ist es beispielsweise derselbe Client, der sowohl den Befehl erzeugt als auch dessen Ausführung auslöst. In der Grundform des Musters, die in Bild 6-2 dargestellt ist, kann die Ausführung eines von einem Aufrufer gespeicherten Befehls von einem Client ausgelöst werden, indem er die Methode `aktivieren()`⁶⁵ 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.

In einer anderen Ausprägung des Musters könnte ein Aufrufer auch 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 6.1.5 erläutert.

6.1.3.2 Teilnehmer

Die folgenden Teilnehmer sind beim Befehlsmuster involviert:

- **IBefehl**

Diese Schnittstelle definiert den Methodenkopf der Methode `fuehreAus()` für das Ausführen von Befehlen.

- **KonkreterBefehl**

Die Klasse `KonkreterBefehl` implementiert die Schnittstelle `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 Parameter. Zu diesen Informationen gehört auch eine Referenz auf das Objekt der Klasse `Empfaenger`. Der Rumpf der Methode `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 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`. Bei Einhaltung der Verträge kann diese Referenz nach dem liskovschen Substitutionsprinzip zur Laufzeit

⁶⁵ Die Methode `aktivieren()` wird hier beispielhaft benutzt.

auf jedes Befehlsobjekt zeigen, dessen Klasse die Schnittstelle `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 Befehlsmusters unterschiedlich sein. Hier wird beispielhaft eine Methode `aktion()` der Klasse `Empfaenger` benutzt. Soll ein Befehl ausgeführt werden, wird die Methode `aktion()` des im Befehlsobjekt gespeicherten Empfängerobjekts ausgeführt. Die Ausführung der Methode `aktion()` eines Empfängerobjekts wird also durch ein Befehlsobjekt gekapselt.

6.1.3.3 Dynamisches Verhalten

Nachdem der Client ein Objekt der Klasse `KonkreterBefehl` erzeugt und deren Referenz dem Aufrufer übergeben hat, erfolgt in dem folgenden Bild 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, indem es die Methode `aktion()` des Empfängerobjekts aufruft. Das Sequenzdiagramm in folgendem Bild zeigt den grundsätzlichen Ablauf des Befehlsmusters:

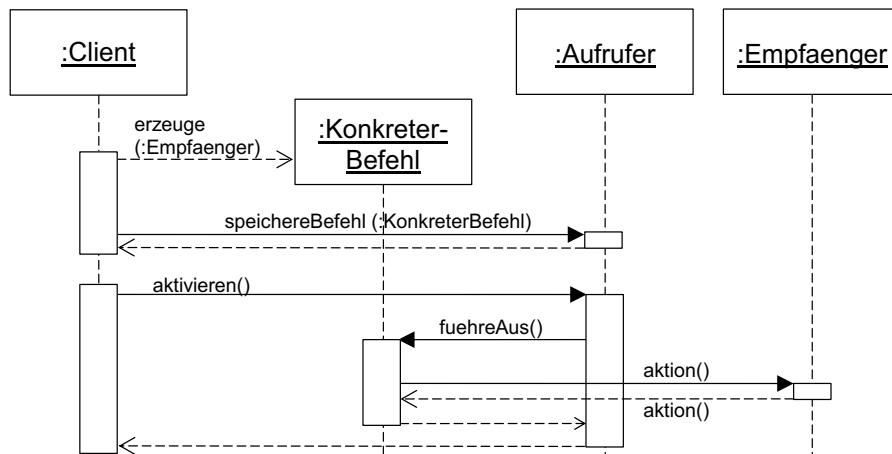


Bild 6-2 Sequenzdiagramm des Befehlsmusters

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. Ein externer 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 ist 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.

6.1.3.4 Programmbeispiel

Das Befehlsmuster wird im folgenden Beispiel verwendet, um Befehle für Lichtquellen⁶⁶ zu kapseln. `LichtQuelle` ist die Empfänger-Klasse. Diese hat zwei Methoden, um ihren Status ändern zu können – also um das Licht ein- bzw. ausschalten zu können. Folglich gibt es Objekte zweier konkreter Befehlsklassen, nämlich der Klassen `LichtEinBefehl` und `LichtAusBefehl`. Die Objekte der Klasse `LichtQuelle` unterscheiden sich noch durch einen Ort, der angibt, wo sich die Lichtquelle befindet. Die Klasse `LichtSchaltung` dient in diesem Beispiel dazu, Lichtquellen zu schalten. Die Klasse `LichtSchaltung` fungiert somit als Aufrufer-Klasse. Sie speichert Referenzen auf einen oder mehrere konkrete Befehle und kann deren Ausführung quasi als Programm zur Schaltung von Lichtquellen veranlassen.

Hier das Klassendiagramm für dieses Beispiel:

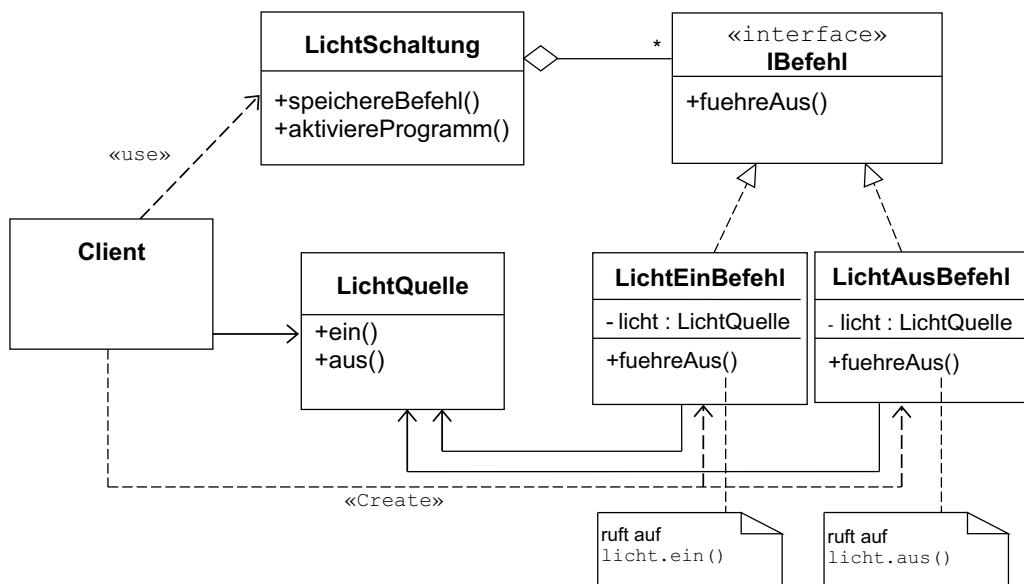


Bild 6-3 Klassendiagramm zum Programmbeispiel einer Lichtquelle

⁶⁶ Eine Lichtquelle ist beispielsweise eine Birne oder eine Lampe.

Im Folgenden wird die Schnittstelle `IBefehl`, welche die abstrakte Schnittstelle für die konkreten Befehlsklassen definiert, gezeigt:

```
// Datei: IBefehl.java

public interface IBefehl {

    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. Lichtquellen unterscheiden sich durch den Ort, an dem sie sich befinden. Der Ort wird im Konstruktor gesetzt. Hier die Klasse `LichtQuelle`:

```
// Datei: LichtQuelle.java

public class LichtQuelle {

    private final String ort;

    public LichtQuelle(String ort) {
        this.ort = ort;
    }

    public void ein() {
        System.out.println(ort + ": Licht wurde eingeschaltet!");
    }

    public void aus() {
        System.out.println(ort + ": Licht wurde ausgeschaltet!");
    }
}
```

Die Klasse `LichtSchaltung` stellt in diesem Beispiel eine **Aufrufer-Klasse** dar. Sie enthält eine Liste von Befehlen vom Typ der Schnittstelle `IBefehl`. Durch Aufruf der Methode `speichereBefehl()` wird ein weiterer Befehl zur Liste hinzugefügt. Die Befehlsliste stellt quasi ein Programm zur Schaltung von Lichtquellen dar, dessen Ausführung durch Aufruf der Methode `aktiviereProgramm()` ausgelöst werden kann. Hier die Klasse `LichtSchaltung`:

```
// Datei: LichtSchaltung.java

import java.util.*;

public class LichtSchaltung {

    private final List<IBefehl> lichtProgramm = new ArrayList<>();
```

```
public void speichereBefehl(IBefehl befehl) {
    this.lichtProgramm.add(befehl);
}

public void aktiviereProgramm() {
    for (IBefehl befehl : lichtProgramm) {
        befehl.fuehreAus();
    }
}

public void leereProgramm() {
    lichtProgramm.clear();
}
}
```

Die Klasse `LichtEinBefehl` stellt eine **konkrete Befehlsklasse** dar. Sie besitzt eine Referenz auf eine Lichtquelle und eine Methode `fuehreAus()`, welche das Licht dieser Lichtquelle einschaltet:

```
// Datei: LichtEinBefehl.java

public class LichtEinBefehl implements IBefehl {

    private final LichtQuelle licht;

    public LichtEinBefehl(LichtQuelle licht) {
        this.licht = licht;
    }

    @Override
    public void fuehreAus() { // Befehl schaltet das Licht ein
        licht.ein();
    }
}
```

Die Klasse `LichtAusBefehl` stellt ebenfalls eine **konkrete Befehlsklasse** dar. Sie besitzt eine Referenz auf eine Lichtquelle und eine Methode `fuehreAus()`, welche das Licht dieser Lichtquelle ausschaltet:

```
// Datei: LichtAusBefehl.java

public class LichtAusBefehl implements IBefehl {

    private final LichtQuelle licht;

    public LichtAusBefehl(LichtQuelle licht) {
        this.licht = licht;
    }
}
```

```

@Override
public void fuehreAus() { // Befehl schaltet das Licht aus
    licht.aus();
}
}
}

```

Die Klasse TestBefehl stellt in diesem Beispiel den **Client** dar. Sie erzeugt ein Programm zur Lichtschaltung, wie es beispielsweise bei einer Ankunft mit dem Auto zu Hause ablaufen sollte: man braucht Licht in der Garage, danach im Flur. Wenn man sich im Flur befindet, kann das Licht in der Garage ausgeschaltet werden. Danach benötigt man Licht im Wohnzimmer und, wenn man dort angekommen ist, kann das Licht im Flur ausgeschaltet werden. Das Lichtprogramm wird in dieser kleinen Simulation einfach sequentiell ausgeführt. In der Realität würde man die Ausführung der Befehle beispielsweise durch Bewegungsmelder steuern. Im Hauptprogramm werden die konkreten Befehle erzeugt und in einem Objekt der Klasse LichtSchaltung abgespeichert. Durch den Aufruf der Methode aktiviereProgramm() wird die Ausführung der gespeicherten Befehle ausgelöst und somit die Simulation des Lichtprogramms angestoßen.

Hier die Klasse TestBefehl:

```

// Datei: TestBefehl.java

public class TestBefehl{

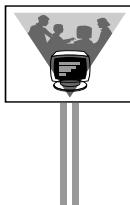
    public static void main(String[] args) {
        // Befehl-Aufrufer
        LichtSchaltung lichtProgramm = new LichtSchaltung();

        // Befehl-Empfaenger
        LichtQuelle lichtW = new LichtQuelle("Wohnzimmer");
        LichtQuelle lichtF = new LichtQuelle("Flur      ");
        LichtQuelle lichtG = new LichtQuelle("Garage     ");

        // Befehle dem Lichtschalterprogramm hinzufuegen
        lichtProgramm.speichereBefehl(new LichtEinBefehl(lichtG));
        lichtProgramm.speichereBefehl(new LichtEinBefehl(lichtF));
        lichtProgramm.speichereBefehl(new LichtAusBefehl(lichtG));
        lichtProgramm.speichereBefehl(new LichtEinBefehl(lichtW));
        lichtProgramm.speichereBefehl(new LichtAusBefehl(lichtF));

        // Gespeichertes Befehlsprogramm aktivieren
        lichtProgramm.aktiviereProgramm();
    }
}

```



Die Ausgabe des Programms ist:

```

Garage      : Licht wurde eingeschaltet!
Flur       : Licht wurde eingeschaltet!
Garage      : Licht wurde ausgeschaltet!
Wohnzimmer: Licht wurde eingeschaltet!
Flur       : Licht wurde ausgeschaltet!

```

Im konkreten, vorliegenden Beispiel ruft der Client die Methode `aktiviereProgramm()` aus Gründen der Einfachheit selbst auf. Hier würde es sich anbieten, die Ausführung zu starten, wenn beispielsweise ein Bewegungsmelder die Einfahrt des Autos registriert.

Eine andere Art der Befehlausführung könnte man durch eine Aufrufer-Klasse, welche eine Zeitschaltuhr implementiert, realisieren. Die Ausführung eines konkreten Befehls (z. B. `LichtEinBefehl`) könnte dann zu einem bestimmten Zeitpunkt veranlasst werden. Eine Lichtquelle könnte so abends automatisch eingeschaltet und am nächsten Morgen wieder automatisch ausgeschaltet werden.

6.1.4 Bewertung

6.1.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **zeitliche Entkopplung von Befehl und Ausführung**

Das Erzeugen eines Befehls und seine Ausführung sind zeitlich entkoppelt.

- **asynchroner Aufruf möglich**

Durch die Kapselung eines Befehls in einem Befehlsobjekt wird eine Zwischenschicht eingeführt. Damit kann ein Befehl die Methode `aktion()` asynchron aufrufen.

- **Befehle austauschbar**

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.

- **dynamischer Austausch von Befehlen**

Man kann Befehle dynamisch zur Laufzeit austauschen und damit das Verhalten eines Aufrufers wie beispielsweise eines Buttons einer grafischen Oberfläche verändern.

- **Wiederverwendung von Befehlen**

Befehlsobjekte können wiederverwendet werden. So kann beispielsweise bei einem Befehlsobjekt die Referenz auf ein Empfängerobjekt geändert werden. Das Befehlsobjekt kann dann erneut ausgeführt werden, wobei der Client nicht für jeden Empfänger ein komplett neues Befehlsobjekt erzeugen muss.

6.1.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **Performance-Verlust**

Durch das Verpacken eines Befehls in einem Objekt entsteht ein zusätzlicher Aufwand beim Aufruf des Befehls (Verschachtelung der Aufrufe).

- **eigene Klasse für jeden konkreten Befehl erforderlich**

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.

- **Komplexität der Implementierung**

Die Implementierung dieses Musters in verteilten Systemen wird komplex.

6.1.5 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, welche die Schnittstelle `IBefehl` implementiert. In ähnlicher Weise können die Aktionen, die durch Auswahl eines Menüpunktes in einem Menü ausgelöst werden sollen, wie beispielsweise das Speichern eines Dokumentes, realisiert werden.

Die Verwendung des Befehlsmusters bietet sich in weiteren Fällen an:

- **Logging – vereinfachte Protokollierung**

Man kann die Referenzen auf die Befehlsobjekte in einem Stream (Kanal) aufreihen, wodurch man eine Logging-Möglichkeit für Befehle hat.

- **zeitversetzte Ausführung – Einreihen der Befehle in eine Warteschlange**

Werden die Referenzen auf die Befehle in eine Warteschlange (engl. queue) einge-reiht, so kann ein Server-Prozess sie von dort aus zu gegebener Zeit zur Ausführung bringen.

- **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.

- **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. Die Schnittstelle `IBefehl` übernimmt die Rolle eines Knotens der Komponente im Kompositum-Muster, ein einfacher konkreter Befehl ist ein Blatt und ein Makro entspricht einem Kompositum.

- **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 **Memento-Pattern** (siehe Kapite 6.5) eingesetzt werden.

- **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.

6.1.6 Ähnliche Entwurfsmuster

Das Entwurfsmuster **Strategie** ist ähnlich dem Entwurfsmuster Befehl: beide kapseln einen Methodenaufruf in einem eigenständigen Objekt. Die Teilnehmer haben andere Namen, entsprechen sich aber wie folgt: der Aufrufer heißt beim Strategie-Muster Kontext, die Schnittstelle `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 6.1.5 genannten Variante des Befehlsmusters, bei der Befehle in eine Warteschlange eingereiht werden. 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.

6.2 Das Verhaltensmuster Beobachter

6.2.1 Name/Alternative Namen

Beobachter (engl. observer), Publizieren/Abonnieren (engl. publish/subscribe), Zuhörer (engl. listener).⁶⁷

6.2.2 Problem

Das Muster Beobachter soll es erlauben, dass ein Objekt von einem Beobachtbaren (engl. Publisher) abhängige Objekte der Klasse Beobachter (engl. Subscriber) von einer Änderung seines Zustands informiert, so dass eine Aktualisierung automatisch eingeleitet werden kann.



Die beteiligten Objekte sollen lose gekoppelt sein, um eine Wiederverwendung der Klassen zu ermöglichen.

Publish/Subscribe beschreibt ein Muster für den Austausch von Nachrichten, bei welchem ein Publisher automatisch einen bei ihm angemeldeten Subscriber informiert, wenn Daten des Publishers abgeändert werden.

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.



6.2.3 Lösung

Das Muster **Beobachter** ist ein **objektbasiertes Entwurfsmuster**. Ein Objekt vom Typ Beobachtbar kann eine beliebige Anzahl Objekte der Klasse Beobachter automatisch informieren, wenn sich sein Zustand ändert.

Die Struktur des Entwurfsmusters **Beobachter** enthält:

- ein Daten haltendes Objekt (**beobachtbares Objekt**/engl. **observable** oder **publisher**), dessen Daten und damit dessen Zustand sich ändern können, und
- mehrere Interessenten (**Beobachter**/engl. **observer** oder **subscriber**) für die im beobachtbaren Objekt gekapselten Daten, die gegebenenfalls auf Änderungen der Daten reagieren wollen.

Die **Beobachter** müssen aber in der **Grundform des Musters zur Laufzeit** beim Beobachtbaren **angemeldet** sein, sonst können sie bei Änderungen des Beobachtbaren

⁶⁷ In der deutschen Literatur ist praktisch nur der deutsche Name Beobachter geläufig, ansonsten alle genannten englischen Namen.

nicht von diesem informiert werden. Ein Beobachter erhält solange bei jeder Zustandsänderung des zu beobachtenden Objekts eine Nachricht, bis er sich selbst abmeldet bzw. abgemeldet wird. Dabei soll ein Daten haltendes Objekt die Interessenten noch nicht zur Kompilierzeit, sondern erst zur Laufzeit nach erfolgter Anmeldung kennen.

Bei einer Änderung der Daten generiert der Beobachtbare ein Ereignis für einen angemeldeten Beobachter.



Ob sich die Beobachter selbst anmelden oder ob sie angemeldet werden, wird durch das Muster nicht festgelegt.



Klassisch würde ein Beobachter per Polling⁶⁸ anfragen, ob sich die Daten des beobachtbaren Objekts geändert haben. Beim Muster Beobachter ist jedoch die Kontrolle vertauscht – die Kontrolle hat der Beobachtbare. Diese Vorgehensweise beim Beobachter-Muster entspricht dem Prinzip **Inversion of Control** (Umkehr der Kontrolle).

Das folgende Bild zeigt die Abhängigkeit einer Beobachter-Klasse von der beobachtbaren Klasse:

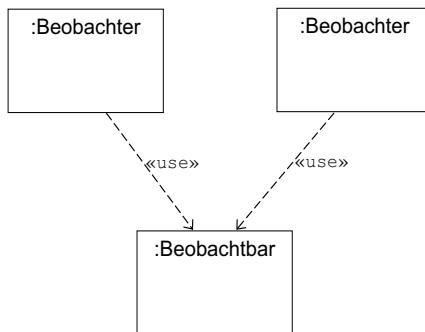


Bild 6-4 Abhängigkeit eines Beobachters vom beobachtbaren Objekt

Eine **Verwendungsbeziehung** ist eine spezielle semantische Ausprägung einer **Abhängigkeitsbeziehung**. Eine Verwendungsbeziehung wird durch den Stereotyp `<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**:

⁶⁸ To poll (engl.) bedeutet abfragen. In der Informatik hat Polling die Bedeutung, dass mittels regelmäßiger Abfragen der Zustand von Daten (Objekten, Geräten etc.) ermittelt wird.

- **Push**

Im **Push-Verfahren** wird der neue Zustand an die Aktualisierungs-Operation übergeben – die Daten werden einem Beobachter "zugeschoben".

- **Pull**

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 6.2.3.4 im Detail vorgestellt. Dort wird auch ein Vergleich zwischen beiden Verfahren durchgeführt.

6.2.3.1 Klassendiagramm

Das folgende Bild zeigt das Klassendiagramm des Beobachtermusters für das **Pull-Verfahren**:

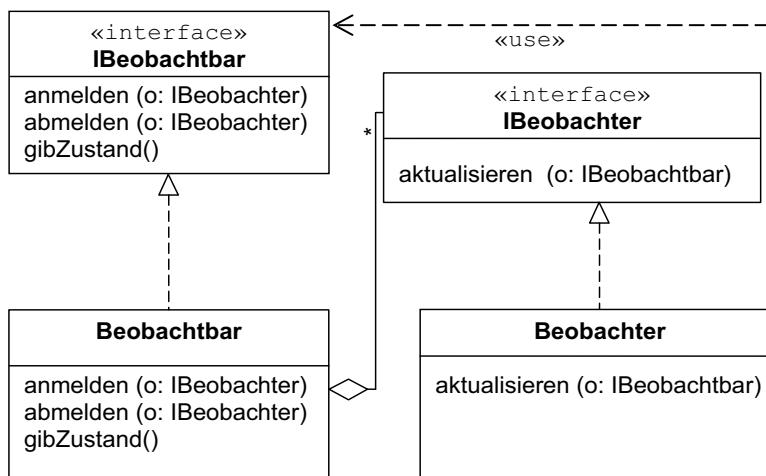


Bild 6-5 Klassendiagramm des Beobachter-Entwurfsmusters

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 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.

Beim Muster Beobachter (siehe Bild 6-5) werden Nachrichten in beide Richtungen zwischen den Objekten vom Typ **Beobachter** und dem Objekt vom Typ **Beobachtbar** ausgetauscht.

Das beobachtbare Objekt verpflichtet den Aufrufer, eine spezielle Schnittstelle (**Callback-Schnittstelle**⁶⁹⁾ zu implementieren, die vom beobachtbaren Objekt vorgegeben wird.



Ändert sich ein Beobachter, so hat dies keinen Einfluss auf das beobachtbare Objekt, solange die Callback-Schnittstelle **IBeobachter** und die Schnittstelle **IBeobachtbar** eingehalten werden. Dagegen wirken sich Änderungen an der Callback-Schnittstelle **IBeobachter** oder an der Schnittstelle **IBeobachtbar** auf die Beobachter aus.

Merkmale des Musters sind:

- Das **zu beobachtende Objekt** kennt zur Kompilierzeit nur die Schnittstelle **IBeobachter** des Beobachters, d. h. die Callback-Schnittstelle, und seine eigene Schnittstelle **IBeobachtbar**, aber nicht die Implementierung eines Beobachters.
- Das **zu beobachtende Objekt** hält zur Laufzeit nach der Anmeldung eines Beobachters eine **Referenz auf diesen Beobachter**.
- Der **Beobachter** kennt zur Kompilierzeit nur die Schnittstelle **IBeobachtbar** des Beobachtbaren und die Callback-Schnittstelle **IBeobachtbar**, welche er selbst implementiert.

Die Methodenaufrufe gehen in beide Richtungen: Zum einen benutzt ein Objekt der Klasse **Beobachter** die Schnittstelle **IBeobachtbar** des Beobachtbaren und ruft hierbei eine Methode eines Objektes auf, das diese Schnittstelle implementiert. Zum anderen hält ein beobachtbares Objekt zur Laufzeit eine Referenz auf ein Objekt, das die Schnittstelle **IBeobachter** implementiert, und ruft dessen Methode **aktualisieren()** auf. Da aber die Schnittstelle **IBeobachter** vom Beobachtbaren vorgegeben wird und damit zum Beobachtbaren gehört, ist der Beobachtbare in der Tat nicht von der Schnittstelle **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 6-4 dargestellt wurde.

Ein **Beobachter** ist nur dann ein Beobachter, wenn er

- die Schnittstelle des beobachteten Objekts, **IBeobachtbar**, benutzt und
- selbst die vom beobachteten Objekt verlangte Callback-Schnittstelle, nämlich die Schnittstelle **IBeobachter**, implementiert.



⁶⁹ Die Callback-Funktion (Rückruffunktion) wird vom Objekt der Klasse **Beobachtbar** aufgerufen.

Zum **Beobachtbaren** gehören deswegen:

- die Vorgabe der Schnittstelle des beobachteten Objekts, `IBeobachtbar`,
- die Implementierung dieser Schnittstelle und
- die Vorgabe der Schnittstelle `IBeobachter` des Beobachters (Callback-Schnittstelle).

6.2.3.2 Teilnehmer

Für die beiden Rollen eines Beobachtbaren und eines Beobachters gibt es jeweils eine **Schnittstelle**⁷⁰, nämlich die Schnittstellen `IBeobachtbar` und `IBeobachter`. Diese werden im Folgenden charakterisiert:

- **IBeobachtbar**

Diese Schnittstelle 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 die Callback-Schnittstelle des Beobachtbaren. Der Beobachtbare kennt nur diese Schnittstelle eines Beobachters. Diese Schnittstelle wird von einem Beobachter realisiert. Sie 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`:

- **Beobachtbar**

Ein Beobachtbarer hält zur Laufzeit eine aktuelle Liste von Referenzen auf Objekte, welche die Schnittstelle `IBeobachter` implementieren. Meldet sich ein Beobachter an, so wird ein neuer Eintrag zu dieser Liste hinzugefügt. Bei Abmeldung eines Beobachters wird dessen Eintrag wieder aus der Liste entfernt. Hierdurch hat der Beobachtbare immer eine aktuelle Liste seiner angemeldeten Beobachter.

Änderungen bei den Beobachtern sind für den Beobachtbaren irrelevant, solange die Beobachter die Callback-Schnittstelle `IBeobachter` einhalten. Zur Kompilierzeit kennt ein Beobachtbarer von einem Beobachter nur die Callback-Schnittstelle `IBeobachter`. Nach dem liskovschen Substitutionsprinzip kann bei Einhaltung der Verträge an die Stelle einer Schnittstelle stets ein Objekt treten, das diese Schnittstelle implementiert. Demnach besteht zwar eine Abhängigkeit des Beobachtbaren zu dieser Schnittstelle beim Kompilieren, da die Schnittstelle aber vom Beobachtbaren selbst vorgegeben wird, ist es de facto keine wirkliche Abhängigkeit.

⁷⁰ Alternativ kann auch eine abstrakte Klasse verwendet werden.

Wenn sich sein Zustand ändert, benachrichtigt der Beobachtbare die bei ihm anmeldeten Beobachter durch Aufruf der Methode `aktualisieren()`.

- **Beobachter**

Ein Beobachter implementiert die Schnittstelle `IBeobachter` und damit auch die Methode `aktualisieren()`. Wenn die Beobachter über eine Änderung benachrichtigt werden sollen, iteriert der Beobachtbare in der Methode `benachrichtigen()`⁷¹ über die gespeicherten Referenzen der Beobachter. Dabei führt er für jeden Beobachter die Operation `aktualisieren()` zur Benachrichtigung des Beobachters aus.

6.2.3.3 Dynamisches Verhalten

Das folgende Sequenzdiagramm zeigt exemplarisch die Anmeldung und Aktualisierung eines Beobachters:

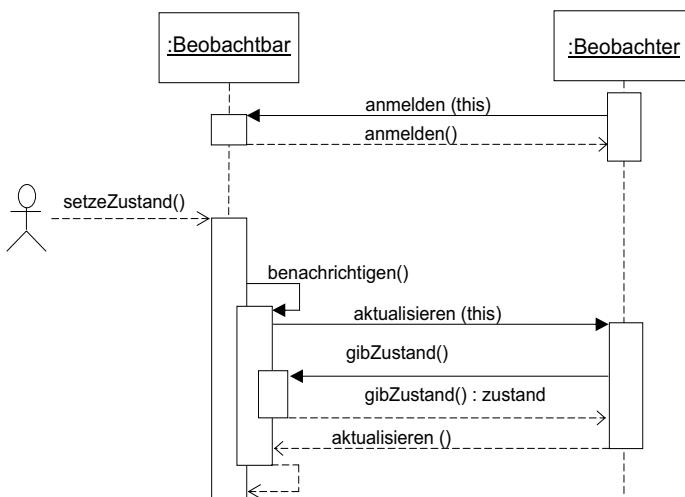


Bild 6-6 Sequenzdiagramm für die Anmeldung und Aktualisierung eines Beobachters

Jeder Beobachter meldet sich beim Objekt der Klasse `Beobachtbar`, das er beobachten will, an. Werden die Daten eines beobachtbaren Objekts geändert, was in Bild 6-6 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 anmeldeten Beobachter durch einen Aufruf der Methode `aktualisieren()` und übergibt dabei eine Referenz auf sich selbst. Ein Beobachter

⁷¹ Der genaue Ablauf wird im folgenden Kapitel beschrieben.

kann in der hier gezeigten Pull-Variante 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 Ausführungsspezifikation⁷².

Die Methode `setzeZustand()` wurde bislang nicht erwähnt und spielt im Rahmen des Beobachter-Musters keine Rolle. Diese Methode steht stellvertretend für eine anwendungsspezifische Methode einer konkreten Klasse der beobachtbaren Objekte, in welcher die für die Beobachtung relevanten Daten eines Objektes geändert werden.

6.2.3.4 Lösungsvariante mit dem Push-Verfahren

Die bisher beschriebene **Pull-Variante** ist dadurch charakterisiert, dass ein Beobachter zwar über eine erfolgte Zustandsänderung durch Aufruf der Methode `aktualisieren()` informiert wird, er sich diese Informationen über den neuen Zustand aber selbst durch Aufruf der Methode `gibZustand()` beim beobachteten Objekt abholen muss.

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 beschreiben⁷³. Durch einen Aufruf dieser **modifizierten Methode `aktualisieren()`** beim Beobachter "drückt" (engl. push) das beobachtete Objekt gleich alle Informationen zu dem Beobachter hin. Die Methode `gibZustand()` kann deshalb entfallen.

Ein **Vorteil des Push-Verfahrens** ist, dass der Rückruf des Beobachters, um sich Informationen zu beschaffen, entfällt. Allerdings hat bei der Push-Schnittstelle jede Anwendung ihre eigenen Parameter. Damit kann die entsprechende Methode in der Regel nicht wiederverwendet werden. 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 aber darin zu sehen, dass statt der bisher genannten Methode `gibZustand()` mehrere get-Methoden zur Verfügung gestellt werden können.

Ein Beobachter kann bei Vorliegen mehrerer get-Methoden in der **Pull-Methode** entscheiden, ob er Informationen abholt und ggf. welche. In der **Push-Variante** wird ein Beobachter immer mit der maximalen Information "gefüttert".



Die Lösung mit mehreren get-Methoden bei der **Pull-Variante** hat auch noch einen kleinen Vorteil bezüglich der Wartbarkeit: Wenn die Klasse `Beobachtbar` erweitert wird und man zusätzliche Daten zu einem Zustandswechsel bereitstellen möchte, kann dies

⁷² Eine Ausführungsspezifikation – in UML 1 hieß der Begriff Aktivitätsbalken oder Steuerungsfokus (engl. focus of control) – zeigt sowohl die Zeitspanne, während der eine Ausführung aktiv ist, als auch die Kontrollbeziehung zwischen der Ausführung und dem dazugehörigen Aufrufer.

⁷³ Damit verringert sich die Wiederverwendbarkeit der Methode `aktualisieren()`. Aber auch die Methode `gibZustand()` beim Pull-Verfahren hat spezielle Rückgabetypen!

in manchen Fällen über **zusätzliche get-Methoden** in der Schnittstelle `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 in der Schnittstelle `IBeobachtbar` als die gerade beschriebenen auch in der Pull-Variante zu Änderungen in den Beobachter-Klassen führen.

6.2.3.5 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. Dieses **Beispiel** ist gemäß dem **Pull-Verfahren** aufgebaut.

Die **Schnittstelle IBeobachter** definiert den **Aufruf eines Beobachters**, der durch die Methode `aktualisieren()` über Änderungen informiert werden kann:

```
// Datei: IBeobachter.java

public interface IBeobachter {

    void aktualisieren(IBeobachtbar beobachtbar);

}
```

Die Schnittstelle `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 die Schnittstelle `IBeobachtbar`:

```
// Datei: IBeobachtbar.java

public interface IBeobachtbar {

    void anmelden(IBeobachter beobachter);

    void abmelden(IBeobachter beobachter);

    String gibZustand();

}
```

Die **Klasse Abonent** stellt eine Implementierung der Schnittstelle `IBeobachter` dar:

```
// Datei: Abonent.java

public class Abonent implements IBeobachter {
```

```
private final String name;

public Abonnent(String name) {
    this.name = name;
}

@Override
public void aktualisieren(IBeobachtbar beobachtbar) {
    System.out.printf("Neue Nachricht fuer %s.%n", name);
    System.out.printf("Nachricht: %s%n", beobachtbar.getZustand());
}
```

Die Klasse **Newsletter** stellt eine Implementierung der Schnittstelle **IBeobachtbar** dar:

```
// Datei: Newsletter.java

import java.util.*;

public class Newsletter implements IBeobachtbar {

    private final List<IBeobachter> abonnenten = new ArrayList<>();
    private String nachricht;

    public void aendereNachricht(String nachricht) {
        this.nachricht = nachricht;
        benachrichtigen();
    }

    @Override
    public void anmelden(IBeobachter beobachter) {
        abonnenten.add(beobachter);
    }

    @Override
    public void abmelden(IBeobachter beobachter) {
        abonnenten.remove(beobachter);
    }

    @Override
    public String getZustand() {
        return nachricht;
    }

    private void benachrichtigen() {
        abonnenten.forEach(b -> b.aktualisieren(this));
    }
}
```

Die Klasse `TestBeobachter` zeigt die Funktionalität eines Newsletter:⁷⁴

```
// Datei: TestBeobachter.java

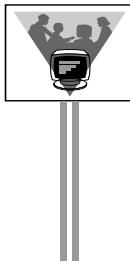
public class TestBeobachter {

    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
```

Wenn die "Neuigkeit 3" beim Newsletter eintrifft, ist kein Beobachter mehr angemeldet. Folglich kann diese Nachricht nicht mehr ausgeliefert werden.

6.2.4 Bewertung

6.2.4.1 Vorteile

Der **Nutzen** des Beobachter-Musters liegt in der hieraus resultierenden **losen Koppelung**. Hierbei ergeben sich unter anderem die folgenden Vorteile:

- **Kapselung der Implementierung der Beobachter**

Der Beobachtbare braucht zur Kompilierzeit die Beobachter nicht zu kennen, sondern nur deren Schnittstelle, welche aber von ihm selbst vorgegeben wird.

⁷⁴ Aus Gründen der Einfachheit werden hier die Beobachter angemeldet.

- **Erweiterbarkeit um neue Beobachter**

Neue Beobachter können jederzeit hinzugefügt werden.

- **Gültigkeit des Beobachtbaren auch für neue Beobachter**

Der Beobachtbare muss nie modifiziert werden, um neue Beobachter zu unterstützen.

- **unabhängige Wiederverwendung von Beobachter und Beobachtbarem**

Beobachter und Beobachtbarer können unabhängig voneinander wiederverwendet werden.

- **Implementierungen gekapselt**

Änderungen am Beobachtbaren, ohne dessen Schnittstelle zu ändern, haben keine Auswirkung auf die Gegenseite. Ein Beobachter kann auch beliebig geändert werden, solange er die Callback-Schnittstelle einhält.

6.2.4.2 Nachteile

Es können jedoch auch Nachteile entstehen:

- **Verfahren kann bei vielen Beobachtern aufwendig werden**

Gibt es besonders viele Beobachter, so kann deren Benachrichtigung durch den Beobachtbaren sehr zeitaufwendig sein.

- **Gefahr von Endlosschleifen**

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 könnte.

- **Information eines Beobachters, obwohl dieser eine Änderung nicht benötigt**

Jeder Beobachter wird sowohl beim Push-Verfahren als auch beim Pull-Verfahren informiert, auch wenn er eine Änderung nicht braucht.

- **aufwendiger Synchronisationsbedarf**

In Anwendungen, bei denen Beobachter und beobachtbare Objekte in parallelen Threads laufen, müssen die beobachtbaren Objekte bzw. deren Methoden synchronisiert werden. Sonst könnte sich beispielsweise der Zustand eines beobachtbaren Objekts ändern, bevor ein Beobachter die Zustandsänderung abfragen kann, für die er benachrichtigt wurde.

6.2.5 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 von dieser Änderung informiert werden müssen. Besonders häufig findet man eine solche Problemstellung bei der Programmierung grafischer Oberflächen. Eine View im Architekturmuster **Model-View-Controller** ist ein Beobachter im Sinne des Beobachter-Musters.

Das Beobachter-Muster kann auch Verwendung im **Vermittler-Muster** finden: Objekte – im Vermittler-Muster Kollegen genannt – 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 nicht um eine Schnittstelle, 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 die Schnittstelle `java.util.Observer` vorgegeben, welche ein Beobachter implementieren muss.

6.2.6 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 potenziell jedes Objekt mit jedem anderen über den Vermittler kommunizieren kann, haben die am Beobachter-Muster beteiligten Objekte bestimmte Rollen, welche 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 jedoch die komplexe Zusammenarbeit zwischen den Kollegen beim Vermittler-Muster 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.

6.3 Das Verhaltensmuster Besucher

6.3.1 Name/Alternative Namen

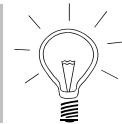
Besucher (engl. visitor).

6.3.2 Problem

Unter einer **Objektstruktur** wird im Folgenden ein – mehr oder weniger zusammenhängender – Satz von **Objekten unterschiedlicher Klassen** verstanden. Als einfaches Beispiel für eine Objektstruktur sei hier eine Liste von Objekten genannt.

Wenn eine **neue Funktionalität** realisiert werden muss, die **auf allen Objekten einer solchen heterogenen Objektstruktur** aus verschiedenen Klassen arbeitet, dann könnten Teilfunktionalitäten lokal in jedem einzelnen Objekt der Objektstruktur durchgeführt werden. Damit wäre aber die Logik der neuen Funktion über viele Klassen verstreut und die Klassen selbst wären mit zusätzlichen Funktionen erweitert, die nicht zu ihrem eigentlichen Kern gehören. Ziel ist es deshalb, die neue Funktionalität, welche die Objekte einer Objektstruktur bearbeitet, als **Operation in einer eigenen Klasse separat zu der Objektstruktur** zu implementieren. Die **neue Operation "besucht"** die Objekte der Objektstruktur und holt sich dabei die benötigten Informationen bei den Objekten der Objektstruktur ab. Die Klassen der Objektstruktur sollen dabei möglichst unverändert bleiben.

Das Besucher-Muster soll es ermöglichen, eine neue Operation auf den Objekten einer heterogenen Objektstruktur zu implementieren, wobei die Klassen der Objekte möglichst wenig geändert werden sollen.



6.3.3 Lösung

Das Besucher-Muster kapselt die Logik einer Operation, welche **Daten von Objekten verschiedener Klassen einer Objektstruktur** benötigt, komplett in einem separaten Objekt einer **Besucher-Klasse**, einem sogenannten Besucher-Objekt. Das **Besucher-Muster** ist ein **objektbasiertes Verhaltensmuster**.

Ein wichtiges Kennzeichen des Besucher-Musters ist, dass die Objekte der Objektstruktur Instanzen von unterschiedlichen Klassen sein können. Die Klassen, deren Instanzen sich in der Objektstruktur befinden, werden im Folgenden als **Element-Klassen** bezeichnet.



Die Klasse **Element** als Basisklasse aller Element-Klassen wird in diesem Kapitel zur einfacheren Beschreibung und zum leichteren Verständnis des Musters eingeführt. Das Besucher-Muster fordert eine solche Basisklasse für die Element-Klassen jedoch nicht.



Zur Lösung wird eine abstrakte Klasse⁷⁵ **Besucher** eingeführt, von der alle konkreten Besucher-Klassen abgeleitet sind. Die abstrakte Klasse **Besucher** definiert die Köpfe von überladenen **besuchen()**-Methoden und zwar **eine besuchen() -Methode für**

⁷⁵ Die Beschreibung des Musters in diesem Buch verwendet für die Abstraktion der Besucher-Klassen eine abstrakte Klasse. Die Abstraktion könnte aber genauso gut mittels einer Schnittstelle definiert werden. Die konkreten Besucher-Klassen müssten dann diese Schnittstelle realisieren, statt von der abstrakten Klasse abzuleiten.

jede in der Objektstruktur vorhandene Element-Klasse. In einer konkreten Besucher-Klasse wird die benötigte neue Operation auf den Elementen der Datenstruktur realisiert und in den jeweiligen `besuchen()`-Methoden implementiert.

Die abstrakte Klasse `Besucher` wird als Typ für den jeweiligen Übergabeparameter einer `akzeptieren()`-Methode gewählt, die in den Element-Klassen implementiert werden muss.



Der Zweck der Methode `akzeptieren()` und das Zusammenspiel von `akzeptieren()`- und `besuchen()`-Methoden 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.



Dadurch dass die Element-Klassen nur die erwähnte `akzeptieren()`-Methode implementieren müssen, kann das Besucher-Muster auf heterogenen Objektstrukturen angewendet werden, bei denen die beteiligten Element-Klassen keine Beziehung untereinander haben.



Besucher-Klassen und ihre `besuchen()`-Methoden

Für jede zu realisierende Operation wird im Rahmen des Besucher-Musters 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**. Damit ist die Logik dieser neuen Operation selbst nicht über viele Objekte verteilt, sondern nur die Datenbeschaffung.



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.

Element-Klassen und ihre `akzeptieren()`-Methoden

Die bereits bestehenden Element-Klassen müssen für einen Besuch durch ein Besucher-Objekt um eine `akzeptieren()`-Methode ergänzt werden, wenn sie diese nicht bereits schon aufweisen. Das Besucher-Objekt wird als Parameter an die `akzeptieren()`-Methode übergeben. Somit kennt ein Element-Objekt in seiner `akzeptieren()`-Methode das Besucher-Objekt.

Die `akzeptieren()`-Methode einer Element-Klasse hat als Überparameter ein Besucher-Objekt. Das Element-Objekt, dessen `akzeptieren()`-Methode aufgerufen wird, kennt damit das Besucher-Objekt und kann so im Rumpf dieser Methode die für eine Element-Klasse spezifische `besuchen()`-Methode des übergebenen Besuchers aufrufen. Dabei übergibt das Element-Objekt eine Referenz auf sich selbst.



Wie bereits erwähnt wird als Typ des Parameters der `akzeptieren()`-Methode die abstrakte Klasse `Besucher` gewählt. Dadurch 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 `akzeptieren()`-Methode muss von allen Daten tragenden Klassen, deren Objekte besuchbar sein sollen, implementiert werden. Dabei besteht der Rumpf jeder dieser `akzeptieren()`-Methode 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 Klassendiagramm angedeuteten get-Methoden der konkreten Element-Klasse (siehe Kapitel 6.3.3.1) sich Daten vom aufrufenden Objekt zu besorgen, um seine zentrale Funktion zu realisieren.

Obwohl die Implementierung der `akzeptieren()`-Methode in jeder Element-Klasse gleich ist, muss sie trotzdem in jeder konkreten Element-Klasse stattfinden und darf nicht in eine Basisklasse verlagert werden, da dem Besucher sonst nur eine Referenz vom Typ der Basisklasse übergeben würde.



Aus diesem Grund ist die in der folgenden Beschreibung verwendete Basisklasse `Element` für die Element-Klassen eine abstrakte Klasse. Sie definiert nur die Schnittstelle der Methode `akzeptieren()`, aber sie implementiert diese Methode nicht.

Zusammenspiel von `besuchen()` - und `akzeptieren()`-Methoden als simuliertes Double-Dispatch

Die abstrakten Klassen `Element` und `Besucher` sind so miteinander verbunden, dass jedes konkrete Besucher-Objekt jedes konkrete Daten tragende Objekt besuchen kann und dass jeder Besuchte jeden Besucher akzeptieren muss.



Durch den Aufruf der entsprechenden `besuchen()`-Methode innerhalb der `akzeptieren()`-Methode 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 **simulierte "double dispatch"** bezeichnet.

Eigentlich hängt die Auswahl einer 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 aller Element-Klassen deklarierte `akzeptieren()`-Methode – wie bereits beschrieben – implementieren. Die `akzeptieren()`-Methode erhält ein konkretes Besucher-Objekt als Übergabeparameter. Im Rumpf der `akzeptieren()`-Methode rufen die konkreten Elemente die `besuchen()`-Methode dieses Besucher-Objektes 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 Hin- und Her-Aufrufe kommen also durch die Simulation von Double-Dispatch in Java zustande. Die Technik des Double-Dispatch wird ausführlich in [Bec08] beschrieben.

Zusammenfassung des Besucher-Musters

Einerseits weicht der Ansatz des Besucher-Musters von dem Grundgedanken der Objektorientierung ab, die Daten eines Objekts und die zu den Daten gehörenden Prozeduren, die auf den Daten operieren, in ein einziges Objekt zu kapseln. Denn beim Besucher-Muster verarbeiten und beschaffen Operationen, die jeweils in einer separaten Klasse, der Besucher-Klasse, gekapselt werden, die Daten von Objekten verschiedener Klassen einer Objektstruktur mit jeweils unterschiedlichen Schnittstellen. Die beschafften Daten werden dann zentral im Besucher-Objekt ausgewertet.

Andererseits hat diese Vorgehensweise den Vorteil, dass die Klassen der Daten tragenden Objekte der Objektstruktur nicht für neu hinzukommende Operationen verändert werden müssen⁷⁶. Der Ansatz folgt damit dem **Open-Closed-Prinzip** (siehe Kapitel 3.1.5): die Element-Klassen bleiben geschlossen, sind aber über die `akzeptieren()`-Methode offen für neue Besucher. Daneben kann noch das Prinzip **Separation of Concerns** (siehe Kapitel 3.1.3) angeführt werden. Das Besucher-Muster folgt diesem Prinzip, da für einen neuen Belang ein neuer Besucher implementiert wird und nicht die Element-Klassen mit diesem neuen Belang belastet werden.

6.3.3.1 Klassendiagramm

Das folgende Klassendiagramm beschreibt die statische Struktur der Teilnehmer des Besucher-Musters:

⁷⁶ Dies gilt natürlich nur, solange die neue Operation nur solche Daten benötigt, die bereits über die öffentliche Schnittstelle der Objekte zur Verfügung stehen.

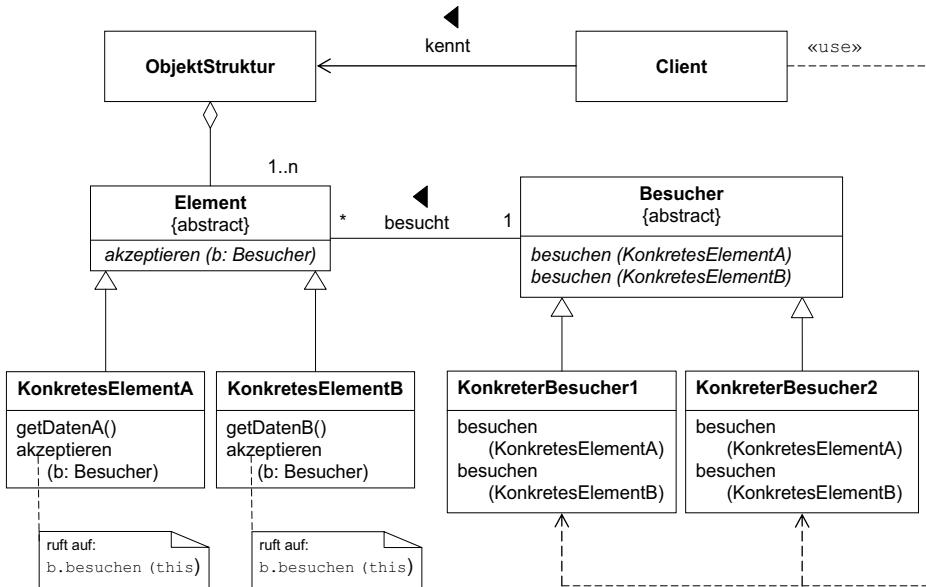


Bild 6-7 Klassendiagramm des Besucher-Musters unter Verwendung einer Objektstruktur

In der linken Hälfte des Klassendiagramms 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 6.3.3.3 im Detail eingegangen.

Die Elemente der Objektstruktur sind Instanzen der Klassen `KonkretesElementY` ($Y = A \dots Z$). 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.

In der rechten Hälfte des Klassendiagramms 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-Klasse⁷⁷.



Die konkreten Besucher-Klassen `KonkreterBesucherX` ($X = 1 \dots n$) implementieren diese `besuchen()`-Methoden, indem sie im Rumpf einer `besuchen()`-Methode auf die Daten des übergebenen Objekts der Objektstruktur – beispielsweise über die `get-`

⁷⁷ 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.

Methoden der entsprechenden Element-Klasse – zugreifen und diese Daten zur Realisierung der gewünschten Funktionalität der jeweiligen Besucher-Klasse nutzen.

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.



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 stellvertretend für eine Anwendung des Besucher-Musters. Der Client kennt die Objektstruktur und möchte auf dieser Objektstruktur eine Besuchsoperation ausführen.

Das Muster legt nicht fest, ob der Client ein Besucher-Objekt, das eine Operation realisiert, selbst erzeugt oder ob ein Besucher-Objekt ihm von außen übergeben wird.



Der Client stößt den Besuch der Objektstruktur durch das Besucher-Objekt an. Der Ablauf kann je nach Objektstruktur unterschiedlich sein. 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.

6.3.3.2 Teilnehmer

Am Besucher-Entwurfsmuster nehmen die folgenden Klassen teil:

- **Besucher**

Die abstrakte Klasse `Besucher` deklariert eine `besuchen()`-Methode für jede Klasse `KonkretesElementY` ($Y = A \dots Z$). 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` abgeleitet⁷⁸. 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` deklarierte `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. Durch das Besucher-Muster ist nicht festgelegt, wie ein Client mit der Objektstruktur interagiert und wie ein Besuchsvorgang dynamisch abläuft, solange nicht die zu besuchende Objektstruktur konkret betrachtet wird. In Kapitel 6.3.3.3 werden verschiedene Alternativen ausführlich diskutiert.

- **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 – abhängig von der konkreten Ausprägung der Objektstruktur.

6.3.3.3 Dynamisches Verhalten

Wie bereits erwähnt, wird durch das Besucher-Muster nicht festgelegt, wie ein Client mit der Objektstruktur interagiert. Hier sind verschiedene Varianten denkbar, die im weiteren Verlauf dieses Kapitels noch ausführlich diskutiert werden.

Im Folgenden wird nun zuerst eine Variante benutzt, bei welcher der Client eine Methode der Objektstruktur (Methode `besucheElemente()`) aufruft und dabei das gewünschte Besucher-Objekt übergibt.

⁷⁸ Das Besucher-Muster wird oftmals so implementiert. Wie bereits erwähnt, verlangt das Muster jedoch streng genommen nur, dass die Klassen `KonkretesElementY` eine `akzeptieren()`-Methode haben.

Das folgende Sequenzdiagramm zeigt das dynamische Verhalten der Teilnehmer in dieser Variante:

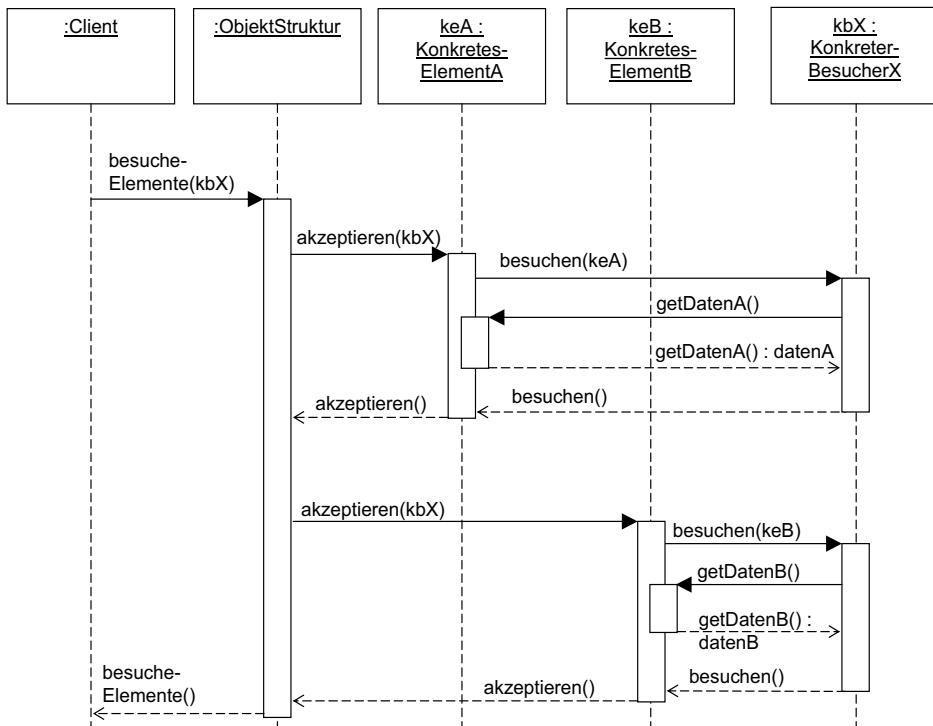


Bild 6-8 Sequenzdiagramm des Besucher-Musters

Der Client kennt die Objektstruktur und möchte die Operation, die in einem Besucher-Objekt `kbX` vom Typ `KonkreterBesucherX` gekapselt ist, ausführen. Wie bereits erwähnt ruft er dazu die Methode `besucheElemente()` der Objektstruktur auf und übergibt dabei das Besucher-Objekt.

Das Objekt der Klasse `ObjektStruktur` sorgt in der hier vorgestellten Variante selbst dafür, dass die zwei Elemente `keA` und `keB`, die in dieser Situation als einzige in der Datenstruktur enthalten sind, besucht werden. Es ruft die `akzeptieren()`-Methode des Objekts `keA` vom Typ `KonkretesElementA` auf und über gibt 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 über gibt 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 6-8 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 6-8 durch einen verschobenen Aufrufbalken für die Aufrufe der get-Methoden bei den Element-Objekten ausdrückt. Diese Hin- und Her-Aufrufe simulieren das bereits angeprochene Double-Dispatch und ermöglichen es, dass jedes Besucher-Objekt jedes Element-Objekt besuchen kann.

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 6-8 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, welche die Elemente enthält (z. B. einem 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 eins nach dem anderen jeweils ein Objekt der vorhandenen Typen `KonkretesElementY` (`Y = A..Z`) 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** (vgl. Kapitel 6.4) 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 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. Diese Variante ist im Programmbeispiel in Kapitel 6.3.3.4 zu sehen.

- **Besucher-Objekt**

Das Besucher-Objekt kann selbst die Rolle der Objektstruktur übernehmen, wenn es alle zu besuchenden Objekte kennt. Allerdings muss dann der Algorithmus für die Besuchsreihenfolge in jeder konkreten Besucher-Klasse dupliziert werden. Hängt die Besuchsreihenfolge von Ergebnissen der Methoden ab, die während des Besuchs auf der Objektstruktur durchgeführt werden, dann bietet es sich an, die Traversierung in einem Besucher-Objekt zu realisieren.

- **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` auf – im Fall eines Baums die `akzeptieren()`-Methode des Wurzelknotens – 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.

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 Elementen auf. Ein Blatt-Element ruft nur die `besuchen()`-Methode für sich auf.

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.

6.3.3.4 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 Klassendiagramm 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

public abstract class MitarbeiterBesucher {

    public abstract void besuchen(Teamleiter mitarbeiter);

    public abstract void besuchen(Sachbearbeiter mitarbeiter);

}
```

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

import java.util.*;

public class Gehaltsdrucker extends MitarbeiterBesucher {

    private static final String SPALTEN_FORMAT = "%17s%23s%12s%8s";
    private static final String GELD_FORMAT = "%.2f";

    // Gibt den Header aus
    public void druckHeader() {
        char[] starChars = new char[60];
        Arrays.fill(starChars, '*');
        String starLine = new String(starChars);
        String centeredHeader = String.format("%36s", "Gehaltsliste");
        String titles = String.format(SPALTEN_FORMAT, "Position",
            "Name", "Gehalt", "Prämie");
        System.out.println(starLine);
        System.out.println(centeredHeader);
        System.out.println(titles);
        System.out.println(starLine);
    }
}
```

```

@Override
public void besuchen(Teamleiter mitarbeiter) {
    System.out.printf(SPALTEN_FORMAT,
        "Leiter " + mitarbeiter.getTeamName(),
        mitarbeiter.getName(),
        String.format(Locale.GERMAN, GELD_FORMAT,
            mitarbeiter.getGrundgehalt()),
        String.format(Locale.GERMAN, GELD_FORMAT,
            mitarbeiter.getPraemie()));
    System.out.println();
}

@Override
public void besuchen(Sachbearbeiter mitarbeiter) {
    System.out.printf(SPALTEN_FORMAT,
        "Sachbearbeiter",
        mitarbeiter.getName(),
        String.format(Locale.GERMAN, GELD_FORMAT,
            mitarbeiter.getGehalt()),
        "---");
    System.out.println();
}
}

```

Durch die Klasse **Gesellschaft** wird die Objektstruktur realisiert. Sie beinhaltet Beispielinstanzen der konkreten Elemente:

```

// Datei: Gesellschaft.java

import java.util.*;

public class Gesellschaft {

    private final List<Mitarbeiter> personal = new ArrayList<>();

    public Gesellschaft() {
        // Teamleiter
        personal.add(
            new Teamleiter("Hirschle, Frank", "Team 1", 80000f, 800f));
        personal.add(
            new Teamleiter("Steib, Corinna", "Team 2", 75000f, 750f));
        // Sachbearbeiter
        personal.add(new Sachbearbeiter("Müller, Markus", 48200f));
        personal.add(new Sachbearbeiter("Neustedt, Silvia", 45500f));
        personal.add(new Sachbearbeiter("Weiss, Alexandra", 37120f));
        personal.add(new Sachbearbeiter("Kienzle, Michael", 43500f));
    }

    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` 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

public abstract class Mitarbeiter {

    private final String name;

    public Mitarbeiter(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract void akzeptieren(MitarbeiterBesucher besucher);
}
```

Die Klasse `Sachbearbeiter` ist eine konkrete Element-Klasse. Sie ist von der abstrakten Klasse `Mitarbeiter` abgeleitet und implementiert die `akzeptieren()`-Methode:

```
// Datei: Sachbearbeiter.java

public class Sachbearbeiter extends Mitarbeiter {

    private final double gehalt;

    public Sachbearbeiter(String name, double gehalt) {
        super(name);
        this.gehalt = gehalt;
    }

    public double getGehalt() {
        return gehalt;
    }

    @Override
    public void akzeptieren(MitarbeiterBesucher besucher) {
        besucher.besuchen(this);
    }
}
```

Die Klasse `Teamleiter` stellt ebenfalls eine konkrete Element-Klasse dar. Sie ist auch von der abstrakten Klasse `Mitarbeiter` abgeleitet und implementiert die von der abstrakten Klasse `Mitarbeiter` vorgegebene Methode `akzeptieren()`:

```
// Datei: Teamleiter.java

public class Teamleiter extends Mitarbeiter {

    private final String teamName;
    private final double grundgehalt;
    private final double praemie;

    public Teamleiter(String name, String teamName,
                      double grundgehalt, double praemie) {
        super(name);
        this.teamName = teamName;
        this.grundgehalt = grundgehalt;
        this.praemie = praemie;
    }

    public String getTeamName() {
        return teamName;
    }

    public double getGrundgehalt() {
        return grundgehalt;
    }

    public double getPraemie() {
        return praemie;
    }

    @Override
    public void akzeptieren(MitarbeiterBesucher besucher) {
        besucher.besuchen(this);
    }
}
```

Innerhalb der `main()`-Methode der Klasse `TestBesucher` 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. Dies geschieht in einer `foreach`-Schleife implizit mithilfe eines **Iterator-Objekts**, das die List-Klasse in Java zur Verfügung stellt. Hier die Klasse `TestBesucher`:

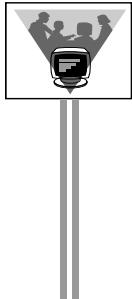
```
// Datei: TestBesucher.java

public class TestBesucher {

    public static void main(String[] args) {
        // Initialisierungen vornehmen
        Gesellschaft firma = new Gesellschaft();

        // Besucher-Objekt fuer die Liste erzeugen
        Gehaltsdrucker besucher = new Gehaltsdrucker();
        besucher.druckHeader();
```

```
// Ueber die Liste iterieren und Besuche durchfuehren
firma.getPersonal().foreach(m -> m.akzeptieren(besucher));
}
}
```



Hier das Protokoll des Programmlaufs:

Gehaltsliste				
Position	Name	Gehalt	Prämie	
Leiter Team 1	Hirschle, Frank	80000,00	800,00	
Leiter Team 2	Steib, Corinna	75000,00	750,00	
Sachbearbeiter	Müller, Markus	48200,00	---	
Sachbearbeiter	Neustedt, Silvia	45500,00	---	
Sachbearbeiter	Weiss, Alexandra	37120,00	---	
Sachbearbeiter	Kienzle, Michael	43500,00	---	

6.3.4 Bewertung

6.3.4.1 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. Wenn die Daten tragenden Element-Klassen die Schnittstelle für den Besuch beliebiger Besucher bereits zur Verfügung stellen, genügt es, 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 Klassenhierarchie-übergreifender Besuche**

Besucher können Element-Objekte besuchen, die nicht Teil derselben Klassenhierarchie sind. Das Muster Besucher setzt 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**

Ein konkreter Besucher ist ein eigenständiges Objekt und kann (Zustands-)Informationen der besuchten Objekte sammeln, was man ansonsten mit Hilfe von globalen Variablen lösen müsste oder dadurch, dass man Argumente an die Traversierungs-

operationen weitergibt und auf diese Weise Informationen "durchschleift". Falls erforderlich, kann ein Besucher-Objekt sogar Informationen über mehrere Besuche hinweg speichern.

- **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.

- **Entkopplung der Element-Klassen von den Besucher-Klassen**

Besucher-Klassen müssen die Schnittstellen aller Element-Klassen kennen, um ihre Funktionalität zu erbringen. Die Element-Klassen bleiben aber weitestgehend von den Besucher-Klassen entkoppelt, da sie nur die abstrakte Besucher-Klasse kennen müssen.

- **Möglichkeit, Frameworks zu erweitern**

Stehen die Daten tragenden Klassen wie beispielsweise bei einem 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 wieder geändert werden muss.

6.3.4.2 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 anzuwenden, wenn sich die Menge der Element-Klassen häufig ändert.

- **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 Informationen von den Elementen. Er kann aber 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 beispielsweise über get-Methoden öffentlich zugänglich macht, wenn ein konkreter Besucher diese Daten benötigt. Die Schnittstellen von Element-Klassen müssen in diesem Fall für einen konkreten Besucher erweitert werden.

6.3.5 Einsatzgebiete

Voraussetzung für den Einsatz des Musters Besucher 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 und dieses Objekt zur Erbringung seiner Funktionalität Informationen von jedem Objekt der Objektstruktur benötigt. 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 Schnittstelle der Basisklasse der Klasse `Knoten` (vgl. Kapitel 5.5.4.2) eingeführt werden. Eine Änderung an der Schnittstelle der Basisklasse 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** (siehe Kapitel 5.5) 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.

Die bisherige Beschreibung des Besucher-Musters beschränkte sich darauf, dass ein Besucher Informationen über die Objekte einer Objektstruktur sammelt. Ein Besucher kann prinzipiell alle öffentlichen Methoden der Element-Klassen nutzen. Wenn beispielsweise die Element-Klassen set-Methoden anbieten, kann ein Besucher auch die Zustände der Objekte einer Objektstruktur ändern.

6.3.6 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 hingegen 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 `akzeptieren()`-Methode dem Besucher zur Verfügung stellen.

Ein **Iterator** und ein **Besucher** des Musters 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, wohingegen 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.

6.4 Das Verhaltensmuster Iterator

6.4.1 Name/Alternative Namen

Iterator, Cursor (engl. cursor).

6.4.2 Problem

In vielen Anwendungen werden Daten aggregiert und in Datenstrukturen wie beispielsweise Arrays, Listen, Tabellen oder Bäumen gespeichert. Häufig müssen alle Elemente einer solchen Datenstruktur bearbeitet werden. Nutzt eine Anwendung das Wissen über den internen Aufbau der Datenstruktur, um auf die einzelnen Elemente zuzugreifen, dann ist sie von der Implementierung der Datenstruktur abhängig.

Mit einem sogenannten **Iterator**⁷⁹ soll nacheinander auf die einzelnen Objekte einer 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.



⁷⁹ Der Name Iterator wird sowohl für ein iterierendes Objekt als auch für das entsprechende objektbasierte Verhaltensmuster verwendet.

6.4.3 Lösung

Ein **Iterator** ist ein Objekt, mit dem sequenziell über alle Objekte einer Datenstruktur "iteriert"⁸⁰ werden kann.



Eine Iterator-Klasse bietet eine Schnittstelle, mit der nacheinander auf die einzelnen Objekte bzw. Elemente einer Datenstruktur zugegriffen wird.

Ein **Iterator-Objekt** "zeigt" auf ein aktuelles Element der entsprechenden Datenstruktur – daher auch der **alternative Name "Cursor"** – und ist in der Lage, das nachfolgende Element in der Datenstruktur zu bestimmen.



Ein Iterator als Objekt trennt den Mechanismus des Traversierens⁸¹ 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.

Die Realisierung der Traversierung wird aus der Datenstruktur herausgehalten. Dadurch wird die Formulierung der Datenstruktur einfacher. Ein Iterator wandert über die Elemente einer Datenstruktur nach einer bestimmten **Durchlaufstrategie (Traversierungsart)**. Dabei besucht er kein Element mehrmals.

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.



Eine Traversierung kann im Falle einer Liste zum Beispiel sequenziell vorwärts oder sequenziell rückwärts erfolgen. Bei einem Baum kann beispielsweise nach den Strategien Depth-First⁸² oder Breadth-First traversiert werden. Ein Iterator könnte auch eine Datenstruktur sortieren und vom kleinsten zum größten Element laufen oder umgekehrt. Bei der Traversierung kann ein Iterator eine Filterbedingung überprüfen und nur solche Elemente liefern, welche der Bedingung genügen.

⁸⁰ Iterieren bedeutet: Es wird schrittweise auf jedes Objekt einer Datenstruktur zugegriffen.

⁸¹ Mit Traversieren bezeichnet man das Durchlaufen einer Datenstruktur in einer bestimmten Reihenfolge.

⁸² 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. Bei der Breadth-First-Strategie geht man in die Breite, besucht also zuerst alle Kindknoten, dann deren Kindknoten usw.

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 Ablauf einer Iteration. Dadurch wird die Iteration wiederverwendbar. Sie muss also nicht wie bei einem externen Iterator in jedem Client 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" [Gam94] der sogenannten "**Gang of Four**" (GoF) vorgestellt. Da interne Iteratoren den Ablauf der Iteration kapseln, ist von außen ihre Eigenschaft als Iterator nicht mehr erkennbar. Interne Iteratoren werden in diesem Kapitel nur am Rande betrachtet.

Im Folgenden werden **externe Iteratoren** behandelt.

Ein externer Iterator hat typischerweise die beiden Methoden `next()` und `hasNext()`. 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, die vom Iterator noch nicht besucht wurden, 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**:

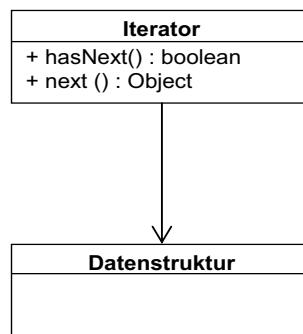


Bild 6-9 Trennung von Iterator und der zu durchquerenden Datenstruktur

Die in diesem Bild gezeigte Schnittstelle eines Iterators ist sehr schmal⁸³, aber genau so auch in vielen Sprachen wie etwa Java zu finden. Die Schnittstelle kann aber auch von Anwendung zu Anwendung oder von Datenstruktur zu Datenstruktur variieren. Beispielsweise kann ein Iterator eine Methode zum Zurücksetzen auf das vorherige Element anbieten, wenn das bei der zugrunde liegenden Datenstruktur sinnvoll ist. Ein Iterator kann aber auch mit einer Methode `remove()` so erweitert werden, dass er das Objekt der Datenstruktur, bei dem er sich gerade befindet, löschen kann.

6.4.3.1 Klassendiagramm

Da die hier besprochenen **externen Iteratoren** alle die gleiche Basisschnittstelle haben, wird eine Schnittstelle `IIterator` eingeführt, die von allen konkreten Iteratoren implementiert werden muss. Im folgenden Klassendiagramm wird aus Gründen der Übersichtlichkeit nur ein einziger konkreter Iterator gezeigt:

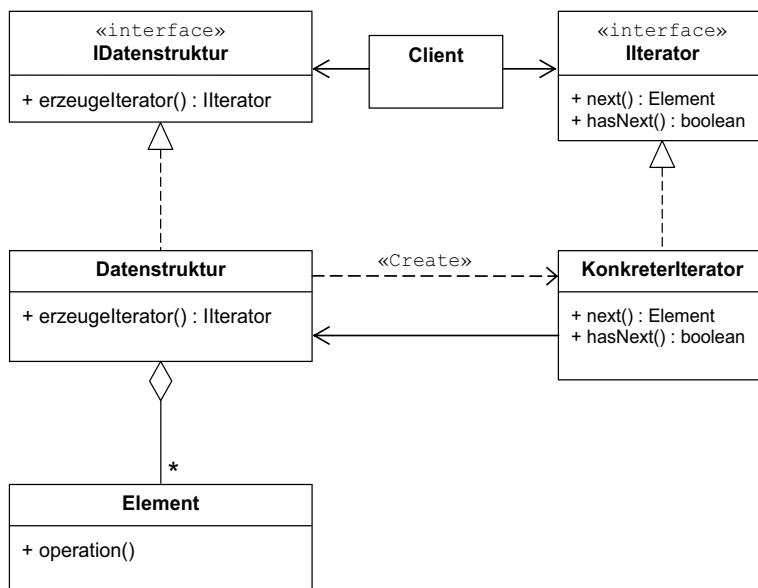


Bild 6-10 Klassendiagramm des Iterator-Musters

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 6-10 zu sehen ist, aggregiert die Klasse `Datenstruktur` eine Menge von Elementen beispielsweise in Form eines Arrays. Die Klasse `Element` steht hier stellvertretend auch für weitere Element-Klassen, die von der Klasse `Element`

⁸³ Theoretisch könnten die beiden Methoden der Schnittstelle noch zu einer einzigen verschmolzen werden.

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 die Schnittstelle `IIterator`. Der Client kann aber dadurch auf Grund des liskovschen Substitutionsprinzips jeden konkreten Iterator nutzen, dessen Klasse die Schnittstelle `IIterator` implementiert und deren Verträge einhält.

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 6-10 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.

Die Schnittstelle `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 7.1). Damit können unterschiedlich implementierte Datenstrukturen aus gleichen Element-Klassen von einem Client auch gleich bearbeitet werden.

Die Schnittstelle `IDatenstruktur` ist nicht durch das Iterator-Muster vorgeschrieben. Das Muster fordert nur, dass eine Datenstruktur eine Methode anbietet, die einen Iterator liefert, der diese Datenstruktur traversieren kann.



Die Einführung einer Fabrikmethode hat den Vorteil, dass konkrete Iteratoren dadurch dynamisch erzeugt werden können und ein Client sich nicht darum kümmern muss, welcher konkrete Iterator zu welcher Datenstruktur passt.

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.

6.4.3.2 Teilnehmer

Das Iterator-Muster hat die folgenden Teilnehmer:

- **Iterator**

Die Schnittstelle `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**

Die Schnittstelle `IDatenstruktur` definiert den Kopf der Methode `erzeugeIterator()` zum Erzeugen eines Iterators. Diese Schnittstelle ist vom Muster nicht vorgeschrrieben. Die Datenstruktur muss nur eine Methode zur Erzeugung des Iterators zur Verfügung stellen.

- **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.

6.4.3.3 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 vom Client 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:

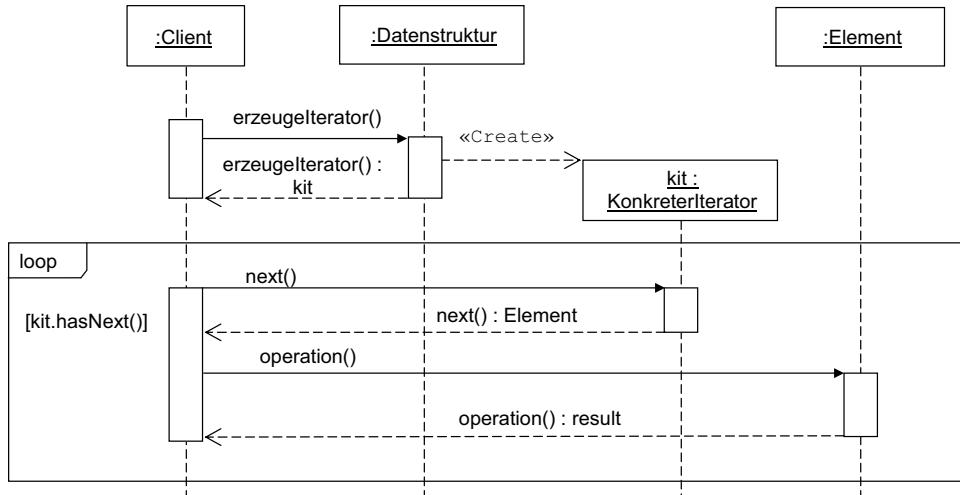


Bild 6-11 Sequenzdiagramm für das Iterator-Muster

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.

Dadurch, dass ein konkreter Iterator dynamisch in einer **Fabrikmethode** erzeugt wird und alle konkreten Iteratoren dieselbe Schnittstelle implementieren, ist die Formulierung einer Schleife im Client praktisch immer gleich. In [Gam15, S. 321] wurde dafür der Begriff **polymorphe Iteration** geprägt, weil trotz desselben Quellcodes ein passender Iterator dynamisch ausgewählt und benutzt wird.



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 der Iterator 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.

6.4.3.4 Programmbeispiel

In folgendem Programmbeispiel für einen externen Iterator 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, welche 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

import java.util.Locale;

public class Mitarbeiter {

    private static int mitarbeiterAnzahl;
    private final int personalNummer;
    private final String nachname;
    private final String vorname;
    private final String abteilung;
    private final double gehalt;

    public Mitarbeiter(String vorname, String nachname,
                       String abteilung, double gehalt) {
        this.personalNummer = ++mitarbeiterAnzahl;
        this.vorname = vorname;
        this.nachname = nachname;
        this.abteilung = abteilung;
        this.gehalt = gehalt;
    }

    // Die Methode toString() entspricht der im Klassendiagramm
    // als operation() bezeichneten Methode.
    @Override
    public String toString() {
        return String.format(Locale.GERMAN, "%s %s, %d, %s, %.2f",
                           vorname, nachname, personalNummer, abteilung, gehalt);
    }
}
```

Die Schnittstelle `IIterator` definiert die Schnittstelle für einen Iterator:

```
// Datei: IIterator.java

public interface IIterator {

    boolean hasNext();
```

```
Mitarbeiter next();
}
```

In der Klasse `MitarbeiterIterator` wird nun die Schnittstelle `IIterator` implementiert und die für die Datenstruktur `MitarbeiterArray` spezifische Funktionalität bereitgestellt:

```
// Datei: MitarbeiterIterator.java

public class MitarbeiterIterator implements IIterator {

    private final Mitarbeiter[] data;
    private int anzahl; // Anzahl der Mitarbeiter im Array
    private int position; // Cursor für den Iterator

    public MitarbeiterIterator(Mitarbeiter[] data, int anzahl) {
        this.data = data;
        this.anzahl = anzahl;
    }

    @Override
    public boolean hasNext() {
        return position < anzahl;
    }

    @Override
    public Mitarbeiter next() {
        return data[position++];
    }
}
```

Die Schnittstelle `IDatenstruktur` definiert die Schnittstelle für iterierbare Datenstrukturen:

```
// Datei: IDatenstruktur.java

// Definiert die Schnittstelle für Datenstrukturen, die mit einem
// Iterator durchlaufen werden sollen
public interface IDatenstruktur {

    IIterator createIterator();
}
```

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

public class MitarbeiterArray implements IDatenstruktur {
```

```
private static final int MAX = 10; // Größe des Arrays
private final Mitarbeiter[] data = new Mitarbeiter[MAX];
private int position; // Aktuelle Position im Array

// Gibt als Rückgabewert den für diese Datenstruktur
// spezifischen Iterator zurück
@Override
public IIIterator createIterator() {
    return new MitarbeiterIterator(data, position);
}

// Funktion zum Hinzufügen eines neuen Mitarbeiters
public void add(Mitarbeiter mitarbeiter) {
    if (position < MAX) {
        data[position++] = mitarbeiter;
    }
}
}
```

Zu Beginn der Klasse `TestIterator` wird mittels der Factory-Methode `erzeugeMitarbeiterTestDaten()` 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 `mitarbeiter` benötigt werden. Im Folgenden die Klasse `TestIterator`:

```
// Datei: TestIterator.java

public class TestIterator {

    // Mitarbeiterliste erzeugen und befüllen
    private static final MitarbeiterArray mitarbeiter =
        erzeugeMitarbeiterTestDaten();

    // Testdatensatz für die Mitarbeiterliste erzeugen
    private static final MitarbeiterArray erzeugeMitarbeiterTestDaten() {
        MitarbeiterArray mitarbeiter = new MitarbeiterArray();

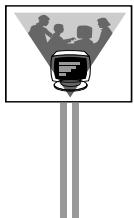
        mitarbeiter.add(new Mitarbeiter("Hermann", "Hinz",
            "MMI-Entwicklung", 3250f));
        mitarbeiter.add(new Mitarbeiter("Thomas", "Kunz",
            "MMI-Entwicklung", 3050f));
        mitarbeiter.add(new Mitarbeiter("Heinz", "Mueller",
            "Unit Tests", 3450f));
        mitarbeiter.add(new Mitarbeiter("Hans", "Maier",
            "Unit Tests", 3400f));
        mitarbeiter.add(new Mitarbeiter("Max", "Muster",
            "Unit Tests", 3500f));
        mitarbeiter.add(new Mitarbeiter("Peter", "Schmidt",
            "Requirements Engineering", 3700f));
    }
}
```

```

        return mitarbeiter;
    }

    public static void main(String[] args) {
        // Iterator erzeugen und durch die Datenstruktur laufen
        IIIterator allMitarbeiter = mitarbeiter.createIterator();
        while (allMitarbeiter.hasNext()) {
            // In der folgenden Zeile wird implizit die toString()-Methode
            // der Klasse Mitarbeiter aufgerufen. Dabei handelt es sich um
            // die im Klassendiagramm als operation() bezeichnete Methode.
            System.out.println(allMitarbeiter.next());
        }
    }
}

```



Hier das Protokoll des Programmlaufs:

```

Hermann Hinz, 1, MMI-Entwicklung, 3250,00
Thomas Kunz, 2, MMI-Entwicklung, 3050,00
Heinz Mueller, 3, Unit Tests, 3450,00
Hans Maier, 4, Unit Tests, 3400,00
Max Muster, 5, Unit Tests, 3500,00
Peter Schmidt, 6, Requirements Engineering, 3700,00

```

6.4.4 Bewertung

6.4.4.1 Vorteile

Folgende Vorteile ergeben sich durch die Verwendung des Musters:

- **Unabhängigkeit von der konkreten Datenstruktur**

Der Quellcode einer Anwendung zur Traversierung einer Datenstruktur ist unabhängig von der konkreten Datenstruktur. Man kann also auf unterschiedliche Datenstrukturen – vorausgesetzt sie enthalten Objekte der gleichen Element-Klassen – in einheitlicher Weise zugreifen und sie durchlaufen.

- **Aufbau der Datenstruktur nicht sichtbar**

Der Aufbau der Datenstruktur, welche die zu durchlaufenden Objekte enthält, wird für die Anwendung nicht sichtbar.

- **Vereinfachung der Schnittstelle der Datenstruktur**

Eine Datenstruktur muss keine Methoden für die verschiedenen Traversierungsarten zur Verfügung stellen. Nur die Fabrikmethode zur Erzeugung eines Iterators ist zusätzlich zu den Fachmethoden notwendig. Auch Element-Klassen benötigen keine speziellen Methoden für Iteratoren.

- **Iteration in eigenem Objekt**

Die Iteration erfolgt in einem eigenen Objekt, dem Iterator. Der Wechsel von einer Traversierungsart zu einer anderen vereinfacht sich zu einem Austausch von Iterator-Instanzen.

- **verschiedene Durchlaufsstrategien durch eine Datenstruktur möglich**

Die Datenstruktur kann je nach Iterator auf verschiedene Arten durchlaufen werden. So können beispielsweise Listen vorwärts oder rückwärts oder auch in einer sortierten Reihenfolge traversiert werden.

- **gleichzeitig mehrere Durchläufe durch eine Datenstruktur**

Mehrere Iteratoren können gleichzeitig über eine Datenstruktur laufen, da jeder Iterator den Zustand der Traversierung für sich selbst verwaltet. Dies gilt nur, falls kein Iterator die Datenstruktur beim Durchlauf verändert.

6.4.4.2 Nachteile

Durch die Verwendung des Musters können auch folgende Nachteile entstehen:

- **starke Kopplung zwischen Datenstruktur und Iterator**

Die konkreten Datenstrukturobjekte müssen ihren Iterator selber erzeugen. Datenstruktur und Iterator sind dadurch stark gekoppelt. Externe Iteratoren müssen die Implementierung der Datenstruktur kennen und durchbrechen dabei möglicherweise die Kapselung der Datenstruktur.⁸⁴

- **Änderungen an der Datenstruktur**

Externe Iteratoren sind potenziell anfällig gegen Änderungen, die durch nebenläufige Prozesse an der Datenstruktur während eines Durchlaufs durchgeführt werden: 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. Beispielsweise könnte das **Beobachter-Muster** eingesetzt werden, damit Iteratoren über die Veränderung einer Datenstruktur informiert werden.

6.4.5 Einsatzgebiete

Der Einsatz des Iterator-Musters ist immer dann sinnvoll, wenn ein einheitlicher Zugriff auf unterschiedliche Datenstrukturen aus gleichen Elementen 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. In diesem Falle werden häufig interne Iteratoren eingesetzt, die auf rekursiven Strukturen besser implementiert werden

⁸⁴ Wenn es die Programmiersprache erlaubt, können Iterator-Klassen in einer Datenstruktur-Klasse geschachtelt werden, um die Kapselung der Datenstruktur aufrechtzuerhalten.

können. Externe Iteratoren müssten einen komplexen Zustand – wie etwa den Pfad zu einem Knoten in einem Baum – speichern, um das nächste Element zu bestimmen. **Interne Iteratoren** können selbst rekursiv implementiert werden und können so durch eine Rückkehr aus einer Rekursion in den nächsten Zustand weitergehen.

Das **Besucher-Muster** kann geschickt mit dem Iterator-Muster kombiniert werden, damit ein Besucher eine Datenstruktur durchlaufen kann. Allerdings setzt das Iterator-Muster voraus, dass alle Element-Klassen der Datenstruktur von einer gemeinsamen Basisklasse abgeleitet sind.

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. Durch den Einsatz bestimmter Sprachmittel sind die Iteratoren in Java teilweise einfacher zu handhaben als die bisher beschriebenen Iteratoren des allgemeinen Musters.

Betrachtet man sich die Schnittstelle `IIterator` im Klassendiagramm in Bild 6-10 genauer, so stellt man fest, dass sie den Typ `Element` benutzt. Diese Schnittstelle muss also für jeden Element-Typ 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 die hier im Buch vorgestellte Schnittstelle `IIterator` hat und die Methoden `next()` und `hasNext()` definiert.

Will man das Interface `Iterator<E>` im vorhergehenden Programmbeispiel nutzen, muss die Klasse `MitarbeiterIterator` das Interface `Iterator<Mitarbeiter>` implementieren:

```
public class MitarbeiterIterator implements Iterator<Mitarbeiter>
```

Wenn eine Datenstruktur einen Iterator zur Verfügung stellt, der das Interface `Iterator<E>` implementiert, dann kann die Datenstruktur zu einer **iterierbaren Datenstruktur** erklärt werden. Dazu definiert Java 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 Programmbeispiel⁸⁵ die Klasse MitarbeiterArray mit dem Zusatz `implements Iterable<Mitarbeiter>` versehen und die Methode `createIterator()` umbenannt in `iterator()`, dann könnte die `main()`-Methode der Klasse TestIterator alternativ auch wie folgt formuliert werden:

```
public static void main(String[] args) {
    // In diesem Beispiel wird der Iterator implizit
    // erzeugt und in der foreach Schleife aufgerufen
    for (Mitarbeiter ma : mitarbeiter) {
        // In der folgenden Zeile wird implizit die toString()-Methode
        // der Klasse Mitarbeiter aufgerufen. Dabei handelt es sich um
        // die im Klassendiagramm als operation() bezeichnete Methode.
        System.out.println(ma);
    }
}
```

Hier ist nun die Nutzung eines Iterators komplett aus dem Quelltext verschwunden und der Compiler hat die Aufgabe übernommen, den Iterator aufzurufen. Sollen aber zusätzliche Funktionen des Iterators benutzt werden, wie etwa ein Element entfernen, dann muss die Nutzung eines Iterators weiterhin im Quellcode ausprogrammiert werden.

6.4.6 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, wohingegen ein Besucher während des Durchlaufs eine zusätzliche Funktionalität erbringt. 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.

Interne Iteratoren sind vergleichbar mit dem Muster **Enumeration Method** – beschrieben beispielsweise in [Bus07]. Eine Enumerationsmethode wird in einer Datenstruktur implementiert und ist verantwortlich für einen kompletten Durchlauf durch die Datenstruktur. Ggf. kann die Aktion, die beim Durchlauf auf den Elementen durchgeführt werden soll, als Argument an die Methode übergeben werden.

6.5 Das Verhaltensmuster Memento

6.5.1 Name / Alternativer Name

Memento⁸⁶, Token (keine deutsche Übersetzung gebräuchlich)

⁸⁵ Der vollständige Quellcode des modifizierten Beispiele ist auf dem begleitenden Webauftritt zu finden.

⁸⁶ Memento (engl.) ist ein Andenken – also ein Ding, um die Erinnerung an etwas aufrecht erhalten zu können.

6.5.2 Problem

Ein Anwender möchte die Möglichkeit haben, Aktionen rückgängig zu machen, die er versehentlich oder irrtümlich ausgeführt hat oder deren Resultate nicht seinen Erwartungen entsprechen. Dafür soll ihm in einer Anwendung die Möglichkeit gegeben werden, ausgeführte Aktionen wie beispielsweise das Hinzufügen, Ändern oder Löschen von Daten ohne großen Aufwand rückgängig zu machen, um so den vorherigen Zustand wiederherstellen zu können.

Ein typisches Anwendungsbeispiel für eine solche Funktionalität ist ein Textverarbeitungsprogramm, das es ermöglicht, eine bestimmte Anzahl von Aktionen wie beispielsweise eine Texteingabe innerhalb eines Dokumentes rückgängig zu machen.

Es sollen also die Zustände eines Objekts gespeichert werden können und das Objekt später in einen der gespeicherten Zustände zurückversetzt werden können.

Eine Lösung des Problems innerhalb des Objekts selbst wäre sehr ineffizient, da im Objekt eine Liste der Kopien seiner Zustände angelegt und verwaltet werden müsste, um das Objekt so in die Lage zu versetzen, seinen vorherigen Zustand wiederherzustellen. Dieser Ansatz würde viel Speicher benötigen und zudem das Objekt sehr groß, komplex und schwergewichtig machen.

Folglich sollte der Zustand eines Objektes außerhalb des Objektes selbst gespeichert und verwaltet werden. Dabei sollte aber auch berücksichtigt werden, dass ein direkter Zugriff auf den Zustand – also auf die Daten – eines Objektes von außen durch andere Objekte nicht möglich sein darf, da dadurch das Konzept der Datenkapselung verletzt und die Zuverlässigkeit sowie Erweiterbarkeit der Anwendung gefährdet würde.

Das Verhaltensmuster Memento soll es erlauben, den **internen Zustand eines Objekts zu erfassen und auszulagern**, so dass dieses Objekt zu einem späteren Zeitpunkt in einen dieser gespeicherten Zustände zurückversetzt werden kann. Dabei darf die Datenkapselung nicht verletzt werden: fremde Objekte sollen keinen Zugriff auf die Zustände eines Objekts haben.



6.5.3 Lösung

Memento ist ein **objektbasiertes Verhaltensmuster**.

Dasjenige **Objekt, dessen interner Zustand gespeichert werden soll**, wird als **Urheber** bezeichnet. Sein **Zustand** wird aus dem Objekt ausgelagert und in einem **Memento-Objekt⁸⁷** zwischengespeichert.



Die Anwendung weiß, wann der interne Zustand des Urhebers gespeichert oder wiederhergestellt werden muss.

⁸⁷ Memento ist sowohl der Name des Musters als auch der Name der zentralen Klasse dieses Musters.

Die Anwendung ist zuständig für die Verwaltung und Speicherung der Memento-Objekte und wird daher im Folgenden als Aufbewahrer bezeichnet.



Um eine Manipulation der Daten des Urhebers zu verhindern, erhält der Aufbewahrer keinen Zugriff auf den intern im Memento-Objekt gespeicherten Zustand des Urhebers.



Dadurch wird verhindert, dass der Inhalt des Memento-Objekts geändert werden kann und somit falsche Daten an den Urheber zurückgegeben werden.

Dadurch, dass die Verwaltung der Memento-Objekte durch den Aufbewahrer erfolgt, vereinfacht man den Urheber, der sich nicht mit der Speicherung beschäftigen soll.



Als Beispiel für die Nutzung des Entwurfsmusters Memento kann ein Text-Editor betrachtet werden, in welchem der Anwender Text abspeichern, rücksetzen oder wiederherstellen kann. Der Text wird als Variable in einem Objekt der Klasse `Urheber` gespeichert. Nach der Eingabe von neuem Text wird das Urheber-Objekt geändert und der neue Text darin abgespeichert. Die Anwendung möchte sich den vorherigen Zustand des Urhebers merken, um die Änderung ggf. rückgängig machen zu können. Um die Datenkapselung nicht zu verletzen, kann die Anwendung nicht selbst auf die Daten des Urhebers zugreifen. Stattdessen fordert die Anwendung ein Memento-Objekt von dem Urheber an. Der Urheber erzeugt daraufhin ein Objekt der Klasse `Memento`, initialisiert es mit seinem aktuellen inneren Zustand und gibt es an die Anwendung zurück. Wird durch den Anwender ein Rücksetzen oder Wiederholen der vorherigen Aktion angefordert, dann gibt die Anwendung das Memento-Objekt an den Urheber zurück. Basierend auf den Informationen im Memento-Objekt ändert der Urheber seine Variablen so, dass sie exakt ihren früheren Zustand widerspiegeln.

Je nachdem, wie viele und welche Memento-Objekte aufbewahrt werden, kann ein Anwender entweder nur die zuletzt ausgeführte Aktion rückgängig machen oder mehrere ausgeführte Aktionen auf einmal.

6.5.3.1 Klassendiagramm

Das Muster Memento definiert zwei Schnittstellen, die der Aufbewahrer nutzt: eine Schnittstelle zur Verwaltung von Mementos und eine Schnittstelle, um Mementos anzufordern bzw. zurückzugeben.

Das folgende Klassendiagramm zeigt die Struktur des Memento-Verhaltensmusters:

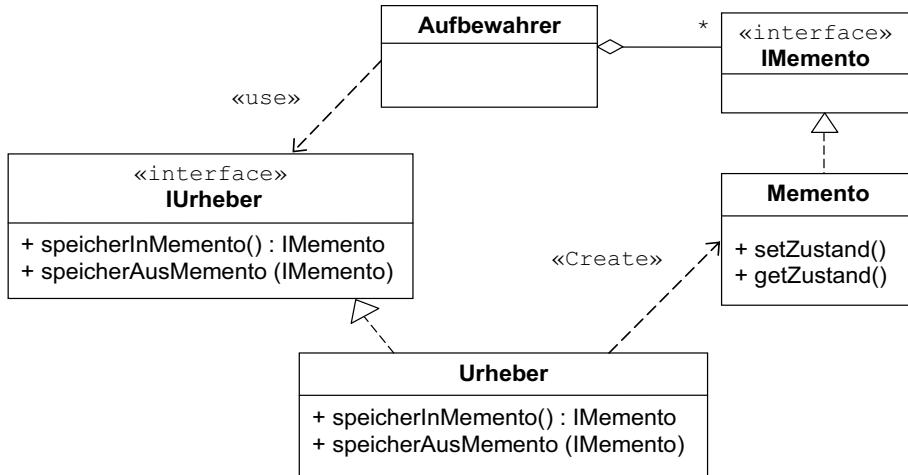


Bild 6-12 Klassendiagramm des Verhaltensmusters Memento

Die Klasse Urheber implementiert die Schnittstelle **IMemento** mit den beiden Methoden `speicherInMemento()` und `speicherAusMemento()`. Die Methode `speicherInMemento()` dient dazu, ein Objekt der Klasse **Memento** zu erzeugen und es dabei mit dem aktuellen Zustand des Urheber-Objekts zu initialisieren.

Der **Urheber** ist somit lediglich der **Erzeuger des Memento-Objekts**, die **Verwaltung** wird durch die Klasse **Aufbewahrer** übernommen.



Ein Aufbewahrer kann ein oder mehrere Memento-Objekte aggregieren. Er fordert bei Bedarf eine Momentaufnahme des inneren Zustandes beim Urheber an. Dabei bestimmt der Aufbewahrer, zu welchem Zeitpunkt ein Memento-Objekt angefordert wird.

Soll ein Objekt der Klasse Urheber in einen vom Aufbewahrer gespeicherten Zustand versetzt werden, beispielsweise um eine Aktion rückgängig zu machen, übergibt der Aufbewahrer das dem gewünschten vorherigen Zustand entsprechende Memento-Objekt an den Urheber über die Methode `speicherAusMemento()`. Der Urheber übernimmt vom Memento-Objekt die darin gespeicherten Daten und ändert seinen internen Zustand gemäß dieser Daten.

Die beiden Methoden `speicherInMemento()` und `speicherAusMemento()` haben als Parameter- bzw. Rückgabetyp das Interface **IMemento**. Diese Schnittstelle enthält keine Methoden. Eine Anwendung wie zum Beispiel der Aufbewahrer kann deswegen nicht auf die in einem Memento-Objekt gespeicherten Informationen zugreifen, solange sie nur dieses Interface nutzt.

Um die Kapselung der Daten eines Memento-Objektes abzusichern, darf der Aufbewahrer nicht die Klasse `Memento` nutzen. Der Aufbewahrer darf nur das Interface `IMemento` nutzen, um auf Memento-Objekte zuzugreifen – aber nicht die Klasse `Memento` selbst bzw. deren Methoden.



Die Klasse `Memento` implementiert die Schnittstelle `IMemento`, die vom Aufbewahrer zu benutzen ist. Gleichzeitig stellt sie aber auch Methoden zur Verfügung, mit deren Hilfe ein Urheber-Objekt seinen Zustand speichern und holen kann. Diese Methoden, in Bild 6-12 `setZustand()` und `getZustand()` genannt, dürfen nicht vom Aufbewahrer aufgerufen werden.⁸⁸

Ein Schutz der Klasse `Memento` vor dem Zugriff durch den Aufbewahrer ist je nach verwendeter Programmiersprache unterschiedlich – manchmal aber auch gar nicht möglich.



In C++ beispielsweise könnte die Klasse `Memento` die Klasse `Urheber` zu einer Freund-Klasse erklären, damit ein Urheber private Methoden der Klasse `Memento` nutzen oder sogar direkt auf den im Memento gespeicherten Zustand zugreifen kann. In Java kann man ggf. die Paket-Sichtbarkeit nutzen oder die Klasse `Memento` kann in der Klasse `Urheber` geschachtelt werden.

6.5.3.2 Teilnehmer

Im Folgenden werden die Teilnehmer des Memento-Musters beschrieben:

- **IMemento**

Die Schnittstelle `IMemento` ist leer. Nutzt man ein Objekt über diese Schnittstelle, kann man das Objekt nur speichern bzw. an andere Objekte weitergeben.

- **Memento**

Ein Memento-Objekt speichert den inneren Zustand eines Urheber-Objektes. Die Klasse `Memento` implementiert die leere Schnittstelle `IMemento`. Der Aufbewahrer nutzt ein Memento nur über diese Schnittstelle und kann das Memento-Objekt daher nur speichern bzw. an andere Objekte weitergeben. Der Urheber auf der anderen Seite benötigt Methoden, die es ihm erlauben, auf alle benötigten Daten zuzugreifen, um seinen vorherigen Zustand wiederherzustellen. Der Umfang der Informationen, die an das Memento-Objekt übergeben werden, muss vom Urheber nach Bedarf festgelegt werden. Wie sichergestellt werden kann, dass nur die Klasse `Urheber` auf die im Memento gespeicherten Daten zugreifen kann, ist – wie bereits in Kapitel 6.5.3.1 erwähnt – abhängig von der verwendeten Programmiersprache.

⁸⁸ In UML kann eine solche Beziehung zwischen den Klassen nicht ausgedrückt werden. Wenn die Methoden mit + markiert sind wie im Bild 6-12 bedeutet das, dass die Methoden öffentlich sind und somit von allen anderen Klassen benutzt werden können – also auch von der Klasse `Aufbewahrer`. Würden sie als privat mit - gekennzeichnet, dürfte keine Klasse – auch nicht die Klasse `Memento` – diese Methoden aufrufen.

- **IUrheber**

Die Schnittstelle `IUrheber` definiert die beiden Methoden `speicherInMemento()` und `speicherAusMemento()`. Eine Anwendung kann über diese Methoden ein Objekt auffordern, ein Memento zu erzeugen oder seinen Zustand auf den im Memento gespeicherten Zustand zu setzen.

- **Urheber**

Der Urheber erzeugt ein Memento-Objekt, um darin seinen gegenwärtigen internen Zustand abzuspeichern, und benutzt ein Memento-Objekt, um seinen vorherigen Zustand wiederherzustellen. Dazu implementiert er die im Interface `IUrheber` definierten Methoden. Ein Urheber stellt natürlich auch anwendungsspezifische Methoden zur Verfügung, mit deren Hilfe sein Zustand geändert und abgefragt werden kann. Diese sind für das Muster nicht von Interesse.

- **Aufbewahrer**

Der Aufbewahrer weiß, wann der interne Zustand des Urhebers in einem Memento gespeichert oder daraus wiederhergestellt werden muss. Der Aufbewahrer sieht nur über die leere Schnittstelle `IMemento` auf das Memento-Objekt und hat somit keinen Zugriff auf den Inhalt des Mementos. Er übernimmt die Aufbewahrung und Verwaltung der Memento-Objekte. Die Klasse `Aufbewahrer` steht hier stellvertretend für eine Anwendung, die eine Undo- und/oder Redo-Funktionalität in Bezug auf Urheber-Objekte zur Verfügung stellt.

6.5.3.3 Dynamisches Verhalten

Das nachfolgende Sequenzdiagramm zeigt das dynamische Verhalten der am Entwurfsmuster Memento beteiligten Objekte.

Im Sequenzdiagramm ist zu sehen, dass der Aufbewahrer die Interaktion initiiert und eine Momentaufnahme des aktuellen inneren Zustandes eines Urhebers anfordert, indem er dessen Methode `speicherInMemento()` aufruft. Die Methode `speicherInMemento()` erzeugt ein Memento-Objekt und initialisiert es mit dem aktuellen Zustand des Urhebers. Der Aufbewahrer erhält eine Referenz auf das Memento-Objekt, um dieses aufzubewahren und zu verwalten.

Entsteht in der Anwendung die Notwendigkeit, eine Zustandsänderung des Urheber-Objekts rückgängig zu machen, dann übergibt der Aufbewahrer das entsprechende Memento-Objekt, das den vorherigen Zustand speichert, an den Urheber, indem er dessen Methode `speicherAusMemento()` aufruft. Der Urheber fordert vom Memento-Objekt die darin gespeicherten Daten an und stellt damit seinen vorherigen internen Zustand wieder her. Das folgende Bild zeigt das Sequenzdiagramm des Verhaltensmusters Memento:

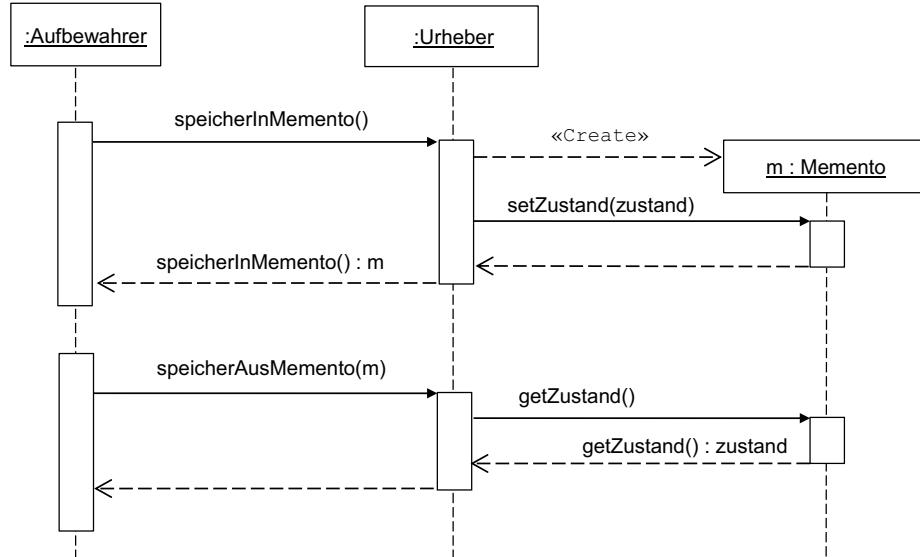


Bild 6-13 Sequenzdiagramm des Verhaltensmusters Memento

Dadurch, dass Sequenzdiagramme die Kommunikation zwischen Objekten darstellen, ist die Schnittstelle `IMemento` nicht im Bild 6-13 zu sehen. Sie sorgt für die Kapselung der Daten – also des Zustands (im Bild: `zustand`) eines Urheber-Objekts –, da auf Grund dieses Typs `IMemento` der Aufbewahrer keinen direkten Zugriff auf das Memento-Objekt erhält. Dadurch dass `IMemento` eine leere Schnittstelle ist, wird verhindert, dass der Aufbewahrer auf den Inhalt des Memento-Objektes zugreifen und ihn verfälschen kann.

6.5.3.4 Programmbeispiel

Als Beispiel für das Entwurfsmuster Memento soll ein einfacher Text-Editor dienen, der es dem Anwender ermöglicht, Text einzugeben, abzuspeichern und bei Bedarf die letzte Speicher-Aktion rückgängig zu machen beziehungsweise wiederherzustellen. Rücksetzen (Undo) bedeutet, dass eine Aktion wie beispielsweise das Hinzufügen von Text ohne großen Aufwand rückgängig gemacht werden kann. Wiederherstellen (Redo) bedeutet, dass eine vorherige Undo-Aktion wie das Entfernen von Text zurückgenommen wird. Macht ein Anwender zum Beispiel die Eingabe eines Wortes in einem Dokument rückgängig (Undo), ist dann aber mit dem Ergebnis unzufrieden, kann er den ursprünglichen Text über Redo wiederherstellen.

Das Hauptprogramm in der Klasse `TestMemento` erzeugt die abgebildete grafische Oberfläche mit einem großen Textfeld, in welchem Text eingegeben werden kann:

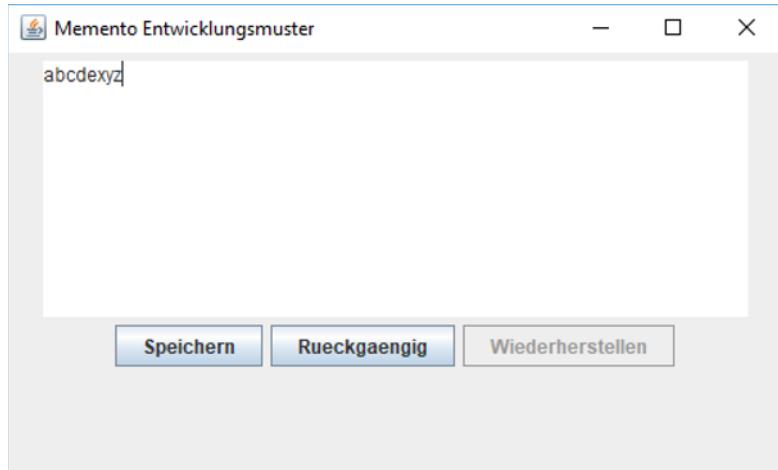
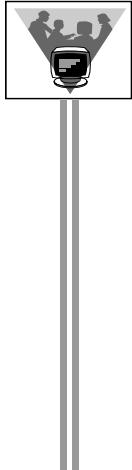


Bild 6-14 Graphische Oberfläche des Programmbeispiels

Über einen ActionListener werden die drei Schaltflächen überwacht und die zugehörigen Aktionen Speichern, Rueckgaengig und Wiederherstellen erfasst. Die Schaltflächen Rueckgaengig und Wiederherstellen sind nur aktiv, wenn bereits Text gespeichert wurde bzw. zurückgenommene Änderungen wiederhergestellt werden können. Zum Speichern des Textes enthält die Klasse ein Objekt der Klasse Urheber. Zum Implementieren der Aktionen Rueckgaengig und Wiederholen dient ein Objekt der Klasse Aufbewahrer.

Die Klasse TestMemento spielt für das Muster keine Rolle. Aus Platzgründen ist der Quellcode dieser Klasse nur auf dem begleitenden Webauftritt enthalten. Um das Zusammenspiel der Aktionen in Bezug auf den jeweils aktuell gespeicherten Text zu veranschaulichen, enthält die Klasse zusätzlich zu der grafischen Oberfläche auch eine normale Textausgabe. In einem Beispieldurchlauf des Programms wurde die folgende Ausgabe erzielt:



Hier das Protokoll des Programmlaufs:

```

SPEICHERN,      Textfeld enthaelt: abcde
SPEICHERN,      Textfeld enthaelt: abcdexyz
RUECKGAENGIG,   Textfeld enthaelt: abcde
WIEDERHERSTELLEN, Textfeld enthaelt: abcdexyz
SPEICHERN,      Textfeld enthaelt: abcdexyz12345
SPEICHERN,      Textfeld enthaelt: abcdexyz12345uvwxyz
RUECKGAENGIG,   Textfeld enthaelt: abcdexyz12345
RUECKGAENGIG,   Textfeld enthaelt: abcde
WIEDERHERSTELLEN, Textfeld enthaelt: abcdexyz
WIEDERHERSTELLEN, Textfeld enthaelt: abcdexyz12345
WIEDERHERSTELLEN, Textfeld enthaelt: abcdexyz12345uvwxyz
RUECKGAENGIG,   Textfeld enthaelt: abcdexyz12345
RUECKGAENGIG,   Textfeld enthaelt: abcdexyz

```

Die Klasse Urheber ist mit dem Textfeld der grafischen Oberfläche verknüpft, die von der Klasse TestMemento erzeugt und bearbeitet wird. In diesem Textfeld ist der aktuelle Text gespeichert, der auf der grafischen Oberfläche zu sehen ist und der dort auch geändert werden kann.

Wird die Schaltfläche Speichern gedrückt, fordert der Text-Editor das Urheber-Objekt mittels der Methode speicherInMemento() auf, den aktuellen Zustand in einem Memento-Objekt zu speichern. Wird die Aktion Rueckgaengig oder Wiederherstellen vom Anwender aufgerufen, verwendet der Text-Editor die Methode speicherAusMemento() und übergibt dem Urheber dabei ein entsprechendes Memento-Objekt. Der Urheber holt den abgespeicherten Text aus dem Memento-Objekt und stellt damit einen vorherigen Zustand wieder her. Diese beiden Methoden sind im Interface IUrheber definiert, welches von der Klasse Urheber implementiert wird. Der Rückgabetyp der Methode speicherInMemento() sowie der Typ des Parameters der Methode speicherAusMemento() ist das leere Interface IMemento.

Hier nun der Quellcode der beiden Interfaces:

```

// Datei: IMemento.java

public interface IMemento {

}

// Datei: IUrheber.java

public interface IUrheber {

    void speicherAusMemento(IMemento m);

    IMemento speicherInMemento();

}

```

Jetzt folgt die Klasse `Urheber`, welche die Schnittstelle `IUrheber` implementiert und intern die Methoden der Klasse `Memento` nutzt. Die Klasse `Urheber` ist – wie bereits erwähnt – mit dem Textfeld der grafischen Oberfläche verknüpft (`meinText`), in welchem der aktuelle Text gespeichert ist, der auf der grafischen Oberfläche zu sehen ist und der dort auch geändert werden kann.

Die Klasse `Memento` ist als private Klasse in der Klasse `Urheber` definiert. Diese Schachtelung erlaubt der Klasse `Urheber` den Zugriff auf die privaten Methoden der Klasse `Memento`. Dadurch dass die Klasse `Memento` als privat deklariert ist, kann außerhalb der Klasse `Urheber` niemand auf die Klasse `Memento` zugreifen.



Durch diese Konstruktion kann in Java die Kapselung der Daten der Memento-Objekte gewährleistet werden. Die Klasse `Urheber` und die darin geschachtelte Klasse `Memento` sind in der folgenden Java-Datei zu sehen:

```
// Datei: Urheber.java

import javax.swing.*;

public class Urheber implements IUrheber {

    private JTextArea meinText;

    public Urheber(JTextArea jta) {
        meinText = jta;
    }

    // Erzeugung des Memento-Objekts aus dem Textfeld
    public IMemento speicherInMemento() {
        Memento m = new Memento(meinText.getText());
        return m;
    }

    // Aus Memento-Objekt den Text für das Textfeld holen
    public void speicherAusMemento(IMemento m) {
        Memento memo = (Memento) m;
        meinText.setText(memo.gibText());
    }

    // Memento-Klasse nur nutzbar vom Urheber
    private static class Memento implements IMemento {

        private String merkText;

        private Memento(String t) {
            merkText = t;
        }

        private String gibText() {
            return merkText;
        }
    }
}
```

Wenn ein Objekt der Klasse `Memento` erzeugt wird, wird dem Konstruktor der aktuelle innere Zustand des Urheber-Objekts – also der im Textfeld gespeicherte Text – übergeben. Auf eine entsprechende `set`-Methode, wie sie im Klassendiagramm in Bild 6-12 zu sehen war, kann in diesem Fall verzichtet werden. Will das Urheber-Objekt seinen ursprünglichen Zustand wiederherstellen, erfolgt über die Methode `gibText()` die Rückgabe des gespeicherten Inhaltes an den Urheber, der damit das Textfeld entsprechend ändert.

Die Klasse `Aufbewahrer` verwaltet die einzelnen Memento-Objekte. Dazu gibt es zwei Array-Listen: die eine Liste enthält Mementos zum Rückgängigmachen (`undoMemos`) und die andere (`redoMemos`) speichert die Mementos, die rückgängig gemacht wurden, damit ihr Zustand ggf. wiederhergestellt werden kann.

Über die Methode `speicherNeuesMemento()` wird ein neues Memento-Objekt in der Undo-Liste abgelegt. Die Methode `gibUndoMemento()` entfernt das letzte Memento aus der Undo-Liste und gibt es zurück. Gleichzeitig wird es in die Redo-Liste eingefügt. Die Methode `gibRedoMemento()` entfernt das letzte Memento aus der Redo-Liste und gibt es zurück. Hier der Quellcode der Klasse `Aufbewahrer`:

```
// Datei: Aufbewahrer.java

import java.util.*;

public class Aufbewahrer {

    // Liste zur Verwaltung von Undo-Mementos
    private List<IMemento> undoMemos = new ArrayList<>();

    // Liste zur Verwaltung von Redo-Mementos
    private List<IMemento> redoMemos = new ArrayList<>();

    // Hinzufügen eines neuen Memento in die Undo-Liste
    public void speicherNeuesMemento(IMemento m) {
        undoMemos.add(m);
    }

    // Rückgabe des vorletzten Elementes aus der Undo-Liste
    public IMemento gibUndoMemento() {
        IMemento letztes = undoMemos.get(undoMemos.size() - 1);
        IMemento vorletztes = undoMemos.get(undoMemos.size() - 2);
        redoMemos.add(letztes);      // Letztes Memento in Redo-Liste
        undoMemos.remove(letztes);   // und aus Undo-Liste löschen
        return vorletztes;
    }

    // Rückgabe des letzten Memento aus der Redo-Liste
    public IMemento gibRedoMemento() {
        IMemento letztes = redoMemos.get(redoMemos.size() - 1);
        undoMemos.add(letztes);      // Letztes Memento in Undo-Liste
        redoMemos.remove(letztes);   // und aus Redo-Liste löschen
        return letztes;
    }
}
```

6.5.4 Bewertung

6.5.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Datenkapselung wird nicht verletzt**

Entscheidend für das Memento-Entwurfsmuster ist, dass das Prinzip der Datenkapselung erhalten bleibt. Durch den Einsatz ist es dem Urheber möglich, seinen inneren Zustand außerhalb abzuspeichern und trotzdem sicherzustellen, dass die Integrität seiner Daten unverletzt bleibt.

- **Aufteilung der Verantwortlichkeiten erleichtert die Implementierung des Urhebers**

Da der Aufbewahrer für die Verwaltung der Memento-Objekte zuständig ist, muss der Urheber nicht zusätzlich die Verwaltung der unterschiedlichen Versionen seines internen Zustandes übernehmen. Seine Implementierung wird nicht unnötig komplex.

6.5.4.2 Nachteile

Das Muster Memento kann aber auch Nachteile zur Folge haben:

- **eventuell werden große Datenmengen gespeichert**

Je nach Komplexität des Urhebers und je nachdem, wie viele Aktionen rückgängig gemacht werden können, können große Datenmengen entstehen. Dadurch kann die Leistung einer Anwendung negativ beeinflusst werden.

- **Kapselung der Memento-Daten ist nicht in jeder Programmiersprache möglich**

Damit die Kapselung der Daten im Memento gewährleistet ist, darf nur der Urheber auf sie direkt oder indirekt über Methoden zugreifen. Alle anderen Klassen bzw. Objekte – insbesondere der Aufbewahrer – dürfen aber nicht auf die internen Daten eines Memento-Objektes in irgendeiner Form zugreifen. Dazu bietet nicht jede Programmiersprache eine Unterstützung an.

- **Memento-Klasse und Urheber-Klasse sind stark gekoppelt**

Die Memento-Klasse ist stark abhängig von den Informationen, die den Zustand der Urheber-Objekte charakterisieren. Die Memento-Klasse kann daher kaum wiederverwendet werden und muss bei jeder Änderung der Urheber-Klasse betrachtet und ggf. ebenfalls geändert werden.

6.5.5 Einsatzgebiet

Das Entwurfsmuster Memento kann eingesetzt werden, wenn man die Wiederherstellung eines vorherigen Zustandes eines Objektes ermöglichen will.

Das Memento-Muster kann zudem in Verbindung mit anderen Entwurfsmustern eingesetzt werden. So kann es zusammen mit dem Entwurfsmuster **Iterator** verwendet werden, um den aktuellen Zustand einer Iteration zu speichern.

Das Muster Memento kann weiterhin mit dem Entwurfsmuster **Befehl** kombiniert werden. Dabei kann ein Memento verwendet werden, um den Zustand vor der Ausführung eines Befehls zu speichern, damit er ggf. rückgängig (Undo) gemacht werden kann. Falls gewünscht, kann ein Memento auch dazu dienen, den Zustand vor dem Rückgängigmachen eines Befehls zu speichern, damit dieser Zustand bei Bedarf einfach wiederhergestellt (Redo) werden kann. Eine solche Anwendung des Musters war im Programmbeispiel in Kapitel 6.5.3.4 gezeigt worden.

6.5.6 Ähnliche Entwurfsmuster

Ähnliche Entwurfsmuster sind den Autoren nicht bekannt.

6.6 Das Verhaltensmuster Rolle

6.6.1 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.

6.6.2 Problem

Objekte einer Software stehen in der Regel nicht alleine da. Sie wirken vielmehr mit anderen Objekten zusammen, indem sie dabei Rollen annehmen können. 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:

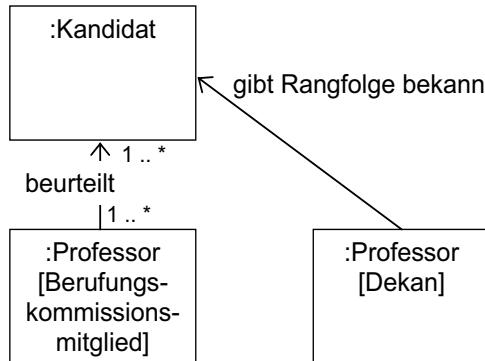


Bild 6-15 Rollen in einem Objektdiagramm

In diesem Beispiel können Objekte der Klasse `Professor` Mitglieder einer Berufungskommision sein. Jeder Professor als Mitglied einer Berufungskommision beurteilt die verschiedenen Kandidaten, die sich um eine neue Professorenstelle beworben haben. Hat die Berufungskommision ihre Entscheidung gefällt, gibt dasjenige Objekt der Klasse `Professor`, welches 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 Berufungskommision, 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 Berufungskommision. Darüber hinaus können die Rollen wechseln wie beispielsweise bei der Besetzung einer neuen Berufungskommision 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.



Weiterhin wird für die Entwicklung von Rollen gefordert:

Das Rollenmuster soll es ermöglichen, dass die Rollen dynamisch weiterentwickelt werden können und unabhängig voneinander sind, damit ihre Entwicklung entkoppelt ist.



6.6.3 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 um zusätzliche Rollen, die ein Objekt der Kernklasse spielen kann.

Zur Darstellung und zur Diskussion verschiedener Lösungen wird das folgende Beispiel verwendet: Die Klasse `Mitarbeiter` sei die Kernklasse, die Klassen `Verkauf`, `Entwicklung` und `Controlling` seien Rollenklassen und `maier`, `mueller` und `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. Eine solche Vorgehensweise widerspricht dem **Single Responsibility Principle** (siehe Kapitel 3.1.3), weil eine solche Klasse viele Verantwortlichkeiten in sich beinhaltet. 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 Programmiersprachen⁸⁹ 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 nach einem Rollenwechsel 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 6.10). Jedes Objekt der Klasse `Mitarbeiter` erhält – analog zu Bild 6-19 – 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. Somit haben die Rollenklassen viele Verantwortlichkeiten, was dem **Single Responsibility Principle** (siehe Kapitel 3.1.3) widerspricht.

Grundidee 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.

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 die konkrete Rolle `RolleX` spielt, wie das folgende Bild zeigt:

⁸⁹ Eine Ausnahme bilden typenlose Sprachen wie JavaScript.

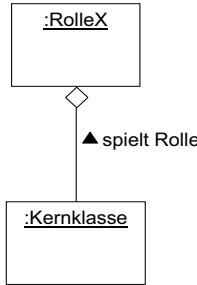


Bild 6-16 Objektdiagramm des Rollen-Musters

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 6.6.2 gestellten Anforderungen.

Die folgenden Bilder zeigen in Form von Objektdiagrammen die Lösung durch das Rollenmuster. Zunächst werden verschiedene Mitarbeiter in der gleichen Rolle gezeigt:

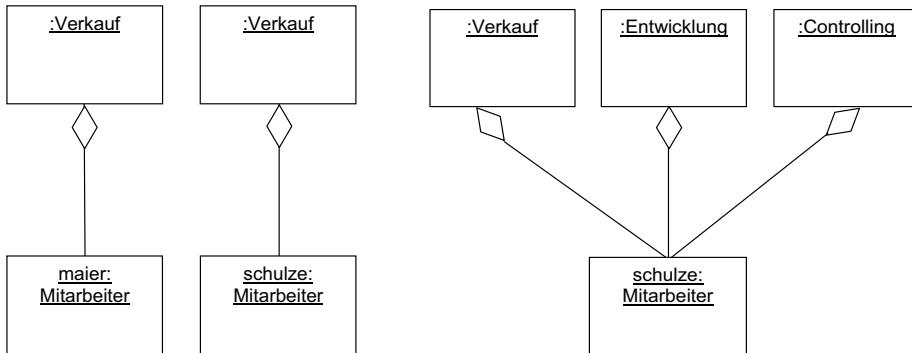


Bild 6-17 Mitarbeiter haben die gleiche Rolle

Bild 6-18 Ein Mitarbeiter hat mehrere Rollen

Das folgende Bild zeigt einen Mitarbeiter, der die Rolle wechselt:

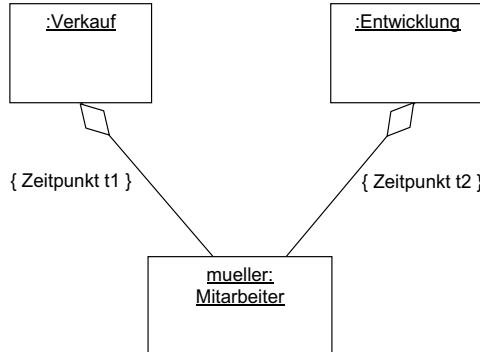


Bild 6-19 Ein Mitarbeiter hat zu verschiedenen Zeitpunkten t_1 und t_2 jeweils eine andere Rolle

Aus der bisherigen Lösungsbeschreibung wird klar, dass das Rollenmuster ein **objekt-basiertes Muster** ist.

Lösungsansatz des Rollenmusters

Eine Anwendung interessiert sich in der Regel sowohl für die Datenfelder und Methoden des Kernobjekts selber als auch für die Datenfelder und Methoden 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, welche das gerade aggregierte Kernobjekt zurückgibt. Damit kann eine Anwendung dann auch Methoden des Kernobjekts aufrufen.



Ebenso gibt es Anwendungen, die das Kernobjekt kennen und die an den Rollen dieses Objektes interessiert sind. Beispielsweise muss bei der Lohnabrechnung zu jedem Kernobjekt dessen Rollen bekannt sein, damit die unterschiedlichen Gehaltskomponenten berechnet werden können.

Die Kernklasse stellt Methoden zur Verfügung, um die Rollen zu verwalten, die ein Kernobjekt gerade spielt.



Diese Konstellation wird im folgenden Klassendiagramm schematisch gezeigt. Sie entspricht dem in [Gam97] beschriebenen Muster **Extension Object**.

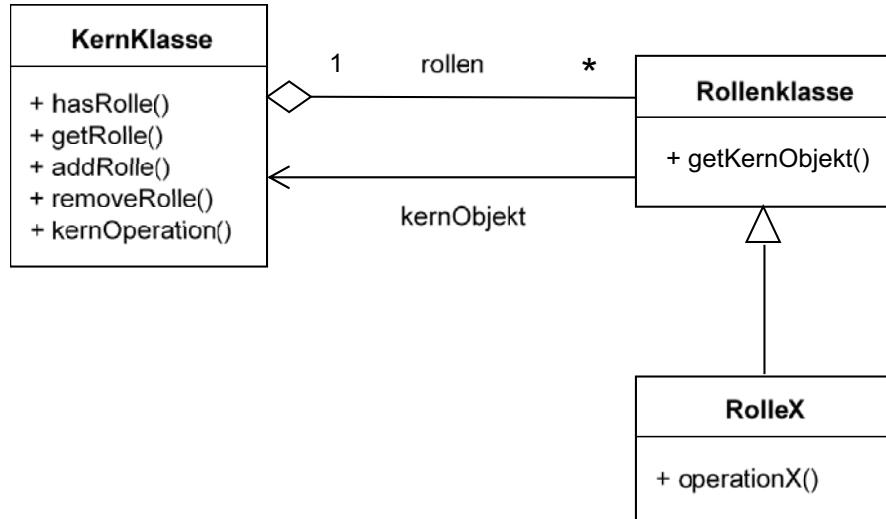


Bild 6-20 Klassendiagramm des Musters Extension Object

Diese Lösung hat den Nachteil, dass eine Anwendung unterscheiden muss, ob sie mit dem Kernobjekt interagiert oder mit einer seiner Rollen. Um diese Transparenz herzustellen, wird die Idee des Musters **Dekorierer** eingesetzt und eine gemeinsame Schnittstelle für das Kernobjekt und für die abstrakte Rollenklasse eingeführt. Im Unterschied zum Muster Dekorierer wird die Aggregation so definiert, dass ein Rollenobjekt nur ein Kernobjekt und nicht etwa ein anderes Rollenobjekt aggregieren kann. Dieser Lösungsansatz ist in [Bäu97] zu finden und wird dort als **Role Object Pattern** bezeichnet. Das im Folgenden beschriebene Rollenmuster basiert auf diesem Lösungsansatz.

6.6.3.1 Klassendiagramm

Die bereits erwähnte gemeinsame Schnittstelle für die Kernklasse und die Rollenklassen wird im Folgenden **Komponente** genannt (in Analogie zum Muster Dekorierer). Sie definiert die Methodenköpfe für die Verwaltung von Rollen und für den anwendungsspezifischen Teil der Kernklasse. In Bild 6-21 ist **Komponente** als abstrakte Klasse dargestellt – je nach Programmiersprache würde sich auch ein Interface anbieten. Von der Klasse **Komponente** ist die Kernklasse abgeleitet, welche die abstrakten Methoden implementiert, sowie die Basisklasse aller Rollenklassen, die Klasse **RollenKlasse**. Die Klasse **RollenKlasse** implementiert ebenfalls die Methoden, die in der Klasse **Komponente** definiert sind, gemäß dem Muster Dekorierer jedoch dadurch, dass die Methodenaufrufe an das gespeicherte Kernobjekt delegiert werden. Die Klasse **RollenKlasse** dient als Basisklasse für alle konkreten Rollenklassen und ist abstrakt, weil von ihr keine Objekte erzeugt werden sollen. Die konkreten Klassen der verschiedenen Rollen werden mit **RolleX** ($X = A \dots Z$) bezeichnet. Jede dieser Klassen enthält rollenspezifische Methoden, für die exemplarisch jeweils die Methode `operationX()` ($X = A \dots Z$) stehen soll. Bild 6-21 zeigt das Klassendiagramm des Rollenmusters:

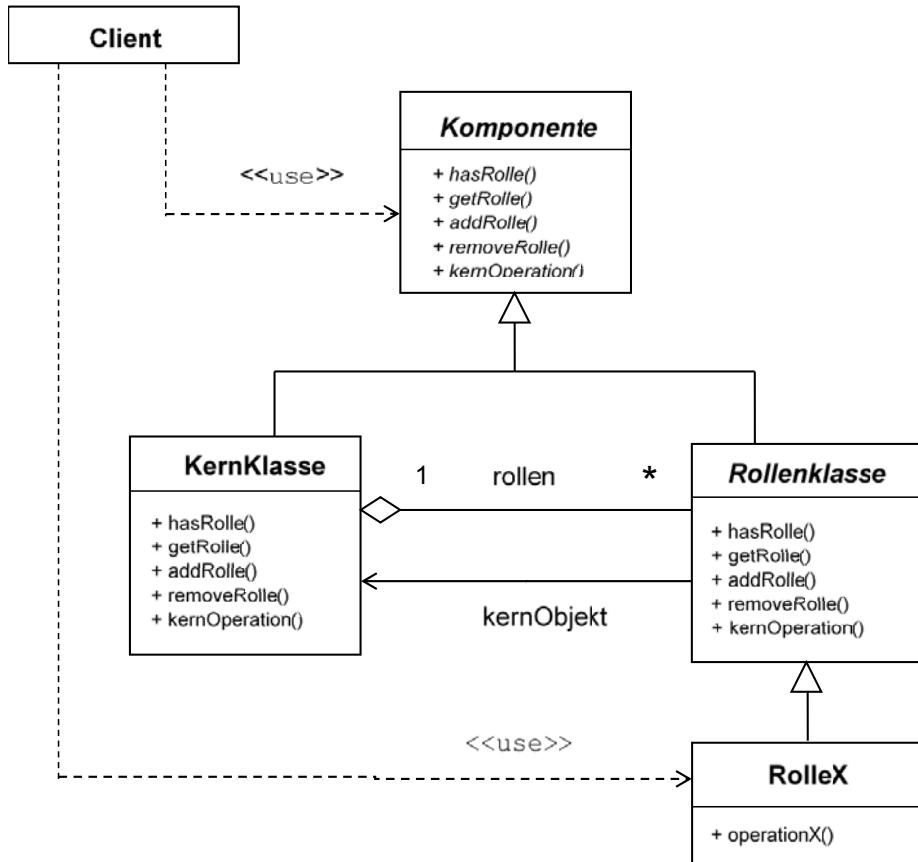


Bild 6-21 Klassendiagramm des Rollenmusters

Die Klasse **KernKlasse** repräsentiert den rollenunabhängigen Teil der Objekte. Die Klasse **KernKlasse** wird hier nicht weiter detailliert, weil es sich im Wesentlichen um eine anwendungspezifische Klasse handelt. Beispiele hierfür sind die bereits genannten Klassen **Mitarbeiter** und **Professor**. Objekte der Klasse **KernKlasse** können die Rollen spielen, für welche im Bild 6-21 stellvertretend die Klasse **RolleX** steht.

6.6.3.2 Teilnehmer

Das Verhaltensmuster Rolle umfasst die folgenden Teilnehmer:

- **Client**

Ein Client steht stellvertretend für eine Anwendung, die mit Objekten in verschiedenen Rollen interagiert.

- **Komponente**

Komponente ist eine abstrakte Klasse – oder ein Interface – und definiert die Methodenköpfe für die rollenunabhängigen Eigenschaften von Objekten. Ein Client kann diese Schnittstelle nutzen, um transparent sowohl auf ein Kernobjekt als auch auf ein Rollenobjekt zuzugreifen.

- **KernKlasse**

Die Klasse KernKlasse repräsentiert die rollenunabhängigen Eigenschaften von Objekten. Eine Instanz dieser Klasse ist ein Objekt, das die Rollen RolleX ($X = A..Z$) annehmen kann. Die Kernklasse implementiert die Methoden zur Verwaltung der Rollen, die ein Kernobjekt gerade spielt. Spielt ein Kernobjekt eine bestimmte Rolle, dann wird es andererseits von dem entsprechenden Rollenobjekt referenziert.

- **RollenKlasse**

Die abstrakte Klasse RollenKlasse implementiert die für alle konkreten Rollenklassen RolleX ($X = A..Z$) gemeinsamen Eigenschaften. Die Klasse RollenKlasse referenziert dasjenige Kernobjekt, welches eine Rolle gerade spielt. Mit Hilfe dieser Referenz delegiert ein Rollenobjekt die Aufrufe der rollenunabhängigen Methoden an das Kernobjekt.

- **RolleX**

Eine Klasse RolleX ($X = A..Z$) beschreibt eine Rolle, die von einem Kernobjekt gespielt werden kann. Stellvertretend für die spezifischen Eigenschaften einer Rolle steht die Methode operationX() ($X = A..Z$).

6.6.3.3 Dynamisches Verhalten

Das folgende Sequenzdiagramm zeigt, wie eine Anwendung – hier durch ein Objekt der Klasse Client repräsentiert – sowohl spezifische Rollenoperationen ausführen als auch rollenunabhängige Kernfunktionen bei einem Rollenobjekt aufrufen kann:

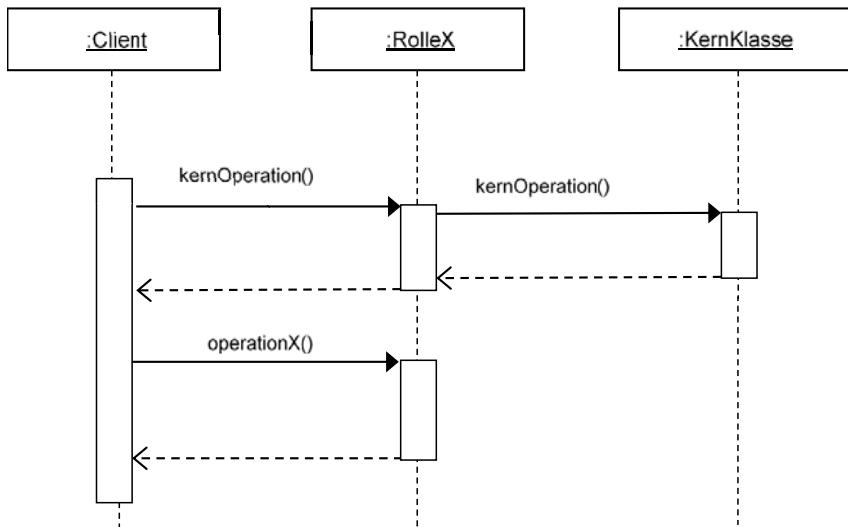


Bild 6-22 Sequenzdiagramm: Client und Rollenobjekt

Dem Client ist nur das Rollenobjekt bekannt, außerdem kennt der Client die Schnittstelle Komponente, welche die Operationen der Kernklasse definiert. Diese Schnittstelle kann in einem Sequenzdiagramm, das die Kommunikation auf Objektebene zeigt, nicht dargestellt werden. Der Client kann rollenspezifische Operationen wie beispielsweise `operationX()` aufrufen. Diese Aufrufe werden von der Instanz der Klasse `RolleX` selbst beantwortet. Der Client kann aber auch Kernfunktionen aufrufen, wie etwa die Methode `kernOperation()`. Solche Aufrufe nimmt ebenfalls das Rollenobjekt entgegen und delegiert dann aber diese Aufrufe an das referenzierte Kernobjekt.

Das nachfolgende Sequenzdiagramm zeigt die umgekehrte Situation, nämlich wie ein Client mit einem Kernobjekt interagiert und dabei auch dessen Rollenoperationen ausführen kann:

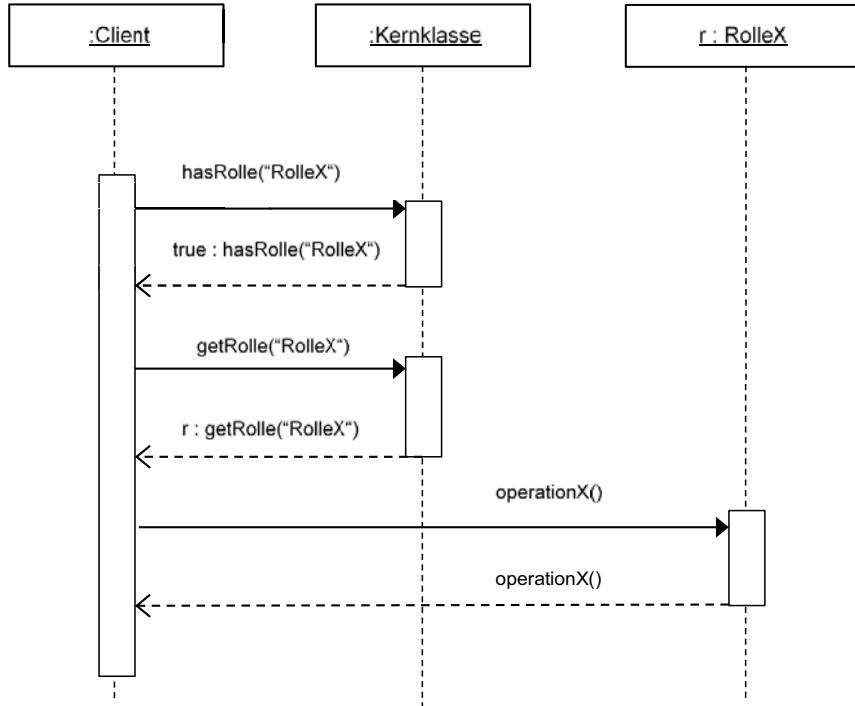


Bild 6-23 Sequenzdiagramm: Client und Kernobjekt

Das Muster legt nicht fest, wie eine bestimmte Rolle spezifiziert wird, wenn Rollenoperationen aufgerufen werden. Wie in Bild 6-23 zu sehen ist, wird hier der Name der Rollenklasse als Parameter für die Rollenoperationen verwendet. Diese Implementierungsentscheidung wird am Ende von Kapitel 6.6.3.4 diskutiert.

6.6.3.4 Programmbeispiel

Das Programmbeispiel greift das bisher schon benutzte Szenario auf und realisiert einen kleinen Ausschnitt einer Unternehmensverwaltung. Die Klasse `Mitarbeiter` ist in diesem Beispiel die Kernklasse. 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 in diesem Beispiel Rollenklassen dar.

Die Kernklasse hat im Rahmen des Entwurfsmusters Rolle nur eine untergeordnete Bedeutung. Die Klasse `Mitarbeiter` ist daher im Beispiel einfach und übersichtlich gehalten: sie speichert nur den Namen eines Mitarbeiters. Die Methode `getName()` spielt somit die Rolle der Methode `kernOperation()` in der allgemeinen Beschreibung des Musters. Zusammen mit den Operationen für die Verwaltung der Rollen ergeben sich somit die Methodenköpfe, die in der abstrakten Klasse `MitarbeiterKomponente` definiert werden:

```
// Datei: MitarbeiterKomponente.java

public abstract class MitarbeiterKomponente {

    // Rollenverwaltung:
    public abstract void addRolle(MitarbeiterRolle r);
    public abstract boolean hasRolle(String rollenName);
    public abstract MitarbeiterRolle getRolle(String rollenName);
    public abstract void removeRolle(String rollenName);
    // Kernmethode:
    public abstract String getName();

}
```

Es folgt nun der Quellcode der Klasse Mitarbeiter:

```
// Datei: Mitarbeiter.java

public class Mitarbeiter extends MitarbeiterKomponente {

    private final String name;

    public Mitarbeiter(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }

    // Implementierung der Rollenverwaltung:
    private java.util.HashMap<String, MitarbeiterRolle> rollen =
        new java.util.HashMap<String, MitarbeiterRolle>();

    @Override
    public void addRolle(MitarbeiterRolle r) {
        r.setMitarbeiter(this);
        String rollenName = r.getClass().getName();
        rollen.put(rollenName, r);
    }
    @Override
    public boolean hasRolle(String rollenName) {
        return rollen.containsKey(rollenName);
    }
    @Override
    public MitarbeiterRolle getRolle(String rollenName) {
        return rollen.get(rollenName);
    }
    @Override
    public void removeRolle(String rollenName) {
        rollen.remove(rollenName);
    }
}
```

Zur Verwaltung der Rollen eines Mitarbeiters wird in der Klasse Mitarbeiter eine Hashmap benutzt. Als Schlüssel fungiert der Name der Rollenklasse, welchem durch

die Hashmap ein entsprechendes Rollenobjekt zugeordnet wird. In der Methode `addRolle()` ist diese Zuordnung zu sehen: Zuerst wird der Name der Klasse des übergebenen Rollenobjekts geholt und dann das Paar Rollenname und Rollenobjekt in die Hashmap eingetragen.

In den Klassen `Mitarbeiterkomponente` und `Mitarbeiter` war bereits der Name `MitarbeiterRolle` zu sehen. Bei dieser Klasse handelt sich um die abstrakte Basisklasse aller konkreten Rollen, die Mitarbeiter spielen können. Sie enthält die vom Muster vorgegebene Referenz auf einen Mitarbeiter und die Delegation der Methoden der Rollenverwaltung an das referenzierte Kernobjekt der Klasse `Mitarbeiter`:

```
// Datei: MitarbeiterRolle.java

public abstract class MitarbeiterRolle extends MitarbeiterKomponente {

    // Kernobjekt:
    protected Mitarbeiter kernObjekt;
    protected void setMitarbeiter(Mitarbeiter m) {
        kernObjekt = m;
    }
    @Override
    public String getName() {
        return kernObjekt.getName();
    }

    // Rollenverwaltung:
    @Override
    public void addRolle(MitarbeiterRolle r) {
        kernObjekt.addRolle(r);
    }
    @Override
    public boolean hasRolle(String rollenName) {
        return kernObjekt.hasRolle(rollenName);
    }
    @Override
    public MitarbeiterRolle getRolle(String rollenName) {
        return kernObjekt.getRolle(rollenName);
    }
    @Override
    public void removeRolle(String rollenName) {
        kernObjekt.removeRolle(rollenName);
    }
}
```

Die erste konkrete Rollenklasse ist die Klasse `Entwicklung`. Sie enthält 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 extends MitarbeiterRolle {
```

```

private String projekt;

public void setProjekt(String projekt) {
    this.projekt = projekt;
}
public String getProjekt() {
    return projekt;
}
@Override
public String toString() {
    return String.format("%s arbeitet momentan an %s",
        kernObjekt.getName(), projekt);
}
}

```

Es folgt nun als zweite konkrete 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:

```

// Datei: Verkauf.java

public class Verkauf extends MitarbeiterRolle {

    private int umsatz;

    public void setUmsatz(int umsatz) {
        this.umsatz = umsatz;
    }
    public int getUmsatz() {
        return umsatz;
    }
    @Override
    public String toString() {
        return String.format("Aktueller Umsatz von %s: %d Euro.",
            kernObjekt.getName(), umsatz);
    }
}

```

Das folgende Hauptprogramm demonstriert nun das Rollenmuster. In der Klasse `TestRolle` wird in Form von statischen Variablen die Organisation einer Firma modelliert: Es gibt zwei Mitarbeiter mit den Namen `maier` und `schulze` und in der Entwicklungsabteilung gibt es zwei Rollen ebenso wie in der Verkaufsabteilung. In der `main()`-Methode wird beiden Kernobjekten 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 ausgegeben. Im weiteren Verlauf des Hauptprogramms sind dann zwei unterschiedliche Sichten auf die Firma zu sehen: zum einen die Kundensicht und zum anderen die Firmensicht. Wenn ein Kunde in den Verkauf kommt, wendet er sich an ein Rollenobjekt der Verkaufsabteilung. Über dieses Rollenobjekt kann er auch Kernklassen-spezifische Methoden aufrufen, hier die Methode `getName()`. Aus Firmensicht ist beispielsweise die Gehaltsabrechnung mitarbeiterbasiert. Das Rollenmuster ermöglicht es, dass bei der Gehaltsabrechnung über ein Mitarbeiterobjekt alle seine Rollen abgefragt werden können und dadurch wiederum die rollenspezifischen Methoden aufgerufen werden

können, um so die Gehaltskomponenten der einzelnen Rollen eines Mitarbeiters zu addieren. Die Klasse TestRolle enthält das beschriebene Hauptprogramm als main()-Methode:

```
// Datei: TestRolle.java

public class TestRolle {

    // Kernobjekte werden erzeugt:
    static Mitarbeiter maier = new Mitarbeiter("Maier");
    static Mitarbeiter schulze = new Mitarbeiter("Schulze");

    // Rollenobjekte der Abteilung Entwicklung:
    static Entwicklung entwicklung1 = new Entwicklung();
    static Entwicklung entwicklung2 = new Entwicklung();

    // Rollenobjekte der Abteilung Verkauf:
    static Verkauf verkauf1 = new Verkauf();
    static Verkauf verkauf2 = new Verkauf();

    public static void main(String[] args) {

        // Zuordnung der Rollen zu den Kernobjekten:
        entwicklung1.setProjekt("Produkt 2.0");
        entwicklung2.setProjekt("Produkt Addon 1.0");
        schulze.addRolle(entwicklung1);
        maier.addRolle(entwicklung2);

        // Ausgabe der Projekte:
        System.out.printf("Aktuelle Projekte der Mitarbeiter:"
            + "%n%s%n%s%n", entwicklung1, entwicklung2);

        // Ein Kernobjekt spielt eine weitere Rolle:
        System.out.println();
        System.out.println("Schulze erhält"
            + " zusätzliche Aufgaben im Verkauf.");
        schulze.addRolle(verkauf1);
        verkauf1.setUmsatz(15000);

        // Ausgabe der Umsatzzahlen:
        System.out.println();
        System.out.printf("Nur Schulze ist in der"
            + " Abteilung Verkauf:%n%s%n", verkauf1);

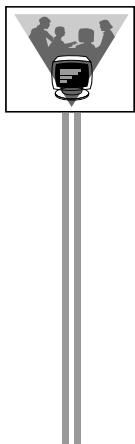
        // Ein Kunde kennt nur die Rolle:
        System.out.println();
        System.out.printf("Ein Kunde kommt in den Verkauf:\n");
        System.out.printf("Guten Tag, Herr %s.\n", verkauf1.getName());

        // Die Gehaltsabteilung kennt nur Mitarbeiter:
        System.out.println();
        System.out.printf("Gehaltsberechnung für %s:\n",
            schulze.getName());
    }
}
```

```

int gehalt = 0;
if (schulze.hasRolle("Entwicklung"))
    // Grundgehalt für Entwickler:
    gehalt = gehalt + 1000;
if (schulze.hasRolle("Verkauf")) {
    Verkauf rolle = (Verkauf) schulze.getRolle("Verkauf");
    // 3 % Provision vom Umsatz für Verkäufer:
    gehalt = gehalt + rolle.getUmsatz() * 3/100;
}
System.out.printf("Gehalt von %s ist %d Euro.\n",
    schulze.getName(), gehalt);
}
}

```



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 erhält zusätzliche Aufgaben im Verkauf.

Nur Schulze ist in der Abteilung Verkauf:
 Aktueller Umsatz von Schulze: 15000 Euro.

Ein Kunde kommt in den Verkauf:
 Guten Tag, Herr Schulze.

Gehaltsberechnung für Schulze:
 Gehalt von Schulze ist 1450 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. Weiterhin wird gezeigt, dass ein Client – hier das Hauptprogramm – sowohl über ein Rollenobjekt auf Kernaufgaben als auch über ein Kernobjekt auf dessen rollenspezifischen Funktionen zugreifen kann.

Implementierungsalternativen

Bei der hier gezeigten Implementierung des Rollenmusters wurden einige Entscheidungen getroffen, die durch das Rollenmuster nicht festgelegt sind, sondern je nach Anwendung unterschiedlich ausgeprägt sein können:

- Wie werden Rollen spezifiziert?

Im vorliegenden Beispiel wurden Rollen durch Strings (`rolleName`) spezifiziert, die den Klassennamen der Rollen entsprechen müssen. Hierdurch können leicht Fehler-situationen entstehen, wenn beispielsweise Klassennamen nachträglich geändert werden. Alternativ kann man etwa eine Enum-Klasse einsetzen – eine statische Lösung –

oder die Anwendung verwaltet eine Map, die einen Rollennamen auf einen Klassennamen abbildet. In Java beispielsweise könnte man auch mit den `Class`-Objekten der Rollenklassen arbeiten.

- Wer erzeugt die Rollenobjekte?

Die Idee bei dem Beispiel war, dass es in einer Firma klar ist, welche und wie viele Rollen es gibt. Daher wurden die Rollenobjekte statisch erzeugt – ohne Bezug zu einem Kernobjekt. Dies kann zu Fehlersituationen führen, wenn beispielsweise eine Rolle (im Beispiel der Klasse `TestRolle` die Rolle `verkauf2`) noch unbesetzt ist. Um solche Fehler abzufangen, müsste das Beispiel entsprechend erweitert werden, was aus Gründen der Übersichtlichkeit unterlassen wurde. Alternativ könnte beispielsweise in der Methode `addRolle()` der Klasse `Mitarbeiter` das Rollenobjekt für den Mitarbeiter erzeugt und mit dem Mitarbeiter als Kernobjekt initialisiert werden. Dies erfordert natürlich eine andere Parametrisierung der Methode: wie oben bereits erwähnt, müsste die Rolle spezifiziert werden und die Methode müsste auf die zu instanzierende Klasse schließen können.

- Wann werden die Rollenobjekte gelöscht?

Im Programmbeispiel sind die Rollenobjekte als statische Klassenvariablen angelegt. Folglich bleiben sie bis zum Ende des Programms erhalten. Werden Rollenobjekte jedoch dynamisch erzeugt, muss die Frage geklärt werden, wann sie nicht mehr benötigt werden und – wenn kein Garbage Collector existiert – wer wann die Objekte löschen (bzw. freigeben) darf. Ggf. muss auch überlegt werden, ob die Objekte wiederverwendet werden können.

6.6.4 Bewertung

6.6.4.1 Vorteile

Folgende Vorteile werden gesehen:

- **dynamisches Einnehmen von Rollen**

Rollen können zur Laufzeit dynamisch angenommen und abgelegt werden.

- **mehrere Rollen gleichzeitig möglich**

Ein Objekt kann mehrere Rollen gleichzeitig spielen.

- **unabhängige Entwicklung**

Rollenklassen können unabhängig voneinander und unabhängig von der Kernklasse entwickelt werden.

- **neue Rollen leicht möglich**

Neue Rollenklassen können sehr leicht zu einer Kernklasse hinzugefügt werden, ohne dass die Kernklasse geändert werden muss.

- **schmale Schnittstellen**

Die Schnittstellen der Kernklasse und der Rollenklasse können genau für ihre jeweiligen Aufgaben entworfen werden (**Single Responsibility Principle**) und enthalten daher keinen zusätzlichen Ballast.

- **Entkopplung von Anwendungen**

Eine Anwendung, welche eine bestimmte Rollenklasse nutzt, um mit einem Kernobjekt zu interagieren, ist unabhängig von einer anderen Anwendung, die über eine andere Rollenklasse auf das Kernobjekt zugreift.

- **rekursive Anwendung des Musters möglich**

Durch rekursive Anwendung des Rollenmusters ist es möglich, hierarchische Abhängigkeiten zwischen Rollen zu implementieren⁹⁰. Um ein Beispiel zu geben, wird das bisher benutzte Szenario – Verkäufer und Entwickler sind die bekannten Rollen von Mitarbeitern – erweitert: Backoffice und Frontoffice seien Rollen von Verkäufern, d.h., die Rolle Verkauf ist Voraussetzung für das Annehmen der Rolle Backoffice.

6.6.4.2 Nachteile

Folgende Nachteile werden gesehen:

- **benötigter Aufwand zur Ermittlung der Rolle(n) eines Kernobjekts**

Es ist etwas aufwendig, wenn man feststellen möchte, welche Rolle(n) ein Kernobjekt einnimmt: man muss alle bekannten Rollennamen überprüfen, ob sie auf das Kernobjekt zutreffen.

- **Verwaltung vieler Objekte erforderlich**

Es werden deutlich mehr Objekte erzeugt und diese müssen auch verwaltet werden.

- **Abhängigkeiten zwischen Rollenklassen schlecht überprüfbar**

Abhängigkeiten zwischen Rollenklassen – beispielsweise wenn ein Kernobjekt eine bestimmte Rolle nur annehmen darf, wenn es bereits eine bestimmte andere Rolle spielt – können nicht vom Typ-System der Programmiersprachen überprüft werden, sondern müssen codiert und zur Laufzeit überprüft werden.

- **zusätzlicher Implementierungsaufwand für Objektidentität**

Durch das Rollenmuster wurde ein Objekt aufgetrennt in Kernobjekt und Rollenobjekte. Eigentlich bilden sie aber eine konzeptionelle Einheit, deren Identität jedoch

⁹⁰ Siehe hierzu auch [Bäu97].

nicht einfach überprüfbar ist. Kernklasse und Rollenklassen müssten um Methoden erweitert werden, welche die Identität der konzeptionellen Einheit auf Basis der Objektidentität auf Programmiersprachenebene realisieren.

6.6.5 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 Principle**. Das Single Responsibility Principle 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 Analyse 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 Analyse identifizierte Rolle in einen Entwurf umgesetzt werden kann. Das hier vorgestellte Entwurfsmuster Rolle ist eine dieser Alternativen.

6.6.6 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. 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 wie beim Rollenmuster gleichzeitig einnehmen.

Das Entwurfsmuster **Extension Object** nach [Gam97] kann ebenfalls zur Implementierung von Rollen eingesetzt werden. Im Gegensatz zum hier beschriebenen Rollenmuster haben Clients beim Muster Extension Object keinen transparenten Zugriff auf die Objekte – also auf Kern- und Rollenobjekte.

Das Rollenmuster ist von der Struktur her ähnlich zum Entwurfsmuster **Dekorierer**. Während ein Dekorierer auch andere Dekorierer „dekorieren“ kann, sind beim Rollenmuster die Assoziationen zwischen den Klassen so eingeschränkt, dass ein Rollenobjekt kein anderes Rollenobjekt referenzieren kann.

Wenn Kernobjekte ihre Rollen nicht dynamisch ändern, kann das Entwurfsmuster **Adapter** eingesetzt werden. Die Implementierung des Adaptermusters ist einfacher als die des Rollenmusters.

In [Cra02] ist das Entwurfsmuster **Facet** veröffentlicht. Eine Facet-Klasse ist ähnlich einer Rollenklasse. Eine Facet-Klasse interagiert jedoch nicht mit einem Kernobjekt, sondern mit einer Art Fassadenobjekt (siehe Kapitel 5.4). Dieses Fassadenobjekt delegiert die Methodenaufrufe dann weiter an potentiell mehrere Kernobjekte, die für die Ausführung der Methodenaufrufe verantwortlich sind. Die Fassadenklasse enthält somit sämtliche Methoden, die alle Facet-Klassen zusammen genommen den Clients zur Verfügung stellen. Dadurch ist es bei dem Muster Facet schlecht möglich, neue Rollen hinzuzufügen.

6.7 Das Verhaltensmuster Schablonenmethode

6.7.1 Name/Alternative Namen

Schablonenmethode (engl. template method).

6.7.2 Problem

Die Struktur eines Algorithmus soll in der Basisklasse festgelegt werden, Einzelheiten in einer Unterklassie.

Das Muster **Schablonenmethode** soll bereits in einer Basisklasse die Struktur eines Algorithmus festlegen. Bestimmte Teile des Algorithmus sollen in der Basisklasse nur deklariert werden. Realisiert werden sollen diese deklarierten Teile in den Unterklassen.



6.7.3 Lösung

Das Muster Schablonenmethode wird für Algorithmen eingesetzt, die in Einzeloperationen zerlegt werden können.

Die Struktur eines Algorithmus für die **Schablonenmethode** aus

- abstrakten Einzeloperationen und
- eventuell konkreten Einzeloperationen



wird in der Basisklasse der Schablonenmethode vorgegeben. Dabei werden bestimmte Einzeloperationen als **Einschubmethoden** (engl. **hooks**) in abstrakter Form in der Schablonenmethode eingeführt⁹¹.

⁹¹ In der Regel ist eine Einschubmethode abstrakt. Es gibt aber auch Varianten des Musters, bei denen Einschubmethoden nicht abstrakt sind, sondern mit einem Rumpf, der eine Standardimplementierung enthält, definiert werden. Diese Einschubmethoden werden dann in den Unterklassen überschrieben.

Eine **Schablonenmethode** enthält also das Grundgerüst eines Algorithmus mit der Zerlegung des Algorithmus in abstrakte Einschubmethoden, deren Aufrufreihenfolge und eventuell konkreten Einzeloperationen.

Das heißt, dass für diese **abstrakten Einzeloperationen** in der **Basisklasse** nur deren **Verträge** vorgegeben werden, die Implementierung wird an die Unterklassen delegiert. Eine Unterklasse ist von der Basisklasse abhängig, während im klassischen Fall eine aufrufende Klasse von der Schnittstelle einer aufgerufenen Klasse abhängig ist. Da die Schablonenmethode den Vertrag der Einschubmethoden vorgibt und eine Unterklasse den Vertrag der Basisklasse nach dem liskovschen Substitutionsprinzip einhalten muss, wenn ein Objekt einer Unterklasse an die Stelle eines Objekts der Basisklasse treten können soll, ist die Schablonenmethode nicht von der speziellen Implementierung einer Unterklasse abhängig. Man spricht daher auch von **Dependency Inversion**.

Die Basisklasse ist von den Unterklassen unabhängig.



Das **Grundgerüst der Schablonenmethode** wird als **invariant** angesehen und darf daher von Subklassen nicht überschrieben werden.



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**, deren Vertrag von der Schablonenmethode vorgegeben wird.



Einschubmethoden werden **in Unterklassen implementiert**, wobei jede Unterklasse die Einschubmethoden unterschiedlich implementieren kann.



So kann jede Unterklasse eine andere Variante des Algorithmus realisieren.

Das Entwurfsmuster Schablonenmethode versucht, soviel Code wie möglich in abstrakten Basisklassen 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.



Einschubmethoden werden speziell für den jeweiligen Algorithmus der Schablonenmethode geschaffen. Sie werden daher nicht von außerhalb der Schablonenmethode aufgerufen, sondern nur von der Schablonenmethode selbst.

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.

Um zu verdeutlichen, dass eine Schablonenmethode als invariant angesehen wird, 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 die Einschubmethoden werden als `protected` deklariert, sodass sie nur von Unterklassen überschrieben werden können.

6.7.3.1 Klassendiagramm

Hier ein beispielhaftes Klassendiagramm für das Muster Schablonenmethode:

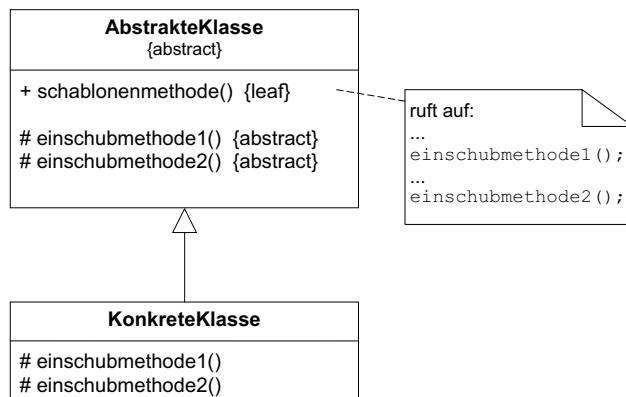


Bild 6-24 Beispiel für ein Klassendiagramm der Schablonenmethode

Die **Schablonenmethode** der Klasse `AbstrakteKlasse` in Bild 6-24 ist vollständig implementiert und benutzt in diesem Beispiel für ihren Algorithmus die Einschubmethoden `einschubmethode1()` und `einschubmethode2()`. Das Besondere an diesen beiden Einschubmethoden ist, dass sie abstrakt sind – die Einschubmethoden werden erst in einer spezialisierenden Unterklasse implementiert.

Die **Einschubmethoden** wurden in Bild 6-24 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.

6.7.3.2 Teilnehmer

Die Schablonenmethode umfasst die folgenden 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.

6.7.3.3 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:

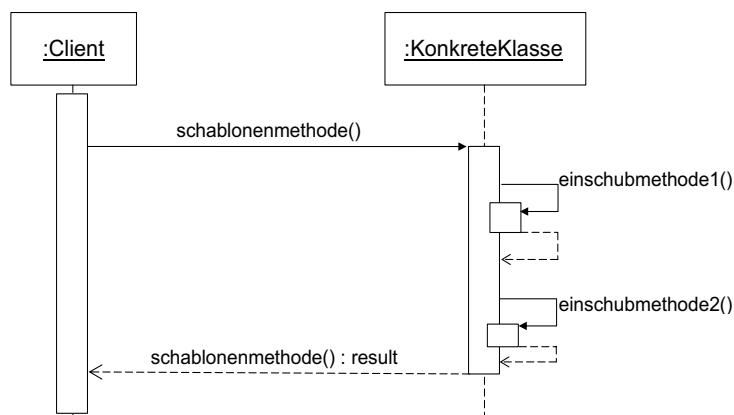


Bild 6-25 Sequenzdiagramm des Musters Schablonenmethode

Einschubmethoden werden beispielsweise dann gebraucht, wenn in der Schablonenmethode Objekte erzeugt werden müssen, deren spezieller Typ aber noch nicht festgelegt werden kann, sondern nur ihr Vertrag. 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.

6.7.3.4 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 Klasse `Urlaubskarte` definiert und in deren Schablonenmethode verwendet. Die Klasse `Urlaubskarte` hat folgendes Aussehen:

```
// Datei: Urlaubskarte.java

public abstract class Urlaubskarte {    // abstrakte Klasse

    public final 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.");
    }

    protected abstract 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 {

    @Override
    protected void zusatzSchreiben() { // Einschubmethode überschreiben
        System.out.println("Ich treibe viel Sport.");
    }
}
```

```
// Datei: UrlaubskarteAnFirma.java

public class UrlaubskarteAnFirma extends Urlaubskarte {

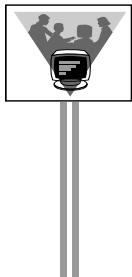
    @Override
    protected void zusatzSchreiben() { // Einschubmethode überschreiben
        System.out.println("Ich freue mich wieder auf die Arbeit.");
    }
}
```

Die Klasse TestSchablone zeigt, wie die Schablonenmethode karteSchreiben() in verschiedenen Unterklassen aufgerufen wird, dabei aber jeweils eine andere übergeschriebene Einschubmethode benutzt wird:

```
// Datei: TestSchablone.java

public class TestSchablone {

    public static void main(String[] args) {
        System.out.println("Karte an die Freunde:");
        UrlaubskarteAnFreunde karteFreunde = new UrlaubskarteAnFreunde();
        karteFreunde.karteSchreiben(); // dabei erfolgt der Aufruf einer
                                       // ueberschreibenden Methode
        System.out.println();
        System.out.println("Karte an die Firma:");
        UrlaubskarteAnFirma karteFirma = new UrlaubskarteAnFirma();
        karteFirma.karteSchreiben(); // dabei erfolgt der Aufruf einer
                                   // ueberschreibenden Methode
    }
}
```



Hier das Protokoll des Programmlaufs:

Karte an die Freunde:
Ich bin gut an meinem Urlaubsziel angekommen.
Das Essen schmeckt gut und die Gegend gefaellt mir.
Ich treibe viel Sport.

Karte an die Firma:
Ich bin gut an meinem Urlaubsziel angekommen.
Das Essen schmeckt gut und die Gegend gefaellt mir.
Ich freue mich wieder auf die Arbeit.

6.7.4 Einsatzgebiete

Die Schablonenmethode setzt voraus, dass mehrere Arbeitsschritte in mehreren Klassen gleichartig 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, welche die Schablonenmethode enthält, 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 6-26). Es sollen verschiedene Klassen für Datenstrukturen von Objekten implementiert werden. Unabhängig von der verwendeten Datenstruktur (Liste, Baum etc.) soll festgestellt werden können, ob sich ein bestimmtes Objekt in der betreffenden Datenstruktur befindet. Der dazu benötigte Algorithmus kann als Schablonenmethode `contains()` in einer gemeinsamen Basisklasse definiert werden.

Jetzt das bereits erwähnte Bild:

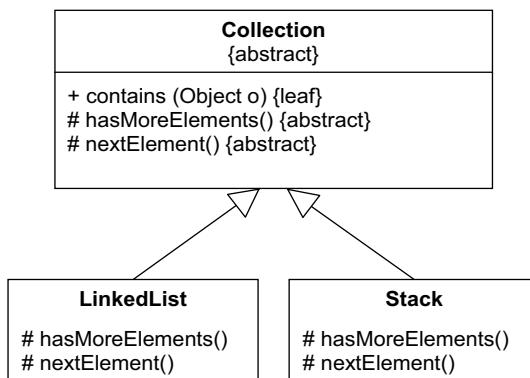


Bild 6-26 Container-Klassen zur Veranschaulichung der Schablonenmethode

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()`:

```

public abstract class Collection {

    // Schablonenmethode
    public final boolean contains(Object o) {
        while(hasMoreElements()) {
            if(o.equals(nextElement()))
                return true;
        }
        return false;
    }

    // Einschubmethoden
    protected abstract boolean hasMoreElements();
    protected abstract 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 benutzen dabei den Algorithmus der Methode `contains()` mit. Sie dürfen den Algorithmus nicht überschreiben. Die von der abstrakten Basisklasse abgeleiteten Klassen implementieren die Einschubmethoden in Abhängigkeit von der jeweiligen Datenstruktur der speziellen Ausprägung der Collection.

6.7.5 Bewertung

6.7.5.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Unabhängigkeit von den Unterklassen – Dependency Inversion**

Durch das Muster Schablonenmethode wird eine Klasse nicht von ihren Unterklassen abhängig, obwohl die Klasse Einschubmethoden aufruft, die in den Unterklassen implementiert sind. Der Mechanismus des Musters Schablonenmethode führt zu einer Invertierung der Abhängigkeiten (**Dependency Inversion**).

- **Abstraktion von Verhalten**

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.

- **Wiederverwendbarkeit der Spezifikation**

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.

6.7.5.2 Nachteile

Aus diesem Verhaltensmuster resultieren – wenn sein Einsatz Sinn macht – keine Nachteile.

6.7.6 Ähnliche Entwurfsmuster

Die **Fabrikmethode** verlagert wie die **Schablonenmethode** die Implementierung in Unterklassen.

Das Entwurfsmuster **Strategie** tauscht einen ganzen Algorithmus statt nur einzelner Teile aus. Es verwendet die Objektkomposition anstelle einer statischen 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 Objektkomposition beim Entwurfsmuster Strategie wird die Implementierung nicht zur Kompilierzeit, sondern zur Laufzeit austauschbar.

6.8 Das Verhaltensmuster Strategie

6.8.1 Name/Alternative Namen

Strategie (engl. strategy oder policy).

6.8.2 Problem

Das Entwurfsmuster Strategie ist einzusetzen, wenn eine Anwendung über mehrere alternative Strategien verfügt, zum Erreichen eines bestimmten Ergebnisses zu einer bestimmten Zeit aber jeweils nur eine einzige dieser Strategien benötigt.

Das Strategie-Muster soll es erlauben, dass ein ganzer Algorithmus in Form einer Kapsel, einer sogenannten Strategie, von der Umgebung zur Laufzeit komplett ausgetauscht wird.



Ein Objekt einer Anwendung, das den auszutauschenden Algorithmus nutzt, wird im Folgenden als **Kontextobjekt** bzw. als Objekt der Klasse **Kontext**⁹² bezeichnet.

Ein Kontextobjekt darf nicht von einer speziellen Implementierung des Algorithmus abhängen. Es soll zur **Kompilierzeit** eine **Abstraktion** aggregieren.

Dem Kontextobjekt soll **zur Kompilierzeit nur die Abstraktion der Strategie**, d. h. eine Schnittstelle oder eine abstrakte Basisklasse, bekannt sein.



Zur **Laufzeit** soll die Programmumgebung die Abstraktion durch ein Objekt, dessen Klasse die **Abstraktion implementiert**, ersetzen.



Wenn das Kontextobjekt die konkrete Strategie kennen würde, wäre es von der konkreten Strategie voll abhängig. Es soll aber gerade nur von dessen Abstraktion abhängig sein. Es geht beim Strategie-Muster also um die **Verringerung von Abhängigkeiten**.

6.8.3 Lösung

Damit die verschiedenen Algorithmen einer Gruppe von miteinander verwandten Algorithmen untereinander austauschbar werden, benötigen sie dieselbe Abstraktion,

⁹² Unterschiedliche Objekte der Klasse **Kontext** können mit unterschiedlichen Strategien arbeiten.

welche im Folgenden mit `IStrategie` bezeichnet wird. Jeder einzelne Algorithmus wird nun als sogenannte konkrete 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 der Abstraktion `IStrategie` treten.

Das Strategie-Muster zieht die alternativen Strategien eines Kontextobjektes heraus, abstrahiert sie durch die Abstraktion `IStrategie` und aggregiert zur Kompilierzeit diese Abstraktion.



Verwendet wird dabei das **Entwurfsprinzip "Ziehe Objektkomposition der Klassenvererbung vor"** (siehe Kapitel 3.2.2), welches die Reduktion von Abhängigkeiten zum Ziele hat.

Das Entwurfsmuster Strategie ist ein objektbasiertes Verhaltensmuster, da es einen Algorithmus in einem Objekt kapselt, das von einem anderen Objekt, dem Kontextobjekt, zur Laufzeit benutzt wird.

6.8.3.1 Klassendiagramm

Die Methode `operation()` der Klasse `Kontext` ist im Folgenden beispielhaft die Methode, welche ganz oder teilweise mit unterschiedlichen Algorithmen – sprich Strategien – ausgeführt werden soll. Zu diesem Zweck referenziert die Klasse `Kontext` eine gemeinsame Schnittstelle oder abstrakte Klasse für den Aufruf der verschiedenen Algorithmen.

Im Folgenden das Klassendiagramm des Strategiemusters bei Verwendung einer Schnittstelle als Abstraktion:

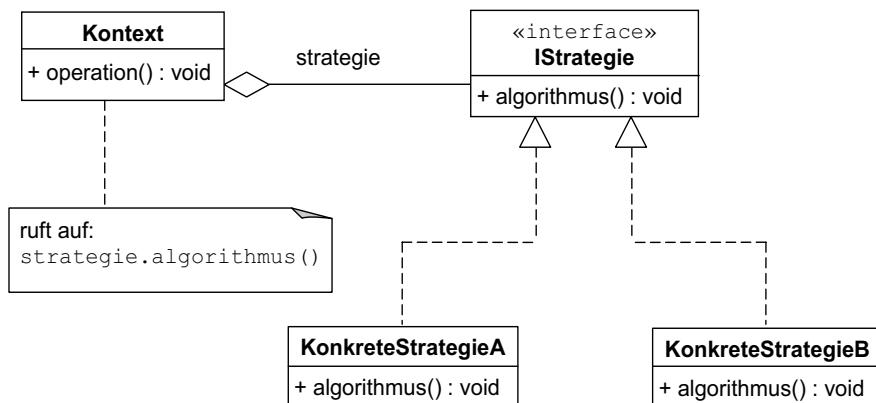


Bild 6-27 Klassendiagramm des Strategiemusters

Beim Strategiemuster in diesem Klassendiagramm wird eine Strategie, die ein Kontextobjekt benutzt, in ein eigenes Objekt vom Typ `IStrategie` ausgelagert. Das Kontextobjekt wird mit der Schnittstelle `IStrategie` über eine Aggregationsbeziehung verbunden und kann so die Strategie nutzen.

Das Kontextobjekt ist dabei nicht von der konkreten Implementierung der Schnittstelle abhängig. Die Aggregationsbeziehung erlaubt es dem Kontextobjekt, alle möglichen unterschiedlichen konkreten Strategien zu nutzen, solange die konkreten Strategien die Schnittstelle `IStrategie` realisieren und den Vertrag der Schnittstelle nicht brechen. Alle verschiedenen Algorithmen der Klassen `KonkreteStrategieX` ($X = A..Z$) implementieren dabei die definierte Schnittstelle `IStrategie`.

Die Schnittstelle `IStrategie` wird vom Kontext vorgegeben.



Da jede konkrete Strategie diese Schnittstelle erfüllen muss, hängt der Kontext nicht von der jeweiligen konkreten Strategie ab. Durch die Verwendung einer Schnittstelle ist der Kontext nur von der Schnittstelle abhängig und nicht von deren Implementierung. Das wird als **Dependency Inversion** bezeichnet.

Die Klasse `Kontext` aggregiert die Schnittstelle `IStrategie`. Das diese Schnittstelle implementierende Objekt kann – falls gewünscht – jederzeit zur Laufzeit ausgetauscht werden, falls das liskovsche Substitutionsprinzip eingehalten wird.

Die Umgebung des Kontextobjekts muss eine Möglichkeit bieten, um die Referenz auf das implementierte Objekt zu setzen. Dies wird auch als **Konfiguration des Kontextes** bezeichnet.



6.8.3.2 Teilnehmer

Folgende Klassen bzw. Schnittstellen sind an dem Entwurfsmuster von Bild 6-27 beteiligt:

- **Kontext**

Die Klasse `Kontext` aggregiert als Abstraktion die Schnittstelle `IStrategie`. Die Klasse `Kontext` enthält in Java dazu eine Referenz `strategie` vom Typ der Schnittstelle `IStrategie`. Durch diese Referenz wird die zu verwendende Strategie festgelegt. Durch `strategie.algorithmus()` wird die gewählte Strategie ausgeführt, um den Aufruf der Operation `operation()` zu realisieren.

- **`IStrategie`**

Die Schnittstelle `IStrategie` wird von allen unterstützenden Algorithmen realisiert. Sie 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 in der Schnittstelle `IStrategie` deklariert wurde.

6.8.3.3 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.

Als Beispiel wird ein Sequenzdiagramm mit dem Wechsel zwischen zwei verschiedenen Strategien in Bild 6-11 gezeigt:

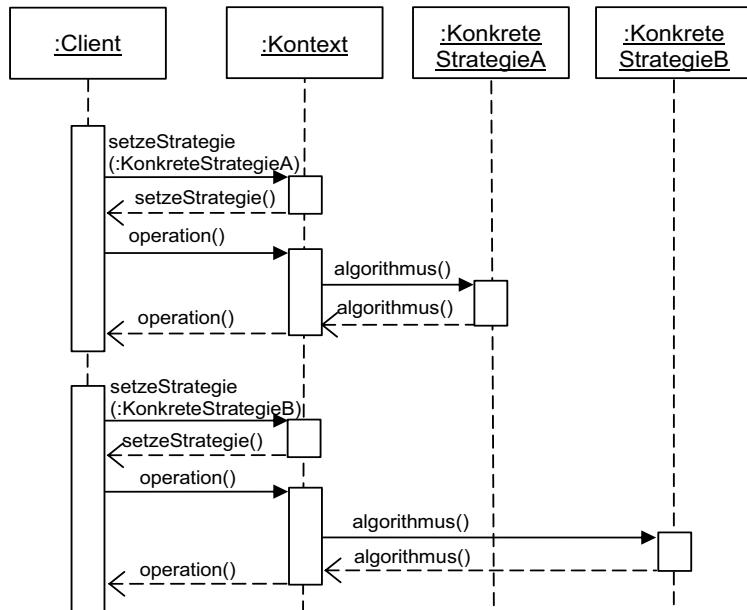


Bild 6-28 Sequenzdiagramm Strategie

Die Aufrufe der Methode `operation()` laufen im Kontextobjekt unterschiedlich ab, je nachdem, welche aktuelle Strategie vom Client gesetzt wurde. Beim ersten Aufruf in Bild 6-28 wird die Methode `algorithmus()` eines Objekts der Klasse `KonkreteStrategieA` aufgerufen, beim zweiten Aufruf hingegen die Operation `algorithmus()` eines Objekts der Klasse `KonkreteStrategieB`.

6.8.3.4 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 als Abstraktion die Methode `datumAusgeben()`, die ein Datum in einem bestimmten Format ausgeben soll:

```
// Datei: IDatumsFormat.java
public interface IDatumsFormat {
```

```
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 final int tag, monat, jahr;
    private IDatumsFormat format;

    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() {
        // Ausgabe wird an das gesetzte Format delegiert.
        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 {
    @Override
    public void datumAusgeben(int tag, int monat, int jahr) {
        System.out.printf("Europaeisches Format: %02d.%02d.%04d%n",
            tag, monat, jahr);
    }
}
```

Die Klasse `AmerikanischesFormat` gibt ein Datum in der Form `mm/dd/yyyy` aus:

```
// Datei: AmerikanischesFormat.java

public class AmerikanischesFormat implements IDatumsFormat {

    @Override
    public void datumAusgeben(int tag, int monat, int jahr) {
        System.out.printf("Amerikanisches Format: %02d/%02d/%04d",
                           monat, 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 Programmablaufs:

Europaeisches Format: 21.09.1985
Amerikanisches Format: 09/21/1985

6.8.4 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 12.1) 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.

6.8.5 Bewertung

6.8.5.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Festlegung zur Laufzeit und Vermeiden statischer Unterklassen**

Das Strategie-Muster kann wesentlich flexibler eingesetzt werden als eine Lösung, bei der die unterschiedlichen Algorithmen in statischen Unterklassen der Kontextklasse realisiert würden. Eine Strategie kann zur Laufzeit ausgetauscht werden, wenn sie nach dem liskovschen Substitutionsprinzip die vorgegebene Abstraktion einhält. Bei der statischen Lösung müsste man eine Methode der Basisklasse in einer Unterklassie statisch überschreiben, um sie zu variieren.

- **Kontext nur von der Abstraktion abhängig**

Durch die Verwendung einer Schnittstelle ist der Kontext nur von der Abstraktion abhängig und nicht von deren Implementierung (**Dependency Inversion**).

- **ganze Familie von Algorithmen möglich**

Eine ganze Familie von Algorithmen ist möglich. Jeder Algorithmus ist für sich gekapselt und hat dieselbe Abstraktion. Dadurch ist er flexibel zur Laufzeit austauschbar. Ein gekapselter Algorithmus kann leichter wiederverwendet werden.

- **Vermeidung von Mehrfachverzweigungen**

Mehrfachverzweigungen aufgrund von Fallunterscheidungen können im Programmcode der Kontextklasse vermieden werden, was die Übersichtlichkeit des Programmtextes erhöht. Der Inhalt jeder Verzweigung – also die entsprechende Anweisungsfolge – kann als eine eigene Strategieklsasse umgesetzt werden.

6.8.5.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **Applikation muss Strategien kennen**

Eine Applikation bzw. die Umgebung des Kontextobjekts muss die unterschiedlichen Strategien kennen, um ein Kontextobjekt passend initialisieren bzw. konfigurieren zu können. Die Applikation muss also das Wissen besitzen, in welcher Situation welche Strategie angebracht ist. Ggf. müssen konkrete Strategieklassen Informationen über ihre Nutzung von Ressourcen – und damit über Implementierungseigenschaften – zur Verfügung stellen, damit eine Applikation die richtige Entscheidung treffen kann.

- **viele Klassen**

Es werden viele, oft kleine Klassen geschrieben.

- **Erhöhung des Kommunikationsaufwandes**

Das Muster fordert eine einzige Schnittstelle für alle konkreten Strategieklassen. In der Konsequenz wird diese Schnittstelle oft etwas breit, damit alle Parameter für alle möglichen Strategien übergeben werden können. Es entsteht ein höherer Kommunikationsaufwand zwischen Strategie und Kontext gegenüber der herkömmlichen Implementierung der Algorithmen im Kontext selbst. Der Kontext muss ggf. Werte für Parameter zur Verfügung stellen, die eigentlich von der konkreten Strategie gar nicht gebraucht werden.

6.8.6 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 wie auch die **Strategie** mit der Änderung eines Algorithmus. Bei der Strategie gibt es nur eine einzige auszutauschende Einheit. Alle konkrete Strategien müssen dieselbe Schnittstelle aufweisen. Bei der Schablonenmethode können mehrere Einschubmethoden mit verschiedenen Schnittstellen verwandt werden. Eine Schablonenmethode legt das Grundgerüst eines Algorithmus statisch fest und erlaubt nur den Austausch einzelner Schritte des Algorithmus in verschiedenen Unterklassen. Eine Kontextklasse benutzt im Falle des Strategiemusters 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 Strategiemuster 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 Strategiemuster eine spezifische Anfrage eines Kontextobjekts an ein anderes Objekt gesandt, das eine Strategie (Algorithmus) zum Ausführen der Anfrage repräsentiert.

6.9 Das Verhaltensmuster Vermittler

Das Verhaltensmuster Vermittler darf nicht mit dem Architekturmuster Broker (siehe Kapitel 11.1) verwechselt werden. Ein Broker stellt umgangssprachlich auch einen Vermittler dar, aber keinen Vermittler im Sinne des Vermittler-Musters.

6.9.1 Name/Alternative Namen

Vermittler, Mediator (engl. mediator).

6.9.2 Problem

Ein Vermittler soll es erlauben, das Zusammenspiel zwischen vielen Objekten im Vermittler zu kapseln und zentral durch den Vermittler zu steuern, damit die einzelnen Objekte voneinander entkoppelt werden. Objekte sollen sich wechselseitig nicht mehr kennen, sondern nur noch den **Vermittler**, welcher die Verteilung von Nachrichten als **zentrale Steuerung** übernimmt.



Der Vermittler soll das gewünschte Gesamtverhalten durch die Benachrichtigung der Objekte erzeugen. Damit soll die Wiederverwendbarkeit der Objekte erhöht und außerdem das System übersichtlicher werden.

6.9.3 Lösung

Der Vermittler ist ein objektbasiertes Verhaltensmuster. Die miteinander kommunizierenden Objekte werden als **Kollegen** bezeichnet.

Bei Änderungen eines Objekts benachrichtigt dieses Objekt den Vermittler. Der Vermittler wiederum benachrichtigt die anderen Objekte über die erfolgte Änderung. Ein Vermittler muss also wissen, welche konkreten Kollegen er benachrichtigen soll.



Der Vermittler muss folglich benachrichtigt werden, wenn das Zustellungsschema geändert werden soll.

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.

Die Beseitigung der Komplexität der Beziehungen zwischen den Objekten wird mit der Komplexität des Vermittlers erkauft.



Im Folgenden werden die beiden Topologien vermaschtes Netz und Sterntopologie schematisch dargestellt:

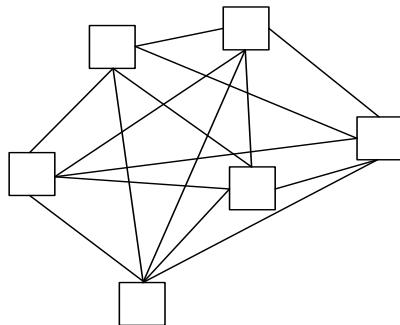


Bild 6-29 Vermaschtes Netz

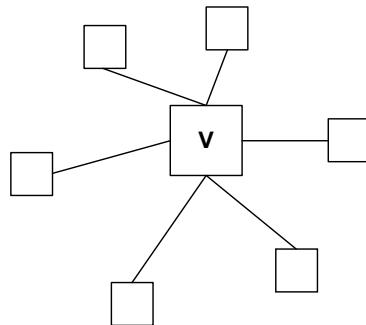


Bild 6-30 Sterntopologie

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.



6.9.3.1 Klassendiagramm

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. Prinzipiell kann für die Abstraktion eines Kollegen sowohl eine abstrakte Klasse als auch eine Schnittstelle verwendet werden. Dabei muss das liskovsche Substitutionsprinzip durch die konkreten Kollegen eingehalten werden.

Das Klassendiagramm in Bild 6-14 zeigt eine Assoziation zwischen der Klasse `Vermittler` und den Klassen `KonkreterKollegeX` ($X = A..Z$). Wie in Bild 6-14 zu sehen ist, soll jeder konkrete Kollege den Vermittler kennen, andererseits soll der Vermittler alle konkreten Kollegen kennen. Es handelt sich also um eine bidirektionale Assoziation.

Im Fall der Veränderung eines konkreten Kollegen-Objektes wird der assozierte Vermittler benachrichtigt. Dieser informiert dann alle betroffenen weiteren konkreten Kollegen.

Hier das Klassendiagramm:

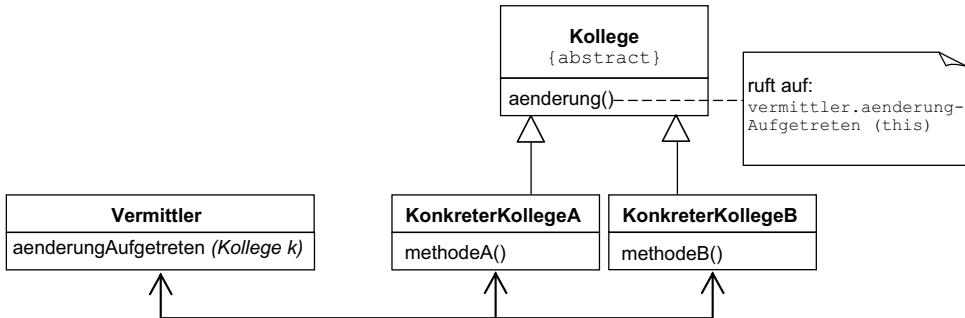


Bild 6-31 Klassendiagramm des Vermittlermusters

Die Kommunikation aller konkreten Kollegen untereinander erfolgt nur zentral über den 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 ändern 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.



6.9.3.2 Teilnehmer

Das Vermittlermuster umfasst die folgenden Teilnehmer:

- **Vermittler**

Der 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.⁹³

- **Kollege**

Die Klasse **Kollege** ist eine abstrakte Basisklasse oder auch eine Schnittstelle für alle konkreten Kollegen. Ein Kollege hält eine Referenz auf einen Vermittler. Er informiert den Vermittler, wenn bei ihm eine Änderung eingetreten ist. Ein Kollege

⁹³ Es kann mehrere Klassen für Vermittler geben, die an die jeweils unterschiedlichen Gegebenheiten der Kollegen-Objekte und deren Zusammenspiel angepasst sind (siehe [Gam94]).

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.

6.9.3.3 Dynamisches Verhalten

Das folgende Sequenzdiagramm zeigt ein Beispiel für das dynamische Verhalten der Beteiligten am Vermittler-Muster:

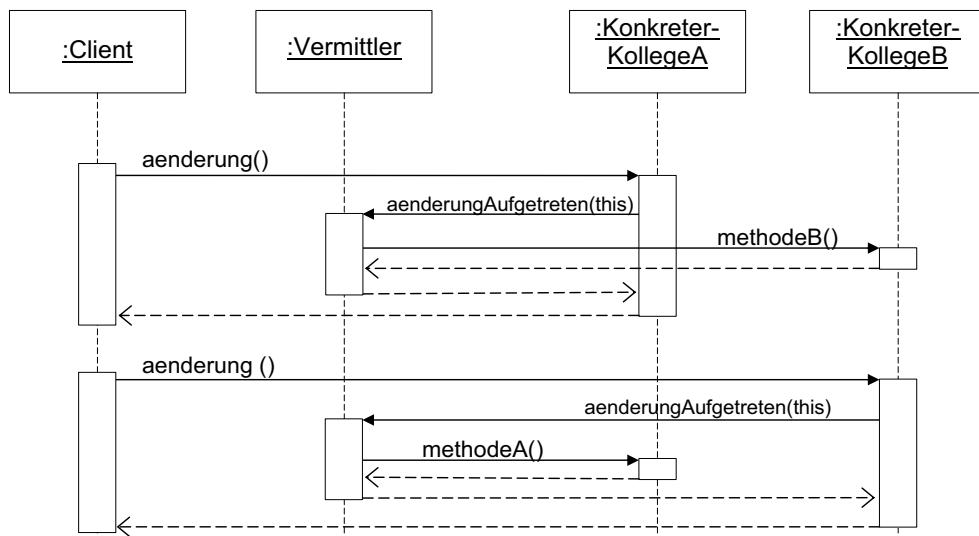


Bild 6-32 Sequenzdiagramm des Vermittler-Musters

In diesem Bild führt ein Client eine Veränderung am Objekt der Klasse `Konkreter-KollegeA` durch. Danach teilt dieses Objekt seinem Vermittler mit, dass eine Änderung eingetreten ist. Das Objekt der Klasse `Vermittler` 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 6-32 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.

6.9.3.4 Offene Punkte des Vermittlermusters

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 [Gra02a] wird eine Lösung am Beispiel eines Vermittlers zwischen grafischen Objekten einer kleinen Anwendung gezeigt, bei der die oben genannten Punkte exemplarisch berücksichtigt sind.

6.9.3.5 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 zu sehen, 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.

Die abstrakte Klasse Kollege ist die Basisklasse für alle Kollegen:

```
// Datei: Kollege.java

public abstract class Kollege {

    private final Vermittler vermittler; // Referenz auf den Vermittler

    public Kollege(Vermittler vermittler) {
        this.vermittler = vermittler;
    }

    // Wird von den ableitenden Klassen nicht überschrieben
    public final void aenderung() {
        System.out.printf("%s wurde geändert durch den Aufruf der"
            + " Methode aenderung() und informiert den Vermittler.%n",
            getClass().getSimpleName());
        vermittler.aenderungAufgetreten(this); // Vermittler informieren
    }
}
```

Die Klasse KonkreterKollegeA ist von der abstrakten Klasse Kollege abgeleitet. Über die Methode aenderung() informiert sie einen Vermittler über Zustandsänderungen:

```
// Datei: KonkreterKollegeA.java

public class KonkreterKollegeA extends Kollege {

    public KonkreterKollegeA(Vermittler vermittler) {
        super(vermittler);
        System.out.println("KollegeA: instanziert.");
    }

    // Wird vom Vermittler aufgerufen, wenn sich ein Kollege ändert
    public void methodeA() {
        System.out.println("KollegeA wird in methodeA() ueber die"
            + " Aenderungen eines Kollegen informiert.");
    }
}
```

Die Klasse KonkreterKollegeB ist ebenfalls von der abstrakten Klasse Kollege abgeleitet und kann über die Methode aenderung() einen Vermittler informieren:

```
// Datei: KonkreterKollegeB.java

public class KonkreterKollegeB extends Kollege {

    public KonkreterKollegeB(Vermittler vermittler) {
        super(vermittler);
        System.out.println("KollegeB: instanziert.");
    }

    // Wird vom Vermittler aufgerufen, wenn sich ein Kollege ändert
    public void methodeB() {
        System.out.println("KollegeB wird in methodeB() ueber die"
            + " Aenderungen eines Kollegen informiert.");
    }
}
```

Die Klasse Vermittler informiert bei Änderungen eines Kollegen den jeweils anderen Kollegen. Über set-Methoden erhält ein Vermittler Kenntnis über die beiden an der Kommunikation beteiligten Objekte. Hier nun der Quellcode der Klasse Vermittler:

```
// Datei: Vermittler.java

public class Vermittler {

    private KonkreterKollegeA kollegeA;
    private KonkreterKollegeB kollegeB;

    public Vermittler() {
        System.out.println("Vermittler: instanziert.");
    }

    public void aenderungAufgetreten(Kollege kollege) {
        if (kollegeA.equals(kollege)) {
            System.out.println("Vermittler: Informiere KollegeB.");
            kollegeB.methodeB();
        } else if (kollegeB.equals(kollege)) {
            System.out.println("Vermittler: Informiere KollegeA.");
            kollegeA.methodeA();
        }
    }
}
```

```

        }
    }

    public void setKollegeA(KonkreterKollegeA konkreterKollegeA) {
        this.kollegeA = konkreterKollegeA;
    }

    public void setKollegeB(KonkreterKollegeB konkreterKollegeB) {
        this.kollegeB = konkreterKollegeB;
    }
}

```

Das Client-Programm ist in zwei Phasen aufgeteilt. In der ersten Phase werden alle Objekte instanziert. 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:");
        Vermittler Vermittler = new Vermittler();
        KonkreterKollegeA kollegeA = new KonkreterKollegeA(Vermittler);
        Vermittler.setKollegeA(kollegeA);
        KonkreterKollegeB kollegeB = new KonkreterKollegeB(Vermittler);
        Vermittler.setKollegeB(kollegeB);

        // KollegeA ändern
        System.out.println("\nKollegeA ändern:");
        kollegeA.aenderung();

        // KollegeB ändern
        System.out.println("\nKollegeB ändern:");
        kollegeB.aenderung();
    }
}

```

Es ist aber anzumerken, 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 durch Interpretation dieser Tabelle hergestellt werden.



Die folgende Programmausgabe zeigt die erwähnten Phasen sehr deutlich.



Die Ausgabe ist:

Initialisierung:

Vermittler: instanziert.
KollegeA: instanziert.
KollegeB: instanziert.

KollegeA aendern:

KonkreterKollegeA wurde geaendert durch den Aufruf der Methode aenderung() und informiert den Vermittler.
Vermittler: Informiere KollegeB.
KollegeB wird in methodeB() ueber die Aenderungen eines Kollegen informiert.

KollegeB aendern:

KonkreterKollegeB wurde geaendert durch den Aufruf der Methode aenderung() und informiert den Vermittler.
Vermittler: Informiere KollegeA.
KollegeA wird in methodeA() ueber die Aenderungen eines Kollegen informiert.

6.9.4 Bewertung

6.9.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **lose Kopplung der Kollegen-Objekte**

Kollegen-Objekte sind untereinander lose gekoppelt und können wiederverwendet werden.

- **Verständlichkeit, Verwaltbarkeit und Erweiterbarkeit**

Zwischen den Kollegen spannt sich ohne Vermittler eine n-zu-m-Beziehung auf. Das Vermittler-Muster reduziert dies jeweils auf eine 1-zu-n-Beziehung zwischen Vermittler und Kollegen und die Kollegen müssen nur den Vermittler kennen. Dadurch wird die Verständlichkeit, Verwaltbarkeit und Erweiterbarkeit verbessert.

- **zentralisierte Steuerung der Kommunikation zwischen den Kollegen-Objekten**

Die Steuerung der Kommunikation zwischen den Kollegen-Objekten ist zentralisiert.

- **keine Unterklassenbildung der konkreten Kollegen nötig**

Die Bildung von Unterklassen der konkreten Kollegen entfällt, da bei einer Änderung der Kommunikation zwischen Kollegen lediglich neue Vermittler erzeugt bzw. vorhandene Vermittler geändert werden müssen und keine neuen konkreten Kollegenklassen statisch erzeugt werden müssen.

- **bei Änderungen keine Rückwirkung auf den Kommunikationspartner**

Objekte können geändert werden, ohne den Kommunikationspartner anzupassen. (Rückwirkungsfreiheit auf Kommunikationspartner). Unter Umständen muss der Vermittler geändert werden. Damit sind Änderungen lokalisiert.

6.9.4.2 Nachteile

Die folgenden Nachteile werden festgestellt:

- **Komplexität des Vermittlers**

Da der Vermittler die Kommunikation mit den Kollegen in sich kapselt, kann er unter Umständen sehr komplex werden.

- **Fehleranfälligkeit**

Der zentrale Vermittler ist fehleranfällig und bedarf fehlertoleranter Maßnahmen.

- **bei Änderungen der Kollegen muss der Vermittler angepasst werden**

Wenn sich die Kollegenschaft oder ihr Zusammenspiel ändert, muss der Vermittler angepasst werden.

6.9.5 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 zusammenwirken. 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 Unterklassie 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 –

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 [Gam94] und wird dort ausführlich beschrieben.

6.9.6 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 die betroffenen 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 eingegangene Nachricht jedoch nur an den gewünschten Empfänger weiter. Außerdem können beim Broker-Muster 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 im Falle des Broker-Musters 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 potenziell jedes Objekt mit jedem anderen über den Vermittler kommunizieren kann, haben die am Beobachter-Muster beteiligten Objekte bestimmte Rollen, welche die Kommunikationsmöglichkeiten einschränken: nur beobachtbare Objekte können ihre Beobachter informieren und nicht umgekehrt. Das bedeutet, dass das Beobachter-Muster für einfache 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 aufrufen, aber der Vermittler wird zuerst die alte Benachrichtigung noch komplett abarbeiten, bevor er sich dem neuen Anruf zuwendet.

6.10 Das Verhaltensmuster Zustand

6.10.1 Name/Alternative Namen

Zustand (engl. state).

6.10.2 Problem

Zustandsbehaftete Probleme werden oft in einem funktionsorientierten Ansatz durch die Programmierung einer Fallunterscheidung gelöst, welche 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 in einem objektorientierten Ansatz so gelöst werden, dass jeder **Zustand**, d. h. jeder Fall einer klassischen Bedingungsanweisung, als **Objekt einer eigenen Klasse** dargestellt wird.



6.10.3 Lösung

Das Zustandsmuster kapselt unterschiedliche, zustandsabhängige Verhaltensweisen eines Objekts. Hierbei ist der konkrete Zustand eines Objekts selbst wieder ein Objekt, welches unabhängig von anderen Objekten ist.

Eine Klasse, deren Objekte ein **zustandsabhängiges Verhalten** besitzen, wird im Zustandsmuster als **Kontextklasse** bezeichnet.



Für die verschiedenen **Zustände eines Kontextobjekts** gibt es in diesem Muster jeweils ein Objekt einer **eigenen Zustandsklasse**, welches das Verhalten im entsprechenden Zustand kapselt.



Im Entwurfsmuster Zustand hängt zur Kompilierzeit der Kontext nicht von den konkreten Zustandsklassen direkt ab, sondern nur von einer Schnittstelle bzw. von einer abstrakten Klasse, welche der Kontext als **Abstraktion** vorgibt. Diese **abstrakte Zustandsschnittstelle** wird von den konkreten Zustandsklassen der entsprechenden Kontextklasse implementiert.



Zur Laufzeit tritt dann bei Einhaltung der Verträge ein Objekt einer konkreten Zustandsklasse an die Stelle der Schnittstelle. Objekte von konkreten Zustandsklassen werden im Folgenden kurz Zustandsobjekte genannt.

Ein **zustandsabhängiges Kontextobjekt** referenziert den aktuellen Zustand und führt in der Grundform des Musters **Zustandsänderungen** durch, indem es die Anfrage des Client-Programms an das jeweilige aktuelle Zustandsobjekt delegiert.



Dadurch, dass 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.

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. Kontextobjekt und Zustandsobjekt sind dadurch schwach gekoppelt. Diese Situation wird in folgendem Bild gezeigt:

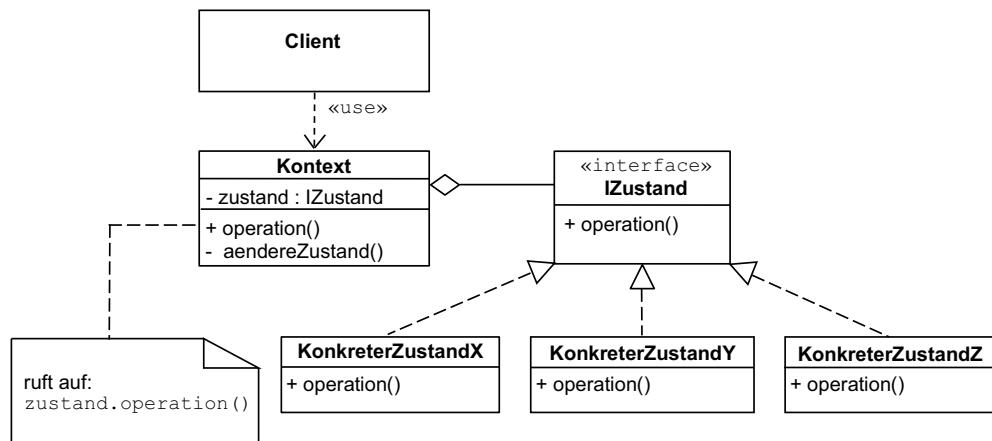


Bild 6-33 Klassendiagramm des Entwurfsmusters *Zustand mit Schnittstelle*

Die Kontextklasse **Kontext** hält eine **Referenz auf das aktuelle Zustandsobjekt**. Als Typ der Referenz wird die gemeinsame Schnittstelle (bzw. die gemeinsame abstrakte Basisklasse) der Zustandsklassen gewählt. Dies wird im Klassendiagramm in Bild 6-33 durch die Aggregationsbeziehung zwischen der Klasse **Kontext** und der Schnittstelle **IZustand** ausgedrückt. Das Klassendiagramm stellt eine Objektkomposition dar (siehe Kapitel 3.2.2).

Zur Laufzeit zeigt die **Referenz des Kontextobjekts 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 zur dynamischen Änderung von Zustandsobjekten und wird daher zu den **objektbasierten Entwurfsmustern** gezählt.

Im Zustandsmuster werden die einzelnen Zustände durch eigene Klassen gekapselt, die von einer gemeinsamen abstrakten Basisklasse ableiten bzw. eine gemeinsame Schnittstelle implementieren.



Die **Anwendung** selbst kennt die Zustandsobjekte nicht, sondern arbeitet nur mit dem Kontextobjekt, das die Übergänge durchführt, zusammen.



Das Zustandsmuster lässt offen, welcher seiner Teilnehmer für die Zustandsübergänge verantwortlich ist.



In der **Grundform des Musters** ist das Kontextobjekt für die Zustandswechsel zuständig.



Häufig ist die Bestimmung des Nachfolgezustandes aber auch eine **zustandsabhängige Operation** und sollte daher vom aktuellen Zustandsobjekt selbst durchgeführt werden. Die Alternativen zur Grundform des Musters werden in Kapitel 6.10.3.4 ausführlich diskutiert. Im Folgenden wird die Grundform des Musters vorgestellt.

Zustandsabhängiges Verhalten wird in der Regel mittels **Zustandsautomaten** (engl. **state machines**) beschrieben. Diese können in UML in Form von Zustandsdiagrammen (siehe Beispiel in Kapitel 6.10.3.5) modelliert werden. Beim Einsatz des Zustandsmusters in der Grundform realisiert das Kontextobjekt den Zustandsautomaten, wobei das Kontextobjekt Zustandsobjekte nutzt, die das Verhalten des Automaten in den jeweiligen Zuständen kapseln.

6.10.3.1 Klassendiagramm

Das Klassendiagramm wurde bereits in Bild 6-33 gezeigt.

Objekte der Klasse **Kontext** zeigen ein zustandsabhängiges Verhalten. Das bedeutet, dass es in dieser Klasse Methoden gibt, deren Wirkung unterschiedlich ist, je nachdem 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 konkreten Zustandsklassen implementieren dabei dieselbe **Schnittstelle** bzw. sind von einer gemeinsamen **abstrakten Basisklasse** abgeleitet. Jede **konkrete Zustandsklasse** ist verantwortlich für das Verhalten eines Kontextobjekts in diesem Zustand. Hierzu realisiert die konkrete Zustandsklasse alle zustandsabhängigen Operationen, die in der gemeinsamen Schnittstelle `IZustand` definiert sind.

Wird 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 in der **Grundform des Musters** allein dem **Kontextobjekt**.



Zu Zustandsänderungen dient die private Methode `aendereZustand()` des Kontextobjekts. Das bedeutet, dass ein Client in diesem Muster unabhängig von der Schnittstelle `IZustand` und von den konkreten Zustandsklassen ist.

6.10.3.2 Teilnehmer

Das gezeigte Klassendiagramm umfasst die folgenden Teilnehmer:

- **Client**

Ein **Client** ruft die öffentlichen Methoden des Kontextobjekts auf. Er kennt dabei die Schnittstelle `IZustand` nicht und auch keine konkreten Klassen, welche diese Schnittstelle 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 konkretes Zustandsobjekt, welches die Schnittstelle `IZustand` implementieren muss, gespeichert. Bei einem Zustandsübergang zeigt die Referenz auf ein anderes Zustandsobjekt.

- **IZustand**

Die Schnittstelle `IZustand` definiert eine einheitliche Schnittstelle aller konkreten Zustandsklassen mit allen zustandsbehafteten Operationen.

- **KonkreterZustandX**

Eine Klasse `KonkreterZustandX(X = A..Z)` implementiert das Verhalten, welches mit einem einzigen Zustand eines Kontextobjekts verknüpft ist.

6.10.3.3 Dynamisches Verhalten

Das dynamische Verhalten des Zustandsmusters soll in folgendem Bild anhand von zwei Zustandsobjekten (`x:KonkreterZustandX` und `y:KonkreterZustandY`) vor gestellt werden:

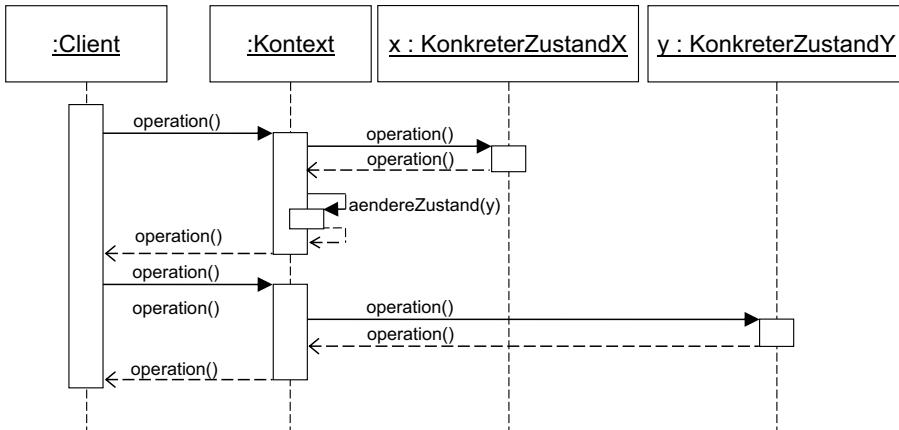


Bild 6-34 Sequenzdiagramm des Entwurfsmusters Zustand

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 soll das Kontextobjekt seinen Zustand über einen Aufruf von `aendereZustand(y)` ändern. 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.

6.10.3.4 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()`, die in der Schnittstelle `Izustand` definiert ist, erhält einen Parameter vom Typ der Klasse `Kontext`. Ruft ein Kontextobjekt die Methode `operation()` bei einem Zustandsobjekt 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 bzw. das Kontextobjekt kennen, gibt es mehrere Implementierungsalternativen:

- **Singleton-Klasse**

Gibt es in einer Applikation nur einen einzigen Automaten einer bestimmten Klasse, dann kann die entsprechende Kontextklasse als **Singleton-Klasse** (siehe Kapitel 7.3) realisiert werden. Die Zustandsobjekte können sich eine Referenz auf das Singleton-Objekt besorgen.

- **Enum-Klasse**

Die konkreten Zustände können in Form einer `enum`-Klasse implementiert werden. Damit sind die Zustandsobjekte öffentlich bekannt.

- **get-Methoden der Kontextklasse**

Kennen die Zustände ihren Automaten, kann der Automat alle seine möglichen Zustände über get-Methoden (wie beispielsweise `getKonkreterZustandX()`) den Zustandsobjekten zur Verfügung stellen.

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.

6.10.3.5 Programmbeispiel

Als Beispiel soll hier die Alarmanlage einer Bank beschrieben werden. Befindet sich die Alarmanlage im Zustand `AlarmanlageAktiv` und wird eine Person erkannt, so soll ein akustischer Alarm als Aktion ausgegeben werden. Befindet sich die Anlage hingegen im Zustand `AlarmanlageInaktiv` (während der Geschäftszeit der Bank), so soll kein

akustisches Signal ausgegeben werden. Bild 6-18 zeigt das Zustandsdiagramm nach UML für dieses Beispiel:

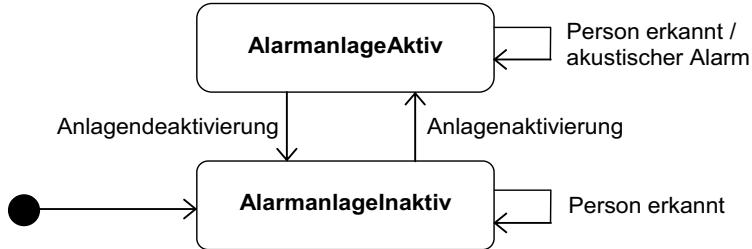


Bild 6-35 Zustandsdiagramm für die Alarmanlage

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 bei einem Ereignis kein Zustandsübergang statt, sondern die Anlage verbleibt im aktuellen Zustand.

Weiterhin wird gezeigt, dass als Startzustand der Zustand `AlarmanlageInaktiv` ausgewählt wird.

Bild 6-19 zeigt das Klassendiagramm dieses Programmbeispiels:

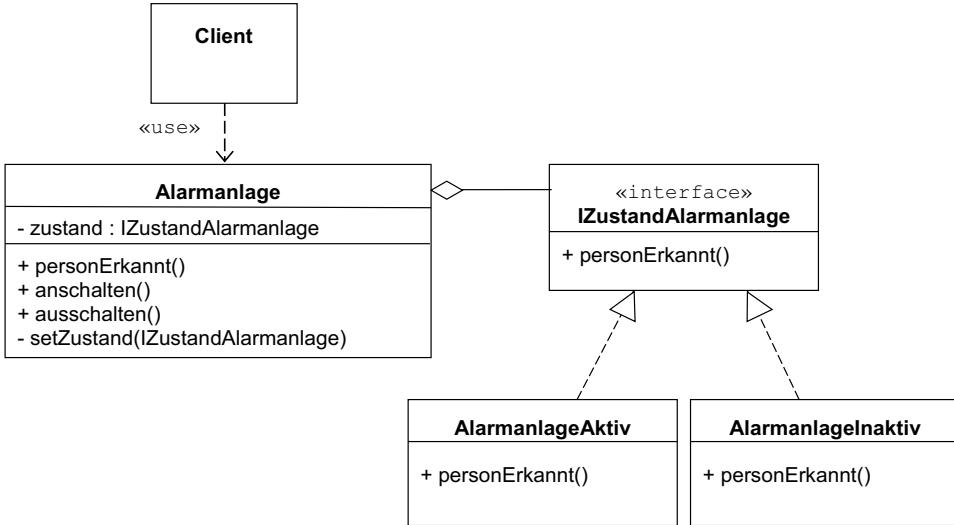


Bild 6-36 Klassendiagramm zum Programmbeispiel Alarmanlage

Die Schnittstelle `IZustandAlarmanlage` deklariert die Methoden aller Zustandsklassen:

```

// Datei: IZustandAlarmanlage.java
// Schnittstelle fuer alle Zustandsklassen
  
```

```
public interface IZustandAlarmanlage {  
    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  
    @Override  
    public void personErkannt() {  
        System.out.println("RING RING");  
    }  
}
```

Im konkreten Zustand `AlarmanlageInaktiv` hingegen soll eine Alarmanlage keinen akustischen Alarm generieren, da dies im normalen Geschäftsbetrieb nur störend wäre:

```
// 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  
    @Override  
    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 6-18 zu sehen ist, muss beim Erkennen einer Person kein Zustandswechsel erfolgen. Hier die Klasse `Alarmanlage`:

```
// Datei: Alarmanlage.java  
  
// Kontext, über den die Alarmanlage gesteuert wird  
public class Alarmanlage {  
  
    private final IZustandAlarmanlage zustandAktiv =
```

```
    new AlarmanlageAktiv();
private final IZustandAlarmanlage zustandInaktiv =
    new AlarmanlageInaktiv();
private IZustandAlarmanlage aktuellerZustand =
    zustandInaktiv; // Startzustand

public void anschalten() {
    setZustand(zustandAktiv);
}

public void ausschalten() {
    setZustand(zustandInaktiv);
}

public void personErkannt() {
    aktuellerZustand.personErkannt();
}

private void setZustand(IZustandAlarmanlage zustand) {
    aktuellerZustand = zustand;
}
}
```

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: TestAlarmanlage.java

public class TestAlarmanlage {

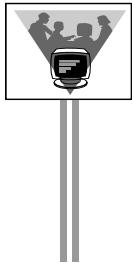
    public static void main(String[] args) {
        Alarmanlage anlage = new Alarmanlage();

        System.out.println("Anlage deaktivieren...");
        System.out.println("Bei Kundenbetrieb stört das.");
        anlage.ausschalten();

        System.out.println("Person erkannt.");
        anlage.personErkannt();

        System.out.println("Feierabend.");
        System.out.println("Aktivierung der Alarmanlage.");
        anlage.anSchalten();

        System.out.println("Person erkannt.");
        anlage.personErkannt();
    }
}
```



Hier das Protokoll des Programmlaufs:

Anlage deaktivieren...
Bei Kundenbetrieb stört 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 des Nachfolgezustands nicht zustandsabhängig ist.

6.10.4 Einsatzgebiete

Ein jedes zustandsabhängiges Verhalten, insbesondere wenn es durch Zustandsautomaten beschrieben werden kann, kann durch dieses Entwurfsmuster beschrieben werden. Beispiele für zustandsbehaftete Probleme sind:

- Bedienelemente mit Zuständen einer grafischen Benutzeroberfläche,
- Zustände von parallelen Einheiten (Prozesssteuerung) und
- Automaten.

6.10.5 Bewertung

6.10.5.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Zustände erhöhen die Übersichtlichkeit**

Zustände werden in Form von Klassen realisiert. Das gesamte Verhalten für einen Zustand wird in einer einzigen Klasse, einer konkreten Zustandsklasse, abgebildet. Ein Kontextobjekt hat immer einen bestimmten konkreten Zustand und enthält damit immer ein Objekt einer solchen Zustandsklasse. Das Verhalten in einem Zustand ist gekapselt, was die Übersichtlichkeit erhöht.

- **leichte Erweiterbarkeit**

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.

- **langwierige Fallunterscheidungen entfallen**

Die langwierigen Fallunterscheidungen, wie sie beim erwähnten funktionsorientierten Ansatz verwendet wurden, entfallen vollständig.

- **Wiederverwendung von Zustandsklassen**

Durch dieses Muster wird es möglich, Zustandsklassen eventuell auch in einem anderen Kontext wiederzuverwenden.

6.10.5.2 Nachteile

Der folgende Nachteil wird festgestellt:

- **ggf. hoher Aufwand**

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.

6.10.6 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 konkretes 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. Beim Strategiemuster setzt die Umgebung des Programms die Strategie, beim Zustandsmuster ist in der Grundform des Musters das Kontextobjekt für die Zustandswechsel verantwortlich.

6.11 Zusammenfassung

Verhaltensmuster beschreiben Interaktionen zwischen Objekten in bestimmten Rollen zur gemeinsamen Lösung eines bestimmten Problems.

Das Befehlsmuster in Kapitel 6.1 verbirgt die Details eines Befehls vor dem Aufrufer des Befehls. Es ermöglicht es, dass die Erzeugungszeit und die Ausführungszeit des Befehls voneinander getrennt werden.

Das Beobachtermuster in Kapitel 6.2 erlaubt es, dass ein Objekt abhängige Objekte von einer Änderung seines Zustands automatisch informiert, so dass eine Aktualisierung automatisch eingeleitet wird.

Das Besucher-Muster in Kapitel 6.3 gestattet es, 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 in Kapitel **6.4** ermöglicht es, eine Datenstruktur in verschiedenen Durchlaufstrategien zu durchlaufen, ohne dass der Client den Aufbau der Datenstruktur kennt.

Das Muster Memento in Kapitel **6.5** beschreibt, wie man den Zustand eines Objekts kapselt und speichert, um das Objekt später dann in einen dieser Zustände zurückzuversetzen. Es müssen zwei Schnittstellen definiert werden, eine für die Verwaltung von Mementos und eine Schnittstelle für das ursprüngliche Objekt, damit es seinen Zustand in einem Memento speichern oder daraus auslesen kann.

Das Rollen-Muster in Kapitel **6.6** erlaubt es, dass ein Objekt dynamisch die Rollen wechseln und mehrere Rollen gleichzeitig annehmen kann, sowie, dass mehrere Objekte die selbe Rolle haben können.

Das Muster Schablonenmethode in Kapitel **6.7** legt die Struktur eines Algorithmus in einer Basisklasse fest. Die Schablonenmethode der Basisklasse enthält sogenannte Einschubmethoden, welche in Unterklassen realisiert werden. Dadurch ist die Schablonenmethode nicht von der speziellen Implementierung einer Unterklasse abhängig. So mit wird Dependency Inversion erreicht.

Das Strategie-Muster in Kapitel **6.8** gestattet es, dass ein ganzer Algorithmus in Form einer Kapsel ausgetauscht wird.

Das Vermittler-Muster in Kapitel **6.9** stellt die Koordination einer ganzen Gruppe von Objekten durch den zentralen Vermittler in den Vordergrund. Die Beseitigung der Komplexität der Beziehungen zwischen den Objekten wird aber mit der Komplexität des Vermittlers erkauft. Jedoch bleiben die einzelnen Objekte unabhängig voneinander und sind austauschbar.

Im Zustandsmuster in Kapitel **6.10** werden die einzelnen Zustände eines Kontextobjekts 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 Zustandsänderungen durch, indem es ein anderes Zustandsobjekt als aktuellen Zustand referenziert.

6.12 Kontrollfragen

Aufgaben 6.12.1: Schablonenmethode

- 6.12.1.1 Was hat das Muster Schablonenmethode als Inhalt?
- 6.12.1.2 Arbeitet das Muster Schablonenmethode klassenbasiert oder objektbasiert?

Aufgaben 6.12.2: Befehl

- 6.12.2.1 Was ist das Ziel des Befehlsmusters?
- 6.12.2.2 Nennen Sie 2 Anwendungsgebiete für das Befehlsmuster.

Aufgaben 6.12.3: Beobachter

- 6.12.3.1 Erklären Sie das Prinzip einer Callback-Schnittstelle anhand des Beobachter-Musters.
- 6.12.3.2 Kennt der Beobachtete beim Beobachter-Muster die komplette Implementierung seiner Beobachter?

Aufgaben 6.12.4: Strategie

- 6.12.4.1 Wozu wird das Strategie-Muster eingesetzt?
- 6.12.4.2 Erklären Sie den Unterschied zwischen den Verhaltensmustern Strategie und Schablonenmethode.

Aufgaben 6.12.5: Vermittler

- 6.12.5.1 Was ist das Ziel des Vermittler-Musters?
- 6.12.5.2 Zeichnen Sie die für das Vermittler-Muster zutreffende Topologie seiner Komponenten.

Aufgaben 6.12.6: Zustand

- 6.12.6.1 Was ist das Ziel des Zustandsmusters?
- 6.12.6.2 Hängt der Kontext zur Kompilierzeit (statisch) von den konkreten Zustandsklassen ab?

Aufgaben 6.12.7: Rolle

- 6.12.7.1 Definieren Sie den Begriff einer Rolle.
- 6.12.7.2 Was ist das Ziel des Rollenmusters?
- 6.12.7.3 Wie wird beim Rollenmuster erreicht, dass ein Objekt zur Laufzeit verschiedene Rollen annehmen kann?
- 6.12.7.4 Was ist der Unterschied zwischen dem Rollen-Muster und der Spezialisierung durch Ableitung?

Aufgaben 6.12.8: Besucher

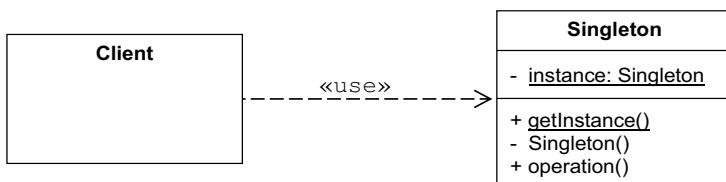
- 6.12.8.1 Was ist das Ziel des Besucher-Musters?
- 6.12.8.2 Beschreiben Sie die Aufgaben eines Besucher-Objekts.

Aufgaben 6.12.9: Iterator

- 6.12.9.1 Was ist das Ziel des Iterator-Musters?
- 6.12.9.2 Nennen Sie ein Entwurfsmuster, das dem Iterator-Muster ähnelt.
- 6.12.9.3 Was ist der grundlegende Gedanke des Iterator-Musters?

Kapitel 7

Erzeugungsmuster



- 7.1 Das Erzeugungsmuster Fabrikmethode
- 7.2 Das Erzeugungsmuster Abstrakte Fabrik
- 7.3 Das Erzeugungsmuster Singleton
- 7.4 Das Erzeugungsmuster Objektpool
- 7.5 Zusammenfassung
- 7.6 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_7.

7 Erzeugungsmuster

Der Begriff "Erzeugungsmuster" wurde von Gamma et al. geprägt [Gam94]. Erzeugungsmuster sind Entwurfsmuster.

Erzeugungsmuster machen ein System bei der Erzeugung von Objekten unabhängig von der gewünschten Spezialisierung einer Basisklasse



Wie in Kapitel 2.2.2 erwähnt betreffen Entwurfsmuster Verarbeitungsfunktionen. Diese existieren bereits im Problembereich, werden im Rahmen von Entwurfsmustern jedoch nur im Lösungsbereich betrachtet. Dies gilt auch für die im Folgenden betrachteten Erzeugungsmuster.

Kapitel 7.1 untersucht das Erzeugungsmuster Fabrikmethode, Kapitel 7.2 das Muster Abstrakte Fabrik und Kapitel 7.3 das Muster Singleton. Das Erzeugungsmuster Objektpool wird in Kapitel 7.4 beschrieben.

7.1 Das Erzeugungsmuster Fabrikmethode

7.1.1 Name/Alternative Namen

Fabrikmethode (engl. factory method), Virtueller Konstruktor.

7.1.2 Problem

Der Begriff Fabrikmethode wird in der Praxis häufig für eine Methode verwendet, welche die Logik und die Regeln für die Erzeugung eines neuen Objekts implementiert und eine Referenz auf das neue Objekt zurückgibt – vergleichbar einem Konstruktor. Beim Erzeugungsmuster Fabrikmethode nach GoF, das im Folgenden vorgestellt wird, ist entscheidend, dass die Fabrikmethode in Unterklassen implementiert wird (Dependency Inversion). Die so entstehenden Fabrikmethoden sind polymorph.

In [Bus07] wird daher zwischen **einfachen und polymorphen Fabrikmethoden** unterschieden. Ein Beispiel für eine einfache Fabrikmethode ist die Methode `getInstance()` im Erzeugungsmuster Singleton, siehe Kapitel 7.3.

Eine wiederverwendbare Anwendung wie ein Framework soll nur Basisklassen bzw. abstrakte Basisklassen oder Schnittstellen aus Methodenköpfen 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, mittels Dependency Inversion zur Verfügung gestellt werden.

Eine erzeugende Klasse soll zur Kompilierzeit nur die Basisklasse des zu erzeugenden Objekts kennen, soll aber zur Kompilierzeit nicht wissen, von welcher Unterklasse das entsprechende Objekt zur Laufzeit im lauffähigen Programm später sein soll.



Kurz gesagt: Bei einer Fabrikmethode wird die Erzeugung von Objekten in Unterklassen verlagert.



Java als stark typisierte Sprache legt beim Aufruf des `new`-Operators den Typ des zu erzeugenden Objekts im Programmtext statisch fest. Es gibt bei Verwendung des `new`-Operators keine Möglichkeit, den Typ eines erzeugten Objekts im Nachhinein zu verfeinern. Zur Initialisierung eines Objektes wird in Java der 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 darf im Falle von Java die Erzeugung auf keinen Fall direkt mit `new` erfolgen, falls der Typ der zu erzeugenden, abgeleiteten Objekte noch offen bleiben soll. Ein Programm soll überdies stabil bleiben, auch wenn sich der Typ des zu erzeugenden Objekts durch Spezialisierung 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, indem die Erzeugung eines Objekts in eine Methode verpackt wird, die polymorph ist und als Fabrikmethode bezeichnet wird.



Das **Muster Fabrikmethode** soll es erlauben, dass die Erzeugung einer konkreten Instanz in der Methode einer Unterklasse gekapselt wird.



7.1.3 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.



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 Erzeugerklasse 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` instanziert 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**.



7.1.3.1 Klassendiagramm

Das folgende Bild zeigt das Klassendiagramm des Fabrikmethode-Musters:

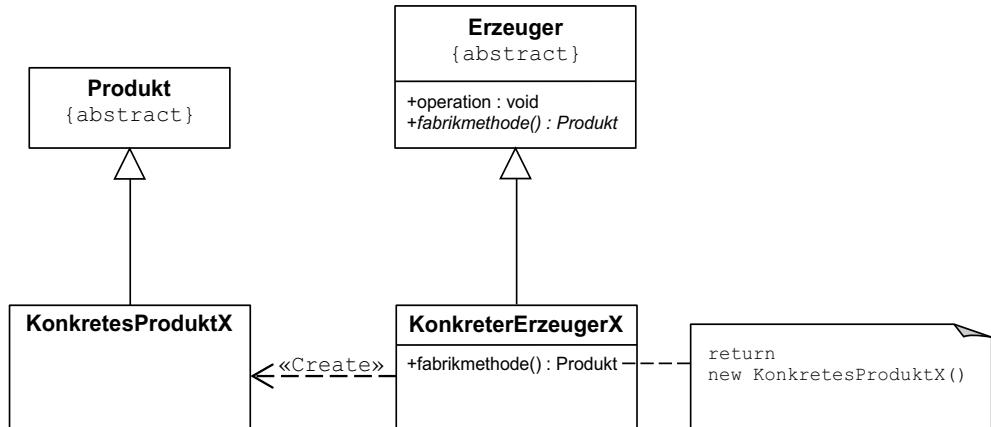


Bild 7-1 Klassendiagramm des Fabrikmethode-Musters mit abstrakten Klassen

Das Muster Fabrikmethode setzt voraus, dass die zu erzeugenden Produkte eine einheitliche Schnittstelle haben. Im Folgenden wird dazu eine abstrakte Klasse namens `Produkt` eingeführt.

Die Klasse `Produkt` muss nicht unbedingt abstrakt sein, sie muss nur die Basisklasse aller konkreten Produkte sein. Sie könnte beispielsweise auch eine Schnittstelle `IProdukt` sein, die von den konkreten Produkten implementiert wird.



Für jede konkrete 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 Rückgabetyp der Fabrikmethode ist vom Typ `Produkt` und die Fabrikmethode ist abstrakt, da hier der abgeleitete 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 zur Kompilierzeit nur die abstrakten Klassen und kennt die zu generierende konkrete Ausprägung des Produkts nicht.

Der spezialisierte Typ der zu erzeugenden Objekte ist also zur Kompilierzeit nicht bekannt.



Eine Anwendung des Musters Fabrikmethode – sie ist im Klassendiagramm nicht dargestellt – 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 konkreten Produkte, die bei Einhaltung der Verträge in der Anwendung nach dem liskovschen Substitutionsprinzip an die Stelle eines abstrakten Produkts treten.

7.1.3.2 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 wie z. B. 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 nach dem liskovschen Substitutionsprinzip auch auf ein Objekt vom Typ `KonkretesProduktX` verweisen.

7.1.3.3 Dynamisches Verhalten

Das folgende Bild gibt ein Beispiel für die Erzeugung konkreter Produkte:

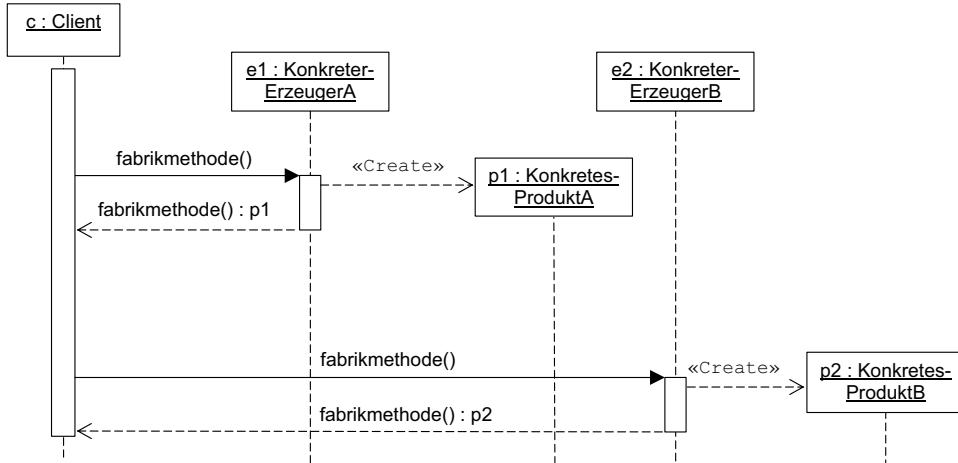


Bild 7-2 Sequenzdiagramm Fabrikmethode

Im Bild 7-2 ruft ein Client die Methode `fabrikmethode()` des ihm bekannten Erzeugers `e1` auf. Daraufhin wird von diesem Objekt `e1` der Klasse `Konkreter-ErzeugerA` 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, so wird eine Instanz des Typs `KonkretesProduktB` erzeugt. Je nachdem, welcher konkrete Erzeuger verwendet wird, werden also verschiedene Objekte erzeugt.

Der Client steht in Bild 7-2 stellvertretend für eine Anwendung wie beispielsweise ein Framework, welche konkrete Produkte flexibel erzeugen und nutzen will. Der Client ist nicht Bestandteil des Musters.

Das Muster lässt es offen, wie ein Client Kenntnis über einen konkreten Erzeuger erlangt.



7.1.3.4 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 {

    public abstract Produkt createProdukt();
}
  
```

In der Unterklasse `KonkreterErzeugerA` werden Objekte der Klasse `Konkretes-ProduktA` erzeugt:

```
// Datei: KonkreterErzeugerA.java

public class KonkreterErzeugerA extends Erzeuger {

    @Override
    public Produkt createProdukt() {
        return new KonkretesProduktA();
    }
}
```

In der Unterklasse KonkreterErzeugerB werden Objekte der Klasse KonkretesProduktB erzeugt:

```
// Datei: KonkreterErzeugerB.java

public class KonkreterErzeugerB extends Erzeuger {

    @Override
    public Produkt createProdukt() {
        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 {

    public abstract void print();
}
```

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 static final String NAME = "ProduktA";

    @Override
    public void print() {
        System.out.println("name = " + NAME);
    }
}
```

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 static final String NAME = "ProduktB";
```

```

@Override
public void print() {
    System.out.println("name = " + NAME);
}
}

```

In der Klasse `TestErzeuger` werden zwei konkrete Erzeuger instanziert, mit deren Hilfe konkrete Produkte durch die Fabrikmethode erstellt werden:

// Datei: `TestErzeuger.java`

```

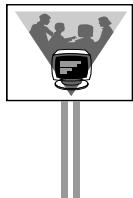
public class TestErzeuger {

    public static void main(String[] args) {
        Erzeuger erzeuger;
        Produkt produkt;

        System.out.println("Erzeuger A");
        erzeuger = new KonkreterErzeugerA();
        produkt = erzeuger.createProdukt();
        produkt.print();

        System.out.println("\nErzeuger B");
        erzeuger = new KonkreterErzeugerB();
        produkt = erzeuger.createProdukt();
        produkt.print();
    }
}

```



Hier das Protokoll des Programmlaufs:

```

Erzeuger A
name = ProduktA

Erzeuger B
name = ProduktB

```

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 Erzeugerklasse. Das Muster lässt offen, wie diese Referenz so gesetzt werden kann, dass sie auf ein Objekt einer konkreten Erzeugerklasse zeigt. Hier kann beispielsweise die Technik der **Dependency Injection**⁹⁴ eingesetzt werden.

7.1.4 Bewertung

7.1.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

⁹⁴ Siehe Kapitel 3.1.3.

- **Anwendungen müssen die konkreten Produktklassen nicht kennen**

Eine Anwendung muss die konkreten Produktklassen der Objekte, die sie erzeugen muss, nicht von vornherein kennen, sondern nur deren Basisklasse. Denn eine explizite Erzeugung von konkreten Produkten – beispielsweise durch einen expliziten Konstruktoraufruf – ist nicht in der Anwendung enthalten. Dadurch kann ein konkreter Erzeuger gegen einen anderen ausgetauscht werden etwa mit Hilfe von Dependency Injection (siehe Kapitel 3.1.3), um damit andere konkrete Produkte für die Anwendung zu erzeugen. Der Einsatz von Fabrikmethoden ist damit flexibler als explizite Erzeugungsanweisungen.

- **Default-Erzeugung möglich**

Sollen die Basisklassen nicht nur abstrakte Methoden enthalten, kann man in der Basisklasse **Erzeuger** eine Default-Erzeugung definieren. Dabei sind die Unterklassen frei, ob sie eine eigene überschreibende Fabrikmethode bereitstellen oder nicht.

- **Sicherheit beim Aufruf**

Durch die Verwendung von Schnittstellen bzw. abstrakten Klassen für Produkte kann das Einbinden von "falschen" anwendungsspezifischen Klassen verhindert werden.

- **konkrete Produktklassen zunächst nicht erforderlich**

Basierend auf den Schnittstellen bzw. abstrakten Klassen für Produkte kann ein Framework bzw. eine Klassenbibliothek entwickelt werden, um Produkte zu bearbeiten, auch wenn noch gar keine spezialisierten, konkreten Produktklassen vorhanden sind. Eine Anwendung des Frameworks kann dann konkrete Produkte, welche 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.

7.1.4.2 Nachteile

Folgende Nachteile werden gesehen:

- **Komplexität erhöht**

Für jede neue Produktklasse muss eine weitere Unterklasse eingeführt werden, um die Fabrikmethode entsprechend zu redefinieren. Damit erhöht sich die Komplexität.

- **Verschlechterung der Performance**

Aus Performance-Gründen kann es sinnvoller sein, Objekte direkt zu erzeugen und nicht die Fabrikmethode einzusetzen.

7.1.5 Einsatzgebiete

Es müssen Objekte einer Klasse erzeugt werden, deren spezialisierter 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 7-1) aufgerufen, dann stellt in dieser Situation die Fabrikmethode eine Einschubmethode und die Methode `operation()` eine Schablonenmethode aus dem Muster Schablonenmethode dar.

Die Methode `createIterator()` aus Kapitel 6.4, 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.

7.1.6 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.

Ähnlich wie das Muster Fabrikmethode verfolgt das Architekturmuster **Plug-in** das Ziel, ein gewünschtes Objekt erst dynamisch zur Laufzeit zu erzeugen, ohne dass der Typ des Objektes statisch festgelegt ist. Allerdings wird die Erzeugung der Objekte beim Muster Plug-in nicht in Unterklassen ausgelagert, sondern ein Plug-in-Manager ist zentral dafür verantwortlich.

7.2 Das Erzeugungsmuster Abstrakte Fabrik

7.2.1 Name/Alternative Namen

Abstrakte Fabrik (engl. abstract factory, kit oder toolkit).

7.2.2 Problem

Die Problemstellung soll zunächst anhand von zwei Beispielen verdeutlicht werden:

Im ersten Beispiel werden Schrauben und Muttern, die es in unterschiedlichen Größen und Gewindeformen gibt, produziert und verpackt. Da es aber keinen Sinn macht, 10er-Schrauben und 6er-Muttern miteinander zu verpacken, sollen immer nur passende Produkte – also 6er-Schrauben und 6er-Muttern bzw. 10er-Schrauben und 10er-Muttern – erzeugt werden können. 6er-Schrauben und 6er-Muttern bilden in diesem Beispiel eine zusammengehörige Produktgruppe, eine andere Produktgruppe besteht aus 10er-Schrauben und 10er-Muttern.

Als zweites Beispiel wird eine Anwendung betrachtet, die mehrere Objekte benötigt, deren Klassen für verschiedene Betriebssysteme implementiert werden müssen. Um diese Anwendung lauffähig zu machen, müssen alle erzeugten Klassen in der jeweils richtigen Variante instanziert werden. Es muss ausgeschlossen werden, dass eine der erzeugten Klassen beispielsweise in der Linux-Variante instanziert wird und eine andere hingegen in der Windows-Variante.

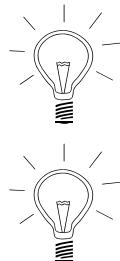
Eine Menge oder Gruppe von Produkten, die zusammengehörig sind oder – allgemein gesprochen – voneinander abhängig sind, wird in diesem Muster als **Produktfamilie** bezeichnet.

Es soll gewährleistet werden, dass immer nur Produkte, die zu derselben Produktfamilie gehören, erzeugt werden. Eine Anwendung sollte selber nicht geändert werden müssen, wenn eine andere Variante der Produktfamilie erzeugt werden soll.

7.2.3 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 Schnittstellen. 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 Schnittstellen 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.

7.2.3.1 Klassendiagramm

Das folgende Bild enthält das Klassendiagramm des Entwurfsmusters Abstrakte Fabrik:

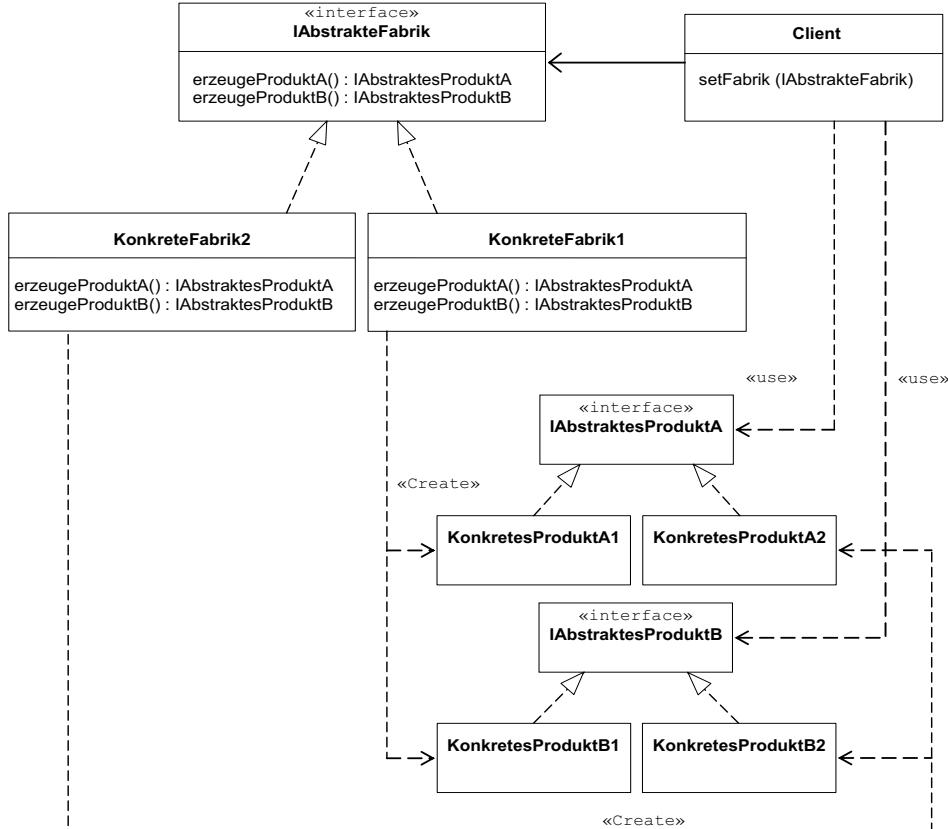


Bild 7-3 Klassendiagramm der Abstrakten Fabrik

Das Klassendiagramm in Bild 7-3 enthält die Schnittstelle **IAbstrakteFabrik**, welche die Schnittstelle für die Erzeugung der Produkte einer Produktfamilie darstellt.

Konkrete Fabriken implementieren die Schnittstelle **IAbstrakteFabrik**. Jede dieser konkreten Fabriken erzeugt **Produkte einer bestimmten Produktfamilie**.



Die **erzeugten Produkte** werden **Produktarten** genannt. Die Summe aller Produktarten, die zusammengehörig sind, gehören zu einer **Produktfamilie**.



So gehören die Produktarten Schrauben und Muttern zu einer Produktfamilie. Die konkreten Produktarten 6-er Schrauben und 6-er Muttern gehören zu einer konkreten Produktfamilie und die konkreten Produktarten 10-er Schrauben und 10-er Muttern zu einer anderen konkreten 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 7.1) 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**, welche die konkreten Objekte erzeugen und damit den **Typ der zu erzeugenden Produkte festlegen**.



Die konkreten Produktklassen `KonkretesProduktA1` und `KonkretesProduktA2` realisieren die Schnittstelle `IAbstraktesProduktA`. Analog dazu gibt es die Produktklassen `KonkretesProduktB1` und `KonkretesProduktB2`, welche die Schnittstelle `IAbstraktesProduktB` realisieren.

Ein Client kennt nur die Schnittstellen der Produkte. Im Programm 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 Konzept **Dependency Injection**⁹⁵ mit Hilfe einer Setter-Methode. Das Programm kennt die entsprechende konkrete Fabrik nicht und hängt damit nicht von deren Ausprägung ab.

7.2.3.2 Teilnehmer

Die Teilnehmer des Entwurfsmusters Abstrakte Fabrik werden im Folgenden kurz vorgestellt:

- **IAbstraktesProduktX**

Die Schnittstelle `IAbstraktesProduktX` ($X = A \dots Z$) definiert die Schnittstelle einer **Produktart X** und damit die Schnittstelle für ein bestimmtes konkretes Produkt dieser Art.

⁹⁵ Siehe Kapitel 3.1.3.

- **IAbstrakteFabrik**

Die Schnittstelle `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 die Schnittstelle `AbstraktesProduktX`. 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 die Schnittstelle `IAbstrakteFabrik` und instanziert in den Erzeugungsmethoden konkrete Produkte, die zur Produktfamilie Y gehören.

- **Client**

Ein Client benutzt die Schnittstellen der abstrakten Fabrik und der abstrakten Produkte. Er besitzt eine Referenz, welche zur Laufzeit auf eine konkrete Fabrik zeigt, um die Produkte einer Produktfamilie zu erzeugen.

7.2.3.3 Dynamisches Verhalten

Das dynamische Verhalten der Teilnehmer des Musters Abstrakte Fabrik wird mit Hilfe des Sequenzdiagramms in Bild 7-4 beschrieben:

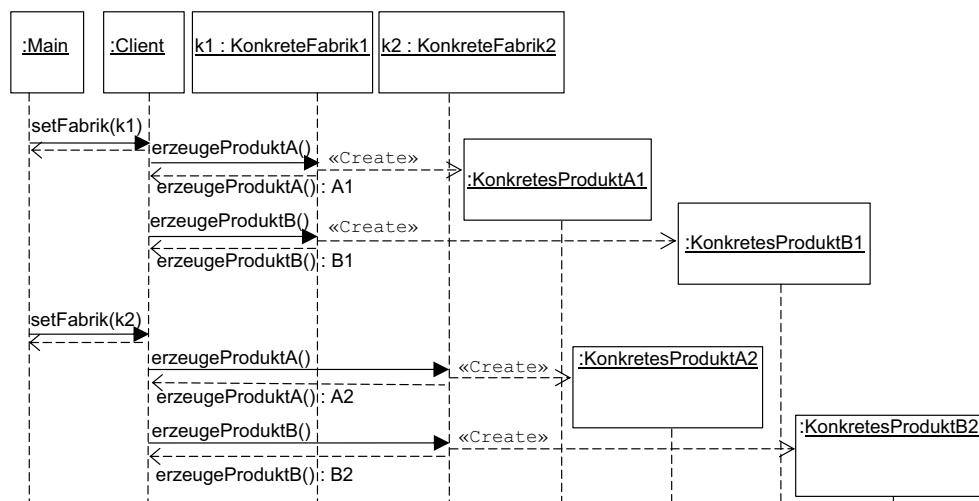


Bild 7-4 Sequenzdiagramm des Musters Abstrakte Fabrik

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.

7.2.3.4 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 die Schnittstelle `IMutter` und die konkreten Produkte, nämlich Muttern mit Gewinde M6 bzw. mit Gewinde M10, vorgestellt:

```
// Datei: IMutter.java

public interface IMutter {

    String gibDetails();
}

// Datei: MutterM6.java

public class MutterM6 implements IMutter {

    @Override
    public String gibDetails() {
        return "Mutter mit M6-Gewinde.";
    }
}

// Datei: MutterM10.java

public class MutterM10 implements IMutter {

    @Override
    public String gibDetails() {
        return "Mutter mit M10-Gewinde.";
    }
}
```

```
    }  
}
```

Um die Muttern auch entsprechend nutzen zu können, sind Schrauben erforderlich. Zuerst wird die Schnittstelle `ISchraube` gezeigt und dann die konkreten Produkte, nämlich Schrauben mit Gewinde M6 und mit M10:

```
// Datei: ISchraube.java  
  
public interface ISchraube {  
  
    String gibDetails();  
  
}  
  
// Datei: SchraubeM6.java  
  
public class SchraubeM6 implements ISchraube {  
  
    @Override  
    public String gibDetails() {  
        return "Schraube mit M6-Gewinde.";  
    }  
}  
  
// Datei: SchraubeM10.java  
  
public class SchraubeM10 implements ISchraube {  
  
    @Override  
    public String gibDetails() {  
        return "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 vom Typ IMutter bzw. ISchraube.  
public interface IAbstrakteFabrik {  
  
    ISchraube erstelleSchraube();  
  
    IMutter erstelleMutter();  
  
}
```

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 Gewindegroßen M6 und M10 werden im Folgenden gezeigt:

```
// Datei: KonkreteFabrikM6.java

public class KonkreteFabrikM6 implements IAbstrakteFabrik {

    @Override
    public ISchraube erstelleSchraube() {
        return new SchraubeM6();
    }

    @Override
    public IMutter erstelleMutter() {
        return new MutterM6();
    }
}

// Datei: KonkreteFabrikM10.java

public class KonkreteFabrikM10 implements IAbstrakteFabrik {

    @Override
    public ISchraube erstelleSchraube() {
        return new SchraubeM10();
    }

    @Override
    public IMutter erstelleMutter() {
        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 der Klasse `Schachtel` gibt es keine Bezüge auf konkrete Produkte. 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

import java.util.*;
import java.util.stream.Collectors;

public class Schachtel {

    private final int groesse;
    private final List<ISchraube> schrauben = new ArrayList<>();
    private final List<IMutter> muttern = new ArrayList<>();

    public Schachtel(int groesse) {
        this.groesse = groesse;
    }

    public int gibGroesse() {
        return groesse;
    }
```

```

public void addSchraube(ISchraube schraube) {
    if (schrauben.size() < gibGroesse()) {
        schrauben.add(schraube);
    }
}

public void addMutter(IMutter mutter) {
    if (muttern.size() < gibGroesse()) {
        muttern.add(mutter);
    }
}

@Override
public String toString() {
    return schrauben.stream().map(ISchraube::gibDetails)
        .collect(Collectors.joining("\n"))
        + '\n'
        + muttern.stream().map(IMutter::gibDetails)
        .collect(Collectors.joining("\n"));
}
}

```

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:

```

// Datei: ProduktionsMaschine.java

public class ProduktionsMaschine {

    private IAbstrakteFabrik fabrik;

    public void setzeFabrik(IAbstrakteFabrik fabrik) {
        this.fabrik = fabrik;
    }

    public void fuelleSchachtel(Schachtel schachtel) {
        for (int i = 0; i < schachtel.gibGroesse(); ++i) {
            schachtel.addSchraube(fabrik.erstelleSchraube());
            schachtel.addMutter(fabrik.erstelleMutter());
        }
    }
}

```

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 {

    public static void main(String[] args) {
        ProduktionsMaschine maschine = new ProduktionsMaschine();
        Schachtel schachtel1 = new Schachtel(5);
        Schachtel schachtel2 = new Schachtel(3);

        maschine.setzeFabrik(new KonkreteFabrikM6());
        maschine.fuelleSchachtel(schachtel1);
        System.out.println(schachtel1);

        System.out.println();
        maschine.setzeFabrik(new KonkreteFabrikM10());
        maschine.fuelleSchachtel(schachtel2);
        System.out.println(schachtel2);
    }
}
```



Hier das Protokoll des Programmlaufs:

```
Schraube 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.
```

7.2.4 Bewertung

7.2.4.1 Vorteile

Folgende Vorteile werden gesehen:

- **Abkopplung von den konkreten Implementierungen**

Die konkreten Klassen werden durch abstrakte Klassen bzw. Schnittstellen ohne Methodenköpfe isoliert. Der Vorteil ist, dass eine Anwendung, welche die erstellten Objekte benutzt, lediglich deren Schnittstellen kennt und so eine Abkopplung von den konkreten Implementierungen möglich ist.

- **leichter Austausch der Produktfamilie**

Produktfamilien können leicht ausgetauscht werden. Es muss nur eine einzige Referenz so geändert werden, dass sie auf eine andere konkrete Fabrik zeigt.

- **getrennte Erstellung von Produktfamilien**

Solange die Referenz auf die konkrete Fabrik nicht geändert wird, ist sichergestellt, dass in diesem Zeitraum nur eine einzige Produktfamilie mit zusammengehörigen Produkten verwendet wird.

- **leichte Einführung neuer konkreter Fabriken**

Eine neue Produktfamilie kann flexibel hinzugefügt werden, ohne dass die Anwendung geändert werden muss – vorausgesetzt die Schnittstellen werden nicht geändert. Es muss nur eine neue konkrete Fabrik implementiert werden.

- **konkrete Fabriken und Produkte werden erst zur Laufzeit festgelegt**

Bei Nutzung des Musters Abstrakte Fabrik möchte der Client seine Objekte – sprich Produkte – erzeugen können, ohne zur Kompilierzeit die konkreten Produkte 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 Schnittstellen `IAbstrakteFabrik` und `IAbstraktesProduktX` zu.

7.2.4.2 Nachteile

Der folgende Nachteil wird gesehen:

- **Aufwand bei der Erweiterung der Produktfamilie**

Neue Produktarten lassen sich nur mit Aufwand zu den Produktfamilien hinzufügen, da eine neue Produktart in der Schnittstelle 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.

7.2.5 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 Produkte 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.

7.2.6 Ähnliche Entwurfsmuster

Eine **Abstrakte Fabrik** hat eine analoge Wirkung wie eine **Fassade**: die Abstrakte Fabrik bietet einen **vereinfachten Zugang** zum Erzeugen von Gruppen untereinander abhängiger Produkte 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 Sequenzen aufeinanderfolgender Methodenaufrufe in einem einzigen Methodenaufruf 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 Mustern 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.

7.3 Das Erzeugungsmuster Singleton

7.3.1 Name/Alternative Namen

Singleton (dt. Einzelstück, dieser Name wird selten verwendet).

7.3.2 Problem

Es soll eine Klasse entworfen werden, von der nur eine einzige Instanz erzeugt werden kann.

Das **Singleton-Muster** soll es gewährleisten, dass eine Klasse nur ein einziges Mal instanziert werden kann.



Die im Folgenden als `Singleton` bezeichnete Klasse stellt sicher, 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 `Singleton`-Klasse selbst verantwortlich sein.

Ein Singleton-Objekt kann nur von der Klasse `Singleton` selbst erzeugt werden.



7.3.3 Lösung

Das Singleton-Muster lässt sich aufgrund seiner einfachen Struktur zwar leicht erklären, seine Implementierung kann aber dennoch komplex sein je nachdem, in welchem Kontext das Muster eingesetzt werden soll. Vorgestellt werden in diesem Kapitel die Implementierungsansätze von Gamma et al. [Gam94], von Grand [Gra02a] und von Bloch [Blo17].

Ein Client, der mit dem Singleton-Objekt arbeiten will, bekommt eine Referenz auf dieses Singleton-Objekt.



Das Muster Singleton ist somit ein **objektbasiertes Erzeugungsmuster**.

Die Klasse `Singleton` darf keine nicht privaten Konstruktoren besitzen. Damit wird verhindert, dass andere Klassen ein Singleton-Objekt unter Verwendung eines Konstruktors erzeugen können.



Für die **Erzeugung des einzigen Exemplars der Klasse `Singleton`** durch die Klasse `Singleton` selbst gibt es drei unterschiedliche Vorgehensweisen:

Bei der **ersten Variante der Klasse Singleton**, wie sie bei **Gamma et al. [Gam94]** zu finden ist, wird das **Singleton-Objekt dann erzeugt, wenn ein Client es zum ersten Mal anfordert.**



Die **zweite Variante für eine Singleton-Klasse** ist unter anderem bei **Grand [Gra02a]** zu finden. Dabei wird das Singleton-Objekt als Klassenvariable **beim Laden der Klasse angelegt und direkt initialisiert.**



In der **dritten Variante wird die Singleton-Klasse als Enum-Klasse⁹⁶ realisiert** [Blo17]. Bei einer Enum-Klasse ist sichergestellt, dass die Enum-Objekte, welche in einer solchen Klasse deklariert werden, nur ein einziges Mal existieren.



Alle drei Implementierungsvarianten werden mit ihren jeweiligen Vor- und Nachteilen in Kapitel 7.3.3.4 ausführlich beschrieben. Die folgende Erklärung des Singleton-Musters basiert auf der Variante von Gamma et al. Für diese Variante gilt:

Clients, welche das Singleton-Objekt verwenden möchten, erhalten von der Klassenmethode `getInstance()` der Klasse `Singleton` eine Referenz auf das einzig existierende Objekt der Klasse `Singleton` zurück.



7.3.3.1 Klassendiagramm für eine Singleton-Klasse

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.

Die Methode `getInstance()` der Klasse `Singleton` muss eine **Klassenmethode** sein, da Client-Objekte bereits auf diese Methode zugreifen können müssen, wenn noch kein Singleton-Objekt existiert.



⁹⁶ Diese Variante setzt voraus, dass in der gewählten Programmiersprache ein entsprechendes Konstrukt zur Verfügung steht, wie es beispielsweise in Java der Fall ist.

Das folgende Bild zeigt das Klassendiagramm:

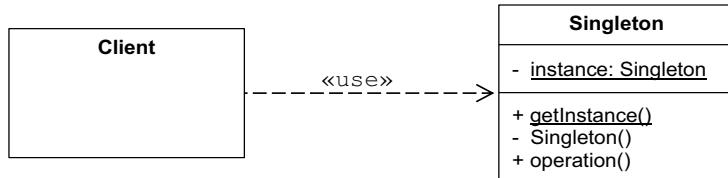


Bild 7-5 Klassendiagramm für das Singleton-Muster

Ein Client nutzt das Singleton-Objekt und ruft Operationen des Singleton-Objekts auf. Stellvertretend für die Funktionalität des Singleton-Objektes steht im Klassendiagramm die Methode `operation()`. Die Klasse `Singleton` muss für den Client sichtbar sein. Man kann vom Client zum Singleton-Objekt gelangen, die Klasse `Singleton` kennt aber den Client nicht.

7.3.3.2 Teilnehmer

Das Erzeugungsmuster Singleton umfasst die folgenden Teilnehmer:

- **Client**

Der Client benötigt das Objekt der Klasse `Singleton`, um dessen Funktionalität zu nutzen.

- **Singleton**

Die Klasse `Singleton` ist diejenige Klasse, von der es nur eine einzige Instanz geben darf.

7.3.3.3 Dynamisches Verhalten

Ein Client, der mit dem Singleton-Objekt arbeiten will, ruft die Klassenmethode `getInstance()` der Klasse `Singleton` auf.

Der Rückgabewert der Methode `getInstance()` ist eine Referenz auf die neu erstellte Instanz der Klasse `Singleton` oder – wenn diese Instanz bereits existiert – auf das einzige, bereits vorhandene Exemplar der Klasse `Singleton`.



Wenn der Client eine Referenz auf die Instanz der Klasse `Singleton` hat, kann er Operationen auf dieser Instanz aufrufen, hier dargestellt durch die Aufrufe der Methode `operation()`:

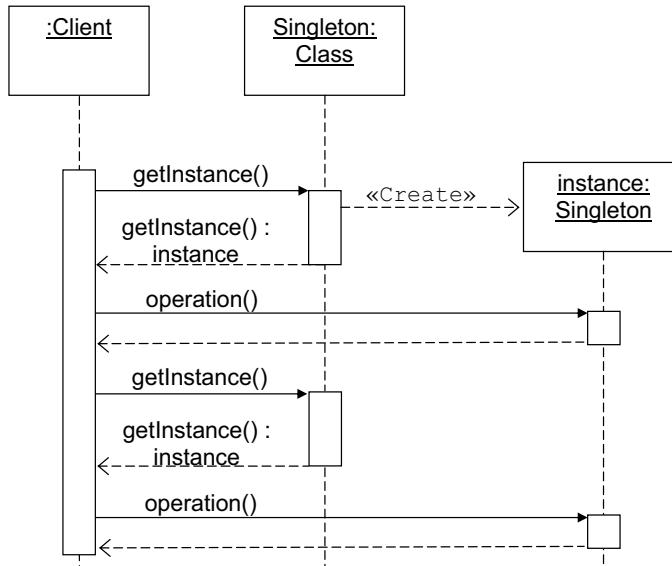


Bild 7-6 Sequenzdiagramm für das Abholen einer Referenz auf das Singleton-Objekt

Die Instanz `Singleton:Class` stellt das Klassenobjekt der Klasse `Singleton` dar.

7.3.3.4 Programmbeispiele

Im Folgenden werden die drei bereits genannten Implementierungsvarianten der Reihe nach beschrieben und miteinander verglichen. Als Beispiel dient eine typische Singleton-Anwendung, nämlich ein Tooltip-Manager einer grafischen Benutzeroberfläche. Ein Tooltip-Manager 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 Tooltip-Manager geben. Daher muss die Klasse des Tooltip-Managers als Singleton ausgelegt werden.

Implementierungsvariante 1 (Gamma et al.)

Die Implementierung der Klasse `ToolTipManager1` folgt der ersten Variante, dem Vorschlag von Gamma et al. [Gam94], und erzeugt das Singleton-Objekt bei Bedarf in der Methode `getInstance()`, wie es auch im Sequenzdiagramm (siehe Bild 7-6) dargestellt ist.

Hier die Klasse `ToolTipManager1`:

```
// Datei: ToolTipManager1.java

public final class ToolTipManager1 {
    private static ToolTipManager1 instance;
```

```
private ToolTipManager1() {
    System.out.println("Neues Singleton erzeugt.");
}

public static ToolTipManager1 getInstance() {
    if (instance == null) {
        // Erzeugung des Singleton.
        // Wenn die Methode getInstance() nicht aufgerufen wird,
        // wird es auch nicht angelegt.
        instance = new ToolTipManager1();
    }
    return instance;
}

public void operation() {
    // Eigentliche Funktionalität des Singleton
    System.out.println("operation() wurde aufgerufen.");
}
}
```

Im Quellcode der Klasse `ToolTipManager1` ist zu sehen, dass die Singleton-Klasse zur Verwirklichung der Singleton-Funktionalität eine Referenz `instance` auf das Singleton-Objekt beinhaltet. Diese Referenz muss eine Klassenvariable sein (static in Java), da auf sie in der Klassenmethode `getInstance()` zugegriffen wird.

Ein Aufruf der Methode `operation()` sieht damit folgendermaßen aus:

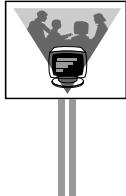
```
ToolTipManager1.getInstance().operation();
```

Hierbei wird über den Klassennamen `ToolTipManager1` auf die Klassenmethode `getInstance()` zugegriffen, um eine Referenz auf das Singleton-Objekt zu erhalten. Über diese Referenz wird dann die Instanzmethode `operation()` aufgerufen, welche die eigentliche Funktionalität des Singleton-Objekts bereitstellt. Die im Folgenden dargestellte Klasse `Client1` ruft die Methode `operation()` auf diese Art und Weise dreimal hintereinander auf:

```
// Datei: Client1.java

public class Client1 {

    public static void main(String[] args) {
        ToolTipManager1.getInstance().operation(); // Laden der Klasse
        ToolTipManager1.getInstance().operation();
        ToolTipManager1.getInstance().operation();
    }
}
```



In der Programmausgabe sieht man, dass das Singleton-Objekt nur ein einziges Mal erzeugt wird:

Neues Singleton erzeugt.
operation() wurde aufgerufen.
operation() wurde aufgerufen.
operation() wurde aufgerufen.

Der interessanteste Teil einer Singleton-Implementierung ist der Rumpf der Methode `getInstance()` sowie die Definition der Referenz `instance`. An diesen beiden Stellen unterscheiden sich die Implementierungsvarianten. Die gerade vorgestellte Variante von Gamma et. al. erzeugt das Singleton-Objekt erst dann, wenn die Methode `getInstance()` aufgerufen wird. Der Nachteil liegt darin, dass bei weiteren Aufrufen der Methode `getInstance()` jedes Mal der Vergleich (`instance == null`) ausgeführt werden muss, der aber nach dem ersten Aufruf immer negativ ausfällt.

Ein weiterer Nachteil der Implementierung der Klasse `ToolTipManager1` zeigt sich in einem Kontext, in dem mehrere, gleichzeitig aktive Threads das Singleton-Objekt benutzen wollen und die Methode `getInstance()` gleichzeitig aufrufen.

Der Code der Klasse `ToolTipManager1` ist nicht **thread-safe**. Der gleichzeitige Aufruf von `getInstance()` aus mehreren Threads als Clients kann dazu führen, dass in bestimmten Konstellationen⁹⁷ doch mehrere Objekte erzeugt werden.



Die Implementierung der Klasse `ToolTipManager1` funktioniert nur korrekt mit einem einzigen Thread als Client, da es bei einem einzigen Thread trivialerweise keine Synchronisationsprobleme gibt. Um die Threadsicherheit im Falle von mehreren Threads zu gewährleisten, müsste die Methode `getInstance()` in Java mit dem Zusatz `synchronized` gekennzeichnet werden. Dies wird im Folgenden gezeigt:

```
public synchronized static ToolTipManager1 getInstance() {
    if (instance == null) {
        instance = new ToolTipManager1();
    }
    return instance;
}
```

Die Kennzeichnung mit dem Schlüsselwort `synchronized` hat allerdings den Nachteil, dass die Aufrufe von `getInstance()` relativ langsam sind [shespj]. Dies ist insbesondere deshalb nachteilig, weil die Synchronisation eigentlich nur für den ersten Aufruf von `getInstance()` nötig wäre.

⁹⁷ Da das Verhalten von Anwendungen mit mehreren Threads nicht deterministisch ist, gibt es auch Konstellationen – sprich zeitliche Abläufe, Unterbrechungen und Wechsel von Threads –, in denen keine Probleme auftreten.

Implementierungsvariante 2 (Grand)

Ist der (fast immer überflüssige) Vergleich und die damit einhergehende Verschlechterung der Performance nicht erwünscht, so kann auf die zweite Variante des Singleton-Musters [Gra02a] zurückgegriffen werden.

Der Vorteil bei dieser Variante ist, dass der oben genannte Vergleich in der Methode `getInstance()` gänzlich entfällt. Die Klasse `ToolTipManager2` zeigt diese Variante:

```
// Datei: ToolTipManager2.java

public final class ToolTipManager2 {

    private static final ToolTipManager2 instance =
        new ToolTipManager2(); // Das Singleton wird bereits beim
                              // Anlegen der Klassenvariable erzeugt.

    private ToolTipManager2() {
        System.out.println("ToolTipManager2 erzeugt.");
    }

    public static ToolTipManager2 getInstance() {
        System.out.println("ToolTipManager2.getInstance() aufgerufen.");
        return instance;
    }

    public void operation() {
        // Eigentliche Funktionalität des Singleton
        System.out.println("operation() wurde aufgerufen.");
    }
}
```

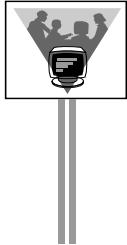
Es ist zu sehen, dass sich die Vereinbarung der Klassenvariable `instance` von der ersten Variante unterscheidet. Die Klassenvariable `instance` wird nun direkt beim Anlegen gleichzeitig auch initialisiert und dabei das Singleton-Objekt erzeugt. Zur Verdeutlichung, dass sich die gespeicherte Referenz nicht mehr ändert, wurde die Klassenvariable `instance` als `final` deklariert.

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(); // Laden der Klasse
        ToolTipManager2.getInstance().operation();
        ToolTipManager2.getInstance().operation();
    }
}
```



Die Ausgabe des Programms ist:

```
ToolTipManager2 erzeugt.  
ToolTipManager2.getInstance() aufgerufen.  
operation() wurde aufgerufen.  
ToolTipManager2.getInstance() aufgerufen.  
operation() wurde aufgerufen.  
ToolTipManager2.getInstance() aufgerufen.  
operation() wurde aufgerufen.
```

Der Quellcode der Klasse `ToolTipManager2` wurde so gestaltet, dass er möglichst ähnlich zu dem der Klasse `ToolTipManager1` in Variante 1 ist. Für die Klasse `ToolTipManager2` bieten sich aber noch Verbesserungsmöglichkeiten an:

Die Klasse `ToolTipManager2` kann bezüglich des Rechenaufwandes noch verbessert werden, in dem man die Klassenvariable `instance` öffentlich macht. Dann kann nämlich die Methode `getInstance()` komplett entfallen.



Diese zweite Implementierungsvariante ist **ohne weitere Maßnahmen threadsicher**. D. h., dass die Methode `getInstance()` der Klasse `ToolTipManager2` problemlos von mehreren Threads aufgerufen werden kann, da das Singleton-Objekt bereits beim Laden der Klasse⁹⁸ angelegt wird. Es spielt bei dieser Implementierungsvariante also keine Rolle, ob `getInstance()` unterbrochen wird.

Die Threadsicherheit soll durch das folgende Programm demonstriert werden, bei dem mehrmalige Aufrufe der Methode `getInstance()` in verschiedenen Threads zu sehen sind. Die folgende Klasse `SingletonTestThread` implementiert das Verhalten der Threads:

```
// Datei: SingletonTestThread.java

public class SingletonTestThread extends Thread {

    private final String threadName;

    public SingletonTestThread(String threadName) {
        this.threadName = threadName;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(500);
            System.out.println(threadName + " - call 1");
        }
    }
}
```

⁹⁸ In Java wird eine Klasse vom Class Loader der Java Virtual Machine (JVM) geladen, wenn sie das erste Mal verwendet wird – beispielsweise wenn eine statische Methode der Klasse aufgerufen oder auf statische Felder zugegriffen wird. Beim Laden der Klasse werden dann **alle statischen Felder** von oben nach unten initialisiert. Erst wenn dieser Vorgang komplett abgeschlossen ist, kann auf die Elemente der Klasse zugegriffen werden.

```
    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 Hauptprogramm werden 3 Threads als Clients angelegt und gestartet. Erzeugt wird ein Thread in Java 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 der Klasse `ToolTipManager2` und ruft jedesmal die Methode `operation()` der zurückgegebenen Instanz auf.

Da in diesem Programm mehrere Threads gleichzeitig und unsynchronisiert voneinander ablaufen, ist das Protokoll einer Ausführung nicht vorhersagbar – also nicht-deterministisch. Das Programm erzeugt bei einer erneuten Ausführung nicht zwangsläufig die gleiche Ausgabe. Die im Folgenden gezeigte Ausgabe ist also nur eine Möglichkeit unter vielen:



Eine mögliche Programmausgabe:

```
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 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.
```

Implementierungsvariante 3 (Bloch)

Bei dieser Variante von [Blo17] wird eine **Enum-Klasse**, wie sie beispielsweise in Java zur Verfügung steht, für die Realisierung der Singleton-Klasse verwendet. Durch die Definition als Enum-Klasse garantiert die Java Virtual Machine (JVM), dass die in der Klasse deklarierten Enum-Objekte nur höchstens einmal instanziert werden. Die statische Methode `getInstance()` und der Konstruktor sind in dieser Variante nicht erforderlich.

Die Java Virtual Machine (JVM) sorgt dafür, dass jedes Enum-Objekt einer Enum-Klasse höchstens einmal angelegt wird. Genau wie bei Variante 2 erfolgt dieser Schritt beim Laden der Klasse.



Hier nun der Tooltip-Manager als Enum-Klasse:

```
// Datei: ToolTipManager3.java

public enum ToolTipManager3 {

    // Singleton-Instanz als Enum-Objekt:
    INSTANCE;
```

```
ToolTipManager3() {  
    // Der Konstruktor dient nur der Veranschaulichung des Beispiels.  
    // Ob überhaupt ein Konstruktor notwendig ist, hängt von der kon-  
    // kreten Anwendung ab.  
    System.out.println("ToolTipManager3 erzeugt.");  
}  
  
public void operation() {  
    // Eigentliche Funktionalität des Singleton  
    System.out.println("operation() wurde aufgerufen.");  
}  
}
```

Das folgende Client-Programm nutzt die Enum-Klasse TooltipManager3:

```
// Datei: Client3.java  
  
public class Client3 {  
  
    public static void main(String[] args) {  
        ToolTipManager3.INSTANCE.operation(); // Laden der Klasse  
        ToolTipManager3.INSTANCE.operation();  
        ToolTipManager3.INSTANCE.operation();  
    }  
}
```



Hier das Protokoll des Programmablaufs:

```
ToolTipManager3 erzeugt.  
operation() wurde aufgerufen.  
operation() wurde aufgerufen.  
operation() wurde aufgerufen.
```

Bei einer Enum-Klasse in Java gilt: Der Default-Konstruktor ist `private` und die Enum-Objekte sind immer `final`. Das sind genau die Eigenschaften, die ein Singleton erfüllen muss. Weiterhin gilt, dass Enum-Objekte wie im Beispiel `INSTANCE` öffentlich sind. Die `getInstance()`-Methode wird deswegen nicht benötigt.

Wie im Falle der Variante 2 wird auch hier die Enum-Klasse von der JVM geladen und komplett initialisiert, bevor auf sie zugegriffen werden kann. Synchronisationsprobleme bei mehreren Threads gibt es damit auch bei der Klasse `ToolTipManager3` nicht.

Vergleich der drei Implementierungsvarianten

- Objekterzeugung

In Variante 1 wird das Singleton-Objekt nur erzeugt, wenn die Methode `getInstance()` aufgerufen wird. In den Varianten 2 und 3 wird es immer erzeugt, wenn die Klasse geladen wird, beispielsweise auch dann, wenn das Singleton-Objekt selbst gar nicht benutzt wird, sondern andere Funktionen der Singleton-Klasse.

- **Rechenaufwand**

In den Varianten 2 und 3 kann die Methode `getInstance()` komplett entfallen. Damit können die Aufrufe dieser Methode eingespart werden und das Programm läuft schneller. In Variante 1 ist die Methode `getInstance()` notwendig und enthält eine Abfrage, die bei jedem Aufruf der Methode ausgeführt wird, obwohl das Ergebnis der Abfrage sich nach dem ersten Aufruf nicht mehr ändert.

- **Threadsicherheit**

Um die Threadsicherheit muss sich ein Programmierer bei den Varianten 2 und 3 nicht kümmern, da das Singleton-Objekt bereits beim Laden der Klasse von der JVM erzeugt wird. Die Variante 1 kann auch threadsicher programmiert werden, was aber die Performanz bei der Ausführung der Methode `getInstance()` mindert.

- **Anwendbarkeit**

Die Variante 3 nutzt eine Enum-Klasse zur Implementierung, die bestimmte Eigenschaften haben muss. Ein solches Konstrukt steht nicht in jeder objektorientierten Programmiersprache zur Verfügung.

7.3.4 Bewertung

7.3.4.1 Vorteile

Der folgende Vorteil wird gesehen:

- **nur ein einziges Exemplar wird erzeugt**

Es wird gewährleistet, dass nur ein einziges Exemplar erzeugt wird.

7.3.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **Synchronisation**

Bei der Verwendung mehrerer Threads als Clients ist bei Variante 1 auf die Synchronisation zu achten.

- **Schwierigkeiten bei verteilten Anwendungen**

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.

7.3.5 Einsatzgebiete

Das Singleton-Muster wird dann verwendet, wenn eine Instanz innerhalb einer Applikation nur in einer einzigen Ausprägung zum Einsatz kommen soll. Beispiele hierfür sind eine Logging-Klasse, die einen Anwendungsablauf protokolliert, eine Klasse, die zentrale Applikations-Einstellungen verwaltet, oder ein authentifizierendes Objekt.

Das Singleton-Muster kann auch in verschiedenen anderen Entwurfsmustern eingesetzt werden. Beispielsweise ist ein **Objektpool** üblicherweise nur ein einziges Mal vorhanden. Ein weiteres Beispiel ist das Muster **Fassade**, bei dem eine Fassade meist als Singleton realisiert wird. Beim Entwurfsmuster **Zustand** kann der Zustandsautomat als Singleton ausgelegt werden, da es den Automaten in einer Anwendung typischerweise nur einmal gibt.

7.3.6 Ähnliche Entwurfsmuster

Wenn die Anzahl der Objekte nicht wie beim Singleton-Muster auf eins begrenzt werden soll, sondern auf eine beliebige andere feste Anzahl, kann das Entwurfsmuster **Objektpool** (siehe Kapitel 7.4) eingesetzt werden.

7.4 Das Erzeugungsmuster Objektpool

7.4.1 Name/Alternative Namen

Objektpool (engl. object pool), Resource Pool.

Das hier vorgestellte Muster Objektpool gehört nicht zu dem Musterkatalog der GoF in [Gam94]. Unter dem Namen Resource Pool wurde es beispielsweise in [Bus07] veröffentlicht, zuvor unter dem einfachen Namen Pooling Pattern in [Kir02].

7.4.2 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.



7.4.3 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.

Ein Objektpool hält Objekte bereit und teilt diese 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 eines Pfandflaschensystems müssen auch die Objekte "gereinigt", d. h. in einen "sauberer" 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 7.3) 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
- 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 eine Methode zur Verfügung stellen, mit denen mehrere Objekte gleichzeitig angefordert werden können und die nicht unterbrechbar ist.

Damit das Entwurfsmuster Objektpool sinnvoll eingesetzt werden kann, sollten gewisse 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 und Initialisierung der ersten Objekte warten muss.

- **Regel 4: ein Client darf ein Objekt nicht mehr benutzen, wenn er es in den Pool zurückgelegt hat.**

In den meisten objektorientierten Programmiersprachen werden Objekte über Referenzen angesprochen. Ein Client erhält beim Anfordern somit eine Referenz auf ein

Objekt des Pools. Wenn er das Objekt wieder in den Pool zurücklegt, wird die Referenz nicht automatisch zurückgesetzt. Er könnte somit das Objekt weiter benutzen mit der Gefahr, dass der Pool das Objekt bereits einem anderen Client zur Verfügung gestellt hat und zwei Clients auf dasselbe Objekt zugreifen.

Ein Client darf ein Objekt nicht mehr benutzen, wenn er es zurückgegeben hat! Um dies sicher zu stellen, muss der Client die Referenz selber zurücksetzen (in Java beispielsweise auf `null` setzen).



Dies ist im Programmbeispiel im Kapitel 7.4.3.4 in der Klasse `Passagier` ausgeführt.

7.4.3.1 Klassendiagramm

Das folgende Klassendiagramm stellt den Objektpool, den Client und die wiederverwendbare Klasse der Pool-Objekte dar:

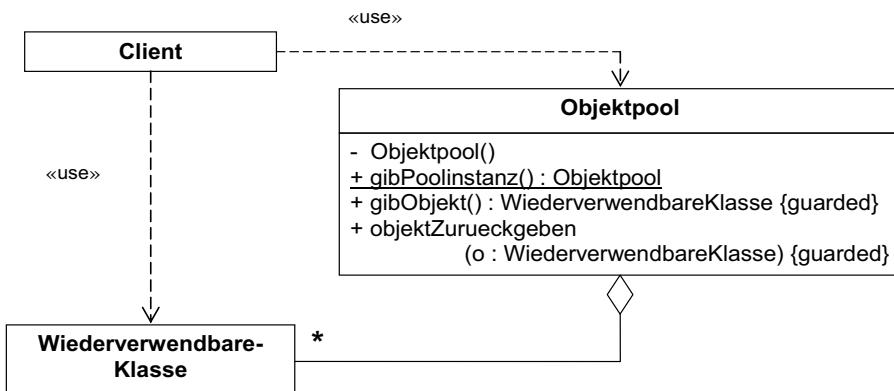


Bild 7-7 Klassendiagramm des Objektpool-Musters

Die Klasse `Objektpool` ist als Singleton (siehe Kapitel 7.3) ausgelegt. Zur Realisierung des Entwurfsmusters **Singleton** wird die Klassenmethode `gibPoolinstanz()` zur Verfügung gestellt und der Konstruktor als `private` deklariert. Auf die Darstellung der Poolinstanz wurde aus Gründen der Übersichtlichkeit verzichtet.

Die Methoden `gibObjekt()` und `objektZurueckgeben()` sind mit dem Zusatz `{guarded}` versehen, um anzudeuten, dass die beiden Methoden threadsicher ausgeführt werden müssen.

7.4.3.2 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 dessen Verwendung wieder in den Pool zurücklegen. Die wiederverwendbare Klasse muss ggf. eine Möglichkeit zur Verfügung stellen, damit ihre Objekte in einen definierten Anfangszustand zurückversetzt werden können.

- **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.

7.4.3.3 Dynamisches Verhalten

Mit dem Aufruf der Klassenmethode Objektpool.gibPoolinstanz() erhält ein Client eine Instanz des Pools.



Neben der Klassenmethode gibPoolinstanz() verfügt der Objektpool noch über zwei Instanzmethoden, die der Client verwenden kann:

gibObjekt() : WiederverwendbareKlasse und
objektZurueckgeben(o : WiederverwendbareKlasse).

Mit gibObjekt() fordert ein Client vom Pool ein Objekt vom Typ WiederverwendbareKlasse an, das er exklusiv – also nur er allein – 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 – im Folgenden wird beispielhaft die Methode operation() benutzt. Wenn der Client dieses Objekt nicht mehr benötigt, ruft der Client objektZurueckgeben() für den Pool auf. Dies ist in dem folgenden Sequenzdiagramm dargestellt:

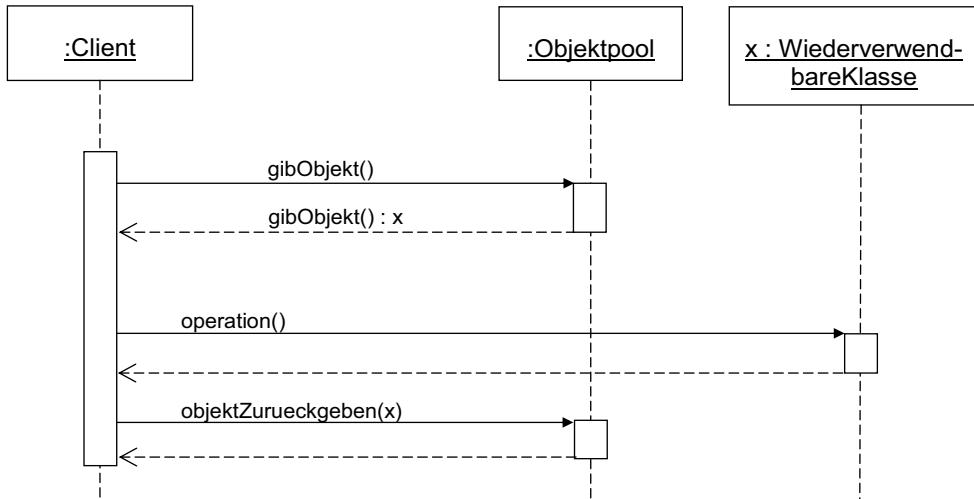


Bild 7-8 Abholen eines Objekts im Objektpool und Rückgabe

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 7-8 nicht dargestellt, sondern das gezeigte Sequenzdiagramm geht davon aus, dass noch freie Objekte verfügbar sind. Ebenfalls nicht gezeigt wird in Bild 7-8, dass der Objektpool das zurückgegebene Element in einen Anfangszustand zurückversetzt.

7.4.3.4 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, welche die Taxis verwenden und nach Gebrauch wieder an den Objektpool zurückgeben.

Die Klasse `Taxi` stellt zwei wichtige Methoden zur Verfügung, nämlich dass ein⁹⁹ Passagier einsteigen und am Ziel wieder aussteigen kann:

```

// Datei: Taxi.java

public class Taxi {

    private final int nummer;
    private Passagier passagier;

    public Taxi(int nummer) {
        ...
    }

    public void einsteigen(Passagier p) {
        ...
    }

    public void aussteigen() {
        ...
    }
}
  
```

⁹⁹ Der Einfachheit halber wird in dem vorliegenden Beispiel nur ein einziger Passagier als Mitfahrer betrachtet.

```
    this.nummer = nummer;
}

public void passagierSteigtAus() {
    System.out.printf("Aus Taxi %d ist Passagier %s ausgestiegen.%n",
        nummer, passagier.getName());
    passagier = null;
}

public void passagierSteigtEin(Passagier passagier) {
    this.passagier = passagier;
    System.out.printf("In Taxi %d ist Passagier %s eingestiegen.%n",
        nummer, passagier.getName());
}
```

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 final String name;
    private Taxi taxi;

    public Passagier(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void taxiNutzen(Taxizentrale taxiZentrale) {
        taxi = taxiZentrale.taxisAnfordern();
        if (taxi == null) {
            System.out.printf("Für Passagier %s ist kein Taxi mehr frei.%n",
                getName());
        } else {
            taxi.passagierSteigtEin(this);
        }
    }

    public void taxiVerlassen(Taxizentrale taxiZentrale) {
        if (taxi != null) {
            taxi.passagierSteigtAus();
            taxiZentrale.taxisFreigeben(taxi);
            taxi = null; // Referenz auf gepooltes Taxi entfernen
        }
    }
}
```

Die Klasse `TaxiZentrale` stellt den eigentlichen Objektpool dar. Diese Klasse verwaltet ein Objekt der Klasse `Deque`¹⁰⁰ 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 Queue verfügbarer Taxis bearbeiten, sollten sie threadsicher gemacht werden. Dies erfolgt in Java 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 7.3) implementiert. Hier der Quellcode der Klasse `TaxiZentrale`:

```
// Datei: TaxiZentrale.java

import java.util.*;

public enum TaxiZentrale {

    INSTANCE;

    private final int size = 2;
    private final Deque<Taxi> taxis;

    TaxiZentrale() {
        taxis = new ArrayDeque<>(size);
        for (int i = 0; i < size; ++i) {
            taxis.add(new Taxi(i + 1));
        }
        System.out.printf("Neue Taxizentrale mit %d verwalteten Taxis"
            + " erzeugt.%n", size);
    }

    public synchronized Taxi taxiAnfordern() {
        return taxis.poll();
    }

    public synchronized void taxiFreigeben(Taxi taxi) {
        taxis.add(taxi);
    }
}
```

Die Klasse `TestTaxiZentrale` erzeugt drei Instanzen der Klasse `Passagier`. Diese Passagiere fordern nacheinander jeweils 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`:

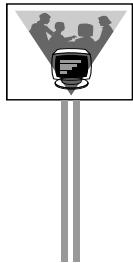
```
// Datei: TestTaxiZentrale.java

public class TestTaxiZentrale {

    public static void main(String[] args) {
```

¹⁰⁰ Die Klasse `Deque` ist im Java Collections Framework enthalten.

```
TaxiZentrale taxiZentrale = TaxiZentrale.INSTANCE;  
  
Passagier hans = new Passagier("Hans");  
Passagier anna = new Passagier("Anna");  
Passagier klaus = new Passagier("Klaus");  
  
hans.taxiNutzen(taxiZentrale);  
anna.taxiNutzen(taxiZentrale);  
klaus.taxiNutzen(taxiZentrale);  
  
hans.taxiVerlassen(taxiZentrale);  
klaus.taxiNutzen(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 Taxi mehr frei.
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.

7.4.4 Bewertung

7.4.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Wiederverwendung erspart ständiges Erzeugen**

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.

- **Rückschlüsse auf Systemlast möglich**

Durch Beobachtung des Pools können Rückschlüsse auf die aktuelle Systemlast gezogen werden.

7.4.4.2 Nachteile

Folgende Nachteile können durch den Einsatz eines Objektpools entstehen:

- **Rückgabe von Objekten kann problematisch sein**

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.

- **Zustand der wiederzuverwendenden Objekte**

Die wiederzuverwendenden Objekte müssen sich vor ihrer Wiederverwendung immer im gleichen Zustand befinden. Die Klasse der wiederverwendbaren Objekte muss entsprechende Methoden zur Verfügung stellen.

- **Deadlocks**

In nebenläufigen Umgebungen synchronisiert der Pool die nebenläufigen Einheiten. Dies kann zu **Deadlocks** führen, wenn ein Thread mehrere Objekte vom Pool haben will.

7.4.5 Einsatzgebiete

Das Entwurfsmuster eines Objektpools wird häufig bei Datenbankzugriffen verwendet, um Verbindungen zur Datenbank zwischenzuspeichern 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**¹⁰¹ eingesetzt. So müssen Threads nicht ständig neu erstellt werden, sondern sie werden einfach wieder-verwendet. Ebenso wie das Erstellen von Datenbankverbindungen ist auch das Erstellen von Threads aufwendig.

7.4.6 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 Wiederverwend-

¹⁰¹ Das Nebenläufigkeitsmuster Thread Pool soll es erlauben, gleichartige Serviceanfragen auf eine Anzahl bereitstehender Threads zu verteilen und ggf. Serviceanfragen in einer Taskqueue zu speichern, falls keine freien Threads verfügbar sind. Damit können gleichartige Aufgaben vom Thread Pool parallelisiert werden und asynchron abgearbeitet werden.

bareKlasse versorgt. Diese Objekte werden allerdings nicht in der Methode gib-Objekt() erzeugt, sondern sie werden im Pool vorgehalten und über die Methode gibObjekt() nur zur Verfügung gestellt.

In [Gra02b] wird das Muster **Thread Pool** als eigenständiges Entwurfsmuster beschrieben. Es wurde im vorherigen Abschnitt als Variante des Musters Objektpool vorgestellt.

Durch das Muster **Object Manager** [Bus07] wird ebenfalls ein Pool von Objekten verwaltet. Die Funktionalität des Objektmanagers ist weitergehend: Eine Anwendung kann auch die Lebensdauer der Objekte kontrollieren – also neue Objekte erzeugen oder existierende löschen.

7.5 Zusammenfassung

Erzeugungsmuster 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.

Das Erzeugungsmuster Fabrikmethode in Kapitel 7.1 kapselt die Erzeugung einer konkreten Instanz in der Methode einer Unterklassie. Die Unterklassen werden durch Vererbung statisch gewonnen.

Das Erzeugungsmuster Abstrakte Fabrik in Kapitel 7.2 erlaubt es, dass durch Wahl der entsprechenden konkreten Fabrik diejenige Produktfamilie zur Laufzeit ausgewählt wird, aus der Objekte erzeugt werden sollen.

Das Erzeugungsmuster Singleton in Kapitel 7.3 gewährleistet es, dass eine Klasse nur ein einziges Mal instanziert werden kann.

Das Erzeugungsmuster Objektpool in Kapitel 7.4 ermöglicht es, Instanzen zur Verfügung zu stellen, dabei diese wiederzuverwenden und so Ressourcen zu schonen, anstatt die Instanzen immer wieder neu zu erzeugen.

7.6 Kontrollfragen

Aufgabe 7.6.1: Fabrikmethode

- 7.6.1.1 Welche Nachteile ergeben sich durch die Verwendung des Erzeugungsmusters Fabrikmethode?
- 7.6.1.2 Wie wird das Erzeugen von Objekten beim Erzeugungsmuster Fabrikmethode realisiert?

Aufgabe 7.6.2: Abstrakte Fabrik

- 7.6.2.1 Was ist der Unterschied zwischen den beiden Erzeugungsmustern Fabrikmethode und Abstrakte Fabrik?
- 7.6.2.2 Mit welchem Problem befasst sich das Entwurfsmuster Abstrakte Fabrik?

Aufgabe 7.6.3: Singleton

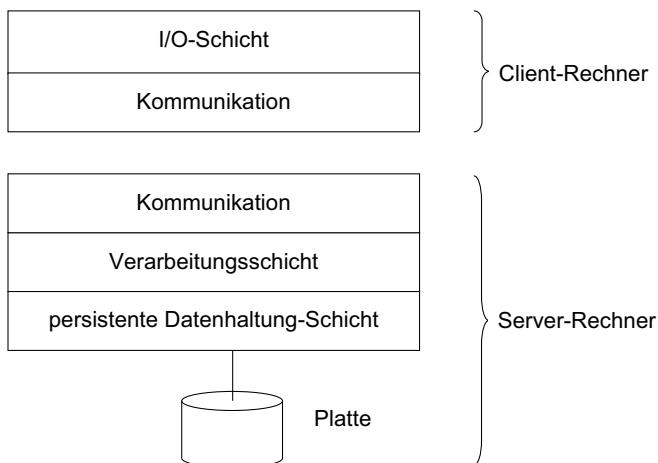
- 7.6.3.1 Skizzieren Sie die Lösung beim Erzeugungsmuster Singleton.
- 7.6.3.2 Beschreiben Sie das dynamische Verhalten des Erzeugungsmusters Singleton.
- 7.6.3.3 Welche Probleme können bei der Realisierung des Erzeugungsmusters Singleton auftreten?

Aufgabe 7.6.4: Objektpool

- 7.6.4.1 Nennen Sie drei Nachteile des Erzeugungsmusters Objektpool.
- 7.6.4.2 Wofür wird im Zusammenhang mit Datenbanken das Erzeugungsmuster Objektpool eingesetzt?

Kapitel 8

Übersicht über die behandelten Architekturmuster



- 8.1 Architekturmuster zur Strukturierung eines Systems
- 8.2 Architekturmuster für adaptierbare Systeme
- 8.3 Architekturmuster für verteilte Systeme
- 8.4 Architekturmuster für interaktive Systeme
- 8.5 Zusammenfassung
- 8.6 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_8.

8 Übersicht über die behandelten Architekturmuster

Der Unterschied zwischen Architekturmustern und Entwurfsmustern wurde in Kapitel 2.2 betrachtet.

Entwurfsmuster stellen feinkörnige Muster eines Systems dar, während **Architekturmuster grobkörnig** sind. Architekturmuster lösen nicht ein Teilproblem, sondern beeinflussen die Grundzüge der Architektur des betreffenden Systems.



Ein **Architekturmuster** kann verschiedene Entwurfsmuster enthalten. Ob überhaupt und wie viele Entwurfsmuster bei einem bestimmtem Architekturmuster miteinander kombiniert werden müssen, damit das Architekturmuster entsteht, ist von Fall zu Fall verschieden. Das Architekturmuster Model-View-Controller (siehe Kapitel 12.1) enthält in der Regel das Entwurfsmuster Beobachter, das Entwurfsmuster Strategie sowie das Entwurfsmuster Kompositum. Das Architekturmuster Layers (siehe Kapitel 9.1) verwendet hingegen kein einziges Entwurfsmuster.

Während alle in diesem Buch gezeigten Muster objektorientiert sind, können Muster generell auch einen nicht objektorientierten Charakter haben. So ist beispielsweise das Architekturmuster Layers nicht an die Verwendung objektorientierter Techniken geknüpft.

Die in diesem Kapitel vorgestellten Architekturmuster sind den Autoren in der Praxis besonders oft begegnet. Sie können in die folgenden Kategorien – angelehnt an Buschmann [Bus98] – eingeteilt werden:

- **Strukturierung eines Systems** ("From Mud to Structure" bei Buschmann et al. [Bus98]) (siehe Kapitel 9),
- **Adaptierbare Systeme** (siehe Kapitel 10),
- **Verteilte Systeme** (siehe Kapitel 11),
- **Interaktive Systeme** (siehe Kapitel 12).

Kapitel 8.1 stellt die Architekturmuster Layers und Pipes and Filters zur Strukturierung eines Systems vor. Kapitel 8.2 geht auf das Muster Plug-in als ein Beispiel für adaptierbare Systeme ein. Kapitel 8.3 stellt Beispiele für verteilte Systeme vor und Kapitel 8.4 betrachtet das Architekturmuster MVC, das zu den Mustern für interaktive Systeme zählt.

8.1 Architekturmuster zur Strukturierung eines Systems

Kapitel 9 analysiert die folgenden Architekturmuster:

- **Layers**

Das Architekturmuster **Layers** (siehe Kapitel 9.1) schneidet die Architektur eines Systems in horizontale Schichten, wobei jede Schicht auf die Dienste der darunterliegenden Schicht zugreifen kann (horizontale Schichtung eines Systems).

- **Pipes and Filters**

Das Architekturmuster **Pipes and Filters** (siehe Kapitel 9.2) 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.

8.2 Architekturmuster für adaptierbare Systeme

Adaptierbare Systeme ermöglichen es, dass Softwaresysteme auch in Situationen, die während ihrer Entwicklung nicht vorhersehbar waren, mit Erfolg eingesetzt werden können. Neue Funktionen müssen hinzugefügt und existierende Dienste müssen geändert werden können. Ebenso müssen beispielsweise neue Versionen von Betriebssystemen oder Frameworks eingeführt werden können,

Behandelt wird in Kapitel 10.1 das Architekturmuster:

- **Plug-in**

Das Architekturmuster **Plug-in** strukturiert eine spezielle Anwendung so, dass sie über Erweiterungspunkte in Form von Schnittstellen verfügt, an denen Plug-ins, welche diese Schnittstellen implementieren, vom sogenannten **Plug-in-Manager** (unter Verwendung der Laufzeitumgebung) instanziert und eingehängt werden können. Die betrachtete Anwendung sollte aber bereits ohne diese zusätzlichen Erweiterungen lauffähig sein. Eine solche Architektur lohnt sich auf jeden Fall, wenn die Basisanwendung ohne Erweiterungen einem breiten Anwenderkreis zur Verfügung steht und die speziellen Erweiterungen für bestimmte Benutzergruppen gedacht sind.

8.3 Architekturmuster für verteilte Systeme

Kapitel 11 erörtert die Beispiele:

- **Broker**

Das Architekturmuster **Broker** (siehe Kapitel 11.1) strukturiert ein System aus mehreren Clients und Servern in eine Architektur mit einem Broker als vermittelnde Instanz für das Finden des passenden Servers bei einer Anfrage eines Clients nach einem Service und das Zurückliefern der Antwort des Servers an den Client. Client- und Server-Komponenten kommunizieren untereinander nur über den Broker. In

einem verteilten System ist der Broker selbst als Middleware auf alle Knotenrechner des Systems verteilt.

- **Service-Oriented Architecture**

Das Architekturmuster **Service-Oriented Architecture**¹⁰² (siehe Kapitel 11.2) gliedert die Architektur eines Systems in Komponenten bzw. Teilkomponenten, die den Anwendungsfällen bzw. Teilen von Anwendungsfällen aus Sicht der Analyse entsprechen und deren Leistungen als Services zur Verfügung stellen. Damit sollen bei einer Abänderung der Geschäftsprozesse nur:

- die entsprechenden Komponenten als Verkörperung der Anwendungsfälle abgeändert werden müssen oder
- ggf. unter Verwendung bestehender elementarer Services neue Komponenten erzeugt werden.

8.4 Architekturmuster für interaktive Systeme

Vorgestellt wird das Architekturmuster:

- **Model-View-Controller**

Das Architekturmuster **Model-View-Controller**¹⁰³ (siehe Kapitel 12.1) 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 plus 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.

8.5 Zusammenfassung

In Kapitel 8.1 werden die Architekturmuster Layers sowie Pipes and Filters zur Strukturierung eines Systems diskutiert.

Kapitel 8.2 skizziert das Architekturmuster Plug-in als Beispiel eines Musters für adaptierbare Systeme.

Kapitel 8.3 gibt eine Hinführung zu den Architekturmustern Broker und Service Oriented Architecture als Vertreter der Architekturmuster für verteilte Systeme.

Kapitel 8.4 führt in das Architekturmuster Model-View-Controller als Beispiel eines Musters für interaktive Systeme ein.

¹⁰² Service Oriented Architecture wird abgekürzt durch SOA.

¹⁰³ Model-View-Controller wird abgekürzt durch MVC.

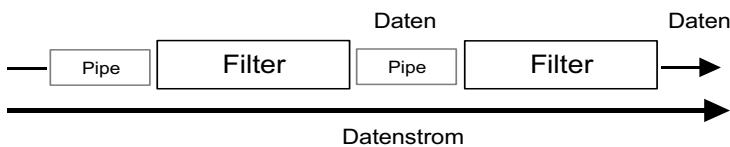
8.6 Kontrollfragen

Aufgaben 8.6.1: Kategorien von Architekturmustern nach Buschmann et al.

- 8.6.1.1 Wie werden nach Buschmann et al. [Bus98] Architekturmuster eingeteilt?
- 8.6.1.2 Nennen Sie zwei Beispiele für die Strukturierung eines Systems durch Architekturmuster.
- 8.6.1.3 Wozu ist ein adaptierbares System gut?
- 8.6.1.4 Nennen Sie ein Muster, das für interaktive Systeme angewendet werden kann.

Kapitel 9

Architekturmuster zur Strukturierung eines Systems



- 9.1 Das Architekturmuster Layers
- 9.2 Das Architekturmuster Pipes and Filters
- 9.3 Zusammenfassung
- 9.4 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_9.

9 Architekturmuster zur Strukturierung eines Systems

Dieses Kapitel über Architekturmuster zur Strukturierung eines Systems umfasst die Architekturmuster Layers sowie Pipes and Filters.

Das Architekturmuster **Layers** (siehe Kapitel 9.1) wird verwendet, wenn die hohe Komplexität eines Systems durch die Einführung **mehrerer horizontaler Schichten** vereinfacht werden soll.

Ein System, das einen **Datenstrom in einer einzigen Richtung** verarbeitet, kann mit Hilfe des Architekturmusters **Pipes and Filters** (siehe Kapitel 9.2) in Komponenten strukturiert werden. Die Verarbeitungsschritte des gesamten Systems werden dabei in einzelne Verarbeitungsstufen zerlegt, die sequenziell hintereinander geschaltet werden.

9.1 Das Architekturmuster Layers

Das Architekturmuster Layers modelliert die Struktur eines Systems **in horizontale Schichten mit Client/Server-Beziehungen** zwischen den Schichten, wobei eine höhere Schicht nur auf tiefere Schichten zugreifen darf.



9.1.1 Name/Alternative Namen

Schichtenmodell, Layers.

9.1.2 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. Der **Name jeder Schicht** soll eine **Abstraktion** der in ihr enthaltenen Funktionalität sein.

In der Grundform des Musters sollen sich die Funktionen einer höheren Schicht auf die Funktionen der nächst tieferen Schicht beziehen, diese aufrufen und von ihnen die Antwort zurück erhalten. Zwischen jeweils zwei Schichten soll also das Client-Server-Prinzip gelten, wobei die Server-Schicht wiederum als Client auftreten kann, wenn sie auf die nächst tiefere Schicht als ihrem Server zugreift.

Eine tiefere Schicht darf dabei nicht von höheren Schichten abhängen. 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.

9.1.3 Lösung

Aufgrund einer zu hohen Komplexität eines Systems wird dessen Softwarearchitektur in mehrere übereinanderliegende horizontale Schichten eines sogenannten **Schichtenmodells** gegliedert. Jede der Schichten (engl. **tier** oder **layer**) enthält eine bestimmte Unteraufgabe eines Systems, wobei eine höhere Schicht nur tiefere Schichten aufrufen darf. Das Aufteilen in Schichten erfolgt nach funktionalen Gesichtspunkten. Im Folgenden die Skizze eines Schichtenmodells:

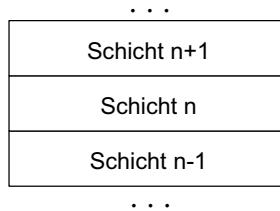


Bild 9-1 Schichtenmodell

Eine tiefere Schicht soll im Rahmen eines Schichtenmodells 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 in der **Grundform des Musters** nur von der jeweils tiefer liegenden Schicht n - 1 ab.
2. Eine tiefere Schicht hängt nicht von den höheren Schichten ab.
3. Jede Schicht bietet in der Grundform des Musters Dienste für ihre jeweils höher liegende, benachbarte 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 von tieferen Schichten zugreifen.



Jede Schicht verfügt in der Grundform des Musters 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, dadurch dass Schicht 4 direkt auf Schicht 2 zugreift:

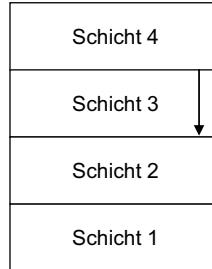


Bild 9-2 Beispiel für Layer Bridging

Jede Schicht stellt eine Abstraktionsebene im System dar. In der älteren Literatur (Beispiel [Dij68]) wird eine Schicht auch als **abstrakte Maschine**¹⁰⁴ bezeichnet. Die über die Schnittstelle einer Schicht nach oben zur Verfügung gestellten Funktionen werden dabei auch als Operationen einer abstrakten Maschine angesehen. Insgesamt entsteht somit eine Hierarchie von abstrakten Maschinen.

9.1.3.1 Klassendiagramm

In diesem Kapitel werden die Schichten und ihre Funktionen in der Grundform des Musters als objektorientierte Lösung mit Klassen vorgestellt:

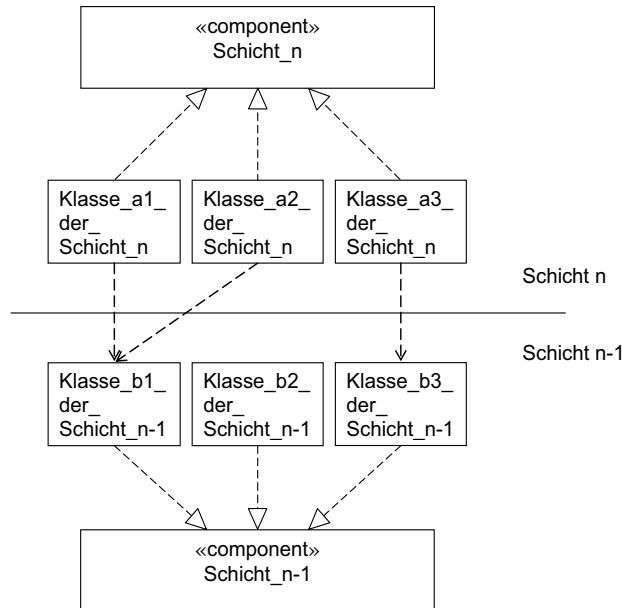


Bild 9-3 Eine Abhängigkeitsbeziehung zwischen Objekten benachbarter Komponenten

¹⁰⁴ 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.

Das Konzept der Schichten ist aber unabhängig von der Objektorientierung. Auf die Darstellung einer alternativen, nicht objektorientierten Variante wird in diesem Kapitel 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 9-3.

Instanziert 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].

9.1.3.2 Teilnehmer

Teilnehmer am Architekturmuster Layers sind in der Grundform des Musters 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 dieses Objekt wiederum Objekte der direkt unter ihm liegenden Schicht aufrufen, falls eine solche Schicht existiert.

9.1.3.3 Dynamisches Verhalten

Das folgende Bild zeigt das Client-Server-Verhalten der Schichten exemplarisch:

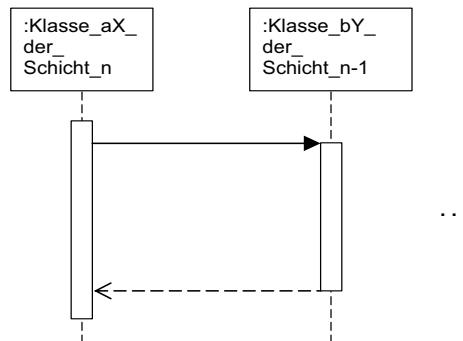


Bild 9-4 Aufruf und Antwort eines Objekts der jeweils benachbarten tieferen Schicht

Eine Schicht n - 1 stellt Dienste für die Schicht n bereit und nutzt selbst wieder die Dienste der Schicht n - 2.

9.1.3.4 Bewertung

Nach dem Prinzip 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 Muster Layers 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.

9.1.3.5 Vorteile

Das Architekturmuster Layers hat die folgenden Vorteile:

- **Abstraktion der Funktionalitäten unterstützt Teile und Herrsche**

Jede Schicht stellt eine Abstraktion einer bestimmten Funktionalität dar und kann leichter verstanden werden als das Ganze.

- **Wiederverwendung von Schichten**

Schichten können unter Umständen wiederverwendet werden.

- **stabile und ggf. standardisierte Schichten**

Schichten sind stabil und können standardisiert werden.

- **Minimierung von Abhängigkeiten**

Code-Abhängigkeiten können minimiert werden, d. h. Änderungen am Quellcode sollten wenige Ebenen betreffen.

- **parallele Entwicklung**

Nach Festlegung der Schnittstellen können die einzelnen Ebenen parallel entwickelt werden.

- **Verbesserung von Test & Integration**

Die Schichten können sehr gut bottom-up integriert und getestet werden.

9.1.3.6 Nachteile

Die folgenden Nachteile treten beim Architekturmuster Layers auf:

- **Schwierigkeit bei der Abgrenzung der Abstraktionen**

Die Identifikation und Abgrenzung der verschiedenen Schichten kann ggf. nicht so einfach sein. Es kann schwierig sein, die "richtige" Anzahl von Schichten zu finden.

- **Strict Layering oft zu einschränkend**

Die Regel des Strict Layering, dass ein Zugriff nur auf die benachbarte Ebene erfolgt, ist oft zu einschränkend. Diese Regel kann aber mit Layer Bridging umgangen werden, wenn die Performance gesteigert werden soll.

- **Zeitaufwand**

Es liegt auf der Hand, dass eine Anfrage, die von Schicht zu Schicht weitergereicht wird, zeitaufwendiger als ein direkter Zugriff ist.

- **Änderungen können ggf. mehrere Schichten betreffen**

Änderungen können sich über Schichten hinaus auswirken. Manche Arbeiten wie z. B. eine Korrektur von Fehlern können alle Schichten betreffen. Dies bedeutet Mehraufwand.

9.1.4 Einsatzgebiete

Der Einsatz des Musters Layers kann überall da erfolgen, wo nach dem Client-Server-Prinzip modelliert wird. Zusammengehörige Funktionen werden jeweils getrennt 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 9.1.4.1), das Schichtenmodell für die Anwendungssoftware von Rechnern (siehe Kapitel 9.1.4.2) oder das in Kapitel 9.1.4.4 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.

Das Strukturmuster **Fassade** (siehe Kapitel 5.4) führt eine zusätzliche Schicht ein, welche die Klassen eines Teilsystems kapselt.

9.1.4.1 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:

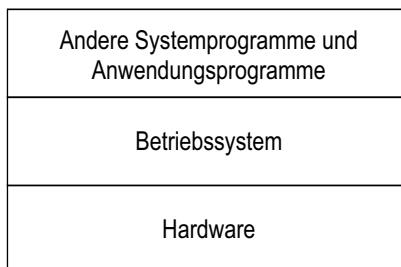


Bild 9-5 Schichtenstruktur eines Rechners

Zu den "anderen Systemprogrammen" gehört beispielsweise die Kommandoschnittstelle Shell bei Unix oder ein DBMS¹⁰⁵.

Das Betriebssystem hat sowohl Schnittstellen zur Hardware als auch Schnittstellen zu den Programmen des Anwenders und zu anderen Systemprogrammen. 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:

- **gerechte Verteilung der Ressourcen des Rechners**

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.

- **Abstraktion der Hardware**

Ein Softwareentwickler soll mit vernünftigem 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 diesen Programmen zur Verfügung und ist eine Softwareschicht, die über der Hardware liegt.

¹⁰⁵ DBMS = Database Management System.

9.1.4.2 Schichtenmodell für die Anwendungssoftware eines Rechners

Schichtenmodelle für Rechner finden nicht nur bei Informationssystemen, sondern auch bei eingebetteten Systemen¹⁰⁶ (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:

- I/O-Schicht,
- Verarbeitungsschicht,
- Datenzugriffsschicht und
- DBMS-Kern



¹⁰⁶ Siehe beispielsweise die AUTOSAR-Architektur für die Automobilindustrie [Kin09].

Im Folgenden werden diese Schichten erläutert:

- **I/O-Schicht**

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.

- **Verarbeitungsschicht**

Die Objekte der **Verarbeitungsschicht** stellen die **Kontrollobjekte** und die **Entity-Objekte** im Arbeitsspeicher (**transiente Datenhaltung**) dar.

- **Datenzugriffsschicht**

Die **Datenzugriffsschicht** abstrahiert die persistente Datenhaltungsschicht auf der Platte. 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 Datenzugriffs- schicht zusätzlich eine selbst programmierte, datenbankunabhängige Schnittstelle anzubieten (siehe Bild 9-6). Dadurch kann das DBMS unterhalb der datenbankunab- hängigen Datenzugriffsschicht ausgetauscht werden, ohne dass die Objekte der Ver- arbeitungsschicht betroffen sind.

Falls das DBMS nie ausgetauscht wird, kann auf eine Kapselung der proprietären Schnittstelle in einer datenbankunabhängigen **Datenabstraktionsschicht** verzichtet werden. In diesem Fall existiert nur der tiefere **herstellerabhängige Anteil der Da- tenzugriffsschicht**. Das folgende Bild zeigt die mögliche Struktur der Datenbankzu- griffsschicht:

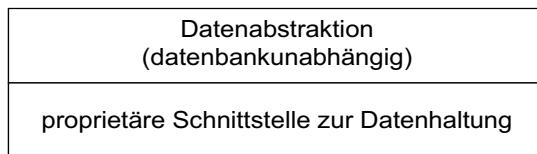


Bild 9-6 Struktur der Datenzugriffsschicht

- **DBMS-Kern**

Im **DBMS-Kern** befindet sich der Kern des DBMS. Er speichert die Daten der Entity- Objekte persistent auf Festplatte bzw. lädt sie.

9.1.4.3 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.

Eine **Zwei-Schichten-Architektur** wird vorwiegend für eine Kombination aus Client- und Server-Rechner eingesetzt.



Bei einer **Drei-Schichten-Architektur** treten ein Client-Rechner, ein Anwendungsserver-Rechner und ein Datenbankserver-Rechner auf.



Diese genannten Architekturen werden im Folgenden genauer vorgestellt:

- **Schichtenmodell eines Einrechner-Systems**

Im Rahmen eines Schichtenmodells werden die Funktionsklassen einer Anwendung auf einem einzigen 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 bei der Analyse 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:

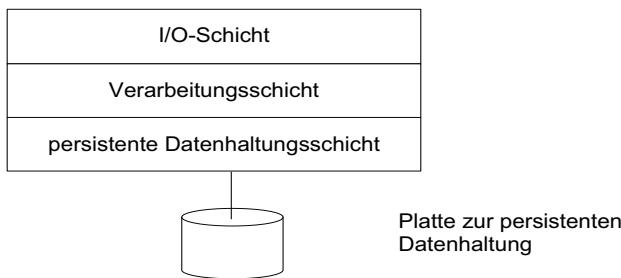


Bild 9-7 Schichtenmodell eines Einrechner-Systems

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.



Abhängig davon, ob sich die Verarbeitungsschicht auf dem Client-Rechner oder dem Server-Rechner befindet, wird zwischen einer **Thin-Client-Architektur** (siehe Bild 9-8) und einer **Fat-Client-Architektur** (siehe Bild 9-9) unterschieden. Dies wird im Folgenden gezeigt:

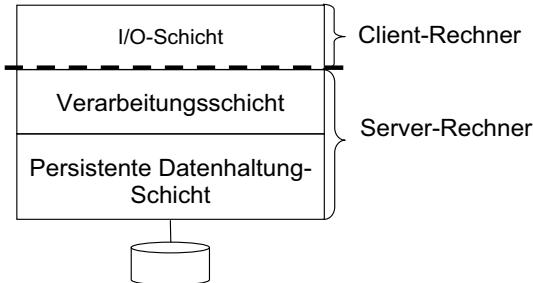


Bild 9-8 Thin Client¹⁰⁷

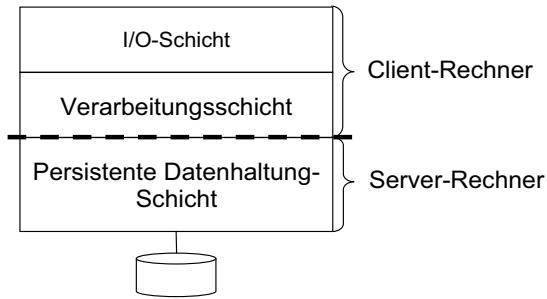


Bild 9-9 Fat Client¹⁰⁸

Auf jedem Rechner wird zusätzlich eine **Kommunikationsschicht** erforderlich, damit die Rechner miteinander kommunizieren können. Diese Kommunikationsschicht ist jeweils über und unter der gestrichelten fetten Linie einzuziehen.

¹⁰⁷ Bei der Thin-Client-Architektur werden nur die Objekte der I/O-Schicht auf dem Client-Rechner ausgeführt.

¹⁰⁸ Bei der Fat-Client-Architektur befindet sich nur die persistente Datenhaltungsschicht auf dem Server-Rechner.

- **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:

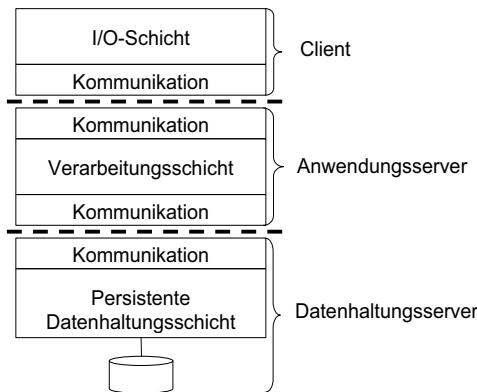


Bild 9-10 Modell der Drei-Schichten-Architektur mit Kommunikation

Vorteile der Zwei- und Drei-Schichten-Architektur

Die folgenden Vorteile werden gesehen:

- 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 diese Funktionen zentral für die verschiedenen Clients.
- Der Vorteil der **Drei-Schichten-Architektur** mit einem **Thin-Client** 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.

9.1.4.4 Das ISO/OSI-Schichtenmodell für die Kommunikation

Ein typisches Beispiel für ein Schichtenmodell ist das ISO/OSI-Schichtenmodell (siehe [Ros90]).

Das **ISO/OSI-Schichtenmodell** beschreibt das **Kommunikationssystem** eines Rechners, in anderen Worten das Netzwerkprotokoll. Ein Kommunikationssystem erlaubt es Anwendungen, über ein Netzwerk miteinander zu kommunizieren.



Das **ISO/OSI Basic Reference Model** strukturiert das Kommunikationssystem in sieben Schichten.¹⁰⁹ Jede Schicht ist für die Lösung eines spezifischen Problems zuständig. Die Probleme sind jeweils mit Hilfe der darunterliegenden Schicht zu lösen.

Im Folgenden die 7 Schichten nach ISO/OSI:

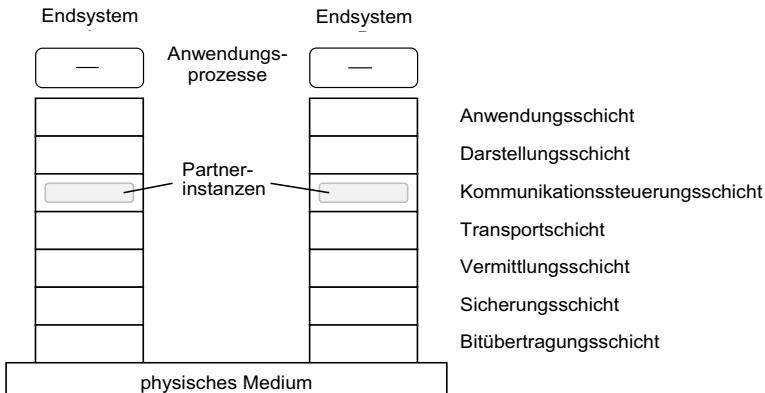


Bild 9-11 Prinzipieller Aufbau des ISO/OSI-Schichtenmodells

Die in Bild 9-11 gezeigten Kommunikationssysteme zweier Rechner werden im Folgenden genauer erläutert:

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 gewisse Arbeitseinheiten (Instanzen) haben, die für das Außenverhalten verantwortlich sind.

ISO/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.

¹⁰⁹ 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 hingegen acht Schichten. SNA 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 verschiedene Hersteller als Nutzer einer Standardisierung die Schichten gemäß demselben Standard implementieren. Damit ist eine Interoperabilität zwischen Rechnern verschiedener Hersteller gewährleistet.

Es verhandeln die Partnerinstanzen einer Schicht, d. h. die entsprechenden Schichteninstanzen des Senders und des Empfängers, miteinander, um die Aufgabe der Schicht gemeinsam zu erfüllen.



Im Folgenden werden die verschiedenen Schichten beschrieben:

- **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.



- **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.



- **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.

- **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.



- **Kommunikationssteuerungsschicht (engl. session layer)**

Die Kommunikationssteuerungsschicht regelt die **Synchronisation zwischen Prozessen in verschiedenen Systemen**.



- **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/BER¹¹⁰ 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.

- **Anwendungsschicht (engl. application layer)**

Die Anwendungsschicht, die höchste Schicht des Referenzmodells, stellt die Schnittstelle der Anwendungsprozesse zum Kommunikationssystem dar.

Bei der Anwendungsschicht handelt es sich um allgemeine Hilfsdienste für die Kommunikation oder aber um spezielle Kommunikationsdienste wie File Transfer, Message Handling System (MHS) etc.



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:

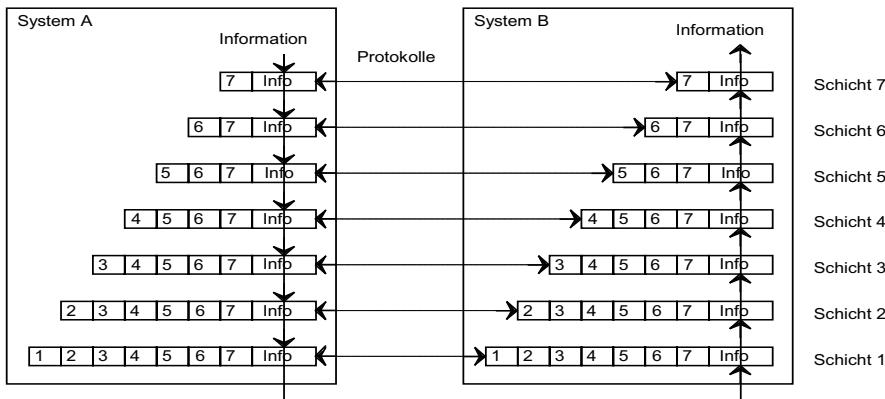


Bild 9-12 Protokollweg durch die OSI-Schichten

9.1.5 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

¹¹⁰ 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 bzw. in umgekehrter Richtung beinhalten.

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 Aufrufsstellen der jeweils höheren Schicht die Ergebnisse der aufgerufenen Dienste zur Verfügung. Ein Dienst gibt eine Antwort dabei 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 beispielsweise die Management Information Base (MIB) beim Netzwerkmanagement [Ros93, S. 76].

9.2 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: Dabei tauschen nur benachbarte Filter Daten aus und es gibt nur eine Richtung des Datenflusses.

Die Ausgabe eines Filters bzw. Prozesses ist die Eingabe des nächsten Filters bzw. Prozesses einer Kette von Filtern einer **datenstromorientierten Architektur** eines Systems.



9.2.1 Name/Alternative Namen

Kanäle und Filter (engl. pipes and filters). Es wird auch die Kurzform Filter-Muster (engl. filter pattern) benutzt.

9.2.2 Problem

Ein **datenstromorientiertes System** der Datenverarbeitung soll nicht als monolithischer Block gebaut werden. Das System soll leicht erweiterbar sein, **aus unabhängigen, verarbeitenden Bausteinen – Filter genannt** – bestehen und eine möglichst parallele Verarbeitung erlauben.

Zwischen zwei benachbarten Filtern können **Daten in Pipes gepuffert** werden, um diese beiden Filter asynchron zu entkoppeln.



Dies bedeutet beispielsweise, dass ein Filter in eine Pipe schreiben kann und dass die Daten aus der Pipe zu einem späteren Zeitpunkt entnommen werden können.

9.2.3 Lösung

Ein System, das Datenströme verarbeitet, wird mithilfe des Musters Pipes and Filters in Systemkomponenten strukturiert.

Die Aufgabe des gesamten Systems wird in einzelne Verarbeitungsschritte, die **Filter**, zerlegt, die sequenziell nacheinander abgearbeitet werden. **Pipes** dienen zur Datenpufferung.



Jeder **Verarbeitungsschritt** wird in Form eines **Filters als Baustein** implementiert. Jeweils zwei Filter können durch eine sogenannte Pipe verbunden werden.



Filter lesen die Daten und bearbeiten sie sequenziell. Ein Filter wandelt eine Dateneingabe des Vorgängers in eine Datenausgabe für den nächsten Filter um. Die Eingabedaten eines Filters werden als **Datenstrom** empfangen. Daraufhin führt der Filter eine Verarbeitungsfunktion 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**

Es werden bestimmte Informationen dem Datenstrom entnommen und nicht weitergereicht.

- **Teile den Daten hinzufügen**

Es werden z. B. weitere Informationen berechnet und an den Datenstrom angehängt wie z. B. Zeilennummern bei der Verarbeitung von Textzeilen.

- **Teile der Daten modifizieren**

Es werden z. B. vorhandene Informationen konzentriert oder in eine andere Darstellung überführt wie z. B. ein Text in Großbuchstaben.

Die hier genannten Verarbeitungsarten können beliebig kombiniert werden.

Die komplette Abfolge von Verarbeitungsstufen beim Muster Pipes and Filters 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. Sie können als spezielle Filter angesehen werden.



Das folgende Bild zeigt den Aufbau einer Pipeline:

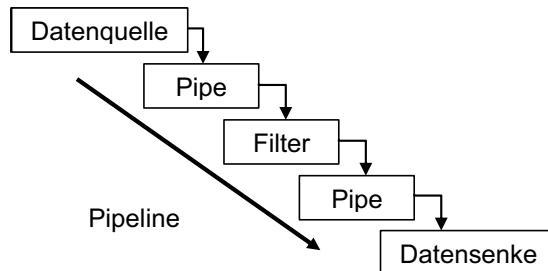


Bild 9-13 Aufbau einer Pipeline beim Muster Pipes and Filters

Um die Verarbeitung eines Filters der Pipeline zu beenden, muss es im Datenstrom eine **Endekennung** geben, da ein Filter wissen muss, ob noch Daten kommen oder nicht. Hierüber muss die Anwendung entscheiden. Das Muster gibt dazu keine Empfehlungen.



In den beiden folgenden Unterkapiteln werden Pipes und Filter genauer charakterisiert.

9.2.3.1 Pipes

Eine **Pipe** transferiert und puffert die Daten.

Eine **Pipe** ist ein **passives Element**, welches einen Puffer für den Datenstrom darstellt.



Eine **Pipe** ermöglicht den **Zugriff** auf die Daten einer **Datenquelle** bzw. einer **Datensenke** genauso wie die **Kommunikation zwischen zwei Filtern**. Die gepufferten Daten werden nach dem **FIFO-Prinzip** weitergegeben.

Da eine Pipe Daten zwischenpuffern kann, kann ein Filter Daten zu einer bestimmten Zeit in eine Pipe schreiben und ein anderer Filter kann dann diese Daten zu einer anderen Zeit entnehmen. Eine Pipe kann also zwei Filter **asynchron entkoppeln**.



9.2.3.2 Aktive und passive Filter

Filter können als aktive oder passive Filter realisiert werden.



Aktive und passive Filter sind charakterisiert durch:

- Ein **aktiver Filter** holt die Daten selbst aus dem ihm aus Sicht des Datenstroms vorangehenden Element wie einer Pipe oder einem Filter und schickt seine Ergebnisse

in das ihm aus Sicht des Datenstroms nachfolgende Element wie einer Pipe oder einem Filter.

- Einem **passiven Filter** werden seine Daten von dem ihm aus Sicht des Datenstroms vorangehenden Filter gesendet. Bei einem **passiven Filter** werden dessen Daten von dem ihm aus Sicht des Datenstroms nachfolgenden Filter geholt.

Für die speziellen Filter Datenquelle und Datensenke gilt:

Eine **aktive Datenquelle** liefert Daten an ein nachfolgendes Element – entweder an eine Pipe oder einen Filter. Ist eine Datenquelle hingegen **passiv**, so wartet sie, bis der nächste Filter Daten von ihr anfordert.



Eine **aktive Datensenke** fordert Daten an. Ist sie **passiv**, so wartet sie auf Daten.



Aktive Filter

Ein **aktiver Filter** stellt einen eigenständigen parallelen Prozess¹¹¹ dar. Ein aktiver Filter ist der typische Fall. Er wird von einem übergeordneten Programm gestartet und läuft immer als parallele Einheit.



Aktive Filter sind vom Betriebssystem Unix bekannt. Sie können ihre Eingabedaten aus dem dem Filter aus Sicht des Datenflusses vorangehenden Element holen und ihre Ausgabedaten in das dem Filter aus Sicht des Datenflusses nachfolgende Element senden. 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, bis er fertig ist. 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 und zum anderen eine gewisse Parallelität erzeugt werden, wenn Filter schon Teilergebnisse liefern, ehe sie fertig sind¹¹².

Ein schneller aktiver Filter vor einem langsamen aktiven Filter in einer Pipeline wird durch eine Pipe zwischen beiden Filtern gebremst, wenn die Pipe voll ist.

¹¹¹ Betriebssystem-Prozess oder Thread.

¹¹² Ein Beispiel, bei dem eine stückweise Vorgehensweise nicht möglich ist, ist ein Filter, der einen Datenstrom sortiert weitergeben soll.

Passive Filter

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. Ein passiver Filter nach dem Push-Prinzip wird wie eine **Prozedur** aufgerufen, ein passiver Filter nach dem Pull-Prinzip entspricht einer **Funktion**.

Beim Zusammenschalten von zwei passiven Filtern wird keine Pipe benötigt, da keine Notwendigkeit zur asynchronen Entkopplung dieser Filter besteht.



9.2.3.3 Kommunikationsdiagramm

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 in Form eines Kommunikationsdiagramms der Analyse skizziert. Anschließend werden die Teilnehmer vorgestellt. Kapitel 9.2.3.5 verdeutlicht das dynamische Verhalten.

Anstelle eines sonst hier üblichen Klassendiagramms des Entwurfs wird im Folgenden ein Kommunikationsdiagramm der Analyse nach UML für Objekte vom Typ **Datenquelle**, **Pipe**, **Filter** und **Datensenke** gezeigt, wobei hier nur ein einziger Filter dargestellt ist. In einem **Kommunikationsdiagramm der Analyse** werden nur die Nachrichten und noch nicht die Aufrufe des Entwurfs mit Methodennamen eingetragen, da diese in der Analyse noch gar nicht bekannt sind. Das Kommunikationsdiagramm visualisiert die Vorzugsrichtung für den Datenstrom beim Muster Pipes and Filters:

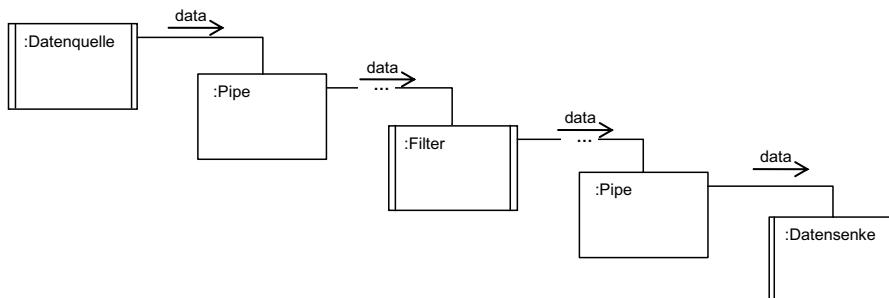


Bild 9-14 Ein Kommunikationsdiagramm der Analyse des Musters Pipes and Filters

Die Klassen Datenquelle, Filter und Datensenke sind in diesem Beispiel aktive Klassen. Eine aktive Klasse¹¹³ ist eine Klasse, deren Objekte einen oder mehrere Threads bzw. Betriebssystem-Prozesse beinhalten.

9.2.3.4 Teilnehmer

Das Architekturmuster Pipes and Filters umfasst die folgenden Teilnehmer:

- **Datenquelle**

Eine **Datenquelle** ist der **Sender von Daten**. 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 (Push-Prinzip). 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) bzw.
- 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 beispielsweise als Thread oder Betriebssystem-Prozess gestartet und fordern zyklisch oder ereignisorientiert Eingabedaten an bzw. senden diese.

- **Datensenke**

Eine Datensenke ist der Empfänger von Daten. Datensenken können sich **aktiv** oder **passiv** verhalten. Aktiv verhält sich eine Senke, wenn sie zyklisch oder ereignisbasiert Daten anfordert (Pull-Prinzip).

¹¹³ Eine aktive Klasse hat seit UML 2.0 jeweils eine doppelte Linie rechts und links.

9.2.3.5 Dynamisches Verhalten

In diesem Kapitel werden 4 verschiedene **Szenarien**¹¹⁴ 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. Mit d_X werden die Daten bezeichnet, die von Filter X erzeugt wurden.

Im Folgenden werden diese 4 Szenarien vorgestellt:

- **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 wiederum 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 (siehe [Bus98]) dargestellt:

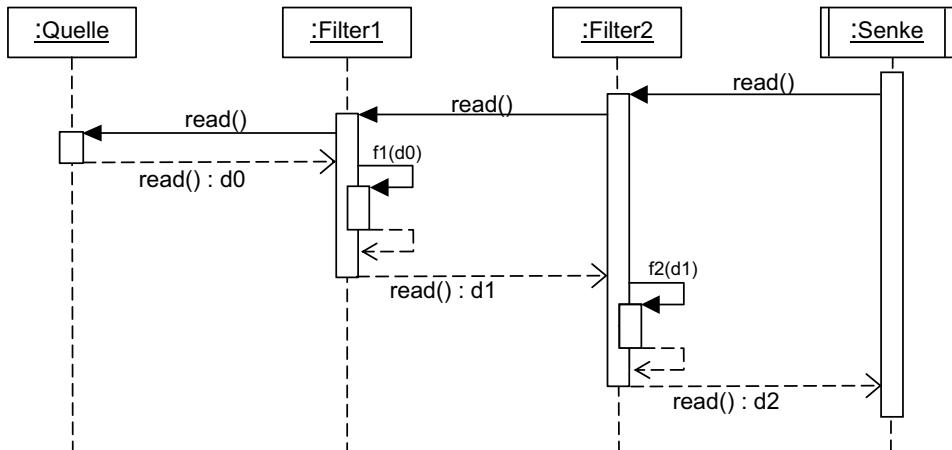


Bild 9-15 Sequenzdiagramm für das Pull-Prinzip

¹¹⁴ In den 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 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 9-16). 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 (siehe [Bus98]):

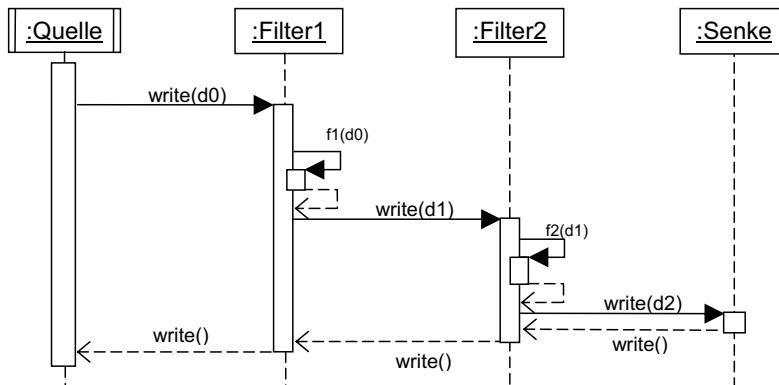


Bild 9-16 Sequenzdiagramm für das Push-Prinzip

- **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 9-17.

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

bearbeitet die übergebenen Daten und gibt das Ergebnis als Rückgabewert zurück. Der Aufruf von `write()` durch Filter 2 entspricht dem **Push-Prinzip**. Hier das Sequenzdiagramm für dieses Szenario:

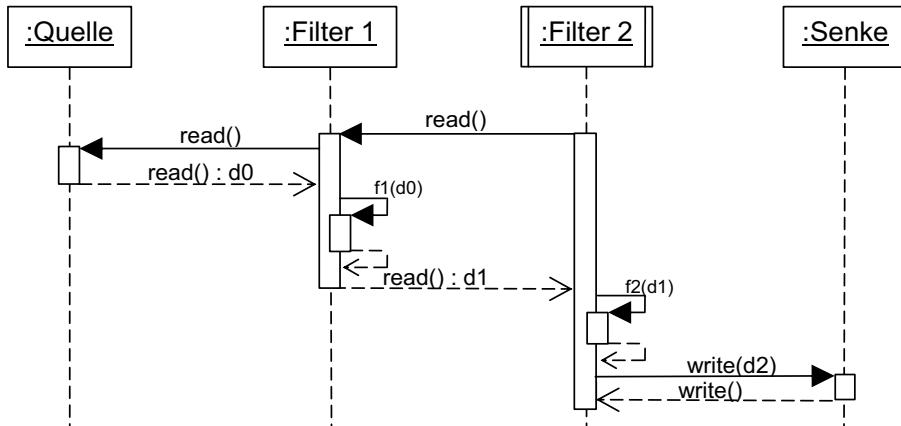


Bild 9-17 Sequenzdiagramm Pull- und Push-Prinzip gemischt

- **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 zwei 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 asynchron entkoppelt**. Die anderen Pipes können weggelassen werden, weil hier nicht zwei Prozesse asynchron entkoppelt werden müssen.

Das Szenario 4 ist in Bild 9-18 zu sehen. Als erstes erfolgt ein Zugriff des aktiven Filters 3 auf die benachbarte Pipe mit `read()`. Da diese Pipe 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:

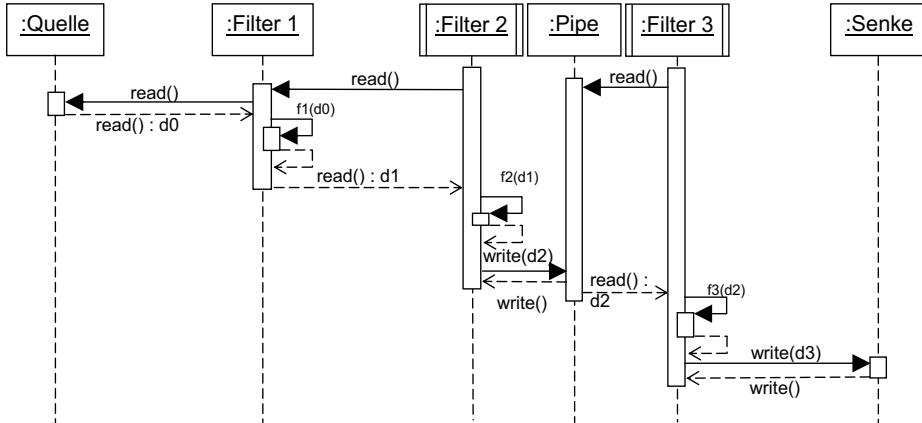


Bild 9-18 Asynchrone Entkopplung von parallelen Filtern

Alle Methodenaufrufe in diesem Diagramm sind synchron. Trotzdem sind die beiden aktiven Elemente asynchron entkoppelt. Die Entkopplung wird durch die Implementierung der Pipe bewerkstelligt.



9.2.4 Bewertung

Beim Muster Pipes and Filters findet man einen klassischen Trade-Off: Will man die Flexibilität der Filter erhöhen, muss man ein allgemeines Transportformat nehmen, das alle Filter nutzen können, weil sie dann beliebig getauscht werden können. So ist es z. B. bei den Pipes in UNIX: deren Transportformat ist Text. Damit verringert sich aber die Performance, denn nun muss jeder Filter die Transportdaten in das Format wandeln, in dem er sie bearbeiten kann und will, und anschließend muss er die Daten wieder in das allgemeine Transportformat konvertieren. Im Beispiel der Pipes von UNIX muss jeder Filter, der mit Zahlen arbeitet, den an kommenden Text in die Binärdarstellung der Zahlen konvertieren und bei der Ausgabe wiederum in umgekehrter Richtung. Will man die Performance der Pipeline verbessern, dann müssen genau diese Konvertierungen vermieden werden – unter der Annahme, dass die eigentliche Verarbeitung in den Filtern performant ist. Um die Konvertierungen zu vermeiden, muss das Transportformat zwischen zwei Filtern individuell definiert werden. Damit ist es aber praktisch unmöglich, diese Filter in anderen Konstellationen wiederzuverwenden. Die so entstandenen Filter verlieren ihre Flexibilität.

Dieser Trade-off zwischen Flexibilität und Performanz muss bei der Anwendung des Musters Pipes and Filters beachtet werden. Weitere Vor- bzw. Nachteile beim Einsatz dieses Musters werden nun im Folgenden diskutiert.

9.2.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Flexibilität gegenüber Modifikationen**

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 ist, hängt jedoch von der Spezifikation des Formats der Datenkanäle ab. Quelle und Senke können bei entsprechendem Transportformat ebenfalls leicht ausgetauscht werden. Filter können auch leicht zusammengefasst werden.

- **leichte Wiederverwendung**

Filter können sehr leicht in anderen Filterketten wiederverwendet werden.

- **leichtes Rapid Prototyping**

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. Hat die Pipeline nicht die gewünschten Eigenschaften beispielsweise bzgl. der Performance, dann können gezielt einzelne Komponenten des Prototyps optimiert werden.

- **Entkopplung von Komponenten**

Nicht benachbarte Verarbeitungsstufen teilen keine Informationen und sind daher entkoppelt.

- **Speicherung von Zwischenergebnissen möglich**

Das Speichern von Zwischenergebnissen ist nicht notwendig, aber möglich

- **parallele Entwicklung**

Die Aufgaben der Filter können bis zu einem gewissen Grad parallel abgearbeitet werden, wenn aktive Filter nebenläufig arbeiten. Dies setzt allerdings voraus, dass dabei ein Datenstrom vorliegt, der von den verschiedenen Filtern bearbeitet werden kann.

9.2.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **Schwierigkeiten bei globalen Daten**

Globale Daten sind schwierig in einer Pipeline umzusetzen. Entweder werden die globalen Daten in einem eigenen Modul realisiert, dann werden die betroffenen Filter von diesem Modul abhängig und können nicht mehr flexibel in einer anderen Pipeline eingesetzt werden oder die globalen Daten müssen zusätzlich über die Pipeline transportiert werden.

- **harte Codierung**

Passive Filter werden direkt aufgerufen. Die Pipeline ist an den entsprechenden Stellen hart codiert. Damit geht die Unabhängigkeit und die Flexibilität der Komponenten verloren.

- **schwierige Fehlerbehandlung**

Die Realisierung einer Fehlerbehandlung ist schwierig, da im System kein gemeinsamer Zustand der Pipeline-Komponenten existiert. Die Fehlerbehandlung wird daher häufig vernachlässigt (siehe [Bus98]).

- **Bottleneck durch den langsamsten Filter**

Der langsamste Filter in der Kette bestimmt die Verarbeitungsgeschwindigkeit. Es gibt keine vollständige Parallelisierung, da die Filter aufeinander warten müssen.

- **Aufwand für das Konvertieren der Daten**

Ist das Format der Datenkanäle zu allgemein spezifiziert, können große Aufwände bei der Datenkonvertierung in einzelnen Filtern entstehen. Je nach Schnittstellenspezifikation der Filter können viele Konvertierungen notwendig sein.

- **Probleme bei der Wartbarkeit**

Änderungen an den Anforderungen können sich oft auf mehrere Bausteine erstrecken.

9.2.5 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:

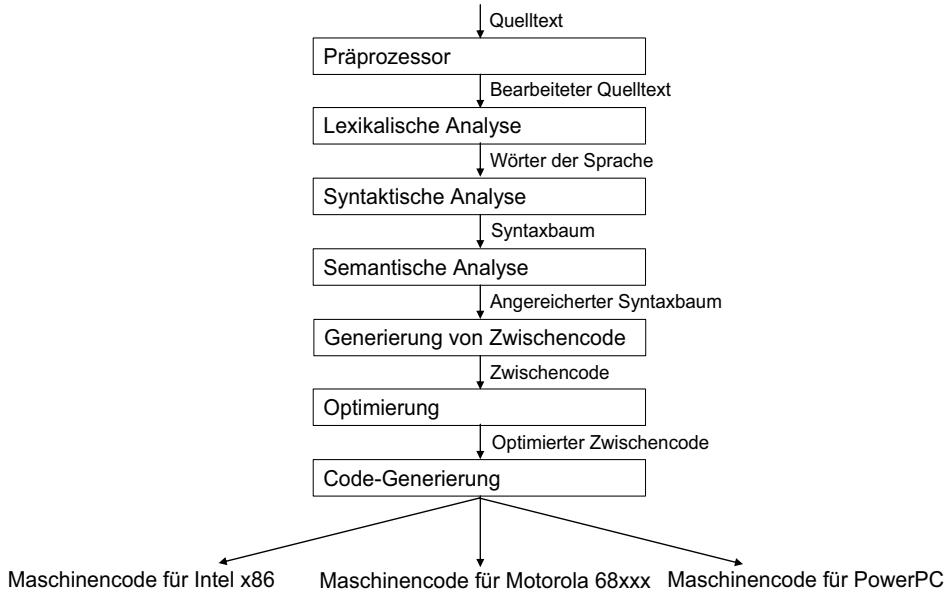


Bild 9-19 Sequenzielle Phasen eines Compilers

9.2.6 Ähnliche Muster

Sowohl beim Architekturmuster **Layers** als auch bei **Pipes and Filters** wird eine Anwendung funktional 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 Aufrufsstellen 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 flexibel und wiederverwendbare Mechanismen. Ein Dekorierer ist jedoch an die dekorierte Klasse gebunden und ist 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 eine Verarbeitung entlang gerichteter Graphen realisieren, bei denen Abzweigungen (Tee), parallel verlaufende Stränge von Filtern und das Zusammenführen (Join) von solchen Strängen möglich sind.

9.3 Zusammenfassung

Die Architekturmuster zur Strukturierung eines Systems in Kapitel 9 umfassen die Architekturmuster Layers sowie Pipes and Filters.

Das Architekturmuster Layers in Kapitel **9.1** 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 einer untergeordneten Schicht aufrufen. Zwischen den Schichten existieren feste Verträge/Schnittstellen.

Ein System, das einen Datenstrom in einer einzigen Richtung verarbeitet, kann mit Hilfe des Architekturmusters Pipes and Filters von Kapitel **9.2** in Komponenten strukturiert werden. Die Aufgabe des gesamten Systems wird dabei in einzelne sequenzielle Verarbeitungsstufen zerlegt. Jeder Verarbeitungsschritt wird in Form einer Filter-Komponente implementiert. Jeweils zwei Filter können durch eine Pipe als Komponente verbunden werden. Eine Pipe wird eingesetzt, wenn eine asynchrone Entkopplung der Filter erforderlich ist. Generell ist die Ausgabe des einen Filters die Eingabe des anderen Filters.

9.4 Kontrollfragen

Aufgaben 9.4.1: Layers

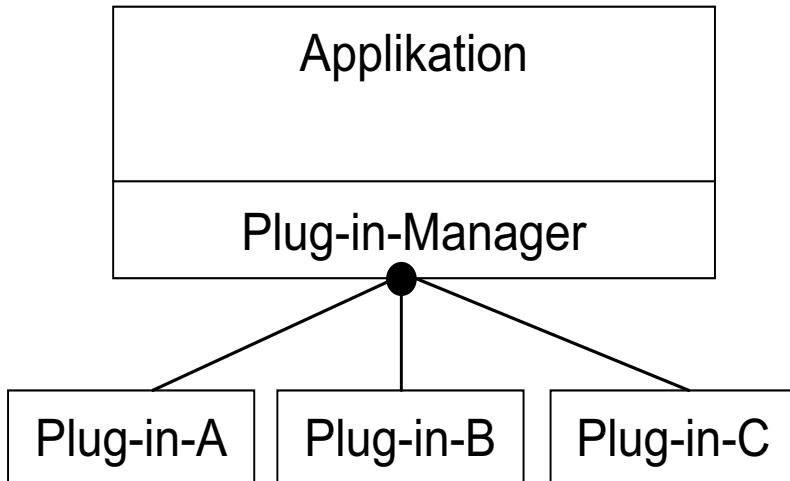
- 9.4.1.1 Erklären Sie die Grundzüge des Musters Layers.
- 9.4.1.2 Kann eine Schicht n die Dienste einer Schicht n-2 direkt nutzen?

Aufgaben 9.4.2: Pipes and Filters

- 9.4.2.1 Erklären Sie die Grundzüge des Musters Pipes and Filters.
- 9.4.2.2 Was ist der Unterschied zwischen einem aktiven und einem passiven Filter?

Kapitel 10

Architekturmuster für adaptierbare Systeme



- 10.1 Das Architekturmuster Plug-in
- 10.2 Zusammenfassung
- 10.3 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_10.

10 Architekturmuster für adaptierbare Systeme

Adaptierbare Systeme ermöglichen es, dass Softwaresysteme auch in Situationen, die während ihrer Entwicklung nicht vorhersehbar waren, mit Erfolg eingesetzt werden können. Neue Funktionen müssen hinzugefügt und existierende Dienste müssen geändert werden können. Ebenso müssen beispielsweise neue Versionen von Betriebssystemen oder Frameworks eingeführt werden können.

Bei einem adaptierbaren System kann ein Drittanbieter eine lauffähige Software erweitern, ohne die Quellen der vorhandenen Software zu kennen. In Kapitel 10.1 wird hierfür das Architekturmuster Plug-in erklärt.

10.1 Das Architekturmuster Plug-in

Das Architekturmuster Plug-in stellt eine Architektur für ein **adaptierbares System** dar.

10.1.1 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.

10.1.2 Problem

Anwendungssoftware soll stabil sein. Sie soll möglichst lange eingesetzt und dabei angepasst oder erweitert werden können. Dabei soll vorhandene lauffähige Software nicht abgeändert werden, sie soll hingegen flexibel – auch durch dritte Parteien – erweitert werden können. Die Forderung, vorhandene Systeme zu erweitern ohne sie zu verändern, ist vom **Open-Closed-Prinzip** her bekannt (siehe Kapitel 3.1.5).

10.1.3 Lösung

Ein System, das im laufenden Betrieb flexibel anpassbar oder erweiterbar sein soll, kann nach dem Architekturmuster Plug-in entworfen und realisiert werden.

Das Architekturmuster Plug-in strukturiert eine Anwendung in der Weise, dass diese über **Erweiterungspunkte in Form von Schnittstellen** verfügt, an denen dann die verschiedenen Plug-ins, welche diese Schnittstellen implementieren, eingehängt werden können.



Die Architektur der Anwendungssoftware muss also **Schnittstellen** definieren, an deren Stelle **zur Laufzeit Komponenten, welche diese Schnittstelle implementieren**, treten können. Plug-ins sind also speziell für eine spezifische Applikation, welche die Schnittstellen vorgibt, und nicht für andere Applikationen vorgesehen. 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 geforderten Schnittstelle an. Ein Plug-in ist in der Regel nicht eigenständig ausführbar.

Eine Applikation definiert **Erweiterungspunkte** in Form von Schnittstellen. Ein Plug-in, das eine dieser Schnittstellen implementiert, stellt eine **Erweiterung** für diesen Erweiterungspunkt dar.



Ein Plug-in ist also eine Softwarekomponente, die einem System neue, zusätzliche oder angepasste Funktionalitäten zur Verfügung stellt. Dritte können unabhängig von den internen Abläufen der Anwendungssoftware solche Funktionalitäten in Form von Plug-ins entwickeln, wenn die zu implementierenden Schnittstellen bzw. die Methodensignaturen einer Anwendungssoftware offengelegt werden.

Ein Plug-in kann selbst auch weitere Schnittstellen zur Erweiterung definieren und kann damit durch eine weitere Ebene von Plug-ins (Kaskade 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. Darauf wird im Folgenden aber nicht weiter eingegangen.

Zum Einfügen von Plug-ins zur Laufzeit dient ein weiterer Teilnehmer dieses Musters, der sogenannte **Plug-in-Manager**.

Der wesentliche Punkt bei einer Plug-in-Architektur ist die vollständige Entkopplung einer speziellen Applikation vom Prozess der Instanzierung von Objekten der Plug-in-Klassen. Plug-ins werden über den Plug-in-Manager instanziert.



Das folgende Bild zeigt den beispielhaften Aufbau einer Plug-in-Architektur:

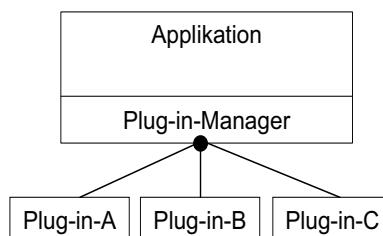


Bild 10-1 Konzept einer Plug-in-Architektur

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. 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.

Ein sogenannter Plug-in-Manager muss zur Laufzeit entscheiden können, welches der verfügbaren Plug-ins zur Abarbeitung der aktuellen Aufgabe passt.



Für die Implementierung von Plug-in-Architekturen gibt es bereits eine Vielzahl von Frameworks, welche 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 – möglicherweise sogar unter Einbeziehung einer Versionsnummer. Das Spring Framework oder die Eclipse Rich Client Platform (kurz Eclipse RCP) bieten u. a. auch Hilfsmittel für eine Plug-in-Architektur in Java.

Bei einer eigenen Implementierung eines Plug-in-Managers müssen die folgenden Punkte geklärt werden, die das Muster offen lässt:

- Wie werden passende Plug-ins gefunden?
Es ist zu entscheiden, ob ein zu ladendes Plug-in beispielsweise über eine selbst-definierte Properties-, XML- oder Textdatei gefunden und anschließend mittels z.B. Java-Reflection instanziert wird oder eine darauf spezialisierte Unterstützung der Laufzeitumgebung wie z.B. ein Aufruf des Java ServiceLoaders genutzt wird.
- Wie werden gefundene Plug-ins der Applikation zur Verfügung gestellt?
Der Plug-in-Manager gibt an die Applikation eine Referenz an ein erzeugtes Plug-in-Objekt weiter und die Applikation benutzt dieses Objekt im weiteren Programmablauf über diese Referenz. Alternativ kann die Applikation Aufrufe an den Plug-in Manager weiterleiten, der diese auf dem entsprechenden Objekt ausführt.
- Können mehrere Plug-ins zur gleichen Schnittstelle gleichzeitig von der Applikation benutzt werden?
In diesem Falle müsste die Applikation entweder selbst die verschiedenen Plug-ins unterscheiden und verwalten können oder müsste dem Plug-in-Manager mitteilen, auf welchem Plug-in-Objekt eine Operation ausgeführt werden soll.

Für die weitere Beschreibung des Musters werden die Entscheidungen zu diesen Fragen gemäß dem Ansatz getroffen, der in [Fow03] veröffentlicht ist. Der Plug-in-Manager findet eine Beschreibung der Plug-ins zu einer Schnittstelle in einer Textdatei, die er interpretieren kann und somit eine zur Schnittstelle passende Klasse finden kann. Diese Klasse kann er instantiiieren und das erzeugte Objekt der Applikation als Plug-in zur Verfügungen stellen. Der Plug-in-Manager stellt zu einem Zeitpunkt immer nur ein Plug-in zu einer Schnittstelle zur Verfügung. Bei Bedarf kann die Applikation ein anderes Plug-in anfordern und so die Funktionalität ändern. Dieser Ansatz ist auch im Programmbeispiel in 10.1.3.4 zu sehen.

10.1.3.1 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 instanzieren. Das Interface `IPlug-in` stellt dabei die von der Anwendungssoftware definierte Schnittstelle dar.

Hier das Klassendiagramm einer Plug-in-Architektur:

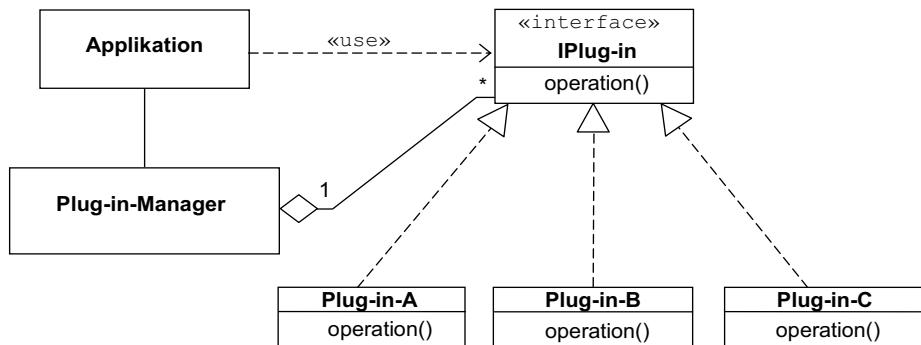


Bild 10-2 Klassendiagramm einer Plug-in-Architektur mit einem einzigen Interface

Der Plug-in-Manager ermittelt mit Hilfe der Laufzeitumgebung verfügbare Plug-ins für die Schnittstelle `IPlug-in` und instanziert passende Objekte (im Bild aus Gründen der Übersichtlichkeit nicht zu sehen). In einer ersten Variante ruft die Applikation ein so erzeugtes Objekt vom Plug-in-Manager ab und benutzt dieses Objekt im weiteren Programmablauf. Alternativ können die instanzierten Objekte auch dauerhaft vom Plug-in-Manager aggregiert werden. Die Applikation leitet dann die Aufrufe an den Plug-in-Manager weiter, der diese auf den entsprechenden Objekten ausführt.

10.1.3.2 Teilnehmer

Die Teilnehmer im Klassendiagramm sind:

- **Applikation**

Eine Applikation repräsentiert diejenige Komponente, die zur Erweiterung ihrer Funktionalität andere Komponenten in Form von Plug-ins nutzt. Sie stellt die Anwendung dar, die gestartet wird. Die Applikation gibt die Schnittstelle für die gewünschte Funktionalität vor, die die Plug-ins implementieren müssen.

- **Plug-in-Manager**

Ein Plug-in-Manager kennt alle verfügbaren Plug-ins anhand der Schnittstelle, welche die Plug-ins 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. Der Plug-in-Manager sucht passende Plug-ins zu dieser Schnittstelle.

- **Plug-in-X (X = A..Z)**

Ein Plug-in erweitert die Applikation um Funktionalität und implementiert die von der Applikation vorgegebene Schnittstelle. Dabei wird ein Plug-in vom Plug-in-Manager instanziert und der Applikation zur Verfügung gestellt. Die Applikation kann dann die Funktionalität des Plug-ins nutzen.

10.1.3.3 Dynamisches Verhalten

Die Applikation definiert eine Schnittstelle zur Erweiterung und stellt Informationen darüber beispielsweise in einer Datei, die dem Plug-in-Manager bekannt ist, zur Verfügung. Unter Verwendung der Informationen in dieser Datei¹¹⁵ findet der Plug-in-Manager passende Klassen, instanziert diese und verwaltet die so erzeugten Plug-ins. Diese Tätigkeit des Plug-in-Managers ist sehr implementierungsspezifisch und wird im Sequenzdiagramm (Bild 10-3) nicht im Detail gezeigt – sie verbirgt sich hinter dem Aufruf von `ladeAllePlug-ins()` wobei der Name der Schnittstelle als Parameter übergeben wird. Im Programmbeispiel in Kapitel 10.1.3.4 ist eine Möglichkeit zur Implementation zu sehen.

Die Applikation kann nun zur Laufzeit Plug-ins zu einer gewünschten Schnittstelle erhalten (`gibPlug-in()`)¹¹⁶. Im folgenden Sequenzdiagramm wird angenommen, dass die Identifikation über den Namen der Klasse eines Plug-ins erfolgt. Die Applikation kann nun die in der Schnittstelle definierten Funktionen eines Plug-ins aufrufen. Das folgende Bild setzt voraus, dass `Plug-in-A` und `Plug-in-B` beide die gewünschte Schnittstelle `IPlug-in` implementieren und dass `operation()` eine in der Schnittstelle definierte Funktion ist.

Im Sequenzdiagramm wird auch gezeigt, wie die Applikation ein weiteres Plug-in anfordern kann. Danach wird ein Teil der Funktionalität der Applikation geändert sein, je nachdem wie das zweite Plug-in die Schnittstelle implementiert.

Hier nun das Sequenzdiagramm des Musters Plug-in:

¹¹⁵ Der Aufbau und Inhalt einer solchen Datei ist im Beispiel des Kapitels 10.1.3.4 gezeigt und beschrieben. Im dort gezeigten Beispiel ist der Name der Datei `META-INF/services/IWoerterbuch`.

¹¹⁶ Der Plug-in-Manager kann auch so implementiert werden, dass ein Plug-in erst bei diesem Aufruf instantiiert wird, anstatt alle gefundenen Plug-ins auf einmal zu erzeugen.

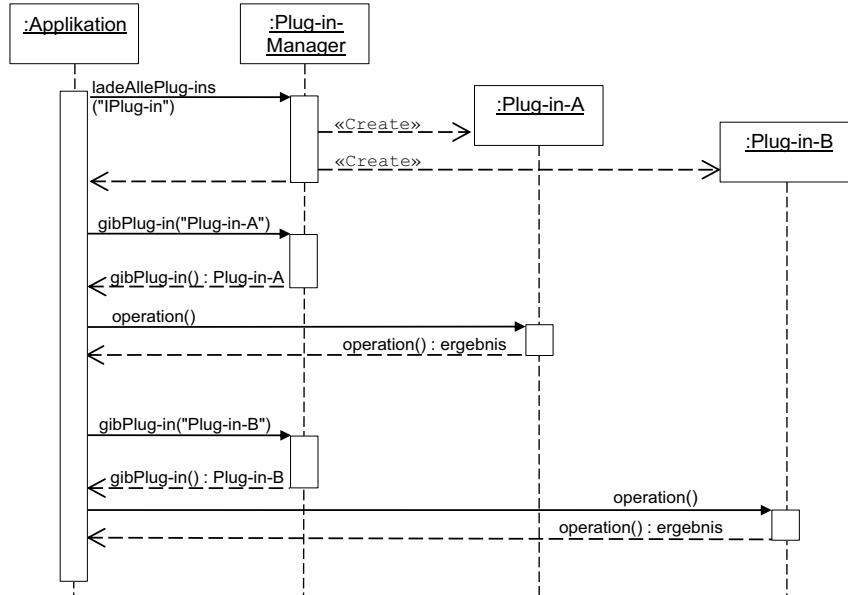


Bild 10-3 Beispiel für ein Sequenzdiagramm eines Basisablaufs nach der Plug-in-Architektur

Wie eine Applikation reagiert, wenn der Plug-in-Manager kein passendes Plug-in findet, welches die gewünschte Schnittstelle implementiert, wird durch das Muster nicht vorgegeben. In der Regel wird dann eine Fehlerbehandlung ausgelöst. Möglicherweise kann aber auch nur der gewünschte Anwendungsfall (z. B. die Wiedergabe einer Musikdatei) nicht durchgeführt werden. Dann kann also die Applikation weiter ausgeführt werden, obwohl das Plug-in nicht geladen werden konnte.

10.1.3.4 Programmbeispiel

An dem folgenden Anwendungsbeispiel eines Wörterbuchs soll eine Plug-in-Architektur in der Programmiersprache Java veranschaulicht werden.

In diesem Beispiel wird eine Anwendung zur Übersetzung von Wörtern von einer Sprache in eine andere implementiert. Der Benutzer dieser Anwendung gibt Wörter ein, die von der Anwendung übersetzt werden aufgrund der Informationen, die in Wörterbüchern enthalten sind. Wörterbücher sollen als Plug-ins geladen werden, da unterschiedliche Benutzer unterschiedliche Sprachpaare für die Übersetzung nutzen wollen. Ein Benutzer muss also zuerst ein Sprachpaar auswählen, damit das passende Plug-in geladen werden kann. Er kann später auch seine Auswahl ändern und ein anderes Wörterbuch benutzen.

Ein Wörterbuch muss die Schnittstelle `IWoerterbuch` implementieren. Die Klasse `WoerterbuchManager` kümmert sich um das Laden eines Wörterbuchs als Plug-in.

Es folgt die Schnittstelle `IWoerterbuch`, die für alle Wörterbücher die Methoden `getSprache()` und `getÜbersetzung(String wort)` vorschreibt:

```
// Datei: IWoerterbuch.java

public interface IWoerterbuch {

    // Liefert das Sprachpaar des Wörterbuchs zurück
    String getSprache();

    // Übersetzt das übergebene Wort
    String getUebersetzung(String wort);

}
```

Beispielhaft für Wörterbücher werden im Folgenden zwei Klassen gezeigt. Die Klasse DeutschEnglischWoerterbuch bietet die gewünschte Funktionalität für das Übersetzen von deutschen Begriffen in englische Begriffe. Die Klasse DeutschEnglischWoerterbuch implementiert das Interface IWoerterbuch:

```
// Datei: DeutschEnglischWoerterbuch.java

import java.util.*;

public class DeutschEnglischWoerterbuch implements IWoerterbuch {

    private static final Map<String, String> WOERTER_LISTE
        = initWoerterListe();

    @Override
    public String getSprache() {
        return "Deutsch-Englisch";
    }

    @Override
    public String getUebersetzung(String wort) {
        String uebersetzung = WOERTER_LISTE.get(wort);
        if (uebersetzung != null)
            return uebersetzung;
        else
            return "Fehler: Wort nicht im Wörterbuch!";
    }

    // Initialisiert die Wörterliste
    private static Map<String, String> initWoerterListe() {
        Map<String, String> result = new HashMap<>();
        result.put("Hallo", "Hello");
        result.put("Auf Wiedersehen", "Goodbye");
        return Collections.unmodifiableMap(result);
    }
}
```

Weiterhin steht ein Deutsch-Französisch-Wörterbuch zur Verfügung. Dazu dient die Klasse DeutschFranzoesischWoerterbuch, welche die Schnittstelle IWoerterbuch implementiert, damit sie als Plug-in in der Anwendung verwendet werden kann:

```
// Datei: DeutschFranzoesischWoerterbuch.java

import java.util.*;

public class DeutschFranzoesischWoerterbuch implements IWoerterbuch {

    private static final Map<String, String> WOERTER_LISTE
        = initWoerterListe();

    @Override
    public String getSprache() {
        return "Deutsch-Französisch";
    }

    @Override
    public String getUebersetzung(String wort) {
        String uebersetzung = WOERTER_LISTE.get(wort);
        if (uebersetzung != null)
            return uebersetzung;
        else
            return "Fehler: Wort nicht im Wörterbuch!";
    }

    // Initialisiert die Wörterliste
    private static Map<String, String> initWoerterListe() {
        Map<String, String> result = new HashMap<>();
        result.put("Hallo", "Salut");
        result.put("Auf Wiedersehen", "Au revoir");
        return Collections.unmodifiableMap(result);
    }
}
```

Die Klasse `WoerterbuchManager` entspricht dem Plug-in-Manager aus der allgemeinen Beschreibung des Musters. Sie wurde als Singleton realisiert, weil nur ein Exemplar für die Anwendung benötigt wird. Für das Laden von Plug-ins wird die externe Komponente `ServiceLoaders` (seit Java 1.6 im JRE enthalten) eingesetzt. Ein `ServiceLoader` kann dynamisch Wörterbücher (Plug-ins) einbinden. Voraussetzung hierfür ist, dass im Quellcodeverzeichnis ein Ordner mit dem Namen `META-INF/services` erzeugt wird. In diesem Ordner muss eine Textdatei angelegt werden, die den Paket- und Klassennamen der Wörterbuch-Schnittstelle trägt (hier: `IWoerterbuch`). Diese Datei enthält Paket- und Klassennamen, in denen Wörterbücher für verschiedene Sprachen zu finden sind und auch zur Laufzeit geändert bzw. ergänzt werden können. Die Datei `IWoerterbuch` wird im Folgenden dargestellt:

Datei: META-INF/services/IWoerterbuch

```
DeutschEnglischWoerterbuch
DeutschFranzoesischWoerterbuch
```

Der Einsatz des `ServiceLoader` bedingt, dass alle Wörterbücher einen Standardkonstruktor haben müssen. Dies ist erforderlich, damit der `ServiceLoader` eine

Instanz dieser Klasse dynamisch erzeugen kann. Es folgt nun der Quelltext der Klasse WoerterbuchManager:

```
// Datei: WoerterbuchManager.java

import java.util.*;

public enum WoerterbuchManager {

    // singleton
    INSTANCE;

    // Die vorhandenen Wörterbücher laden
    private final Map<String, IWoerterbuch> woerterbuecher
        = initWoerterbuecher();

    // Liefert die Sprachen, für die Wörterbücher vorhanden sind
    public Collection<String> getSprachen() {
        return woerterbuecher.keySet();
    }

    // Gibt das Wörterbuch zu einer Sprache zurück
    public IWoerterbuch getWoerterbuch(String sprache) {
        return woerterbuecher.get(sprache);
    }

    // Lädt die Wörterbücher über den ServiceLoader
    private Map<String, IWoerterbuch> initWoerterbuecher() {
        Map<String, IWoerterbuch> result = new LinkedHashMap<>();

        // Passende Plug-ins für das Interface IWoerterbuch laden
        ServiceLoader<IWoerterbuch> woerterbuecher =
            ServiceLoader.load(IWoerterbuch.class);
        for (IWoerterbuch eachWoerterbuch : woerterbuecher) {
            result.put(eachWoerterbuch.getSprache(), eachWoerterbuch);
        }
        return Collections.unmodifiableMap(result);
    }
}
```

Die Klasse Client enthält die `main()`-Methode und stellt die Anwendung dar, die mittels Plug-ins – in diesem Beispiel sind dies Wörterbücher – erweitert werden kann. Sie simuliert Eingaben des Benutzers zur Wahl eines Wörterbuches und zum Übersetzen von Wörtern gemäß dem gewählten Wörterbuch – also mit Hilfe eines Plug-ins. Wörterbücher werden vom WoerterbuchManager verwaltet und dem Client zur Verfügung gestellt. Hier die Klasse Client:

```
// Datei: Client.java

public class Client {

    // WoerterbuchManager besorgen
    private static WoerterbuchManager wm = WoerterbuchManager.INSTANCE;

    // Platzhalter für Plug-in
    private static IWoerterbuch wb;

    public static void main(String[] args) {

        // Alle verfügbaren Wörterbücher ausgeben
        System.out.println("Alle vorhandenen Wörterbücher:");
        wm.getSprachen().forEach(System.out::println);

        // Wahl des Wörterbuchs zum Übersetzen
        String sprache = "Deutsch-Englisch";
        waehleWoerterbuch(sprache);

        // Der Benutzer gibt Wörter zum Übersetzen ein
        uebersetze("Hallo");
        uebersetze("Auf Wiedersehen");

        // Der Benutzer wählt ein anderes Wörterbuch zum Übersetzen
        waehleWoerterbuch("Deutsch-Französisch");

        // Der Benutzer gibt Wörter zum Übersetzen ein
        uebersetze("Hallo");
        uebersetze("Auf Wiedersehen");
        uebersetze("Guten Tag");
    }

    // Laden eines Wörterbuchs mit Fehlerbehandlung
    private static void waehleWoerterbuch(String sprache) {
        wb = wm.getWoerterbuch(sprache);
        if (wb == null) {
            System.out.println (
                "Fehler: Wörterbuch konnte nicht geladen werden:" + sprache);
            System.exit(99);
        }
        else
            System.out.println ("\nWörterbuch " + sprache + " geladen.");
    }

    // Übersetzung eines Wortes mit Ausgabe
    private static void uebersetze(String wort) {
        System.out.println (wort + " --> " + wb.getUebersetzung(wort));
    }
}
```



Zuletzt das Protokoll des Programmelaufs:

Alle vorhandenen Wörterbücher:

Deutsch-Englisch

Deutsch-Französisch

Wörterbuch Deutsch-Englisch geladen.

Hallo --> Hello

Auf Wiedersehen --> Goodbye

Wörterbuch Deutsch-Französisch geladen.

Hallo --> Salut

Auf Wiedersehen --> Au revoir

Guten Tag --> Fehler: Wort nicht im Wörterbuch!

10.1.4 Bewertung

10.1.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Einhalten von Separation of Concerns**

Jedes Plug-in besitzt seine eigene Zuständigkeit (**separation of concerns**, siehe Kapitel 3.1.3).

- **robustes Verhalten**

Plug-in-Architekturen zeigen in der Regel ein robustes Verhalten.

- **Erweiterbarkeit ohne Kenntnis des Programmcodes der zu erweiternden Software**

Der Mechanismus von Plug-ins bietet im Gegensatz zu Bibliotheken bzw. 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.

- **"schlankes" System**

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.

- **Wartungsaufwand wird kleiner**

Der Wartungsaufwand sinkt, da die in Plug-ins ausgelagerte Funktionslogik für einen Anwendungsfall angepasst und als neue Version eines Plug-ins ausgeliefert werden kann.

- **bequemes Teile und Herrsche**

Die Entwicklung komplexer Software kann bequem aufgeteilt werden.

- **verschiedene Versionen der Plug-ins möglich**

Plug-ins können in mehreren verschiedenen Versionen vorhanden sein und auch gleichzeitig ausgeführt werden, wenn der Plug-in-Manager dies unterstützt.

- **unabhängige Tests der verschiedenen Komponenten**

Jedes Plug-in kann getrennt von den anderen Plug-ins getestet werden. Ein Gesamtsystemtest ist leichter, wenn komponentenweise getestet werden kann, da die Komplexität dann geringer ist ("divide et impera" beim Testen).

10.1.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **initialer Aufwand der Implementierung höher**

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.

- **Verwaltungsaufwand bei der Ausführung einer Anwendung steigt**

Der Verwaltungsaufwand während der Ausführung einer speziellen Anwendung steigt.

- **schwieriges Finden einer gemeinsamen Schnittstelle bei Erweiterungen mit Varianten**

Es kann manchmal schwer sein, eine gemeinsame Schnittstelle zu finden, die den Plug-in-Entwicklern möglichst viele Möglichkeiten bietet, aber die Kernanwendung nicht instabil bzw. unsicher macht.

10.1.5 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 in Form von Plug-ins zur Verfügung gestellt werden.

Ein zusätzliches Einsatzgebiet stellt die Möglichkeit der Erweiterung der Applikation dar, falls später festgestellt wird, dass 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 Webbrowsersn. 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.

10.1.6 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 kann dies der Plug-in-Manager übernehmen.

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 bzw. koordiniert den Zugriff auf diese Ressourcen. Daher wird das Muster Microkernel hauptsächlich im Zusammenhang mit dem Entwurf von Betriebssystemen erwähnt.

Das Muster **Fabrikmethode** und das Architekturmuster Plug-in haben ein ähnliches Ziel: ein gewünschtes Objekt erst dynamisch zur Laufzeit zu erzeugen, ohne dass der Typ des Objektes statisch festgelegt ist. Die Methode `gibPlug-in()` in Bild 10-3 wirkt somit wie eine Fabrikmethode, insbesondere wenn das Plug-in-Objekt erst beim Aufruf dieser Methode erzeugt und der Applikation zur Verfügung gestellt wird. Allerdings ist die Methode `gibPlug-in()` gemäß Kapitel 7.1.2 nur eine einfache Fabrikmethode und keine polymorphe, weil die Erzeugung der verschiedenen Plug-in-Objekte nicht in Unterklassen ausgelagert ist, wie es beim Muster Fabrikmethode der Fall ist.

10.2 Zusammenfassung

Das Architekturmuster Plug-in (siehe Kapitel 10.1) stellt eine Architektur für ein **adaptierbares System** dar, bei der ein Drittanbieter eine lauffähige Software erweitern kann, ohne die Quellen der vorhandenen Software zu kennen.

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 in der Form von Plug-ins treten können, welche diese Schnittstelle implementieren. Ist die Schnittstelle offen gelegt, können auch Dritte eine erweiterte Funktionalität für die Anwendungssoftware bereitstellen.

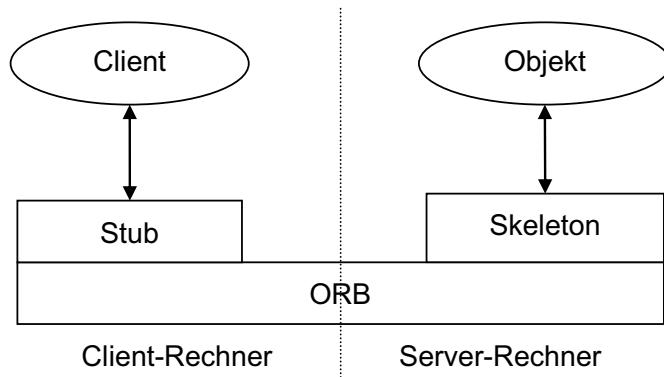
10.3 Kontrollfragen

Aufgaben 10.3.1: Plug-in

- 10.3.1.1 Erklären Sie die Grundzüge des Plug-in-Architekturmusters.
- 10.3.1.2 Welche Aufgaben hat ein Plug-in-Manager?

Kapitel 11

Architekturmuster für verteilte Systeme



11.1 Das Architekturmuster Broker

11.2 Das Architekturmuster Service-Oriented Architecture

11.3 Zusammenfassung

11.4 Aufgaben

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_11.

11 Architekturmuster für verteilte Systeme

Das Architekturmuster **Broker**¹¹⁷ strukturiert ein **verteiltes System** aus mehreren Clients und Servern in eine Architektur mit einem Broker als vermittelnde Instanz für das Finden des passenden Servers bei einer Anfrage eines 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** (kurz SOA) gliedert die Architektur eines Systems in Komponenten bzw. Teilkomponenten, die den Anwendungsfällen bzw. Teilen von Anwendungsfällen aus Sicht der Analyse 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.

Kapitel 11.1 behandelt den Broker und Kapitel 11.2 die Service-Oriented Architecture.

¹¹⁷ Dt. Makler. Üblicherweise wird der englische Begriff „Broker“ verwendet.

11.1 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.

11.1.1 Name/Alternative Namen

Broker heißt "Makler" oder auch "Vermittler"¹¹⁸. Als Name für das Muster ist praktisch nur der englische Begriff "Broker" geläufig. In [Gra02b] wird der Name "Object Request Broker" für das Muster benutzt – in Anlehnung an CORBA, einem wichtigen Einsatzgebiet für das Muster (siehe Kapitel 11.1.5).

11.1.2 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 anderen Komponenten physisch zu erreichen sind 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.

¹¹⁸ Das Architekturmuster Broker entspricht nicht dem Entwurfsmuster Vermittler, siehe dazu Kapitel 11.1.6.

11.1.3 Lösung

In der folgenden Beschreibung des Musters werden die kommunizierenden Komponenten auf Grund ihrer Rolle in einer Kommunikationsbeziehung unterschieden:

- **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 für die 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-Komponenten untereinander nur über den 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 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 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. Eine Server-Klasse bietet ihren Dienst in Form einer öffentlichen Methode an und eine Client-Klasse ruft die Dienstmethode einer 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 einzigen 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 – welches 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 bzw. Deserialisierung übernimmt. In symmetrischer Weise vertritt ein **Server-side Proxy** den **Client auf der Serverseite** und nimmt wiederum die Deserialisierung bzw. Serialisierung 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.

11.1.3.1 Klassendiagramm

Das folgende Klassendiagramm beschreibt die statische Struktur des Architekturmusters Broker:

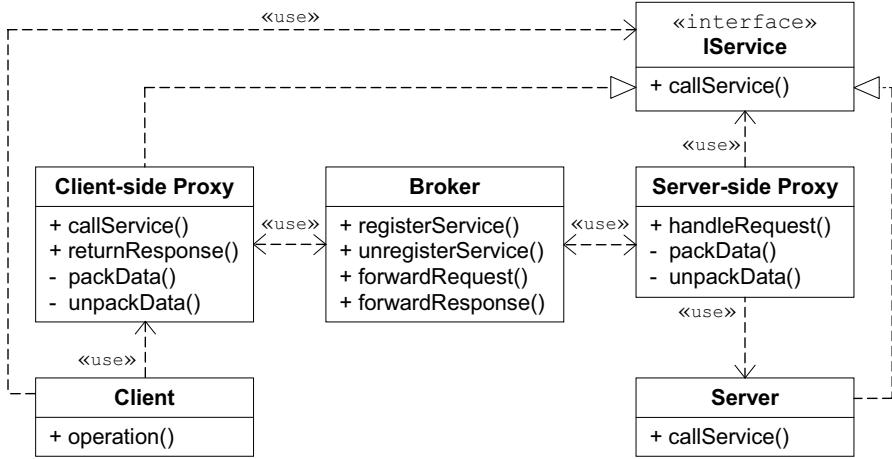


Bild 11-1 Klassendiagramm Broker-Muster

Die gegenseitigen Abhangigkeiten zwischen Broker und den beiden Proxys, die im Bild 11-1 zu sehen sind, sollten in der Praxis mit Hilfe von Interfaces aufgelost werden¹¹⁹. Diese Vorgehensweise wird im Programmbeispiel in Kapitel 11.1.3.4 gezeigt.

Die Klasse `Server` repräsentiert eine Server-Komponente und stellt beispielhaft als Dienst die Methode `callService()` bereit. Die Aufrufsschnittstelle 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 für den Nachrichtenaustausch die Methoden `forwardRequest()` und `forwardResponse()` vorgesehen. Durch Aufruf der Methode `forwardRequest()` des Brokers wird eine Nachricht, die einen Dienstauftrag enthält, ange nommen 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 Dienstauftrags zuständig. Ein Server-side Proxy ruft diese Methode auf und übergibt dabei eine Nachricht, die das Ergebnis des Dienstauftrags enthält. Der Broker reicht die

¹¹⁹ Im Klassendiagramm jedoch wurde wegen der Übersichtlichkeit und auch aus Platzgründen auf diese Interfaces verzichtet.

Nachricht an den Aufrufer 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 11-1 zu sehen ist, implementiert die Klasse `Client-side Proxy` das Interface `IService`. Dies entspricht dem **Entwurfsmuster Proxy** (siehe Kapitel 5.6).

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 der Broker 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.

Ein Client-side Proxy ist von seinem entsprechenden Server-side Proxy inhaltlich abhängig, da er sowohl die Aufrufsschnittstelle des Service kennen muss als auch das Nachrichtenformat, das der Server-side Proxy erwartet, um den Service aufzurufen. Diese Abhängigkeit wurde im Klassendiagramm (Bild 11-1) nicht gezeigt, um das Bild übersichtlich zu halten.



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** röhrt 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 11.1.3.3 genauer eingegangen.

11.1.3.2 Teilnehmer

Das Muster Broker enthält die folgenden 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.

Ein Client muss 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 wie- tergeleitet.

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.

Der Broker muss intern eine Liste der angebotenen Dienste mit den dazugehörigen Servern und Server-side Proxys verwalten.



Natürlich muss der Broker 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 Servers zu erzeugen und dann den Aufruf auch durchzuführen. Der Rückgabewert des Dienstauftrags 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 Dienstauftrags geht wieder auf dem umgekehrten Weg zurück.

11.1.3.3 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 11-2 wird durch punktierte senkrechte Linien¹²⁰ 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 folgendem Bild als "normale" synchrone Methodenaufrufe dargestellt. Diese Aufrufe müssen aber durch entsprechende Mechanismen der Interprozesskommunikation, die das Betriebssystem zur Verfügung stellt, auf dem die Prozesse ablaufen, realisiert werden.



Das dynamische Verhalten der beteiligten Komponenten wird im Folgenden an Hand einer Client-Anfrage beschrieben. Hier nun das Sequenzdiagramm:

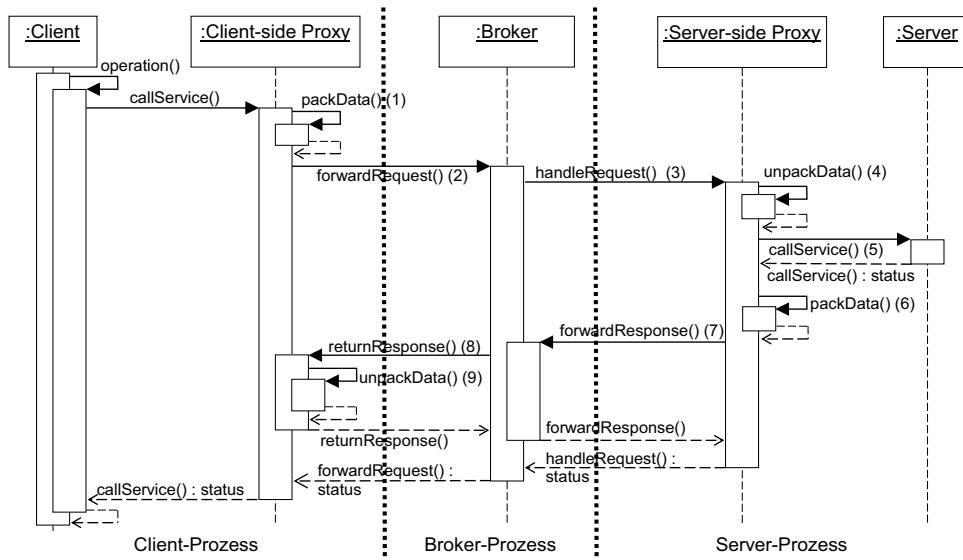


Bild 11-2 Sequenzdiagramm Broker-Muster auf einem Rechner

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()` (1)). Diese Nachricht wird dann zum Broker gesendet

¹²⁰ Diese Linien gehören nicht zum UML-Standard, sondern sollen nur die Grenzen der Betriebssystemprozesse für den Leser verdeutlichen.

(forwardRequest() (2)). Nachdem der Broker den passenden Server gefunden hat¹²¹, sendet er die Nachricht (handleRequest() (3)) an den serverseitigen Stellvertreter des Clients, den Server-side Proxy. Der Server-side Proxy nimmt die Nachricht entgegen, entpackt sie (unpackData() (4)) und ruft den gewünschten Service des Servers auf (callService() (5)). 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() (6)) und sendet die Nachricht weiter an den Broker (forwardResponse() (7)). Der Broker übergibt dann die Antwortnachricht an den clientseitigen Stellvertreter des Servers, den Client-side Proxy (returnResponse() (8)). Der Client-side Proxy entpackt die Daten (unpackData() (9)) und übergibt dem Client das Ergebnis.

Verteilung der Komponenten auf mehrere Rechner

Wie bereits erwähnt wurde, zeigt Bild 11-2 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 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 Dienstaufruf. Das Ergebnis geht wiederum auf dem gleichen Weg zurück.

Das folgende Sequenzdiagramm zeigt diesen Ansatz mit zwei Rechnern:

¹²¹ Die Aktivitäten des Brokers zur Bestimmung des passenden Servers werden der Übersicht halber nicht in Bild 11-2 gezeigt.

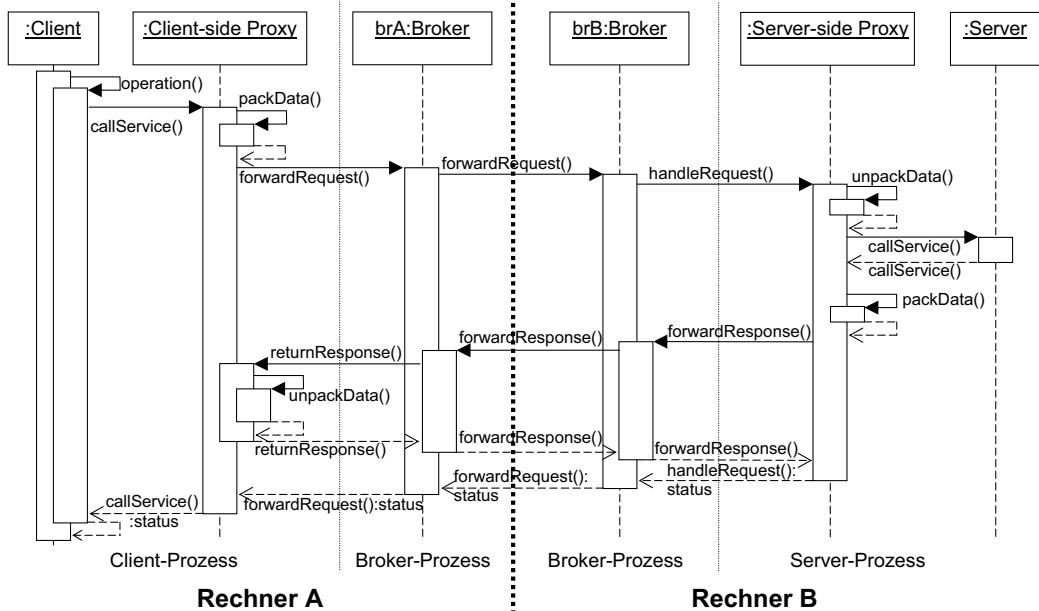


Bild 11-3 Dynamischer Ablauf mit mehreren Brokern

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.

11.1.3.4 Programmbeispiel

Wie bereits in Kapitel 11.1.3.1 erwähnt wurde, müssen in einer Implementierung des Musters einige Punkte geklärt werden, welche das Muster offen lässt. Diese Voraussetzungen werden nun zuerst beschrieben, bevor im Anschluss das eigentliche Programmbeispiel vorgestellt wird.

Die Implementierung des Broker-Musters erfolgt bei diesem Beispiel in Form einer Simulation in einem einzigen Betriebssystem-Prozess. In dieser Simulation werden somit keine Interprozesskommunikations-Mechanismen benötigt. Die Verteilung der Klassen auf mehrere Rechner spielt beim Nachrichtenaustausch in diesem Beispiel ebenfalls keine Rolle. Die beteiligten Klassen können folglich mittels „normaler“ Methodenaufrufe miteinander kommunizieren.

Als Format für Nachrichten wird ein UTF-8 codiertes Byte-Array verwendet. Ein Client-side Proxy wandelt die Parameter für den Dienstauftrag in ein solches Byte-Array, das der entsprechende Server-side Proxy in die für den Methodenaufruf notwendige Daten zurückwandelt. Der Inhalt dieses Byte-Arrays muss also zwischen diesen beiden Kommunikationspartnern abgesprochen sein. Ähnliches gilt für die Antwort, welche der

Server-side Proxy in ein Byte-Array packt und vom Client-side Proxy entpackt werden muss. Der Broker ist nur der Transporteur dieses Byte-Arrays und kümmert sich nicht um den Inhalt.

Zur Identifikation der Objekte werden statisch festgelegte Namen (Java Strings) verwendet. Das ist auf Serverseite der Name der aufzurufenden Dienstmethode und auf der Clientseite der Klassenname des Client-side Proxys.

Nach diesen Vorüberlegungen wird nun der Quellcode des Programmbeispiels vorgestellt und erläutert.

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()`.

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 {

    String DIENST = "erfrageWetter";

    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 {

    String DIENST = "erfrageTemperatur";

    Double 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 WetterDienst vorgestellt:

```
// Datei: WetterDienst.java

public class WetterDienst implements IWetterDienst {

    public WetterDienst() {
        System.out.println(getClass().getSimpleName() + ": instanziert");
    }

    @Override
    public String erfrageWetter(Tag tag) {
        switch (tag) {
            case HEUTE:
                return "Sonnenschein";
            case MORGEN:
                return "Regen";
            default:
                return "Kein Wetter bekannt";
        }
    }
}
```

Die Klasse TemperaturDienst ist die zweite Server-Klasse in diesem Beispiel:

```
// Datei: TemperaturDienst.java

public class TemperaturDienst implements ITemperaturDienst {

    public TemperaturDienst() {
        System.out.println(getClass().getSimpleName() + ": instanziert");
    }

    @Override
    public Double erfrageTemperatur(Tag tag) {
        switch (tag) {
            case HEUTE:
                return 25.3d;
            case MORGEN:
                return 10.7d;
            default:
                return Double.NaN;
        }
    }
}
```

Damit der Broker die Proxys auf der Serverseite ansprechen kann, wird das Interface IServersideProxy definiert, das von den Server-side Proxys implementiert werden

muss. Es enthält die Methoden `bearbeiteAnfrage()` und `getDienstName()`, die vom Broker aufgerufen werden:

```
// Datei: IServersideProxy.java

public interface IServersideProxy {

    // Bearbeitet die Anfrage eines Clients und sendet die Antwort
    // an den Broker, bevor die Methode zurückkehrt
    void bearbeiteAnfrage(String clientName, byte[] anfrage);

    // Welchen Dienst dieser Server-side Proxy anbietet
    String getDienstName();

}
```

Ein Objekt der Klasse `ServersideWetterDienstProxy` ist der Server-side Proxy auf der Seite des Wetterdienstes. 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 `WetterDienst` aussieht, als würde sie vom Client direkt aufgerufen werden. Ebenso wandelt der Server-side Proxy die zurückgegebene Antwort in eine Nachricht und schickt diese über den Broker an den zugehörigen Proxy auf der Seite des Clients. 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 `ServersideWetterDienstProxy`:

```
// Datei: ServersideWetterDienstProxy.java

import java.nio.charset.StandardCharsets;

public class ServersideWetterDienstProxy
    implements IServersideProxy {

    // Ein Server-side Proxy kennt den Broker
    private final Broker broker;
    // Der "echte" WetterDienst
    private final IWetterDienst wetterDienst;

    // Erzeugt einen WetterDienst und meldet sich beim Broker an.
    public ServersideWetterDienstProxy(Broker broker) {
        System.out.println(getClass().getSimpleName() + ": instanziiert");
        this.broker = broker;
        this.wetterDienst = new WetterDienst();
        this.broker anmelden(this);
    }

    @Override
    public void bearbeiteAnfrage(String clientName, byte[] anfrage) {
        // Parameter auspacken
        Tag tag = auspacken(anfrage);
        // Dienstmethode aufrufen
        String ergebnis = wetterDienst.erfrageWetter(tag);
        // Ergebnis verpacken ...
        byte[] antwort = verpacken(ergebnis);
    }
}
```

```

// ...und als Antwort an den Broker weiterleiten
broker.antwortWeiterleiten(clientName, antwort);
}

@Override
public String getDienstName() {
    return IWetterDienst.DIENST;
}

// Hilfsmethode zum Verpacken der Wettervorhersage
// in ein Byte-Array zum Transport
private byte[] verpacken(String wetter) {
    return wetter.getBytes(StandardCharsets.UTF_8);
}

// Hilfsmethode zum Auspacken einer Byte-Nachricht
// und umwandeln in ein Tag-Objekt zum Aufruf des Dienstes
private Tag auspacken(byte [] tag) {
    return Tag.valueOf(new String(tag, StandardCharsets.UTF_8));
}
}

```

Die Klasse ServersideTemperaturDienstProxy hat die gleiche Rolle wie die Klasse ServersideWetterDienstProxy – sie ist allerdings für den Temperaturdienst-Server zuständig:

```

// Datei: ServersideTemperaturDienstProxy.java

import java.nio.charset.StandardCharsets;

public class ServersideTemperaturDienstProxy
    implements IServersideProxy {

    // Ein Server-side Proxy kennt den Broker;
    private final Broker broker;
    // Der "echte" TemperaturDienst
    private final ITemperaturDienst temperaturDienst;

    // Erzeugt einen TemperaturDienst und meldet sich beim Broker an.
    public ServersideTemperaturDienstProxy(Broker broker) {
        System.out.println(getClass().getSimpleName() + ": instanziert");
        this.broker = broker;
        this.temperaturDienst = new TemperaturDienst();
        this.broker.anmelden(this);
    }

    @Override
    public void bearbeiteAnfrage(String clientName, byte[] anfrage) {
        // Parameter auspacken
        Tag tag = auspacken(anfrage);
        // Dienstmethode aufrufen
        Double ergebnis = temperaturDienst.erfrageTemperatur(tag);
        // Ergebnis verpacken ...
        byte[] antwort = verpacken(ergebnis);
        // ...und als Antwort an den Broker weiterleiten
        broker.antwortWeiterleiten(clientName, antwort);
    }
}

```

```

@Override
public String getDienstName() {
    return ITemperaturDienst.DIENST;
}

// Hilfsmethode zum Verpacken der Temperatur
// in ein Byte-Array zum Transport
private byte[] verpacken(Double temperatur) {
    return temperatur.toString().getBytes(StandardCharsets.UTF_8);
}

// Hilfsmethode zum Auspacken einer Byte-Nachricht
// und umwandeln in ein Tag-Objekt zum Aufruf des Dienstes
private Tag auspacken(byte [] tag) {
    return Tag.valueOf(new String(tag, StandardCharsets.UTF_8));
}
}

```

Die Klasse `Broker` ist zuständig für die Implementierung der Funktionalitäten eines Brokers. Zur Registrierung der angebotenen Dienste und zur Speicherung der Dienste anfordernden Clients arbeitet der Broker mit zwei Objekten der Klasse `HashMap`:

- In der Hashmap `servers` werden vom Broker die **Dienste aller Server mit ihrem Namen registriert**. Da der Broker aber nicht mit einem Servern direkt kommunizieren kann, wird in der Hashmap `servers` die Referenz auf den Server-Side Proxy eines Servers gespeichert. Als Schlüssel dient der Dienstname des Servers.
- In der Hashmap `clients` 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. In der Hashmap `clients` legt der Broker eine Referenz auf den Client-side Proxy ab. Hierbei wird als Schlüssel der Klassennamen des Client-side Proxys benutzt.

Über die Methode `anmelden()` wird die Referenz auf einen Server-side Proxy unter dem angegebenen Dienstnamen in die Hashmap `servers` eingetragen. Ein Eintrag kann unter Angabe des Namens mit Hilfe der Methode `abmelden()` aus dieser Hashmap wieder ausgetragen werden.

Ruft ein Client-side Proxy die Methode `anfrageWeiterleiten()` auf, übergibt er eine Referenz auf sich selbst, die Anfragenachricht in Form eines Byte-Array und den Namen des Servers, der die Anfrage bearbeiten soll. Mittels der Referenz kann der Broker den Klassennamen des Proxys ermitteln und zusammen mit der Referenz in der Hashmap `clients` speichern. Danach ruft der Broker die private Methode `findeServer()` auf, welche die Aufgabe hat, den passenden Server zu dem angefragten Dienst zu finden. Die Anfragenachricht wird dann vom Broker an den Server-side Proxy des gefundenen Servers geschickt.

Auf ähnliche Weise leitet der Broker die Antwortnachricht eines Servers beim Aufruf der Methode `antwortWeiterleiten()` durch den Server-side Proxy wieder an den Client zurück. Dabei wird die Hashmap `clients` 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.*;

public class Broker {

    private final Map<String, IServersideProxy> servers;
    private final Map<String, IClientsideProxy> clients;

    public Broker() {
        System.out.println(
            getClass().getSimpleName() + ": instanziert");
        servers = new HashMap<>();
        clients = new HashMap<>();
    }

    // Server-side Proxy meldet sich an:
    public void anmelden(IServersideProxy proxy) {
        servers.put(proxy.getDienstName(), proxy);
        System.out.printf("Broker: Dienst '%s' angemeldet.%n",
            proxy.getDienstName());
    }

    // Server-side Proxy meldet sich ab:
    public void abmelden(String dienstName) {
        servers.remove(dienstName);
        System.out.printf("Broker: Dienst '%s' abgemeldet.%n",
            dienstName);
    }

    // Die Anfrage eines Client wird an einen Server-side Proxy
    // weitergeleitet, der den entsprechenden Dienst anbietet:
    public void anfrageWeiterleiten(IClientsideProxy client,
        byte[] anfrage, String dienstName) {
        String clientName = client.getClass().getSimpleName();
        clients.put(clientName, client);
        IServersideProxy server = findeServer(dienstName);
        server.bearbeiteAnfrage(clientName, anfrage);
    }

    // Die Antwort kommt vom Server-side Proxy und wird
    // an den wartenden Client-side Proxy weitergeleitet:
    public void antwortWeiterleiten(String clientName, byte[] antwort) {
        IClientsideProxy client = findeUndEntferneClient(clientName);
        client.bearbeiteAntwort(antwort);
    }

    // Server-side Proxy anhand des Dienstnamens suchen:
    private IServersideProxy findeServer(String dienstName) {
        return servers.get(dienstName);
    }

    // Client-side Proxy anhand des Clientnamens suchen.
```

```
// Da der Client danach nicht mehr auf eine Antwort wartet,
// wird er aus der Map entfernt:
private IClientsideProxy findeUndEntferneClient(String clientName) {
    return clients.remove(clientName);
}
```

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 {

    void bearbeiteAntwort(byte[] antwort);

}
```

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. Damit der Broker die Antwortnachrichten wieder zurückliefern kann, muss die Klasse `ClientsideWetterDienstProxy` des Weiteren die Schnittstelle `IClientsideProxy` implementieren:

```
// Datei: ClientsideWetterDienstProxy.java

import java.nio.charset.StandardCharsets;

public class ClientsideWetterDienstProxy
    implements IClientsideProxy, IWetterDienst {

    private final Broker broker;
    private String responseWetter;

    public ClientsideWetterDienstProxy(Broker broker) {
        System.out.println(
            getClass().getSimpleName() + ": instanziert");
        this.broker = broker;
    }

    @Override
    public String erfrageWetter(Tag tag) {
        // Erstellt Anfragenachricht als Byte-Array
        // und leitet sie an den Broker weiter:
        byte[] anfrage = verpacken(tag);
        broker.anfrageWeiterleiten(this, anfrage, DIENST);
        // Der Broker ruft dabei die Methode bearbeiteAntwort() auf.
        // Daher kann jetzt das Ergebnis zurückgeliefert werden:
        return responseWetter;
    }

    // Der Broker ruft diese Methode auf und
    // übergibt die verpackte Antwort des Servers:
    @Override
```

```

public void bearbeiteAntwort(byte[] antwort) {
    responseWetter = auspacken(antwort);
}

// Hilfsmethode zum Verpacken der Anfrage
// in ein Byte-Array zum Transport:
private byte[] verpacken(Tag tag) {
    return tag.toString().getBytes(StandardCharsets.UTF_8);
}

// Hilfsmethode zum Auspacken einer Byte-Nachricht
// und umwandeln in einen String, der das Wetter beschreibt:
private String auspacken(byte [] wetter) {
    return new String(wetter, StandardCharsets.UTF_8);
}
}

```

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. Auch hier gilt: damit der Broker die Antwortnachrichten wieder zurückliefern kann, muss die Klasse ClientsideTemperaturDienstProxy die Schnittstelle IClientsideProxy implementieren. Hier nun die Klasse ClientsideTemperaturDienstProxy:

```

// Datei: ClientsideTemperaturDienstProxy.java

import java.nio.charset.StandardCharsets;

public class ClientsideTemperaturDienstProxy
    implements IClientsideProxy, ITemperaturDienst {

    private final Broker broker;
    private Double responseTemperatur;

    public ClientsideTemperaturDienstProxy(Broker broker) {
        System.out.println(getClass().getSimpleName() + ": instanziert");
        this.broker = broker;
    }

    @Override
    public Double erfrageTemperatur(Tag tag) {
        // Erstellt Anfragenachricht als Byte-Array
        // und leitet sie an den Broker weiter:
        byte[] anfrage = verpacken(tag);
        broker.anfrageWeiterleiten(this, anfrage, DIENST);
        // Der Broker ruft dabei die Methode bearbeiteAntwort() auf.
        // Daher kann jetzt das Ergebnis zurückgeliefert werden:
        return responseTemperatur;
    }

    // Der Broker ruft diese Methode auf und
    // übergibt die verpackte Antwort des Servers:
    @Override
    public void bearbeiteAntwort(byte[] antwort) {
        responseTemperatur = auspacken(antwort);
    }
}

```

```

// Hilfsmethode zum Verpacken der Anfrage
// in ein Byte-Array zum Transport:
private byte[] verpacken(Tag tag) {
    return tag.toString().getBytes(StandardCharsets.UTF_8);
}

// Hilfsmethode zum Auspacken einer Byte-Nachricht
// und umwandeln in den Temperaturwert:
private Double auspacken(byte [] temperatur) {
    return Double.valueOf(
        new String(temperatur, StandardCharsets.UTF_8));
}
}

```

Die Klasse `Client` enthält die eigentliche Anwendungslogik des Programmbeispiels in der Methode `druckeWetterdaten()`. Damit die Klasse `Client` mit lokalen wie mit entfernten Witterservern 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 {

    // Es kann sich um echte Server-Objekte
    // oder Client-side Proxys handeln
    private IWetterDienst wetterDienst;
    private ITemperaturDienst temperaturDienst;

    // Hier werden dem Client die Server-Objekte bekannt gemacht
    public void injectServices(IWetterDienst wetterDienst,
                               ITemperaturDienst temperaturDienst) {
        this.wetterDienst = wetterDienst;
        this.temperaturDienst = temperaturDienst;
    }

    // Client-Anwendung zum Abfragen der Wetter-
    // und Temperatur-Daten
    public void druckeWetterdaten() {
        // Daten für alle verfügbaren Tage abfragen
        for (Tag tag : Tag.values()) {
            System.out.printf("%nClient: Wetter und Temperatur"
                + " für %s anfragen:%n", tag);
            String wetter = wetterDienst.erfrageWetter(tag);
            Double temperatur = temperaturDienst.erfrageTemperatur(tag);
            System.out.printf("Wetter für %s: %s bei %.1f Grad.%n",
                tag, wetter, temperatur);
        }
    }
}

```

Die Klasse `TestBroker` enthält das Hauptprogramm des Beispiels. Im Hauptprogramm wird ein Objekt der Klasse `Broker` instanziert 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

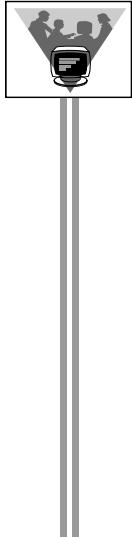
public class TestBroker {

    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 clientWetter =
            new ClientsideWetterDienstProxy(broker);
        ClientsideTemperaturDienstProxy clientTemperatur =
            new ClientsideTemperaturDienstProxy(broker);

        // Client-Objekt erstellen und Referenzen
        // auf die Client-side Proxys übergeben:
        Client anwendung = new Client();
        anwendung.injectServices(clientWetter, clientTemperatur);

        // Jetzt kann die Anwendung gestartet werden:
        anwendung.druckeWetterdaten();
    }
}
```



Hier das Protokoll des Programmlaufs:

```
TestBroker: Broker und Dienste erzeugen
Broker: instanziert
ServersideWetterDienstProxy: instanziert
WetterDienst: instanziert
Broker: Dienst 'erfrageWetter' angemeldet.
ServersideTemperaturDienstProxy: instanziert
TemperaturDienst: instanziert
Broker: Dienst 'erfrageTemperatur' angemeldet.
```

```
TestBroker: Client-side Proxys erzeugen
ClientsideWetterDienstProxy: instanziert
ClientsideTemperaturDienstProxy: instanziert
```

```
Client: Wetter und Temperatur für HEUTE anfragen:
Wetter für HEUTE: Sonnenschein bei 25,3 Grad.
```

```
Client: Wetter und Temperatur für MORGEN anfragen:
Wetter für MORGEN: Regen bei 10,7 Grad.
```

11.1.4 Bewertung

11.1.4.1 Vorteile

Die folgenden Vorteile werden gesehen:

- **Separation of Concerns**

Das Broker-Muster trennt die Kommunikation zwischen Client und Server von der Funktionalität eines Clients bzw. Servers.

- **Stabilität bei unveränderter Schnittstelle**

Solange sich die Schnittstelle für die Dienstaufrufe 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.

- **Ortstransparenz**

Ein Client muss den physischen Ort der Diensterbringung 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.

- **Plattformunabhängigkeit**

Client und Server sind plattformunabhängig. Sie können auch in unterschiedlichen Programmiersprachen erstellt werden.

- **Verteilung möglich**

Durch das Broker-Muster werden sehr große Softwaresysteme auf mehreren verteilten Rechnern möglich.

11.1.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **Fehlertoleranz erforderlich**

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.

- **Verringerung der Performance**

Der Kommunikationsablauf wird durch indirekte Aufrufe über den Broker aufwendiger. Die Performance ist daher schlechter als bei einem Direktzugriff.

- **potenzielles Bottleneck**

Ein Broker kann zu einem Engpass werden, was geeignete Maßnahmen bezüglich des Durchsatzes erfordert.

- **Abhängigkeit der Proxys vom Broker**

Die Proxys sind abhängig vom Broker. Wird der Broker ausgetauscht, müssen sie angepasst werden.

11.1.5 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]¹²² 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 11-3) ist ein ORB selbst eine **verteilte Anwendung**. Das folgende Bild zeigt die Struktur von CORBA:

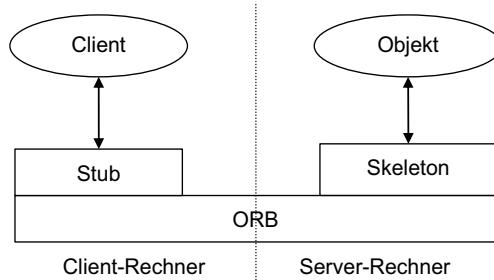


Bild 11-4 Struktur von CORBA

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 Methodenauftrag um. Liefert der Methodenauftrag 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 Methodenauftrag 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 Methodenauftrags an den Client zurückgegeben. Somit entspricht der Stub dem **Client-side Proxy**. Das folgende Bild stellt die Erzeugung von Stub und Skeleton dar:

¹²² Ein Spezifikationsdokument ist in der Regel nicht geeignet als Einstieg in ein Thema. Eine sehr gute Übersicht über CORBA bietet [OMGCOB].

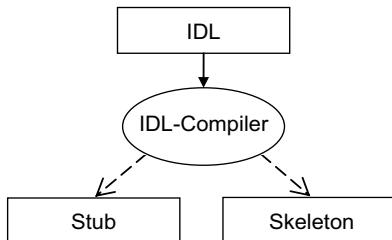


Bild 11-5 Erzeugung von Stub und Skeleton

11.1.6 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 Muster **Vermittler** 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) eingezogen. 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 eingezo- gen. Im Gegensatz zum Broker wird bei einer SOA die Abbildung von Anwendungsfällen aus Verarbeitungssicht auf Komponenten berücksichtigt.

11.2 Das Architekturmuster Service-Oriented Architecture

Das Architekturmuster Service-Oriented Architecture bildet die geschäftsprozessorien- tierte Sicht der Verarbeitungsfunktionen eines Unternehmens direkt auf die Architektur eines **verteilten Systems** ab.

Dienste (Anwendungsservices) in Form von **Komponenten entspre- chen Teilen eines Geschäftsprozesses**.



Durch den Einsatz von Diensten entstehen die Rollen des **Serviceanbieters** und **Servicesnutzers**.

11.2.1 Name/Alternative Namen

Serviceorientierte Architektur (engl. Service-Oriented Architecture, abgekürzt SOA), dienstorientierte Architektur.

11.2.2 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 Geschäfts- prozess auch nur einen einzigen Anwendungsfall enthalten, in der Regel sind es aber mehrere.

Ein Anwendungsfall ist ein Geschäftsprozess oder Teil eines Ge- schäftsprozesses, 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 beein- flussen. Als Folge davon sind viele Systeme sehr schlecht wartbar. Da viele Firmen ge-

zwungen sind, ihre Geschäftsprozesse laufend an die sich rasch ändernden Marktbedingungen anzupassen, führen die erzwungenen Änderungen zu einer Instabilität der Programme und zu einer Kostenexplosion.

11.2.3 Lösung

Mit einer **serviceorientierten Architektur** soll die geschäftsprozess-orientierte Sicht eines Unternehmens direkt auf die Architektur eines DV-Systems abgebildet werden



Dies bedeutet, dass die Anwendungsfälle aus Sicht der Verarbeitung den Komponenten des Entwurfs zugeordnet werden. Dabei sollen die einzelnen Geschäftsprozesse so weit wie möglich voneinander isoliert sein, damit sie unabhängig voneinander ablaufen können. Ziel ist es, dadurch eine Anwendung änderungsfreundlich zu machen. Voraussetzung einer serviceorientierten Architektur ist, dass die Geschäftsprozesse vollständig und klar verständlich sind.

Eine serviceorientierte Architektur kapselt die Verarbeitungsfunktionen einer Software, die Geschäftsprozessen oder Teilen von Geschäftsprozessen entsprechen, als Dienste in Komponenten bzw. Systemteilen.



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 in diesem Kapitel 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. Ziel eines Anwendungsfalls eines Systems ist 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 oder einem Geschäftsprozess aus Sicht der Verarbeitung entsprechen sollen.



Die technischen Funktionen, die beim Entwurf eines Anwendungsfalls zur Verarbeitung 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 und können in der Praxis nicht immer vollständig umgesetzt werden:

- **Verteilung** (engl. **distribution**)

Ein Dienst steht in einem Netzwerk zur Verfügung.

- **Geschlossenheit** (engl. **component appearance**)

Jeder Dienst stellt eine abgeschlossene Einheit dar, die weitestgehend 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" über frühere Aufrufe.

- **Lose Kopplung** (engl. **loosely coupled**)

Die Services sind untereinander möglichst entkoppelt. Dies bedeutet, dass ein beliebiger Service bei Bedarf dynamisch zur Laufzeit vom Servicenutzer gesucht und aufgerufen werden kann. Ein Service kann aus anderen Services zusammengesetzt sein.

- **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 veröffentlichte 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 Verzeichnis¹²³.

Ein Geschäftsprozess wird auf einen oder mehrere Anwendungsfälle abgebildet. Die Verarbeitungsfunktionen der Anwendungsfälle werden als Komponenten realisiert. Ein Anwendungsfall kann aus Teilkomponenten bestehen. Jeder Anwendungsfall kann einem Service oder mehreren elementaren Services zugeordnet werden. Hierbei können elementare Services und Services auch für mehrere unterschiedliche 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 11-6 gezeigt:

¹²³ Dies ist optional.

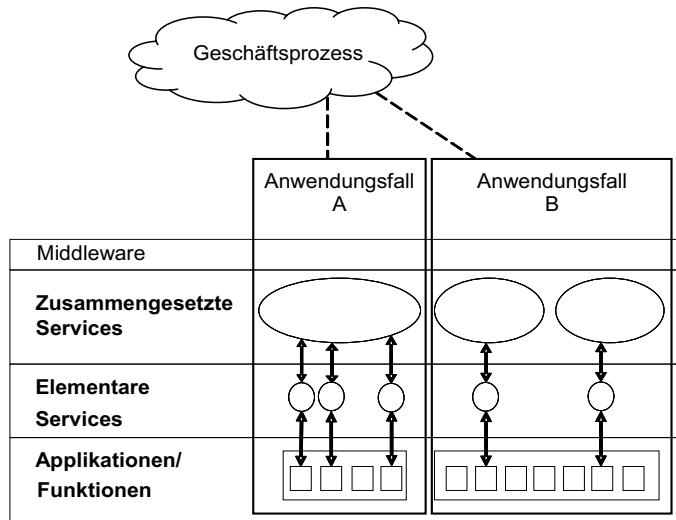


Bild 11-6 Schichtendiagramm für Services

Der sogenannte **Serviceanbieter** (engl. **service provider**) erbringt eine Dienstleistung, die von einem **Servicenutzer**¹²⁴ (engl. **service consumer**) innerhalb oder außerhalb des anbietenden Unternehmens genutzt werden kann. Die beiden Rollen 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 visualisiert die Zusammenarbeit zwischen den erwähnten Rollen:

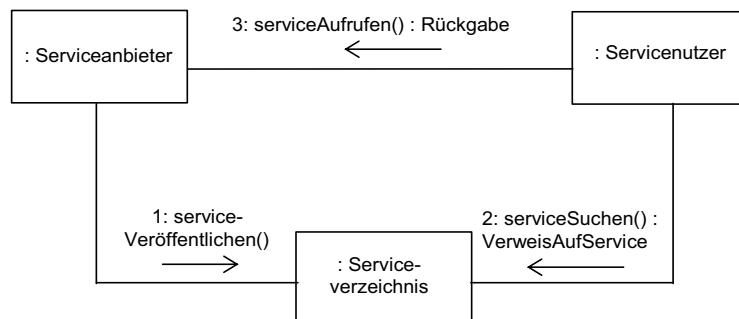


Bild 11-7 Zusammenarbeit in einer SOA als UML-Kommunikationsdiagramm

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, welcher den Service anbietet. Diese

¹²⁴ Statt Servicenutzer ist auch der Begriff Servicekonsument gebräuchlich.

Adresse wird danach vom Servicenutzer genutzt, um einen direkten Serviceaufruf beim Serviceanbieter durchzuführen (3). Der Rückgabewert des Service wird anschließend zurück zum Servicenutzer übertragen.

11.2.3.1 Klassendiagramm

Die im Folgenden gezeigte Architektur ist nicht als Muster, sondern als ein Beispiel zu sehen. Das eigentliche Muster ist die Abbildung der Geschäftsprozesse auf Komponenten der Verarbeitungsfunktionalität der Anwendungsfälle (siehe Bild 11-6) und die Existenz der Rollen Serviceanbieter und Servicenutzer (siehe Bild 11-7).

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:

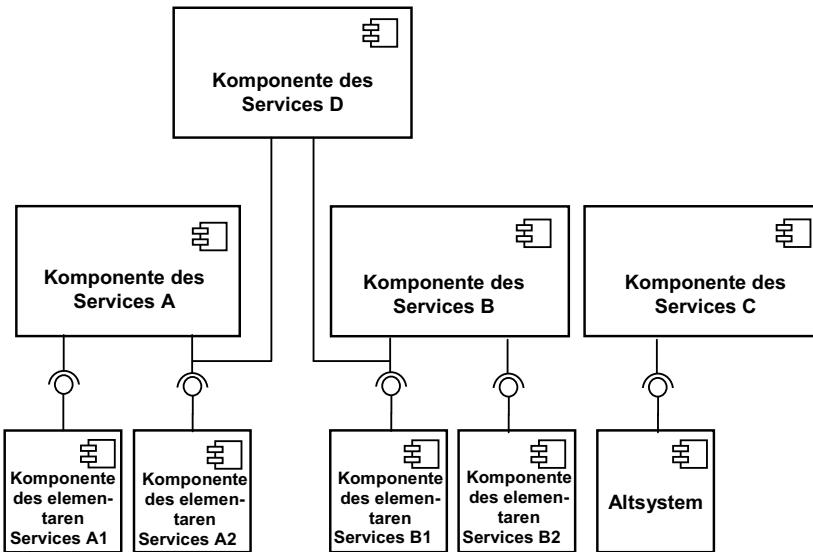


Bild 11-8 Komponentendiagramm SOA

11.2.3.2 Teilnehmer

Im Folgenden werden die Teilnehmer beschrieben:

- **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.

11.2.3.3 Dynamisches Verhalten

Das folgende Bild zeigt ein beispielhaftes Zusammenspiel zwischen Servicenuutzer und Servicekomponenten.

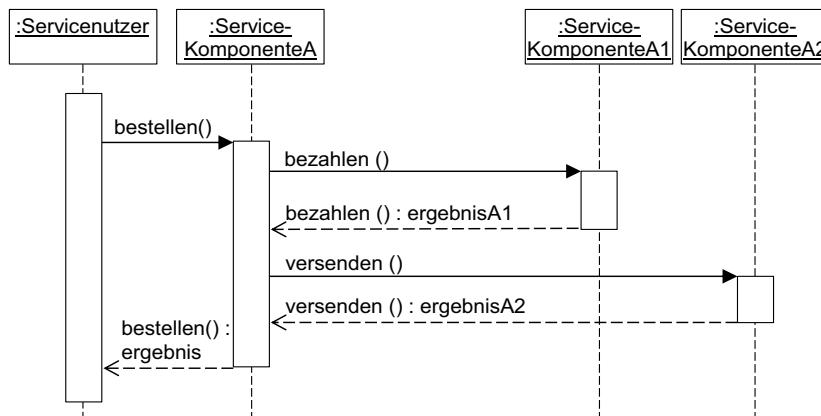


Bild 11-9 Beispielhaftes Sequenzdiagramm einer SOA

Das gezeigte Bild beschränkt sich auf die Servicekomponente A.

11.2.4 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.

11.2.4.1 Vorteile

Die folgende Liste zeigt die Vorteile einer SOA auf:

- **Abbildung von Geschäftsprozessen auf Komponenten des Systems**

Die Abbildung von Geschäftsprozessen auf Dienste in Komponenten schafft eine Übersicht über alle benötigten Dienste und deren Schnittstellen.

- **Komplexität von Systemen wird reduziert**

Die Komplexität von verteilten Systemen wird reduziert durch die Aufteilung in Komponenten von Services bzw. elementaren Services.

- **Wiederverwendung von Services**

Services und elementare Services können mehrfach eingesetzt, d. h. wiederverwendet werden.

- **Stabilität gegen Änderungen**

Solange sich die Schnittstelle eines Service nicht ändert, kann die dahinter liegende Implementierung dieses Service dynamisch ausgetauscht werden.

11.2.4.2 Nachteile

Folgende Nachteile werden gesehen:

- **komplexe Struktur**

Eine zu feine Granularität der Services erzeugt komplexe Strukturen.

- **Mehraufwand für Kommunikation**

Es entsteht ein Mehraufwand durch die Kommunikation über mehrere Schichten hinweg.

- **hoher Durchsatz der Netzwerkverbindungen erforderlich**

Die üblicherweise eingesetzten Protokolle stellen hohe Anforderungen an den Durchsatz der Netzwerkverbindungen.

- **hohe Bedeutung der Erfassung der Geschäftsprozesse**

Eine SOA kann nur dann effektiv eingesetzt werden, wenn die Geschäftsprozesse klar definiert und dokumentiert sind.

11.2.5 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.

11.2.6 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.

11.2.7 Realisierung einer SOA

Die Architektur-Vorstellung einer 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 **Broker **A**rchitecture) sowie**
- OSGi¹²⁵.

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.

11.2.7.1 Java-Standard 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. Bild 11-10 zeigt die für die einzelnen Schritte konkret eingesetzten Protokolle für XML-basierte Web Services.

Der Java-Standard **JAX-WS**¹²⁶ 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 Servicenuutzer erfolgt im Falle von XML-basierten Web Services über das SOA-Protokoll (**SOAP**¹²⁷). Ein Service kann durch die Registrierung in einem Verzeichnis veröffentlicht, anschließend dort gesucht und gefunden werden. Häufig wird hierfür Universal Discovery, Description, Integration

¹²⁵ 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.

¹²⁶ JAX-WS steht für Java API for XML Web Services.

¹²⁷ Die Abkürzung SOAP stand ursprünglich für Simple Object Access Protocol.

(**UDDI**) als Protokoll eingesetzt. Diese Technologien werden im Folgenden kurz vorstellt. Hier eine Visualisierung der typischen Protokollabläufe eines Web Service:

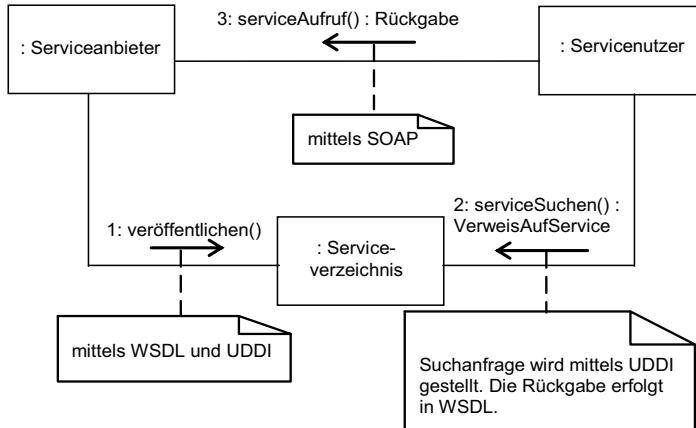


Bild 11-10 Protokollabläufe eines XML-basierten Web Service

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:

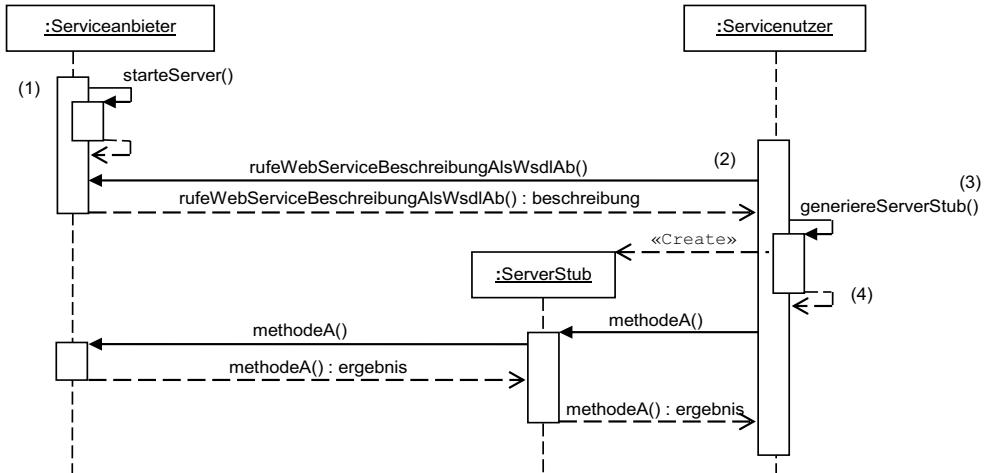


Bild 11-11 JAX-WS Sequenzdiagramm

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, welche 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 einzige Klasse instanziert werden) mit einem Serviceanbieter verbinden, ohne sich mit näheren Details zur Kommunikation beschäftigen zu müssen (4).¹²⁸

In JAX-WS¹²⁹ 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 zeigt eine Klasse als Web Service-Zugangspunkt, dessen WSDL-Beschreibung in Kapitel 11.2.7.3 aufgeführt ist:

```
// Datei Serviceanbieter.java

import jakarta.jws.WebMethod;
import jakarta.jws.WebService;

@WebService(targetNamespace = "serviceanbieter",
            serviceName= "Beispieldienst")
public class Serviceanbieter {

    @WebMethod
    public void methodeA(String para) {
        System.out.println("Server: Methode A wurde aufgerufen.");
        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 Servicenutzer an den Serviceanbieter und eine für die Antwort vom Serviceanbieter an den Servicenutzer.

¹²⁸ Instanziiert der Servicenutzer die ServerStub-Klassen, wird im Hintergrund eine Verbindung zum Serviceanbieter aufgebaut.

¹²⁹ Die in diesem Beispiel verwendeten JAX-WS-Bibliotheken (Jakarta) müssen ab Java 11 manuell zum Java-Class-Path hinzugefügt werden. Über den begleitenden Webauftritt finden sich die genutzten Bibliotheken im Ordner `jaxws/lib`.

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 Service 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 das folgende Beispiel zeigt:

```
// Datei: ServiceanbieterMain.java

import jakarta.xml.ws.Endpoint;

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();
        }
    }
}
```

Nach dem Starten des Serviceanbieters ist mittels eines Internet-Browsers 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 in JAX-WS das Kommandozeilenprogramm `wsimport` bereit¹³⁰. Nach dem Ausführen des Kommandos `wsimport -s src -keep -d bin http://localhost:8080/Serviceanbieter?wsdl`¹³¹ wurde u. a. eine Klasse `ServiceanbieterService` erzeugt, die nach dem

¹³⁰ Im begleitenden Webauftritt finden Sie das Programm `wsimport` im Ordner `jaxws/bin`.

¹³¹ Im Projektverzeichnis ggf. mit vollständiger Pfadangabe zu `wsimport` ausführen. 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`.

Entwurfsmuster **Fabrikmethode** (siehe Kapitel 7.1) eine Instanz der Klasse Serviceanbieter zurückgibt. Hier nun das Programm, das den Web Service aufruft:

```
// Datei: Servicenutzer.java

import serviceanbieter.Serviceanbieter;
import serviceanbieter.Beispieldienst;

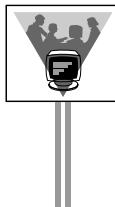
public class Servicenutzer {

    public static void main(String[] args) {
        Beispieldienst s = new Beispieldienst();
        Serviceanbieter anbieter;
        try {
            anbieter = s.getServiceanbieterPort();

            // MethodeA aufrufen
            anbieter.methodeA("Nachricht an Serviceanbieter");

            // MethodeB aufrufen und Ergebnis zeigen
            String result = anbieter.methodeB();
            System.out.println("Ergebnis: " + result);

        } 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.



Hier das Protokoll des Programmlaufs des Servicenutzers:

Ergebnis: Nachricht von Server an Client.

11.2.7.2 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 11-12 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 befinden 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:

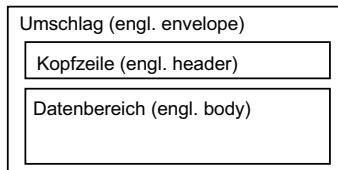


Bild 11-12 Aufbau einer SOAP-Nachricht

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 ... -->
  </soap:Header>
  <soap:Body>
    <soap:message> Inhalt des Datenbereichs </soap:message>
  </soap:Body>
</soap:Envelope>
  
```

11.2.7.3 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 Verfügung¹³²:

¹³² Hinweis: Die fettgedruckten Begriffe spiegeln die Bezeichner der entsprechenden Tags in der WSDL-Beschreibung wieder.

1. Documentation

Die Beschreibung der Funktionalität des Web Services in Prosatext.

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-Schema¹³³. 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 eines 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].

Beispielquelltext für eine WSDL-Beschreibung:

Es folgt ein Beispiel zu einer WSDL-Beschreibung. Dabei handelt es sich um einen unvollständigen Auszug der Beschreibung der in Kapitel 11.2.7.1 gezeigten Web Services. Hier nun das WSDL-Dokument:

```
<?xml version="1.0"?>
<definitions xmlns:tns="serviceanbieter" ...
  name="Beispieldienst" targetNamespace="serviceanbieter">
```

¹³³ Standard spezifiziert vom W3C [w3xmls].

```

<documentation> Beschreibung des Services in textueller Form.
</documentation>

<!-- Datentypen definieren -->
<types>
    <xss:complexType name="methodeA">
        <xss:sequence>
            <xss:element name="arg0" type="xss:string" minOccurs="0">
            </xss:element>
        </xss:sequence>
    </xss:complexType>
</types>
...
<!-- 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>

```

11.2.7.4 Universal Discovery, Description, Integration – UDDI

UDDI wurde vom Industriekonsortium OASIS¹³⁴ spezifiziert [oasudd]. Die federführenden Firmen hinter dem Standard sind IBM, Microsoft und SAP. Ziel von UDDI war es, **ein Verzeichnis zu schaffen**, das eine durchsuchbare Übersicht über eine (große) Menge von Web Services bietet. **Langfristig sollten mit Hilfe von UDDI die Dienste einer beliebigen Firma gefunden werden.** In der Praxis sind UDDI-Verzeichnisse jedoch eher selten anzutreffen. Ein UDDI-Verzeichnis kann mit der Funktionsweise von Telefonbüchern verglichen werden. Der Zugriff auf UDDI-Verzeichnisse kann sowohl

¹³⁴ OASIS steht für Organization for the Advancement of Structured Information Standards.

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.

- **Service Type Registration**

Bietet eine Tabelle der Servicebeschreibungen in Prosatext für die maschinelle Nutzung an.

Aus der **technischen Sicht** besteht ein UDDI-Verzeichnis im Wesentlichen aus zwei Komponenten:

- **UDDI-API,**

Die UDDI-API stellt verschiedene Web Service-Methoden bereit, die zum Suchen, Veröffentlichen und Verwalten eines Service im UDDI-Verzeichnis genutzt werden können. Die Kommunikation erfolgt über SOAP-Nachrichten.

- einem **standardisierten XML-Schema,**

welches das intern für UDDI genutzte Datenmodell beschreibt.

Als **Alternative zu UDDI** kann beispielsweise die **Web Services Inspection Language (WSIL)** [ibmwsil] 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 Entwicklungsteams zu unterstützen und die Wiederverwendung von bereits existierenden Web Services zu fördern.

11.2.7.5 Implementierung eines SOA-Beispiels mit Web Services in JAX-WS

In diesem Kapitel wird der Einsatz von JAX-WS¹³⁵ anhand der Implementierung eines SOA-Beispiels verdeutlicht. 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:

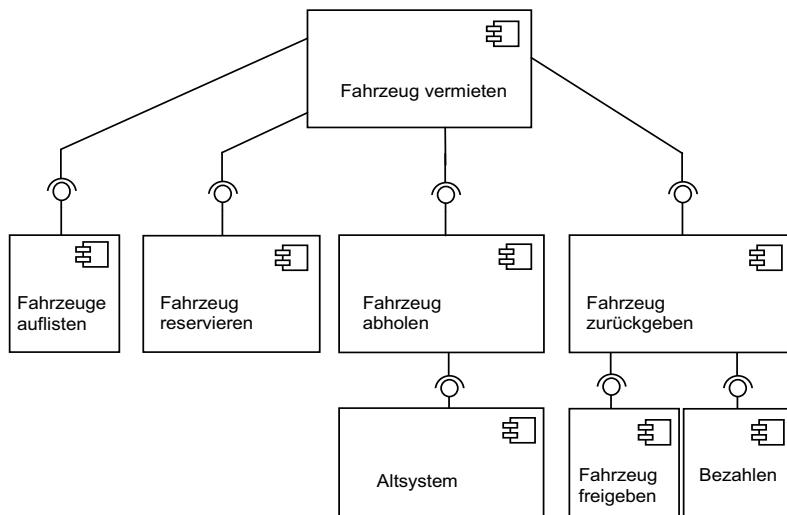


Bild 11-13 Komponentendiagramm der SOA-Beispielimplementierung

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 (`AutovermietungImpl.java`) sowie einem Hauptprogramm (`AutovermietungMain.java`), das den Server über das Netzwerk

¹³⁵ Die in diesem Beispiel verwendeten JAX-WS-Bibliotheken (Jakarta) müssen ab Java 11 manuell zum Java-Class-Path hinzugefügt werden. Über den begleitenden Webauftritt finden sich die genutzten Bibliotheken im Ordner `jaxws/lib`.

aufrufbar macht. Hier zunächst die Schnittstelle, welche die Dienste beschreibt, die zur Verfügung gestellt werden:¹³⁶

```
// Datei: IAutovermietung.java

package autovermietung.server;

import jakarta.jws.WebParam;
import jakarta.jws.WebResult;
import jakarta.jws.WebService;

@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);

    @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: AutovermietungImpl.java

package autovermietung.server;

import jakarta.jws.WebService;
import java.io.IOException;

@WebService(
    serviceName = "AutovermietungService",
    targetNamespace = "autovermietung.server",
    portName = "FahrzeugMieten",
    endpointInterface = "autovermietung.server.IAutovermietung")
public class AutovermietungImpl implements IAutovermietung {
```

¹³⁶ Die Annotationen @WebService, @WebParam und @WebResult werden in Kapitel 11.2.7.1 erklärt.

```
// Diese Methode stellt die 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 die 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
    try {
        Runtime.getRuntime().exec("legacySystem.exe ...");
        return true;
    } catch(IOException e) {
        return false;
    }
}

// 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: AutovermietungMain.java

package autovermietung.server;

import jakarta.xml.ws.Endpoint;

public class AutovermietungMain {

    // Hauptprogramm, das den Web Service im Netz ansprechbar macht
    public static void main(String[] args) {
        String url = "http://localhost:8080/AutovermietungServer";
        IAutovermietung server = new AutovermietungImpl();
        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`¹³⁷ oder direkt mit der Eclipse-Oberfläche¹³⁸ 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:

¹³⁷ `wsimport -s src -d bin -keep http://localhost:8080/AutovermietungServer?wsdl` im Projektverzeichnis des Servicenutzers ausführen. Im begleitenden Webauftritt finden Sie das Programm `wsimport` im Ordner `jaxws/bin`.

¹³⁸ Im Menü Neu -> Andere -> Web Service Client.

```
// Datei: Servicenutzer.java

package autovermietung.servicenutzer;

import server.autovermietung.AutovermietungService;
import server.autovermietung.IAutovermietung;

import jakarta.xml.ws.WebServiceRef;
import java.util.List;

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.");
    }

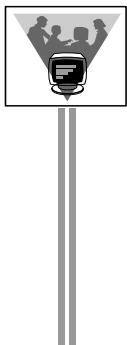
    //Servicenutzer mietet ein Fahrzeug
    public void fahrzeugMieten() {
        String reservierungsNr;
        String standort = "Stuttgart";
        String kreditkartenNr = "abc123456";

        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... ");
            reservierungsNr = service.fahrzeugReservieren(
                standort, "Kleinwagen");
            System.out.println("Reserviertes Fahrzeug hat "
                + "die Reservierungsnr. " + reservierungsNr + ".");
        }

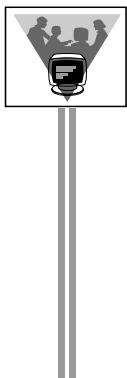
        // Fahrzeug abholen
        service.fahrzeugAbholen(reservierungsNr);
        System.out.println("Fahrzeug abgeholt.");
    }
}
```

```
// 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 ha
FZ_ID_4711.
Fahrzeug abgeholt.
Fahrzeug zurueckgegeben.
Servicenutzer beendet.

11.2.7.6 Java-Web-Service für REST

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. Im folgenden Beispiel wird die JAX-RS Referenzimplementierung Jersey¹³⁹ genutzt. Um das Beispiel ausführen zu können, müssen die entsprechenden Bibliotheken im Class-Path liegen¹⁴⁰.

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, welche 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 `Serviceanbieter` mit den JAX-RS Annotationen:

```
// Datei: Serviceanbieter.java

import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("Serviceanbieter")
public class Serviceanbieter {

    @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();
    }

    @GET
    @Path("methodeB")
}
```

¹³⁹ Die Webseite des Projekts ist: <http://jersey.java.net>.

¹⁴⁰ Über den begleitenden Webauftritt finden sich die genutzten JAX-RS Bibliotheken im Ordner `jersey-libs`.

```

@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

import com.sun.net.httpserver.HttpServer;
import org.glassfish.jersey.jdkhttp.JdkHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;

import java.net.URI;
import java.net.URISyntaxException;

public class ServiceanbieterMain {

    public static void main(String[] args) {
        HttpServer s;
        try {
            s = JdkHttpServerFactory.createHttpServer(
                new URI("http://localhost:8080/rest/"),
                new ResourceConfig(Serviceanbieter.class));
            System.out.println("Serviceanbieter ist nun aktiv "
                + "und wartet auf Anfragen...");
        } catch (IllegalArgumentException | URISyntaxException e) {
            e.printStackTrace();
        }
    }
}
}

```

Anschließend kann ein Servicenutzer implementiert werden, welcher die Methoden beim Serviceanbieter aufruft:

```

// Datei: Servicenutzer.java

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;

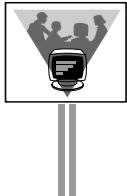
public class Servicenutzer {

    public static void main(String[] args) {
        String restUrl = "http://localhost:8080/rest/Serviceanbieter";
        Client client = ClientBuilder.newClient();
        WebTarget baseService = client.target(restUrl);

        // MethodeA aufrufen
        WebTarget serviceA = baseService.path("methodeA")
            .path("Nachricht an Serviceanbieter");
        serviceA.request(MediaType.TEXT_PLAIN).get(String.class);
    }
}
}

```

```
// MethodeB aufrufen und Ergebnis zeigen
WebTarget serviceB = baseService.path("methodeB");
String result = serviceB.request(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

11.2.7.7 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. 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 12.2.7.5 gezeigte Autovermietung herangezogen werden.

Die Klasse `Autovermietung` stellt die Umsetzung der Autovermietung mit REST dar. Im Beispiel wird die JAX-RS Referenzimplementierung Jersey genutzt. Um das Beispiel ausführen zu können, müssen die entsprechenden Bibliotheken¹⁴¹ im Class-Path liegen. Es werden die Methoden `fahrzeugeAuflisten()`, `fahrzeugReservieren()`, `fahrzeugZurueckgeben()` und `bezahlen()` mit den entsprechenden Übergabewerten implementiert. Hier der Quelltext der Klasse `Autovermietung`:

```
// Datei: Autovermietung.java

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.ArrayList;
import java.util.List;

@Path("Autovermietung")
public class Autovermietung {
```

¹⁴¹ Über den begleitenden Webauftritt finden sich die genutzten Bibliotheken im Ordner `jersey-libs`.

```
// Diese Methode stellt die Servicekomponente "Fahrzeuge auflisten"
// dar. Alle verfuegbaren 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 verfuegbar
    return list.toString();
}

// Diese Methode stellt die Servicekomponente "Fahrzeug reservieren"
// dar. Ein Fahrzeug wird an einem Standort reserviert.
@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;
}

// Diese Methode stellt die Servicekomponente "Fahrzeug abholen"
// dar.
@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.";
}

// Diese Methode stellt die Servicekomponente "Fahrzeug zurueck-
// geben" dar. 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

import com.sun.net.httpserver.HttpServer;
import org.glassfish.jersey.jdkhttp.JdkHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;
import java.net.URI;
import java.net.URISyntaxException;

public class AutovermietungMain {

    public static void main(String[] args) {
        HttpServer s;
        String url = "http://localhost:8080/rest";
        try {
            s = JdkHttpServerFactory.createHttpServer(
                new URI(url),
                new ResourceConfig(Autovermietung.class));
            System.out.println("Der Web Service ist nun im Netzwerk "
                + "unter der Adresse " + url + " aufrufbar.");
            System.out.println("Serviceanbieter lauscht jetzt...");
        } catch(InvalidArgumentException | URISyntaxException e) {
            e.printStackTrace();
        }
    }
}

```

Abschließend wird mit der Datei AutovermietungClient.java ein Nutzer für die Autovermietung implementiert:

```
// Datei: AutovermietungNutzer.java

import org.glassfish.jersey.client.ClientConfig;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;

public class AutovermietungNutzer {

    public static void main(String[] args) {
        String erg;
        WebTarget res;
        String restUrl = "http://localhost:8080/rest/Autovermietung";

        String reservierungsNr;
        String standort = "Stuttgart";
        String kreditkartenNr = "abc123456";
        System.out.println("Servicenutzer gestartet.");

        // Verbindung zum Serviceanbieter aufbauen
        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient(config);
        WebTarget baseService = client.target(restUrl);

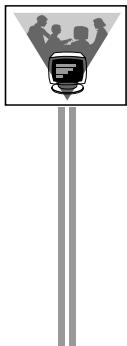
        // Alle verfuegbaren Fahrzeuge auflisten
        res = baseService.path("Fahrzeuge/" + standort);
        erg = res.request(MediaType.TEXT_PLAIN).get(String.class);
        System.out.println("Freie Fahrzeuge in " + standort + ":");
        System.out.println(erg);

        // Ein Fahrzeug reservieren
        System.out.println("Reserviere einen Kleinwagen...");
        res = baseService.path("Reservieren/" + standort
            + "/Kleinwagen");
        reservierungsNr =
            res.request(MediaType.TEXT_PLAIN).get(String.class);
        System.out.println("Reserviertes Fahrzeug hat "
            + "die Reservierungsnr. " + reservierungsNr + ".");

        // Fahrzeug abholen
        res = baseService.path("Abholen/" + reservierungsNr);
        res.request(MediaType.TEXT_PLAIN).get(String.class);
        System.out.println("Fahrzeug abgeholt.");

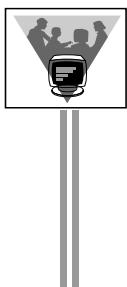
        // Fahrzeug nutzen
        // Nicht Teil des Beispielprogramms

        // Später Fahrzeug zurueckgeben
        res = baseService.path("Zurueckgeben/"
            + reservierungsNr + "/" + kreditkartenNr);
        res.request(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.

11.3 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 in Kapitel 11 behandelten Architekturmuster für verteilte Systeme sind im Folgenden kurz zusammengefasst:

- **Broker** (siehe Kapitel 11.1): Das Broker-Muster entkoppelt in verteilten Softwaresystemarchitekturen Client- und Server-Komponenten, indem zwischen diesen Komponenten ein verteilter Broker als Zwischenschicht eingeschoben wird, über welche Client- und Server-Komponenten untereinander 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, damit der Broker die von Servern angebotenen Dienste kennt. Clients können dann die angebotenen Dienste über den Broker nutzen.
- **Service-Oriented Architecture** (siehe Kapitel 11.2): 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.

11.4 Kontrollfragen

Aufgaben 11.4.1: Broker

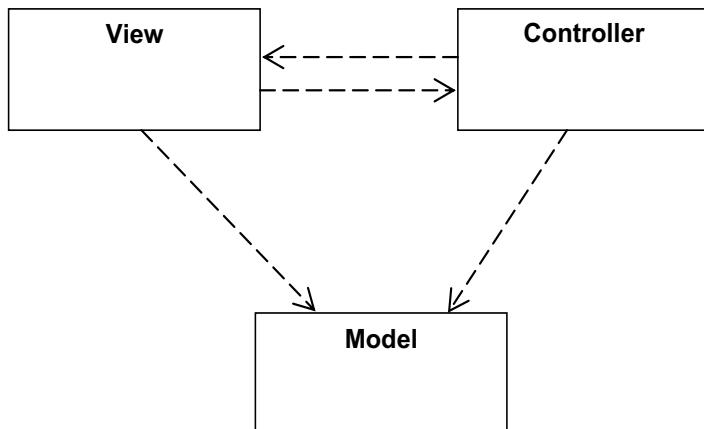
- 11.4.1.1 Erklären Sie die Grundzüge des Broker-Musters.
- 11.4.1.2 Welche Rollen spielen ein Client-side Proxy und ein Server-side Proxy im Rahmen des Broker-Musters?

Aufgaben 11.4.2: Service-Oriented Architecture

- 11.4.2.1 Erklären Sie die Grundzüge einer Service-Oriented Architecture.
- 11.4.2.2 Erläutern Sie die Rollen von Serviceanbieter, Servicenutzer und Serviceverzeichnis.

Kapitel 12

Architekturmuster für interaktive Systeme



12.1 Das Architekturmuster Model-View-Controller

12.2 Zusammenfassung

12.3 Kontrollfragen

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-42384-1_12.

12 Architekturmuster für interaktive Systeme

In diesem Kapitel wird das Architekturmuster Model-View-Controller für interaktive Systeme analysiert (siehe Kapitel 12.1).

12.1 Das Architekturmuster Model-View-Controller

Das Architekturmuster Model-View-Controller trennt ein **interaktives System** in die Komponenten:

- **Model** (Verarbeitung/Datenhaltung der Geschäftslogik),
- **View** (Ausgabe) und
- **Controller** (Eingabe).

Das Model hängt dabei nicht von der Ein- und Ausgabe ab.

12.1.1 Name/Alternative Namen

Model-View-Controller¹⁴², abgekürzt als MVC.

12.1.2 Problem

Die Benutzerschnittstelle eines Systems wird besonders häufig geändert.

Die Benutzerschnittstelle eines Systems soll in einfacher Weise – sogar zur Laufzeit – ausgetauscht werden können, ohne dass das restliche System 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.

12.1.3 Lösung

Eines der bekanntesten Architekturmuster ist das Architekturmuster "**Model-View-Controller**". 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.

¹⁴² 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.

Mit dem Architekturmuster MVC ist es möglich,

- die **Datenhaltung und Verarbeitung (Geschäftslogik)**,
- die **Präsentationslogik** bzw. **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 der zugehörige 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 Programmierung (**Trennung der Belange**, engl. **separation of concerns**).

Die Datenhaltung/Verarbeitung, die Präsentationslogik sowie die Interpretation der Benutzereingaben sollten in jeweils eigenen Komponenten getrennt programmiert und untereinander nicht vermischt werden.



Dies verbessert die Struktur des Quelltextes und somit die Wartbarkeit der Anwendung. Da Model, View und Controller getrennte Komponenten darstellen, wobei ein Benachrichtigungsmechanismus die Kommunikation beispielsweise 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, welche dabei die Daten verschieden darstellen.



12.1.3.1 Teilnehmer

Der Grundgedanke der MVC-Architektur ist die Zerlegung einer interaktiven Anwendung in die Komponenten **Model**, **View** und **Controller**.



Jeder View ist stets ein Controller zugeordnet. Ein Controller kann aber mehrere Views "bedienen".



Die Aufgaben dieser Komponenten sind im sogenannten **Active Model**¹⁴³:

- 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 und an ggf. angemeldete Controller gegeben.



Will ein angemeldeter Beobachter (View bzw. Controller) nicht länger über eine Änderung der Daten des Model informiert werden, weil zum Beispiel die View ausgeblendet wird, kann sich der entsprechende Beobachter bei seinem Model abmelden.

Das folgende Bild zeigt die Interaktionen zwischen den erwähnten drei Komponenten:

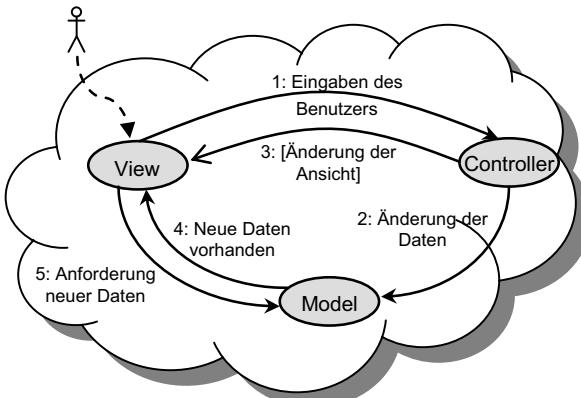


Bild 12-1 Interaktionen der Komponenten der MVC-Architektur

Die Interaktionen dieser Komponenten werden im Folgenden entsprechend der Nummerierung in Bild 12-1 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.

¹⁴³ Das Active Model selbst wird in diesem Kapitel noch detailliert vorgestellt und abgegrenzt. An dieser Stelle werden nur die Aufgaben der verschiedenen Komponenten im Rahmen des Active Model kurz erläutert.

- (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**, 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.



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.

Die Abhängigkeiten der Komponenten der MVC-Architektur sind im folgenden Bild symbolisiert:

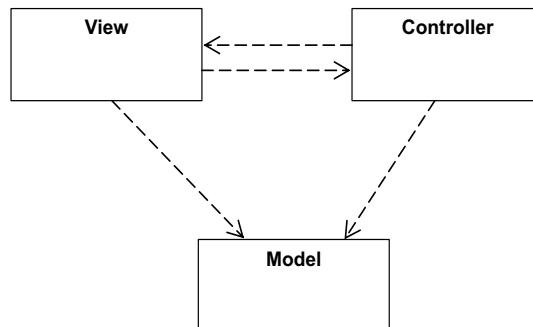


Bild 12-2 Komponenten der MVC-Architektur und ihre Abhängigkeiten

Im Folgenden erfolgt die Beschreibung der Komponenten:

- **Model**

Die Aufgabe des Model ist das Durchführen der Geschäftsprozesse (ohne Ein- und Ausgabe) sowie 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.

Beim Model muss prinzipiell nicht auf die Views bzw. die 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 jedoch 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.

Werden die Daten des Model von einem anderen Geschäftsprozess geändert, so muss der Controller über diese Änderung 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 Strategien, 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 aus Platzgründen nicht eingegangen werden¹⁴⁴.
- 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 6.3) 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.

¹⁴⁴ Mehr Informationen zum Passive Model sind beispielsweise in [Bur92] zu finden.

- 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) sowie Speicherung der Daten durchführen.
- Die An- und Abmeldung von Views entgegennehmen.
- Angemeldete Views über geänderte Daten informieren.
- Eine Aufrufsschnittstelle für das Abholen von Daten durch die Views bzw. Controller bereitstellen.
- In der View eingegebene Daten von Controllern entgegennehmen.



• 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:

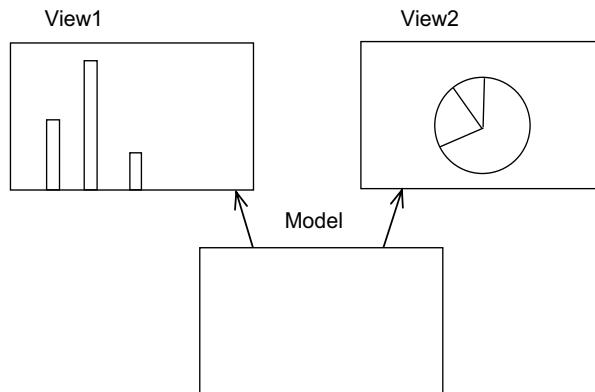


Bild 12-3 Aktualisierung zweier Views durch das Model

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 sogenannten **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.



• 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 eine 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.



12.1.3.2 Klassendiagramm

Ein Klassendiagramm für das MVC-Muster ist beispielhaft in Bild 12-8 in Kapitel 12.1.3.4 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. Die Zusammenarbeit dieser Entwurfsmuster im MVC-Muster wird in Kapitel 12.1.3.4 ausführlich diskutiert.

12.1.3.3 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** (siehe Kapitel 6.3) und ermöglicht es, einem Model mehrere Views zuzuordnen. Zudem können neue Views entwickelt werden, ohne das Model umschreiben zu müssen.

In Bild 12-4 soll das Zusammenspiel der Komponenten Controller, Model und View(s) von der Anmeldung der View(s) beim Model (`anmelden()`)¹⁴⁵ bis zur Aktualisierung der Daten in der View bzw. den Views betrachtet werden. Bei einem Vergleich mit Bild 6-6 (Sequenzdiagramm des Beobachter-Musters) fällt auf, dass der in Bild 12-4 dargestellte Ablauf zwischen Model und View exakt dem des Beobachter-Musters entspricht. Darauf wird in Kapitel 12.1.3.4 noch genauer eingegangen. Hier das bereits erwähnte Sequenzdiagramm:

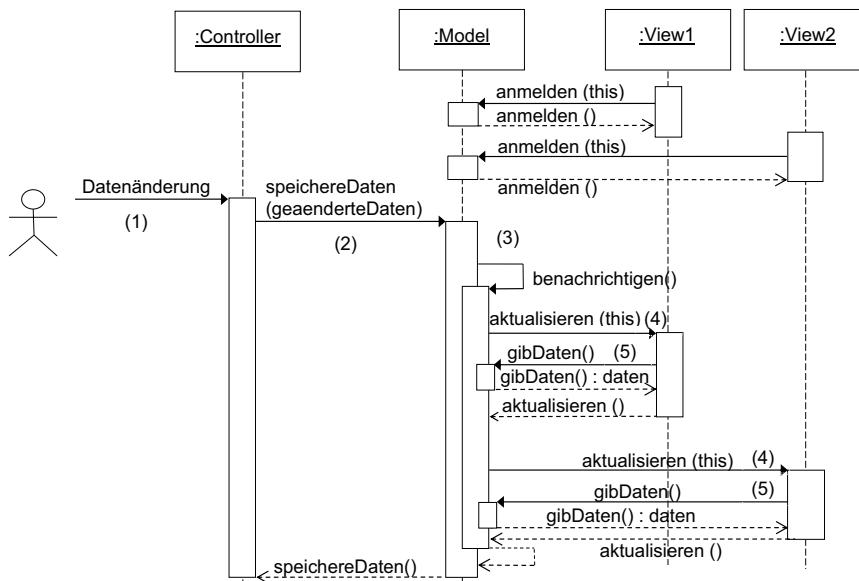


Bild 12-4 Sequenzdiagramm Aktualisierung des Model¹⁴⁶

¹⁴⁵ Es wird hier nicht der Fall betrachtet, dass ein Dritter die Views anmeldet.

¹⁴⁶ Die Nummerierung in Klammern bezieht sich auf die noch kommende Erklärung und ist nicht UML-konform.

Im ersten Schritt in Bild 12-4 melden sich zwei Views mit der Methode anmelden() bei der Model-Komponente an.

Der weitere Ablauf in Bild 12-4 veranschaulicht einen Aktualisierungsvorgang. Die Kommunikation zwischen den MVC-Komponenten ist dabei wie folgt:

- (1) Der Benutzer gibt Daten an der View ein. Die View leitet diese Daten an den Controller weiter. Dies ist hier vereinfacht dargestellt als Datenänderung für den Controller.
- (2) Der Controller übersetzt diese Daten in einen Methodenaufruf für das Model. Hierzu bestimmt der Controller, welche Methoden des Model aufgerufen werden müssen. Werden Parameter benötigt, so werden sie vom Controller ebenfalls übergeben. Dieser Vorgang wird in Bild 12-4 durch den Aufruf der Methode speichereDaten (geaenderteDaten) repräsentiert.
- (3) Das Model ändert innerhalb der Methode speichereDaten() seine Daten und ruft die Methode benachrichtigen() auf.
- (4) Das Model benachrichtigt durch Aufruf der Methode aktualisieren() innerhalb der Methode benachrichtigen() seine zugeordneten Views darüber, dass sich seine Daten geändert haben.
- (5) Die benachrichtigten Views holen die geänderten Daten durch den Aufruf der Methode gibDaten() beim Model ab.
- (6) Die neuen Grafiken werden auf dem Bildschirm ausgegeben. Diese Aktivität ist nicht mehr dargestellt.

12.1.3.4 Entwurfsmuster im MVC-Muster

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 12.1.3.3 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 Elementen der grafischen Oberfläche eingesetzt werden.

Im Folgenden wird der Einsatz der Muster Beobachter, Kompositum und Strategie jeweils einzeln beschrieben und anschließend ihr Zusammenspiel:

- **Beobachter-Muster im MVC-Modell**

Das Beobachter-Muster wird im MVC-Architekturmuster verwendet, um eine möglichst lose Kopplung vom Model zur View herzustellen. In Kapitel 6.3 ist der prinzipielle Aufbau des Musters dargestellt. Hier ein an die Situation im MVC angepasstes Klassendiagramm dieses Musters:

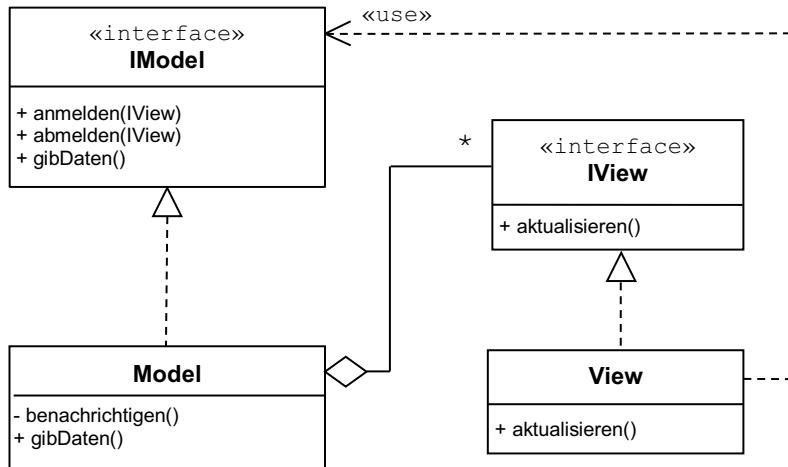


Bild 12-5 Beobachter-Muster zwischen Model und View im Pull-Betrieb des Active Model

Hierbei nimmt das Model die Rolle des Beobachtbaren ein und die View die Rolle des Beobachters. Wird eine View für ein bestimmtes Model instanziert, so muss sie sich beim Model anmelden oder angemeldet werden. Meldet sich die View selbst an, so erfolgt dies, 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 ändern. 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 als Beobachter registriert ist, die Methode `aktualisieren()` aufgerufen. Dieser 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 12.1.3.1).

Views, die nicht mehr gebraucht werden, weil sie zum Beispiel ausgebendet 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.

- **Kompositum-Muster im MVC-Modell**

Beim Zusammensetzen von grafischen Oberflächen – wie es auch die Views im MVC-Muster sind –, ist das Kompositum-Muster¹⁴⁷ 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 12-6). Die Klasse `ViewPanel` kann Elemente gruppieren und entspricht beim Kompositum-Muster einem Kompositum. Die Klassen `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-Modell abgebildet:

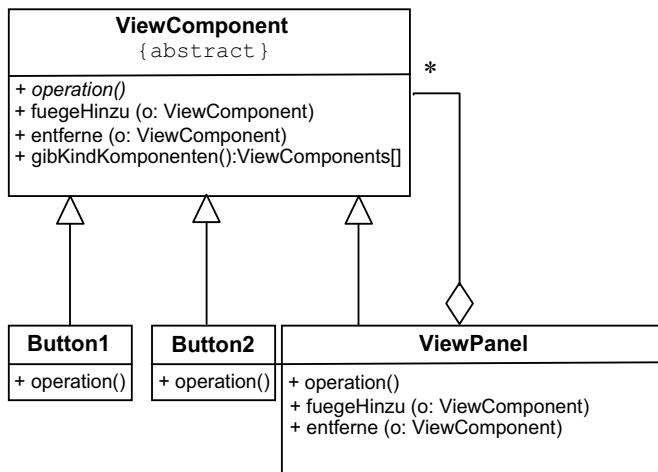


Bild 12-6 Klassendiagramm Kompositum-Muster im MVC

Durch die Verschachtelung entsteht eine Baumstruktur aus Komposita und Blättern. Komposita sind dabei aus weiteren Elementen zusammengesetzt. Blätter – wie z. B. ein Button – tragen keine anderen Elemente in sich.

Um einen Aufruf einer gemeinsamen Methode (z. B. `operation()`) in allen Baumelementen zu veranlassen, wird diese Methode im Wurzelement des Baumes aufgerufen. Von dort wird der Aufruf an die inneren Elemente delegiert (**Delegationsprinzip**).

Als Beispiel für die Methode `operation()` kann man sich eine `resize()`-Methode vorstellen, mit deren Hilfe man eine Größenänderung eines Fensters an alle enthaltenen Elemente weiterleitet.

¹⁴⁷ Das Kompositum-Muster ohne Bezug zur MVC-Architektur ist in Kapitel 5.5 dargestellt.

- **Strategie-Muster im MVC-Modell**

Die Beziehung zwischen View und Controller kann mit dem klassischen Strategie-Muster¹⁴⁸ 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.

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.

Bild 12-7 zeigt das Strategie-Muster im MVC-Kontext:

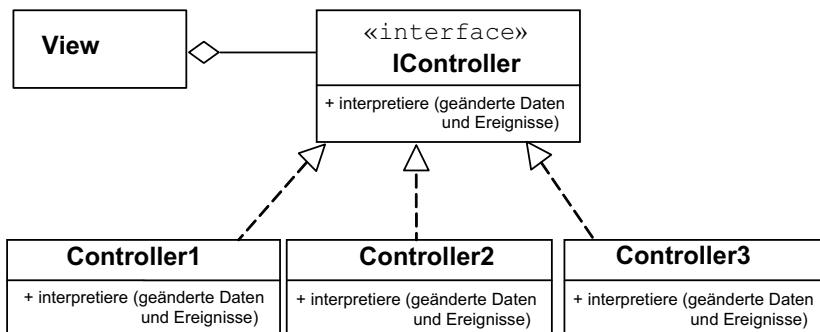


Bild 12-7 Klassendiagramm Strategie-Muster im MVC-Kontext

Dabei ist `interpretiere()` eine Methode des Controllers, 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.

- **Zusammenspiel von Beobachter-, Strategie- und Kompositum-Muster**

In Bild 12-8 wird ein mögliches Zusammenspiel der Entwurfsmuster Beobachter, Strategie und Kompositum im Architekturmuster MVC gezeigt¹⁴⁹:

¹⁴⁸ Das Strategie-Muster ohne Bezug zur MVC-Architektur ist in Kapitel 6.4 dargestellt.

¹⁴⁹ Diese Struktur dient dabei als Vorschlag und nicht als feste Vorgabe. Die Zusammenarbeit kann je nach Architektur an einigen Stellen variieren.

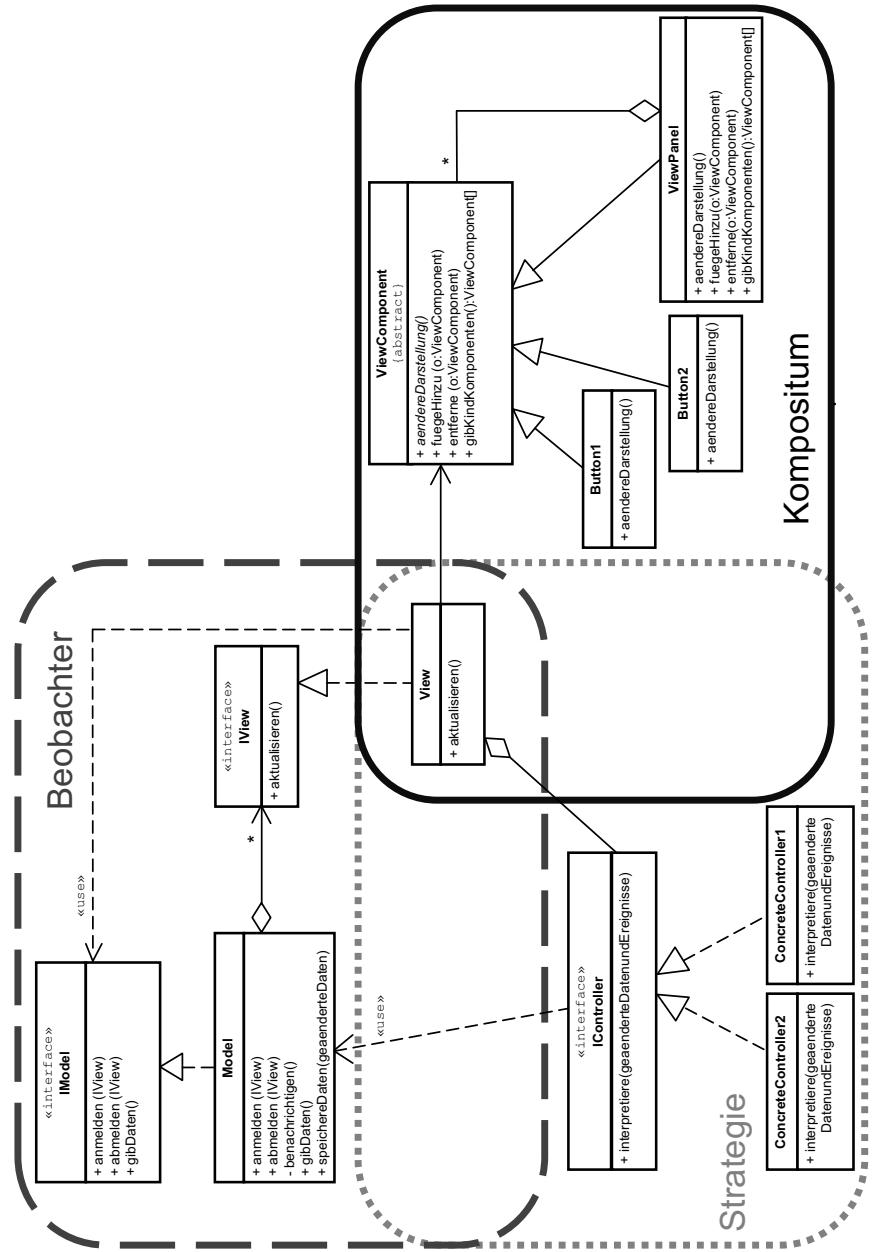


Bild 12-8 MVC-Variante mit Beobachter, Strategie und Kompositum

12.1.3.5 Programmbeispiel

Im Folgenden wird ein Beispiel gezeigt und erklärt. In diesem Beispiel wird das MVC-Muster 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. Im Folgenden werden die Komponenten der MVC-Architektur verkürzt¹⁵⁰ wiedergegeben:

- **Model**

Die beiden Schnittstellen `IObserver` und `IObservable` gehören zum Beobachter-Muster. Sie ermöglichen es dem Model, seine Views über Änderungen zu informieren:

```
// Datei: IObserver.java

public interface IObserver {
    void update();
}

// Datei: IObservable.java

public interface IObservable {
    void registerObserver(IObserver o);
    void removeObserver(IObserver o);
    void notifyObservers();
}
```

Die Schnittstelle `IModel` definiert das Model, das die Sitzplatzanzahl von drei Parteien (rot, grün und blau) speichert. Zu beachten ist hier, dass die Schnittstelle `IModel` von der Schnittstelle `IObservable` ableitet, um für die View beobachtbar zu sein. Hier die Schnittstelle `IModel`:

```
// Datei: IMODEL.java

public interface IMODEL extends IObservable {
    double getRedPercentage();
    double getBluePercentage();
    double getGreenPercentage();
    void setRedValue(int value);
    void setBlueValue(int value);
    void setGreenValue(int value);
}
```

¹⁵⁰ Der vollständige Quellcode ist zu umfangreich für dieses Buch. Er ist im Begleitmaterial zu diesem Buch enthalten.

Die Klasse `DataModel` implementiert die Schnittstelle `IModel` 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;
import java.util.List;

public class DataModel implements IMODEL {

    private int redValue = 0;
    private int greenValue = 0;
    private int blueValue = 0;
    private List<IObserver> observers = new ArrayList<>();

    @Override
    public double getBluePercentage() {
        double total = redValue + greenValue + blueValue;
        if (total > 0) {
            return blueValue / total;
        }
        return 0;
    }

    @Override
    public double getGreenPercentage() {
        double total = redValue + greenValue + blueValue;
        if (total > 0) {
            return greenValue / total;
        }
        return 0;
    }

    @Override
    public double getRedPercentage() {
        double total = redValue + greenValue + blueValue;
        if (total > 0) {
            return redValue / total;
        }
        return 0;
    }

    @Override
    public void setBlueValue(int value) {
        blueValue = value;
        notifyObservers();
    }

    @Override
    public void setGreenValue(int value) {
        greenValue = value;
        notifyObservers();
    }
}
```

```
@Override
public void setRedValue(int value) {
    redValue = value;
    notifyObservers();
}

@Override
public void notifyObservers() {
    for (IObserver observer : observers) {
        observer.update();
    }
}

@Override
public void registerObserver(IObserver o) {
    observers.add(o);
}

@Override
public void removeObserver(IObserver o) {
    observers.remove(o);
}
}
```

• View

Die drei Klassen `TableView`, `PieChartView` und `BarChartView` stellen die unterschiedlichen Sichten auf das Model dar. Der Benutzer kann zwischen Tabelleform, 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 im Begleitmaterial zu diesem Buch verfügbar. Die Views repräsentieren die Beobachter im Beobachter-Muster und implementieren deshalb die Schnittstelle `IObserver`. 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 IObserver {

    private IController controller;
    private IMModel model;
    ...
    public TableView(IMModel model) {
        createComponents();
    }
}
```

```
    this.model = model;
    model.registerObserver(this);
    controller = new TableViewController(model, this);
}

public void createComponents() {
    . . .
    setButton = new JButton("Set");
    setButton.addActionListener(
        new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                controller.setValues(
                    redTextField.getText(),
                    greenTextField.getText(),
                    blueTextField.getText()
                );
            }
        }
    );
    . . .
}
. . .
@Override
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 IObserver {

    private IMModel model;
    private JPanel viewPanel;
    private JFrame viewFrame;

    public PieChartView(IMModel model) {
        createComponents();
        this.model = model;
        model.registerObserver(this);
    }

    public void createComponents() {
        . . .
    }

    @Override
    public void update() {
        . . .
        double red = model.getRedPercentage();
        double green = model.getGreenPercentage();
        double blue = model.getBluePercentage();
    }
}
```

```
viewPanel = createChartPanel(red, green, blue);
. . .
}

private JPanel createChartPanel(double red, double green,
    double blue) {
    . . .
}

// Datei: BarChartView.java
. . .

public class BarChartView implements IObserver {

    private IMModel model;
    private JFrame viewFrame;
    private JPanel viewPanel;

    public BarChartView(IMModel model) {
        createComponents();
        this.model = model;
        model.registerObserver(this);
    }

    public void createComponents() {
        . . .
    }

    @Override
    public void update() {
        . . .
        double red = model.getRedPercentage();
        double green = model.getGreenPercentage();
        double blue = model.getBluePercentage();
        viewPanel = createChartPanel(red, green, blue);
        . . .
    }

    private JPanel createChartPanel(double red, double green,
        double blue) {
        . . .
    }
}
```

• Controller

Die Schnittstelle `IController` definiert einen Controller. Über die Methode `setValues()` kann der Controller von einer View über neue Werte für die Sitzplatzverteilung informiert werden. Hier die Schnittstelle `IController`:

```
// Datei: IController.java

public interface IController {

    void setValues(String red, String green, String blue);
}
```

Die Klasse `TableViewController` implementiert die Schnittstelle `IController`. 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 überprüft werden. Diese Überprüfungen werden hier – wieder aus Gründen der Übersichtlichkeit – nicht dargestellt. Hier die Klasse `TableViewController`:

```
// Datei: TableViewController.java

public class TableViewController implements IController {

    private IMModel model;
    private TableView tableView;

    public TableViewController(IMModel model, TableView tableView) {
        this.tableView = tableView;
        this.model = model;
    }

    // Prüft die eingegebenen Werte für rot, grün und blau
    // und übergibt sie dem Model.
    @Override
    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);
        . .
        model.setBlueValue(b);
        model.setGreenValue(g);
        model.setRedValue(r);
        . .
    }
}
```

Im Folgenden wird die **Anwendung** sowie die **Bildschirmausgabe** gezeigt.

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 13-9 gezeigte Bildschirmausgabe:

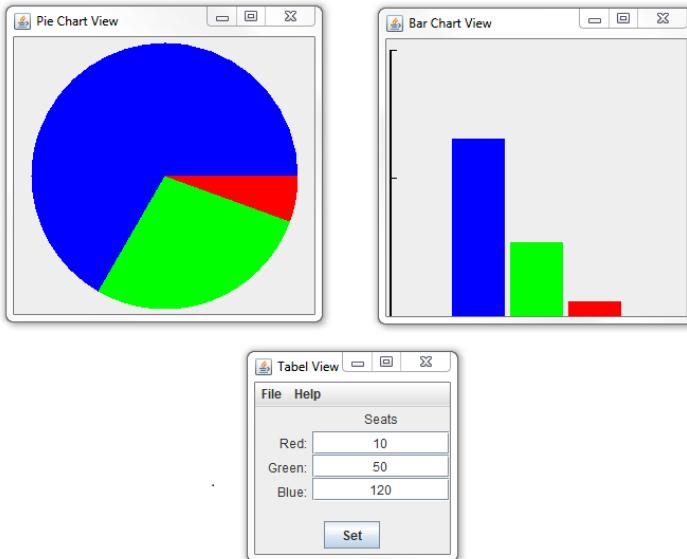


Bild 12-9 Beispielprogramm für das MVC-Architekturmuster

12.1.4 Bewertung

12.1.4.1 Vorteile

Vorteile des MVC-Musters sind:

- **Unabhängigkeit des Model vom MMI**

Die Model-Klassen können völlig unabhängig von der Gestaltung der Benutzeroberfläche entworfen werden.

- **bessere Testbarkeit des Model**

Das Model kann unabhängig von einer Oberfläche getestet werden.

- **Benutzeroberfläche veränderbar**

Die Benutzeroberfläche kann geändert werden, ohne eine Veränderung an den Model-Klassen vornehmen zu müssen.

- **verschiedene Oberflächen für dasselbe Model**

Es können verschiedene Benutzeroberflächen für dieselbe Anwendung entworfen werden.

- **durch Teile und Herrsche Übersichtlichkeit erhöht**

Durch verschiedene Komponenten wird die Struktur des Quelltextes einer Anwendung verbessert. Dies trägt wesentlich zur Übersicht bei. Damit werden Änderungen vereinfacht und die Wartbarkeit erhöht. Zudem wird die Fehlersuche erleichtert, da ein Entwickler auf Grund der gegebenen Verantwortlichkeiten besser eingrenzen kann, in welcher Komponente nach einem aufgetretenen Fehler gesucht werden muss.

12.1.4.2 Nachteile

Die folgenden Nachteile werden gesehen:

- **Komplexität des Systementwurfs steigt**

Bedingt durch die Trennung der Darstellung (View) und der Eingabeverarbeitung (Controller) auf verschiedene Komponenten steigt für den Systementwurf die **Komplexität** und damit auch der Aufwand.

- **starke Abhängigkeiten**

Aufgrund der Abhängigkeiten im MVC-Muster kann nur das Model unabhängig entwickelt und getestet werden. Es gibt **starke Abhängigkeiten**, die unerwünscht sind. Denn im MVC-Muster besteht unter anderem eine wechselseitige Abhängigkeit zwischen der View und dem Controller¹⁵¹. Die View informiert den Controller über Eingaben des Benutzers und der Controller kann andererseits den Zustand und die Ansicht der View verändern. Überdies sind View und Controller vom Model abhängig.

- **Implementierungsaufwand**

Der erhöhte Implementierungsaufwand durch die Unterteilung in viele Klassen ist bei kleinen Anwendungen nicht gerechtfertigt.

12.1.5 Einsatzgebiete

Das MVC-Muster kann bei jeder Art von interaktiver Software zur Strukturierung angewandt werden.

Das MVC-Architekturmuster wird in irgendeiner Variante 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.

¹⁵¹ Oftmals wird aufgrund der wechselseitigen Abhängigkeit der Verbund von View und Controller in einer View-Controller-Komponente zusammengefasst. In Swing wird eine solche Komponente "Delegate" genannt.

12.1.6 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 Java-Beans 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.

12.2 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** in Kapitel 9.1). Bei Architekturmustern für interaktive Systeme wird in diesem Buch das Architekturmuster **Model-View-Controller** (siehe Kapitel 12.1) behandelt. 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.

12.3 Kontrollfragen

Aufgaben 12.3.1: Model-View-Controller

- 12.3.1.1 Erklären Sie die Grundzüge des Model-View-Controller-Musters.
- 12.3.1.2 Welche Abhängigkeiten bestehen zwischen View, Controller und Model?
- 12.3.1.3 Welche Schwachstellen hatten Systeme üblicherweise in der Zeit vor MVC?
- 12.3.1.4 Nennen Sie die Aufgaben des Model im Pull-Betrieb des Active Model.
- 12.3.1.5 Nennen Sie die Aufgaben der View im Pull-Betrieb des Active Model.
- 12.3.1.6 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 ein Buchstabe zur eindeutigen Kennzeichnung an die letzte Stelle angehängt.

Bei privater Mitteilung oder Veröffentlichungen des Internets treten anstelle der beiden letzten Ziffern Buchstaben. Nationale oder internationale Standards mit einer Numerierung 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/rieble.pdf> (Stand 16.11.2022).
- Bec87 Beck, K., Cunningham, W.: "Using Pattern Languages for Object-Oriented Programs" (presented at OOPSLA-87 conference). <http://c2.com/doc/oopsla87.html> (Stand 16.11.2022).
- Bec08 Beck, K.: "Implementation Patterns: Der Weg zu einfacherer und kosten-günstigerer Programmierung". Addison-Wesley, München, 2008.
- Bie00 Bienhaus, D.: "Muster-orientierter Ansatz zur einfacheren Realisierung Verteilter Systeme". Tectum Verlag, 2000.
- Blo17 Bloch, J.: "Effective Java". Addison-Wesley Professional, 2017.
- Bur92 Burbeck, S.: "Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)". http://www.dgp.toronto.edu/~dwigdor/teaching/csc2524/2012_F/papers/mvc.pdf (Stand 16.11.2022).
- Bus07 Buschmann, F., Henney, K., Schmidt, D.C.: "Pattern-Oriented Software Architecture – A Pattern Language for Distributed Systems", Vol. 4, John Wiley & Sons Ltd, 2007.
- Bus98 Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. und Stal, M.: "Pattern-orientierte Software-Architektur: Ein Pattern-System". Addison-Wesley-Longman, München, 1998.

- Cra02 Crahen, E.: "Facet: A pattern for dynamic interfaces", PLoP 2002 (9th Conference on Pattern Languages of Programs).
<https://www.hillside.net/plop/plop2002/proceedings.html>
(Stand 16.11.2022).
- db2cop IBM DB2 Infocenter zum Thema Connection Pooling:
<https://www.ibm.com/docs/en/db2/9.7?topic=management-connection-pooling> (Stand 16.11.2022).
- Dij68 Dijkstra, E.: "The Structure of the "THE"-Multiprogramming System". Comm. ACM, Vol. 11, No. 5, May 1968.
- 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 16.11.2022).
- 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).
<https://martinfowler.com/apsupp/roles.pdf> (Stand 16.11.2022).
- Gal03 Gall, H., Hauswirth, M., Dustdar, S.: "Software-Architekturen für Verteilte Systeme: Prinzipien, Bausteine und Standardarchitekturen für moderne Software". Springer, Berlin, 2003.
- Gam15 Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software". mitp-Verlag, 2015.
- Gam94 Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1994.
- Gam97 Gamma, E.: "Extension Object". In [Mar97]. Chapter 6
- Gol19 Goll, J.: "Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik". Springer Vieweg, Wiesbaden, 2019.
- Gra02a Grand, M.: "Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML". 2. Auflage, John Wiley & Sons; 2002.
- Gra02b Grand, M.: Patterns in Java, Volume 3: Java Enterprise Design Patterns. John Wiley & Sons, 2002.

- 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". <https://www.ibm.com/docs/en/radfw/9.6.1?topic=applications-web-services-inspection-language-wsil> (Stand 16.11.2022).
- 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.
- Kir02 Kircher, M., Prashant, J.: „Pooling Pattern“, EuroPlop 2002, <https://www.kircher-schwanninger.de/michael/publications/Pooling.pdf> (Stand 16.11.2022).
- Lah09 Lahres, B., Rayman, G.: "Objektorientierte Programmierung". 2. Auflage, Galileo Computing, 2009.
- Lew14 Lewis, J., Fowler, M.: "Microservices". <https://martinfowler.com/articles/microservices.html> (Stand 16.11.2022).
- Mar97 Martin, R.C., Riehle, Dirk, Buschmann, Frank: "Pattern Languages of Program Design 3". Addison Wesley, 1997.
- oasudd OASIS, "UDDI Specifications TC – Committee Specifications" <http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm> (Stand 16.11.2022).
- omgcos "Common Object Request Broker Architecture: Core Specification". <http://www.omg.org/cgi-bin/doc?formal/04-03-12.pdf> (Stand 16.11.2022).
- 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 16.11.2022).
- 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.
- shespj Sherzad, R.: "Singleton Pattern in Java".
<http://www.theserverside.de/singleton-pattern-in-java>
(Stand 16.11.2022).
- 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 16.11.2012).
- w3wsdl "Web Services Description Language (WSDL) 1.1".
<http://www.w3.org/TR/wsdl>
(Stand 16.11.2022).
- w3xmls "XML Schema 1.1".
<http://www.w3.org/XML/Schema>
(Stand 16.11.2022).

Abkürzungsverzeichnis

API	Application Programming Interface
ASN.1/BER	Abstract Syntax Notation 1/Basic Encoding Rules
AWT	Abstract Window Toolkit
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
DBMS	Database Management System
EJB	Enterprise JavaBeans
FIFO	First-In-First-Out
GIOP	General Inter-ORB Protocol
GoF	"Gang of Four"
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IIOP	Internet Inter-ORB Protocol
IPC	Interprozesskommunikation
ISO	International Organization for Standardization
JAX-RS	Java API for RESTful Web Services
JAX-WS	Java API for XML Web Services
JRE	Java Runtime Environment
JSP	JavaServer Pages
JVM	Java-Virtuelle Maschine
MEP	Message Exchange Pattern
MHS	Message Handling System
MIB	Management Information Base
MMI	Man-Machine Interface
MP3	MPEG-1 Audio Layer 3, eine Audiokodierung
MPEG	Motion Picture Expert Group
MVC	Model-View-Controller
OMG	Object Management Group
ORB	Object Request Broker
OSGi	Früher: Open Services Gateway Initiative
OSI	Open Systems Interconnection
REST	Representational State Transfer
RCP	Rich Client Plattform
SEI	Service Endpoint Interface
SEU	Software-Entwicklungsumgebung
SNA	Systems Network Architecture
SOA	Service-Oriented Architecture
SOAP	SOA-Protokoll, früher: Simple Object Access Protocol
STL	Standard Template Library
SW	Software

TCP/IP	Transmission Control Protocol/Internet Protocol
UDDI	Universal Discovery, Description, Integration
UML	Unified Modeling Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
WSIL	Web Services Inspection Language
XML	Extensible Markup Language

Begriffsverzeichnis

- **Abgeleitete Klasse**

Eine abgeleitete Klasse (Unterklasse, untergeordnete Klasse, Subklasse) wird von einer anderen Klasse, der sogenannten Basisklasse (Oberklasse, übergeordnete Klasse, 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 Modellementen oder Codestücken, die zum Ausdruck bringt, dass sich eine Änderung des unabhängigen Elementes auf das abhängige Element auswirken kann.

- **Abstract Window Toolkit**

Die Klassenbibliothek AWT ist die Vorgängerin der Klassenbibliothek Swing. AWT-GUI-Komponenten sind in Aussehen und Verhalten abhängig vom Betriebssystem. Aufgrund ihrer Abhängigkeit vom Betriebssystem werden sie als "schwergewichtig" bezeichnet. "Leichtgewichtige" Swing-GUI-Komponenten werden hingegen mit Hilfe der Java 2D-Klassenbibliothek durch die Java Virtual Machine 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 Methodenkopf, d. h. nur die entsprechenden Methodenkörper, hat. Eine Methode ohne Methodenkopf 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 instanziert werden.

- **Abstrakter Dekorierer**

Ein abstrakter Dekorierer aggregiert ein Objekt vom Typ einer Komponente und bietet deren Schnittstelle an. Der abstrakte Dekorierer delegiert seine Methodenaufrufe an das aggregierte Objekt. Der abstrakte Dekorierer ist "abstrakt" in dem Sinne, dass er keine konkrete Dekorationsfunktion implementiert.

- **Abstraktion**

Unter Abstraktion versteht man das Weglassen irrelevanter Einzelheiten und die Konzentration auf das Wesentliche.

- **Administrative Sicht**

Hier betrachtet man nur die erforderliche Funktionalität für die Administratoren. Man unterscheidet generell die Software für die Anwender, die sogenannte operationelle

Software, und die Software für die Administratoren des Systems, die administrative Software.

- **Aggregation**

Eine Aggregation ist eine spezielle Assoziation, die eine Beziehung zwischen einer referenzierenden Komponente und einer referenzierten Komponente ausdrückt. Eine referenzierte Komponente kann von mehreren Objekten referenziert werden. Bei einer Aggregation ist die Lebensdauer eines referenzierten Objekts nicht mit der Lebensdauer des referenzierenden Objekts gekoppelt. Bei einer Komposition hingegen gehört ein Teil genau zu einem einzigen zusammengesetzten Ganzen, wobei ein Teil genauso lange wie das enthaltende Ganze lebt.

- **Akteur**

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

- **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 eines Systems**

Die Beschreibung der Architektur eines Systems, der Systemarchitektur, umfasst

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

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

- **Assoziation**

Eine Beschreibung eines Satzes von Verknüpfungen (engl. links) zwischen Objekten. Dabei verbindet eine Verknüpfung zwei oder mehr¹⁵² Objekte als Gleichberechtigte (engl. peers). Eine Assoziation ist prinzipiell eine symmetrische Strukturbziehung 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:

¹⁵² Man kann Multiplizitäten für Objekte einführen.

1. Ein Attribut ist eine Eigenschaft einer Klasse oder eines Objekts.
 2. Ein Assoziationsattribut 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 Spaltelements in einer Zeile.
- **Basisklasse**
Eine Basisklasse (Superklasse, Oberklasse, übergeordnete Klasse) steht in einer Vererbungshierarchie über einer aktuell betrachteten Klasse.
 - **Belang**
Zusammenhängende Funktionen bilden in der Sprechweise von Separation of Concerns einen Belang (engl. concern). Verschiedene Belange sollen nach dem Prinzip Separation of Concerns sauber getrennt voneinander geführt werden.
 - **Benutzungsabstraktion**
Der Zugang zu einem Modul bzw. einer Komponente erfolgt über eine definierte, schmale Schnittstelle, welche die Leistung eines Moduls abstrahiert und dessen Services anbietet. Benutzer müssen nur wissen, welche Leistung durch ein Modul erbracht wird, nicht aber, wie dessen Implementierung erfolgt. Bei Vorliegen einer Benutzungsabstraktion kann die Verwendung eines Moduls erfolgen, ohne dass man dessen Aufbau kennt.
 - **Beziehung**
Eine Beziehung zwischen zwei oder mehr Elementen beschreibt, dass diese Elemente zueinander Bezug haben. Hierbei gibt es statische und dynamische Beziehungen.
 - **Black-Box-Wiederverwendung**
Da keine internen Details der Objekte sichtbar sind und die Objekte als Black-Boxes erscheinen, wird "Objektkomposition" auch Black-Box-Wiederverwendung genannt [Gam15, S. 50].
 - **Classifier**
Classifier ist ein Begriff aus dem Metamodell von UML. Die Metaklasse Classifier ist eine abstrakte Metaklasse.

Es gibt eine Hierarchie von Classifiern. Die Wurzel ist die Metaklasse Classifier des Pakets Classes::Kernel. Von dieser Wurzel wird abgeleitet und ein ganzer Baum von Classifiern aufgebaut. Eine der Spezialisierungen der Metaklasse Classifier ist die Metaklasse Class, die Abstraktion einer Klasse. Instanzen der Metaklasse Class sind die Klassen.

Jedes Modellement von UML, von dem eine Instanz gebildet werden kann, ist ein Classifier. Ein Classifier bis auf die abstrakte Metaklasse Classifier kann instanziert werden.

Ein Classifier besitzt in der Regel eine Struktur und ein Verhalten. Schnittstellen besitzen als einzige Ausnahme meist keine Attribute, d. h., sie haben keine Struktur.

- **Client**

Siehe Kunde.

- **Concern**

Das engl. Wort "concern" kann mit Belang, Anliegen, Interesse oder Verantwortung übersetzt werden.

- **Delegation**

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

- **Dependency Injection**

Die Erzeugung von Objekten wird an eine dafür vorgesehene Instanz delegiert. Diese Instanz, die auch Injektor genannt wird, erzeugt zur Laufzeit auch die Verknüpfungen zwischen den Objekten. Damit wird die Abhängigkeit zwischen nutzenden und benötigtem Objekt stark abgeschwächt, aber nicht vollständig aufgelöst.

Zur Kompilierzeit kennt ein Objekt einer nutzenden Klasse statt der konkreten Klasse nur eine Abstraktion der Klasse des vom nutzenden Objekt benötigten Objekts. Der Vertrag dieser Abstraktion muss vom Objekt der benutzten Klasse, welches der Injektor an die Stelle der Abstraktion setzt, eingehalten werden.

- **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 Principle**

Das Dependency Inversion Principle 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 einer tieferen Ebene vorgeben und aggregieren. Die untergeordneten Klassen sollen nur von der entsprechenden Abstraktion abhängen.

- **Dependency Lookup**

Bei dem Konzept Dependency Lookup sucht ein Objekt, das ein anderes Objekt braucht, nach diesem anderen Objekt z. B. in einem Register, um die Verknüpfung mit dem benötigten Objekt herzustellen. Zur Suche braucht das suchende Objekt nur einen Schlüssel wie den Namen des gesuchten Objekts zu kennen und ist beim Suchen von dem anderen Objekt weitgehend entkoppelt.

Ein suchendes Objekt muss nicht mehr die konkrete Klasse des von ihm benötigten Objekts, aber die Abstraktion der Klasse des benötigten Objekts kennen und einhalten, um die Methoden des gesuchten Objekts aufrufen zu können.

Das suchende Objekt ist aber ferner auch vom Register abhängig, da es von diesem seine benötigten Objekte bezieht.

- **Design**

Wird im Sinne von Entwurf verwendet.

- **Design Pattern**

Siehe Entwurfsmuster.

- **Design to Test**

Das Prinzip Design to Test betont die hohe Bedeutung einer einfach zu testenden Systemarchitektur. Das betrachtete System muss aus Aufwandsgründen auf leichte Weise getestet werden können! Ist die gefundene Architektur nicht leicht zu testen, weil sie zu komplex ist bzw. zu viele Abhängigkeiten bestehen, so wird sie verworfen. Es sollte komponentenweise getestet werden können.

- **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 Pulsdigramm der Elektrotechnik ist.

- **Domäne**

Eine Domäne umfasst die Aufgaben einer bestimmten Anwendung, auch Fachkonzept genannt. Die Domäne soll durch die zu entwickelnde Software unterstützt werden.

- **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 patterns)**

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.

- **Extreme Programming**

Extreme Programming (XP) von Kent Beck [Bec99] ist eine agile Methode, die im Gegensatz zu spezifikationsorientierten Methoden das Programmieren in den Vordergrund eines Projekts stellt sowie die Planung und die Erstellung von Dokumenten auf das Allernötigste beschränkt.

- **Feature**

Ein Feature ist eine kleine, nützliche Funktion für den Kunden.

- **Framework**

Ein Framework offeriert dem nutzenden System Klassen, von welchen das System abgeleitet werden kann und somit deren Funktionslogik erben kann. Ein Framework bestimmt die Architektur der Anwendung, also die Struktur im Großen. Es definiert weiter die Unterteilung in Klassen und Objekte, die jeweiligen zentralen Zuständigkeiten, die Zusammenarbeit der Klassen und Objekte sowie den Kontrollfluss [Gam15, S. 60].

- **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 UML das Modellement 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. 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**

Wird eine Einheit wie ein Modul oder eine Klasse gekapselt, werden nach außen nur wenige Informationen über die Services dieser Einheit freigegeben (schmale Schnittstelle). Die Implementierung wird gekapselt oder verborgen (Information Hiding).

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

- **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.

- **Interface Segregation Principle**

Clients sollen nur Schnittstellen erhalten, deren Methoden sie auch tatsächlich nutzen (Rollen-Schnittstellen).

- **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-zu-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**

Inversion of Control ist die Umkehr des Kontrollflusses. Dabei 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**

Darunter versteht man die sinnvolle Zusammenfassung von Daten und zugehörigen Funktionen in einer gemeinsamen Kapsel, der Klasse. Daten und zugehörige Funktionen werden nicht getrennt. Die Kapselung eines Moduls umfasst Information Hiding und den Zugriff auf ein Modul über eine Schnittstelle als Abstraktion der Leistung eines Moduls. Die Implementierung der Module ist verborgen und ist nur über deren Schnittstellen zugänglich (Benutzungsabstraktion).

- **Kardinalität**

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

1. Auf dem Gebiet der Datenbanken legt die Kardinalität in der Analyse 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..n.

- **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 sowie 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**

Siehe Modul.

- **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. Ein Teil lebt genauso lange wie das enthaltende Ganze.

- **Konkrete Klasse**

Eine konkrete Klasse kann im Gegensatz zu einer abstrakten Klasse instanziert 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 Rückgabewert und kann nicht vererbt werden.

- **Kontrollobjekt**

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

- **Kunde**

Hier: Teil eines Computerprogramms, welches gewisse Objekte oder Funktionalitäten benutzt.

- **Layer Bridging**

Eine Schicht kann auf alle unter ihr liegenden Schichten zugreifen.

- **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.

- **Link**

Siehe Verknüpfung.

- **Liskovsches Substitutionsprinzip**

Das liskovsche Substitutionsprinzip fordert, dass eine Referenz vom Typ einer Basisklasse nicht nur auf ein Objekt der Basisklasse, sondern auch auf ein Objekt einer von der Basisklasse abgeleiteten Klasse zeigen kann. Damit muss nicht bei jedem Zugriff einzeln geprüft werden, von welchem Typ das referenzierte Objekt einer Vererbungshierarchie eigentlich ist.

- **Logisch zusammenhängend**

Aus Sicht des Problembereichs stark zusammenhängende Module werden als "logisch zusammenhängend" bezeichnet.

- **Lösungsbereich**

Im Problembereich ist man in einer idealen Welt mit unendlicher Performance und keinem technischen Fehler, im Lösungsbereich ist man in der physischen Welt mit ihren Einschränkungen und technischen Fehlern.

Im Problembereich ist man in der Welt der Logik der Anwendung. Im Lösungsbereich ist man in der Welt der Konstruktion.

Im Problembereich gibt es kein technisches System, im Lösungsbereich gibt es ein technisches System.

Im Problembereich kümmert man sich um die Verarbeitungsprozesse in einer idealen Gedankenwelt (Essenz des Systems), im Lösungsbereich um alle Funktionsklassen.

- **Loose Coupling and Strong Cohesion**

Ein gutes Softwaredesign sollte nach "Loose Coupling and Strong Cohesion" eine schwache Kopplung (engl. loose coupling) zwischen den einzelnen Teilsystemen und eine starke Kohäsion (engl. strong cohesion) innerhalb eines einzelnen Teilsystems aufweisen.

- **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.

- **Mock-Objekt**

Ein Mock-Objekt ist ein Objekt, das als Platzhalter für echte Objekte innerhalb eines Komponententests verwendet wird. Der Begriff kommt aus dem Englischen und kann

mit "etwas vortäuschen" übersetzt werden. An die Stelle von Objekten einer tieferen Ebene können beispielsweise Mock-Objekte treten, welche die Schnittstelle erfüllen und das Testen durch die Rückgabe simulierter, sinnvoller Werte unterstützen.

Ein Mock-Objekt hat eine Logik implementiert. Ein Mock-Objekt kann beim Testen wie das tatsächliche Objekt aufgerufen werden und liefert vorher festgelegte, sinnvolle Werte zurück.

- **Multiplizität**

Eine Multiplizität bezeichnet in UML einen Bereich zulässiger Kardinalitäten. Der Bereich ist durch eine untere und eine obere Grenze bestimmt.

- **Modul**

Es gibt keine einheitliche Definition. In diesem Buch wird ein Modul folgendermaßen verwendet:

Ein Modul kapselt seine Implementierung. Solange die Schnittstelle eines Moduls nicht verändert wird, kann man im Inneren eines Moduls beliebige Änderungen durchführen. Die Leistung eines Moduls steht über schmale Schnittstellen nach außen zur Verfügung. Die Verwendung eines Moduls kann also erfolgen, ohne dass man den Aufbau eines Moduls kennt (Benutzungsabstraktion).

Wenn die Schnittstellen der Module feststehen, können die Module unabhängig voneinander entwickelt werden (Parallelität der Entwicklung der Module). Module sind in sich logisch sehr stark zusammenhängend und tragen nach dem Single Responsibility Principle von Robert C. Martin nur eine einzige Verantwortlichkeit. Module sollen getrennt getestet werden können.

- **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.

- **Öffentliche Methode**

Siehe Schnittstellenmethode.

- **Oberklasse**

Siehe Basisklasse.

- **Objekt**

Siehe Klasse.

- **Objektdiagramm**

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

- **Objektkomposition**

Bei Verwendung des Prinzips "Ziehe Objektkomposition der Klassenvererbung vor" aggregiert nach UML zur Kompilierzeit eine Klasse eine Abstraktion. Zur Laufzeit tritt an die Stelle der Abstraktion ein Objekt, dessen Klasse die Abstraktion implementiert.

- **Operation**

Eine Operation stellt in UML die Abstraktion einer Methode dar. Sie umfasst den Namen, die Übergabeparameter, den Rückgabetyp, 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 Rückgabetyp 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. Man unterscheidet generell die Software für die Anwender, die sogenannte operationelle Software, und die Software für die Administratoren des Systems, die administrative Software.

- **Over-Engineering**

Man spricht von Over-Engineering, wenn ein Produkt in höherer Qualität oder mit mehr Aufwand erstellt wird, als es der Kunde wünscht. Dabei wird oft die Zahlungsbereitschaft des Kunden überschritten.

- **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. auf einer Festplatte abgespeichert werden.

- **Polymorphie von Objekten**

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.

- **Produktfamilie**

Das ist eine Gruppe von zusammengehörigen Objekten, die z. B. für ein bestimmtes Betriebssystem erzeugt werden.

- **Paket in Java**

Ein Paket in Java dient zur Gruppierung von Klassen und Schnittstellen sowie von Paketen in einem Paket. Ein Paket stellt einen Namensraum dar, ist eine Einheit für den Zugriffsschutz und erleichtert die Übersicht.

- **Problembereich**

Der sogenannte Problembereich oder Problem Domain ist der Bereich, der automatisiert werden soll. Es ist derjenige Teil der realen Welt, der später durch die zu realisierende Software abgedeckt werden soll. Siehe auch Lösungsbereich.

- **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 (Classifizieren 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 Classifizieren wie Klassen, die Schnittstellen implementieren, und bei Kollaborationen, die Anwendungsfälle implementieren.

- **Rolle**

Der Begriff einer Rolle ist mehrdeutig:

1. Rolle als Stellvertreter¹⁵³.

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 Schnittstelle (engl. interface) stellt das Bindeglied zwischen den Nutzern eines Moduls und der Implementierung dieses Moduls dar. Nur die Schnittstelle eines Moduls ist nach außen sichtbar.

Eine Schnittstelle ist die 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.

Eine Schnittstelle erlaubt es, auf die Implementierung eines Moduls zuzugreifen, ohne die Implementierung dieses Moduls zu kennen. Sind Module nicht gekoppelt, können sie nicht zusammenarbeiten.

Eine Schnittstelle stellt eine Abstraktion des entsprechenden Moduls und dessen angebotener Leistungen dar. Auf der Ebene von Klassen beinhaltet eine Schnittstelle die Methodenköpfe – also Methodennamen, Rückgabetypen und Übergabeparameter – von öffentlichen Methoden der Klasse. Eine bestimmte Schnittstelle kann alle Operationen oder aber auch nur einen Teil der Operationen einer Klasse repräsentieren. Eine Schnittstelle spezifiziert einen Vertrag, den das realisierende Element erfüllen muss.

Schnittstellen können in objektorientierten Programmiersprachen oftmals mit einer formalen Syntax beschrieben werden wie beispielsweise in Form eines Interface in Java.

In der Vergangenheit sah ein Interface in Java keine syntaktische Möglichkeit vor, Methoden zu definieren. Dies ist erst seit Java 8 möglich. Schnittstellen in Java können jetzt auch den Charakter einer abstrakten Klasse¹⁵⁴ haben und dedizierte Methoden vorgeben. Die Einführung von Implementierungen in einer Schnittstelle untergräbt jedoch die eigentliche Bedeutung einer Schnittstelle.

¹⁵³ 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, 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.

¹⁵⁴ Abstrakte Klassen dürfen im Allgemeinen neben abstrakten Methoden auch für einige Methoden Implementierungen enthalten – wie neuerdings auch die Interfaces in Java.

Eine Schnittstelle im Sinne dieses Buches enthält grundsätzlich keine Implementierungen.

- **Schnittstellenmethode**

Über den Methodenkopf einer Schnittstellenmethode (öffentlichen Methode) kann man auf die Daten eines Objektes zugreifen.

- **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.

- **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 Rückgabetypr.
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. Zu den Stakeholdern eines Projekts gehören beispielsweise

- Projektleiter,
- Projektmitarbeiter,
- Kunden, Benutzer oder
- Auftraggeber.

- **Strict Layering**

Eine Schicht kann nur auf die direkt unter ihr liegende Schicht zugreifen.

- **Struktur**

Die Struktur eines Systems ist sein statischer Aufbau.

- **Stub**

Stub bedeutet Platzhalter. Ein Stub ist ein vorläufiger, einfacher Ersatz für eine andere Komponente, die noch nicht erstellt ist, aber benötigt wird, damit ein Programm ablauffähig wird und getestet werden kann. Ein Stub bildet die noch fehlende Komponente in einfacherster Weise nach. Er hat keine Logik. Die Aufgabe des Stubs ist es dabei nur, die Durchführung eines Aufrufs zu gewährleisten. Damit ist die Lauffähigkeit eines Programmes hergestellt. Ein Stub gibt beim Aufruf einer seiner Methoden einen festen Wert zurück.

- **Subklasse**

Siehe abgeleitete Klasse.

- **Subsystem**

Eine Komponente, die einen größeren Teil eines Systems darstellt.

- **Superklasse**

Siehe Basisklasse.

- **Subklasse**

Siehe abgeleitete Klasse.

- **Subtyp**

Eine abgeleitete Klasse stellt einen eigenen Typ dar, der dann als Subtyp bezeichnet wird, wenn das liskovsche Substitutionsprinzip und damit der Vertrag der Basisklasse eingehalten wird. Dann hat die Interpretation "is a" ihre Berechtigung, da ein Subtyp alles hat (Methoden und Datenfelder), was eine Basisklasse auch hat, und sich die Objekte des Subtyps genau so verhalten wie die Objekte der Basisklasse, wenn die Objekte des Subtyps über Methoden der Basisklasse angesprochen werden.

- **Superklasse**

Siehe Basisklasse.

- **Tabelle**

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

- **Technische Funktion**

Bei den Anwendungsfunktionen eines Systems kann man zwischen Verarbeitungsfunktionen und den sogenannten technischen Funktionen unterscheiden.

Um eine untersuchte Verarbeitungsfunktion in Realität ordnungsgemäß auf dem Rechner zum Laufen zu bringen, braucht man beim Entwurf im Lösungsbereich in der Regel zusätzlich zu den betrachteten Verarbeitungsfunktionen noch die bereits genannten technischen Funktionen des Lösungsbereichs.

Sogenannte technische Funktionen sind:

- Funktionen zur Datenhaltung,
- Funktionen zur Ein- und Ausgabe,
- Funktionen zur Rechner-Rechner-Kommunikation,
- Funktionen zur Sicherheit,
- Funktionen zur Parallelität/Interprozesskommunikation.

- **Threadsicherheit**

Eine Methode ist threadsicher, wenn sie aus mehreren Threads gleichzeitig aufgerufen werden kann, ohne dass sich diese Aufrufe gegenseitig stören. Insbesondere wenn diese Aufrufe auf eine gemeinsame Ressource zugreifen wollen, können Wettrennen (sog. Race Conditions) um den Zugriff auf diese Ressource entstehen, die durch entsprechende Synchronisierungsmaßnahmen verhindert werden müssen, um Threadsicherheit zu gewährleisten.

- **Typ**

Der Typ einer konkreten Klasse hat Attribute und Operationen und für seine Attribute einen Wertebereich.

- **Unterklasse**

Siehe abgeleitete Klasse.

- **Use Case**

Siehe Anwendungsfall.

- **Verantwortlichkeit**

"A reason to change" (Robert C. Martin) charakterisiert eine sogenannte Verantwortlichkeit (engl. responsibility) einer Klasse.

- **Verarbeitungsfunktion**

Bei den Anwendungsfunktionen eines Systems kann man zwischen Verarbeitungsfunktionen und den sogenannten technischen Funktionen unterscheiden.

Eine Verarbeitungsfunktion "tut" etwas und ist sozusagen ein Prozessor, der ein Problem löst, das bereits im Problembereich der untersuchten Domäne existiert. Beim Studium des Problembereichs – also der Fachlichkeit – eines Programms stehen im Rahmen der Analyse die fachlichen Aufgaben, also die Verarbeitungsfunktionen, und ihre Verknüpfungen im Zentrum des Interesses. Die Beschränkung in der Analyse auf Verarbeitungsfunktionen verhindert, dass man sich in der Menge der zu analysierenden Funktionen verläuft. Man hat in der Analyse noch kein lauffähiges Programm, sondern befasst sich mit den für das Programm erforderlichen fachlichen Funktionen und deren Abläufen aus logischer Sicht.

- **Vererbung**

In der Objektorientierung kann eine Klasse von einer anderen Klasse statisch zur Kompilierzeit 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.

- **Vertrag**

Ein Vertrag regelt insbesondere die Beziehungen zwischen Aufrufer und Aufgerufenen.

- **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.

- **White-Box-Wiederverwendung**

Wiederverwendung durch Unterklassenbildung wird oft auch White-Box-Wiederverwendung genannt, da die öffentlichen und geschützten Elemente der Basisklasse in der abgeleiteten Klasse direkt sichtbar sind, wenn sie nicht verdeckt bzw. überschrieben werden.

- **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 Regionen¹⁵⁵) des näher zu beschreibenden Classifiers und aus Transitionen. Zustände beschreiben Verzögerungen, Eintritts-, Austritts- und Zustandsverhalten. Transitionen spezifizieren Ereignisse für Zustandsübergänge, deren

¹⁵⁵ Regionen dienen zur Beschreibung von Parallelitäten.

Bedingungen und das Verhalten des Classifiers beim Übergang. Ein Zustandsdiagramm wird zur Modellierung von reaktiven Systemen benötigt.

- **Zustandsübergangsdiagramm**

In Zustandsübergangsdiagrammen werden Zustände von Systemen nach David Harel oder Hatley/Pirhai 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).

Index

Abstrakte Fabrik	
Bewertung	282
Klassendiagramm.....	275
Nachteile	283
Vorteile	282
abstrakte Klasse vs. Schnittstelle.....	26
Adapter	60, 81, 109, 131, 221
Bewertung	68
Klassen-Adapter.....	61
Klassendiagramm.....	62
Nachteile	69
Objekt-Adapter.....	61
Vorteile	68
Aggregation	43
Analysemuster	20
Architekt	
analytische Aufgaben	8
konstruktive Aufgaben.....	9
Architektur.....	5
~muster	20, 23, 310
Abstraktion	3
Adaptierbare Systeme	310
Administration	11
Änderbarkeit und Erweiterbarkeit	6
Architektur- und Entwurfsmuster	11
Auslieferungskonzept.....	12
Ausnahmebehandlung	11
Bauplan	4
Bewährtheit	7
Datenhaltungskonzept.....	10
Definition	4
Dynamik	5
Ein- und Ausgabekonzept	11
Einfachheit und Verständlichkeit	6
Entwurfsmuster	23
IEEE 1471	5
Integrationskonzept	12
Interaktive Systeme.....	310
iteratives Finden	4
konkrete	2
Muster	22
Performance.....	6
Prototypen.....	13
Prozesskonzept.....	10
Qualitäten.....	6
Sicherheitskonzept.....	11
Skalierbarkeit	7
Stabilität	7
Statik	5
Strategie.....	5
Struktur eines Systems	310
Testbarkeit	6
Testkonzept.....	11
Unabhängigkeit der Komponenten...6	
Verteilte Systeme	310
Wartbarkeit.....	6
Arten von Entwurfsmustern.....	25
Befehl	136
Bewertung	146
Klassendiagramm	138
Nachteile	146
Vorteile	146
Beobachter	149, 189, 247, 430
Bewertung	158
Klassendiagramm	151
Nachteile	159
Vorteile	158
Besucher.....	100, 160, 190, 191
Bewertung	175
Klassendiagramm	164
Nachteile	176
Vorteile	175
Bewertung	
Abstrakte Fabrik	282
Adapter	68
Befehl	146
Beobachter	158
Besucher	175
Broker	385
Brücke	80
Dekorierer	94
Fabrikmethode	271
Fassade	107
Iterator	188
Kompositum	118
Layers	320
MVC	441
Objektpool	305
Pipes and Filters	341
Plug-in	358
Proxy	128
Rolle	219

Schablonenmethode	229	Bewertung	271
Singleton	296	Klassendiagramm	267
SOA.....	395	Nachteile	272
Strategie.....	236	Vorteile	271
Vermittler.....	245	Facet.....	222
Zustand	257	Fassade	70, 101, 222, 284, 297, 321
Black-Box-Verhalten	43	Bewertung	107
Black-Box-Wiederverwendung.....	45	Klassendiagramm	102
Broker	247, 365, 397	Legacy-System	109
Bewertung	385	Nachteile	108
Klassendiagramm.....	367	Vorteile	107
Nachteile	386	Filter.....	333
Vorteile	385	Forwarder-Receiver	389
Brücke.....	70	Funktion	
Bewertung	80	technisch	25
Klassendiagramm.....	72	Gang of Four.....	24
Nachteile	81	Generalisierung	26
Vorteile	80	Idiom.....	20, 23
Client-Dispatcher-Server.....	388	Inversion of Control.....	150
Controller	423, 428	Iterator	169, 178
Dekorierer...81, 82, 122, 131, 177, 209,	221, 237, 343, 360	Bewertung	188
Bewertung	94	extern	180
Klassendiagramm.....	83	intern	180
Nachteile	95	Klassendiagramm	181
Vorteile	94	Nachteile	189
Delegationsprinzip	27, 111, 432	robust	189
Deserialisierung	367	Vorteile	188
Design to Test.....	6	JAX-RS	411
Double-Dispatch		JAX-WS	397
simuliert.....	164	Klassendiagramm	
Entscheidungsdokumentation	17	Abstrakte Fabrik	275
Entwurfsmuster.....	20, 23, 24, 310	Adapter	61
Arten.....	25	Befehl	138
Blaupausen	21	Beobachter	151
Erzeugungsmuster	52	Besucher	164
klassenbasiert	26	Broker	367
objektbasiert.....	27	Brücke	72
Strukturmuster.....	52, 60	Dekorierer	83
Verhaltensmuster	52	Fabrikmethode	266
Enumerationsmethode.....	191	Fassade	102
Erzeugungsmuster.....	52	Iterator	181
Abstrakte Fabrik	273	Kompositum	111
Fabrikmethode	264	Layers	318
Objektpool	297	Memento	193
Singleton	285	MVC	428
Extension Object.....	208, 221	Objektpool	300
Fabrik		Plug-in	351
Abstrakte~	109, 273	Proxy	123
Fabrikmethode	184, 229, 264, 284, 307,	Rolle	209
360		Schablonenmethode	224
		Service-Oriented Architecture	394

Strategie.....	231	Fassade	108
Vermittler.....	239	Iterator.....	189
Zustand	250	Kompositum	119
Kommunikationssystem		Layers	320
Software ISO/OSI-Modell	328	Memento	202
Komponente		MVC	442
Architektur-~.....	5	Objektpool.....	305
Komposition	43	Pipes and Filters	341
Kompositum.....	99, 109, 170, 177, 189, 430	Plug-in.....	359
Bewertung	118	Proxy	129
Klassendiagramm.....	111	Rolle	220
Nachteile	119	Schablonenmethode	229
Vorteile	118	Singleton	296
Layer Bridging.....	318, 321	SOA	396
Layers	316, 343	Strategie	236
Bewertung	320	Vermittler.....	246
Klassendiagramm.....	318	Zustand	258
Nachteile	320	Object Request Broker	387
Vorteile	320	Objektkomposition	45, 46
Lösungsbereich	25	Abhängigkeit von Abstraktion.....	46
Marshalling	367	Black-Box-Sicht der beteiligten Klassen.....	46
Memento	191	Kapselung nicht aufgebrochen.....	47
Bewertung	202	Objektpool.....	297
Klassendiagramm.....	193	Bewertung	305
Nachteile	202	Klassendiagramm	300
Vorteile	202	Nachteile	305
Microkernel	360	Vorteile	305
Model.....	423, 425	Pipes.....	333
Model 2	443	Pipes and Filters	100, 330, 331
Model-View-Controller	422	Bewertung	340
Beobachter.....	430	Nachteile	341
Bewertung	441	Vorteile	341
Klassendiagramm.....	428	Plug-in	273, 348
Kompositum	432	Bewertung	358
Nachteile	442	Klassendiagramm	351
Strategie.....	433	Nachteile	359
Vorteile	441	Vorteile	358
Muster		Problembereich.....	25
bewährt	22	Programmiere gegen Schnittstellen,	
erweiterbar	22	nicht gegen Implementierungen	39
Wiederverwendbarkeit.....	21	abstrakte Klasse.....	40
Nachteile		Bewertung	41
Abstrakte Fabrik	283	Nachteile	42
Adapter.....	69	Schnittstelle.....	40, 41
Befehl	146	Vorteile	41
Beobachter.....	159	Wiederverwendbarkeit	41
Besucher.....	176	Prototyp	
Broker.....	386	horizontal.....	13
Brücke	81	Realisierbarkeit	13
Dekorierer	95	vertikal.....	13
Fabrikmethode	272	Proxy	70, 81, 100, 122

Bewertung	128	Vorteile	236
Klassendiagramm.....	123	Strict Layering.....	318
Nachteile	129	Strukturmuster	52, 60
Vorteile	128	Adapter.....	60
Qualitäten eines Systems		Brücke.....	70
nicht funktional	6	Dekorierer	82
Ressourcenplanung.....	11	Fassade	101
Role Object Pattern.....	209	Kompositum	109
Rolle.....	203	Proxy	122
Analysemuster.....	221	Systemadministration	
Bewertung	219	Single System View	11
Klassendiagramm.....	210	technische Funktion	
Nachteile	220	Datenhaltung.....	25
Vorteile	219	Ein- und Ausgabe.....	25
Schablonenmethode	222, 237	Parallelität/IPC	25
Bewertung	229	Rechner-Rechner-Kommunikation	25
Klassendiagramm.....	224	Sicherheit	25
Nachteile	229	Typen von Architekturen.....	7
Vorteile	229	Unmarshalling.....	367
Schichtenmodell		User Interface	423
Client/Server	325	Vererbung	46
Drei-Schichten-Architektur	327	Verhaltensmuster.....	52
Einrechner-System.....	325	Befehl	136
Rechner.....	322	Beobachter	149
Zwei-Schichten-Architektur	325	Besucher	160
Schnittstelle vs. abstrakte Klasse	26	Iterator	178
Serialisierung	367	Memento	191
Service-Oriented Architecture	389	Rolle	203
Anwendungsservice	391	Schablonenmethode	222
Bewertung	395	Strategie	230
Klassendiagramm.....	394	Vermittler	238
Nachteile	396	Zustand	248
UDDI	405	Vermittler	109, 160, 238, 365, 388
Vorteile	396	Bewertung	245
Service-Registry	393	Klassendiagramm	239
Service-Verzeichnis	393	Nachteile	246
Single Responsibility Principle	205, 206	Vorteile	245
Singleton	285, 306	View	423, 427
Bewertung	296	Vorteile	
Klassendiagramm.....	287	Abstrakte Fabrik	282
Nachteile	296	Adapter	68
Sequenzdiagramm	288	Befehl	146
Vorteile	296	Beobachter	158
SOA	389	Besucher	175
Softwarearchitekt	13	Broker	385
Spezialisierung	26	Brücke	80
Strategie	100, 148, 229, 230, 258, 360, 430	Dekorierer	94
Bewertung	235	Fabrikmethode	271
Klassendiagramm.....	231	Fassade	107
Nachteile	236	Iterator	188
		Kompositum	118

Layers	320	Web Services Description Language	402
Memento	202	White-Box-Verhalten.....	43
MVC	441	White-Box-Wiederverwendung	45
Objektpool	305	Ziehe Objektkomposition der	
Pipes and Filters.....	341	Klassenvererbung vor	42
Plug-in	358	Bewertung	46
Proxy	128	Nachteile	47
Rolle	219	Vorteile	46
Schablonenmethode	229	Zustand.....	206, 221, 237, 248, 297
Singleton	296	Bewertung	257
SOA.....	396	Klassendiagramm	250
Strategie.....	236	Nachteile	258
Vermittler.....	245	Vorteile	257
Zustand	257		