



Erich
Gamma
Richard
Helm
Ralph
Johnson
John
Vlissides

Design Patterns

Entwurfsmuster als Elemente wieder-
verwendbarer objektorientierter Software

Design-Patterns-Katalog

Erzeugungsmuster (Creational Patterns)

Abstract Factory (Abstrakte Fabrik, siehe [Abschnitt 3.1](#))

Bereitstellung einer Schnittstelle zum Erzeugen verwandter oder voneinander abhängiger Objektfamilien ohne die Benennung ihrer konkreten Klassen.

Builder (Erbauer, siehe [Abschnitt 3.2](#))

Getrennte Handhabung der Erzeugungs- und Darstellungsmechanismen komplexer Objekte zwecks Generierung verschiedener Repräsentationen in einem einzigen Erzeugungsprozess.

Factory Method (Fabrikmethode, siehe [Abschnitt 3.3](#))

Definition einer Schnittstelle zur Objekterzeugung, wobei die Bestimmung der zu instanziierenden Klasse den Unterklassen überlassen bleibt. Das Design Pattern *Factory Method (Fabrikmethode)* gestattet einer Klasse, die Instanziierung an Unterklassen zu delegieren.

Prototype (Prototyp, siehe [Abschnitt 3.4](#))

Spezifikation der zu erzeugenden Objekttypen mittels einer prototypischen Instanz und Erzeugung neuer Objekte durch Kopieren dieses Prototyps.

Singleton (Singleton, siehe [Abschnitt 3.5](#))

Sicherstellung der Existenz nur einer einzigen Klasseninstanz sowie Bereitstellung eines globalen Zugriffspunkts für diese Instanz.

Strukturmuster (Structural Patterns)

Adapter (Adapter, siehe [Abschnitt 4.1](#))

Anpassung der Schnittstelle einer Klasse an ein anderes von den Clients erwartetes Interface. Das Design Pattern *Adapter (Adapter)* ermöglicht die Zusammenarbeit von Klassen, die ansonsten aufgrund der Inkompatibilität ihrer Schnittstellen nicht dazu in der Lage wären.

Bridge (Brücke, siehe [Abschnitt 4.2](#))

Entkopplung einer Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.

Composite (Kompositum, siehe [Abschnitt 4.3](#))

Komposition von Objekten in Baumstrukturen zur Abbildung von Teil-Ganzes-Hierarchien. Das Design Pattern *Composite (Kompositum)* gestattet den Clients einen einheitlichen Umgang sowohl mit individuellen Objekten als auch mit Objektkompositionen.

Decorator (Dekorierer, siehe [Abschnitt 4.4](#))

Dynamische Erweiterung der Funktionalität eines Objekts. *Decorator*-Objekte stellen hinsichtlich der Ergänzung einer Klasse um weitere Zuständigkeiten eine flexible Alternative zur Unterklassenbildung dar.

Facade (Fassade, siehe [Abschnitt 4.5](#))

Bereitstellung einer einheitlichen Schnittstelle zu einem Schnittstellensatz in einem Subsystem. Das Design Pattern *Facade (Fassade)* definiert eine Schnittstelle höherer Ebene, die die Nutzung des Subsystems vereinfacht.

Flyweight (Fliegengewicht, siehe [Abschnitt 4.6](#))

Gemeinsame Nutzung feingranularer Objekte, um sie auch in großer Anzahl effizient nutzen zu können.

Proxy (Proxy, siehe [Abschnitt 4.7](#))

Bereitstellung eines vorgelagerten Stellvertreterobjekts bzw. eines Platzhalters zwecks Zugriffssteuerung eines Objekts.

Verhaltensmuster (Behavioral Patterns)

Chain of Responsibility (Zuständigkeitskette, siehe [Abschnitt 5.1](#))

Unterbindung der Kopplung eines Request-Auslösers mit seinem Empfänger, indem mehr als ein Objekt in die Lage versetzt wird, den Request zu bearbeiten. Die empfangenden Objekte werden miteinander verkettet und der Request wird dann so lange entlang dieser Kette weitergeleitet, bis er von einem Objekt angenommen und abgearbeitet wird.

Command (Befehl, siehe [Abschnitt 5.2](#))

Kapselung eines Requests als Objekt, um so die Parametrisierung von Clients mit verschiedenen Requests, Warteschlangen- oder Logging-Operationen sowie das Rückgängigmachen von Operationen zu ermöglichen.

Interpreter (Interpreter, siehe [Abschnitt 5.3](#))

Definition einer Repräsentation für die Grammatik einer gegebenen Sprache und Bereitstellung eines Interpreters, der sie nutzt, um in dieser Sprache verfasste Sätze zu interpretieren.

Iterator (Iterator, siehe [Abschnitt 5.4](#))

Bereitstellung eines sequenziellen Zugriffs auf die Elemente eines aggregierten Objekts, ohne dessen zugrunde liegende Struktur offenzulegen.

Mediator (Vermittler, siehe [Abschnitt 5.5](#))

Definition eines Objekts, das die Interaktionsweise eines Objektsatzes in sich kapselt. Das Design Pattern *Mediator (Vermittler)* begünstigt lose Kopplungen, indem es die explizite Referenzierung der Objekte untereinander unterbindet und so eine individuelle Steuerung ihrer Interaktionen ermöglicht.

Memento (Memento, siehe [Abschnitt 5.6](#))

Erfassung und Externalisierung des internen Zustands eines Objekts, ohne dessen Kapselung zu beeinträchtigen, so dass es später wieder in diesen Zustand zurückversetzt werden kann.

Observer (Beobachter, siehe [Abschnitt 5.7](#))

Definition einer 1-zu-n-Abhängigkeit zwischen Objekten, damit im Fall einer Zustandsänderung eines Objekts alle davon abhängigen Objekte entsprechend benachrichtigt und automatisch aktualisiert werden.

State (Zustand, siehe [Abschnitt 5.8](#))

Anpassung der Verhaltensweise eines Objekts im Fall einer internen Zustandsänderung, so dass es den Anschein hat, als hätte es seine Klasse gewechselt.

Strategy (Strategie, siehe [Abschnitt 5.9](#))

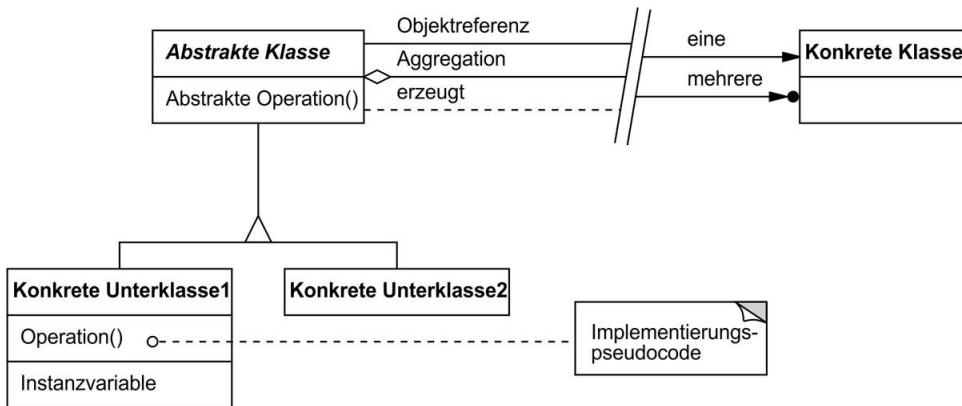
Definition einer Familie von einzeln gekapselten, austauschbaren Algorithmen. Das Design Pattern *Strategy (Strategie)* ermöglicht eine variable und von den Clients unabhängige Nutzung des Algorithmus.

Template Method (Schablonenmethode, siehe [Abschnitt 5.10](#))

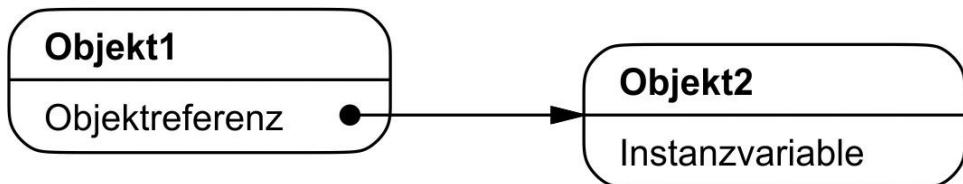
Definition der Grundstruktur eines Algorithmus in einer Operation sowie Delegation einiger Ablaufschritte an Unterklassen. Das Design Pattern *Template Method (Schablonenmethode)* ermöglicht den Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne dessen grundlegende Struktur zu verändern.

Visitor (Besucher, siehe [Abschnitt 5.11](#))

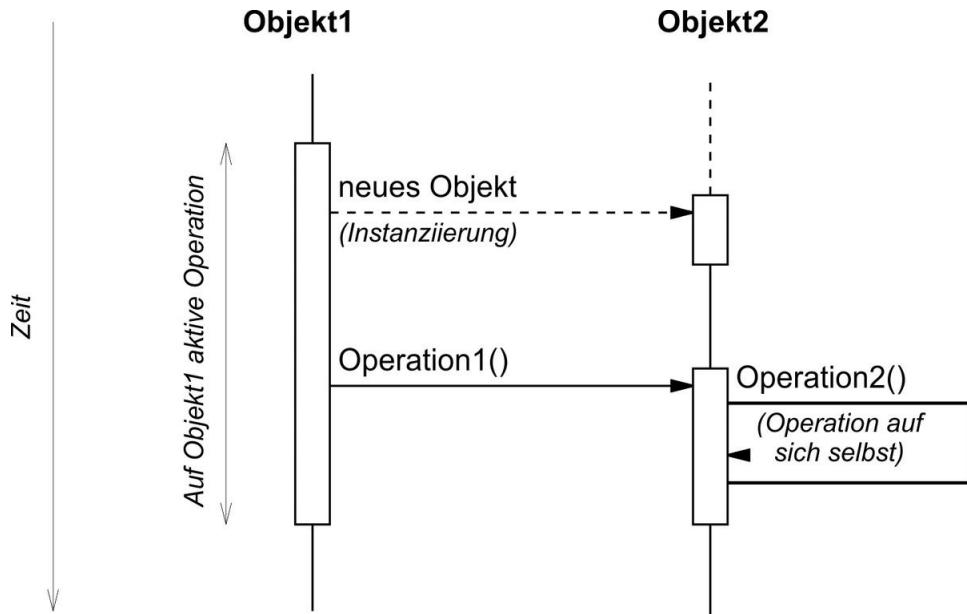
Darstellung einer auf die Elemente einer Objektstruktur auszuführenden Operation. Das Design Pattern *Visitor (Besucher)* ermöglicht die Definition einer neuen Operation, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.



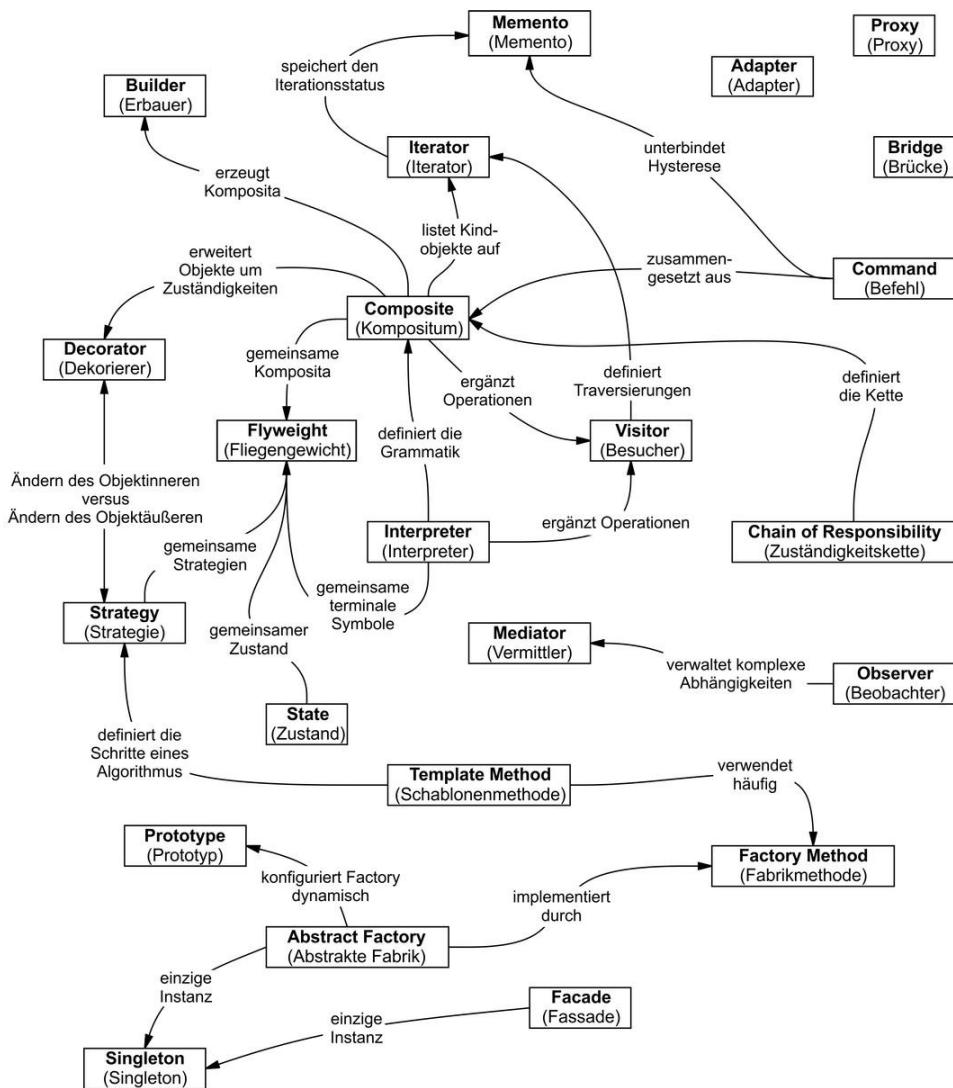
Notation für Klassendiagramme



Notation für Objektdiagramme



Notation für Interaktionsdiagramme



Beziehungen der Design Patterns

Inhaltsverzeichnis

[Impressum](#)

[Vorwort](#)

[Geleitwort von Grady Booch](#)

[Einleitung](#)

[Kapitel 1: Einführung](#)

[1.1 Was ist ein Design Pattern?](#)

[1.2 Design Patterns im Smalltalk MVC](#)

[1.3 Beschreibung der Design Patterns](#)

[1.4 Der Design-Patterns-Katalog](#)

[1.5 Aufbau des Katalogs](#)

[1.6 Die Anwendung von Design Patterns zur Behebung von Designproblemen](#)

[1.6.1 Passende Objekte finden](#)

[1.6.2 Objektgranularität bestimmen](#)

[1.6.3 Objektschnittstellen spezifizieren](#)

[1.6.4 Objektimplementierungen spezifizieren](#)

[1.6.5 Wiederverwendungsmechanismen einsetzen](#)

[1.6.6 Strukturen der Laufzeit und beim Komplizieren abstimmen](#)

[1.6.7 Designänderungen berücksichtigen](#)

[1.7 Auswahl eines Design Patterns](#)

[1.8 Anwendung eines Design Patterns](#)

[Kapitel 2: Fallstudie: Erstellung eines Texteditors](#)

[2.1 Designprobleme](#)

[2.2 Dokumentstruktur](#)

[2.2.1 Rekursive Komposition](#)

[2.2.2 Glyphen](#)

[2.2.3 Das Design Pattern Composite \(Kompositum\)](#)

[2.3 Formatierung](#)

[2.3.1 Kapselung des Formatierungsalgorithmus](#)

[2.3.2 Die Unterklassen Compositor und Composition](#)

[2.3.3 Das Design Pattern Strategy \(Strategie\)](#)

[2.4 Ausgestaltung der Benutzeroberfläche](#)

[2.4.1 Durchsichtige Umhüllung \(Transparent Enclosure\)](#)

[2.4.2 Die Unterklasse MonoGlyph](#)

[2.4.3 Das Design Pattern Decorator \(Dekorierer\)](#)

[2.5 Unterstützung mehrerer Look-and-Feel-Standards](#)

[2.5.1 Abstrahierung der Objekterzeugung](#)

[2.5.2 Factories und Produktklassen](#)

2.5.3 Das Design Pattern Abstract Factory (Abstrakte Fabrik)

2.6 Unterstützung mehrerer Fenstersysteme

2.6.1 Eignung des Design Patterns Abstract Factory (Abstrakte Fabrik)

2.6.2 Kapselung von Implementierungsabhängigkeiten

2.6.3 Die Klassenhierarchien Window und WindowImp

2.6.4 Das Design Pattern Bridge (Brücke)

2.7 Userseitige Operationen

2.7.1 Kapselung eines Requests

2.7.2 Die Command-Klasse und ihre Unterklassen

2.7.3 Die Funktion Undo (Rückgängig)

2.7.4 Befehlshistorie

2.7.5 Das Design Pattern Command (Befehl)

2.8 Rechtschreibprüfung und Silbentrennung

2.8.1 Zugriff auf verteilte Informationen

2.8.2 Kapselung von Zugriff und Traversierung

2.8.3 Die Iterator-Klasse und ihre Unterklassen

2.8.4 Das Design Pattern Iterator (Iterator)

2.8.5 Traversierung kontra Traversierungsaktionen

2.8.6 Kapselung der Analyse

2.8.7 Die Visitor-Klasse und ihre Unterklassen

2.8.8 Das Design Pattern Visitor (Besucher)

2.9 Zusammenfassung

Design-Patterns-Katalog

Kapitel 3: Erzeugungsmuster (Creational Patterns)

3.1 Abstract Factory (Abstrakte Fabrik)

3.2 Builder (Erbauer)

3.3 Factory Method (Fabrikmethode)

3.4 Prototype (Prototyp)

3.5 Singleton (Singleton)

3.6 Weitere Erläuterungen zu den Erzeugungsmustern

Kapitel 4: Strukturmuster (Structural Patterns)

4.1 Adapter (Adapter)

4.2 Bridge (Brücke)

4.3 Composite (Kompositum)

4.4 Decorator (Dekorierer)

4.5 Facade (Fassade)

4.6 Flyweight (Fliegengewicht)

4.7 Proxy (Proxy)

4.8 Weitere Erläuterungen zu den Strukturmustern

4.8.1 Adapter (Adapter, siehe Abschnitt 4.1) kontra Bridge (Brücke, siehe

Abschnitt 4.2)

4.8.2 Composite (Kompositum, siehe Abschnitt 4.3) kontra Decorator
(Dekorierer, siehe Abschnitt 4.4) kontra Proxy (Proxy, siehe Abschnitt 4.7)

Kapitel 5: Verhaltensmuster (Behavioral Patterns)

5.1 Chain of Responsibility (Zuständigkeitskette)

5.2 Command (Befehl)

5.3 Interpreter (Interpreter)

5.4 Iterator (Iterator)

5.5 Mediator (Vermittler)

5.6 Memento (Memento)

5.7 Observer (Beobachter)

5.8 State (Zustand)

5.9 Strategy (Strategie)

5.10 Template Method (Schablonenmethode)

5.11 Visitor (Besucher)

5.12 Weitere Erläuterungen zu den Verhaltensmustern

 5.12.1 Variieren der Kapselung

 5.12.2 Objekte als Argumente

 5.12.3 Kommunikation: Kapseln oder Verteilen?

 5.12.4 Absender und Empfänger entkoppeln

 5.12.5 Zusammenfassung

Kapitel 6: Schlusswort der Autoren

6.1 Was kann man von Design Patterns erwarten?

 6.1.1 Ein gemeinsames Designvokabular

 6.1.2 Eine Dokumentations- und Lernhilfe

 6.1.3 Eine Ergänzung zu existierenden Methoden

 6.1.4 Zielsetzungen für Refactorings

6.2 Eine kleine Kataloggeschichte

6.3 Die Pattern-Gemeinde

 6.3.1 Christopher Alexanders »Muster-Sprache«

 6.3.2 Patterns in Softwaresystemen

6.4 Eine Einladung

6.5 Ein abschließender Gedanke

Anhang A: Glossar

Anhang B: Notationshinweise

 B.1 Klassendiagramme

 B.2 Objektdiagramme

 B.3 Interaktionsdiagramme

Anhang C: Fundamentale Klassen

 C.1 Die Klasse List

[C.2 Iterator](#)

[C.3 ListIterator](#)

[C.4 Point](#)

[C.5 Rect](#)

[Anhang D: Quellenverzeichnis](#)

Design Patterns

**Entwurfsmuster als Elemente wiederverwendbarer
objektorientierter Software**

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Übersetzung aus dem Amerikanischen von Maren Feilen und Knut Lorenzen



Impressum

Bibliografische Information der Deutschen Nationalbibliothek

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

ISBN 978-3-8266-9904-7

1. Auflage 2015

www.mitp.de

E-Mail: mitp-verlag@sigloch.de

Telefon: +49 7953 / 7189 - 079

Telefax: +49 7953 / 7189 - 082

Authorized translation from the English language edition, entitled DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, 1st Edition, 0201633612 by GAMMA, ERICH; HELM, RICHARD; JOHNSON, RALPH; VLISSIDES, JOHN, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 1995 by Addison-Wesley.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. GERMAN language edition published by mitp, an imprint of Verlagsgruppe Hüthig Jehle Rehm GmbH, Copyright © 2015.

© 2015 mitp-Verlags GmbH & Co. KG

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

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Lektorat: Sabine Schulz

Sprachkorrektorat: Maren Feilen, Knut Lorenzen

electronic publication: III-satz, Husby, www.drei-satz.de

Coverbild: © art_of_sun @ Fotolia.com

Dieses Ebook verwendet das ePub-Format und ist optimiert für die Nutzung mit dem iBooks-reader auf dem iPad von Apple. Bei der Verwendung anderer Reader kann es zu Darstellungsproblemen kommen.

Der Verlag räumt Ihnen mit dem Kauf des ebooks das Recht ein, die Inhalte im Rahmen des geltenden Urheberrechts zu nutzen. Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und Einspeicherung und Verarbeitung in elektronischen Systemen.

Der Verlag schützt seine ebooks vor Missbrauch des Urheberrechts durch ein digitales Rechtemanagement. Bei Kauf im Webshop des Verlages werden die ebooks mit einem nicht sichtbaren digitalen Wasserzeichen individuell pro Nutzer signiert.

Bei Kauf in anderen ebook-Webshops erfolgt die Signatur durch die Shopbetreiber. Angaben zu diesem DRM finden Sie auf den Seiten der jeweiligen Anbieter.

Für Karin
– E.G.

Für Sylvie
– R.H.

Für Faith
– R.J.

Für Dru Ann und Matthew
Josua 24:15b
– J.V.

Vorwort

Dieses Buch wird Ihnen keine Einführung in die objektorientierte Programmierung oder das Anwendungsdesign als solches bieten – denn immerhin gibt es bereits zahlreiche sehr gute Nachschlagewerke, die diese Themen ausführlich behandeln. Die Inhalte dieses Buches setzen vielmehr voraus, dass Sie mindestens eine objektorientierte Programmiersprache relativ gut beherrschen und über grundlegende Erfahrungen im objektorientierten Design verfügen, so dass Sie nicht zum nächstbesten Fachlexikon greifen müssen, wenn Begriffe wie »Typen«, »Polymorphie« oder »Schnittstellenvererbung« statt »Implementierungsvererbung« zur Sprache kommen.

Andererseits handelt es sich bei diesem Buch aber auch nicht um eine technische Abhandlung für fortgeschrittene Entwickler. Es befasst sich mit **Design Patterns**, sprich Entwurfsmustern, die einfache und elegante Lösungen für spezifische Problemstellungen im objektorientierten Softwaredesign anbieten. Design Patterns repräsentieren im Grunde genommen Problemlösungen, die sich in der Praxis als sinnvoll und hilfreich erwiesen haben. Sie beschreiben Designs, die man normalerweise nicht ohne Weiteres in dieser Art entwickelt. Die Patterns basieren auf zahllosen Design- und Programmrevisionen, die Softwareentwickler auf der Suche nach besseren Wiederverwendungsmöglichkeiten und größerer Flexibilität im Laufe der Zeit erarbeitet haben – und sie stellen diese Lösungen in einer konzentrierten und einfach anzuwendenden Form zur Verfügung.

Die in dem hier vorgestellten Katalog enthaltenen Patterns erfordern weder außergewöhnliche programmiersprachliche Funktionen noch irgendwelche raffinierten Programmiertricks, mit denen man Freunde und Vorgesetzte beeindrucken kann. Sie können allesamt in den standardmäßigen objektorientierten Programmiersprachen implementiert werden, sind allerdings mit etwas mehr Aufwand verbunden als Ad-hoc-Lösungen – dieser Mehraufwand wird jedoch immer mit einer deutlich besseren Wiederverwendbarkeit und höherer Flexibilität belohnt.

Wer die Arbeitsweise der Design Patterns erst einmal verstanden hat und diesbezüglich zu einem »Aha!«-Erlebnis gelangt ist, wird fortan völlig anders über das objektorientierte Design denken. Die Patterns vermitteln Ihnen Einsichten, die es Ihnen ermöglichen, Ihre Designs flexibler, modularer, wiederverwendbarer und verständlicher zu gestalten – und genau das zählt schließlich zu den Hauptgründen,

sich der objektorientierten Programmierung zuzuwenden, richtig?

An dieser Stelle sei aber auch gleich angemerkt: Machen Sie sich keine Gedanken, wenn Sie den Inhalt dieses Buches nicht schon bei der ersten Lektüre vollumfänglich verstehen. Selbst wir Autoren haben nicht immer alle Zusammenhänge dessen, was wir da zu Papier gebracht haben, sofort zu 100 % verstanden! Dieses Buch ist kein »Schmöker«, den man einmal liest und dann ins Bücherregal stellt – es ist im wahrsten Sinne des Wortes ein Nachschlagewerk, von dem wir hoffen, dass Sie es zukünftig immer wieder aufschlagen werden, um neue Erkenntnisse und Anregungen für Ihre eigenen Designs zu gewinnen.

Die Fertigstellung dieses Buches hat sich recht lange hingezogen. Es hat sozusagen vier Länder »bereist«, die Hochzeiten von drei seiner Autoren und sogar die Geburten von zwei (nicht miteinander verwandten) Sprösslingen erlebt. Und es waren sehr viele Menschen an der Entstehung dieses Werkes beteiligt. Besonderer Dank gebührt in diesem Zusammenhang Bruce Anderson, Kent Beck und André Weinand für ihre Inspiration und ihre hilfreichen Ratschläge. Unser Dank richtet sich auch an all jene, die unsere Manuskriptentwürfe gelesen und kritisch kommentiert und bewertet haben: Roger Bielefeld, Grady Booch, Tom Cargill, Mashall Cline, Ralph Hyre, Brian Kernighan, Thomas Laliberty, Mark Lorenz, Arthur Riel, Doug Schmidt, Clovis Tondo, Steve Vinoski und Rebecca Wirfs-Brock. Weiterhin danken wir dem Team von Addison-Wesley für seine Hilfe und Geduld: Kate Habib, Tiffany Moore, Lisa Raffaele, Pradeepa Siva und John Wait. Ein ganz besonderer Dank geht an Carl Kessler, Danny Sabbah und Mark Wegman von IBM Research für ihre unermüdliche Unterstützung für dieses Projekt.

Selbstverständlich danken wir auch all den Menschen, die uns per Internet und auf anderen Wegen ihre Erfahrungen mit und Meinungen zu den verschiedenen Patterns mitgeteilt haben, uns ermutigt haben und uns wissen ließen, dass unsere Arbeit die Mühe wert war. Stellvertretend für all diese Menschen bedanken wir uns an dieser Stelle bei Jon Avotins, Steve Berczuk, Julian Berdych, Matthias Bohlen, John Brant, Allan Clarke, Paul Chisholm, Jens Coldewey, Dave Collins, Jim Coplien, Don Dwiggins, Gabriele Elia, Doug Felt, Brian Foote, Denis Fortin, Ward Harold, Hermann Hueni, Nayeem Islam, Bikramjit Kalra, Paul Keefer, Thomas Kofler, Doug Lea, Dan LaLiberte, James Long, Ann Louise Luu, Pundi Madhavan, Brian Marick, Robert Martin, Dave McComb, Carl McConnell, Christine Mingins, Hanspeter Mössenböck, Eric Newton, Marianne Ozkan, Roxsan Payette, Larry Podmolik, George Radin, Sita Ramakrishnan, Russ Ramirez, Alexander Ran, Dirk Riehle, Bryan Rosengrub, Aamod Sane, Duri Schmidt, Robert Seidl, Xin Shu und Bill Walker.

Wir betrachten den in diesem Buch vorgestellten Design-Pattern-Katalog keineswegs als vollständig und unveränderlich – er repräsentiert vielmehr eine Momentaufnahme unserer Überlegungen und Erwägungen in Bezug auf gutes Softwaredesign. Wir freuen uns über Meinungen und Kommentare jeder Art, sei es Kritik oder Lob zu unseren Beispielen, Hinweise auf Referenzen und Praxisbeispiele, die wir übersehen haben, oder Vorschläge für weitere Patterns, die wir ebenfalls in den Katalog hätten aufnehmen sollen. Ihre diesbezüglichen Anfragen erreichen uns per E-Mail an design-patterns@cs.uiuc.edu. Die in diesem Buch verwendeten Codebeispiele stehen unter www.mitp.de/9700 zum Download zur Verfügung oder können alternativ per E-Mail mit dem Inhalt »send design pattern source« an design-patterns-source@cs.uiuc.edu bei uns angefordert werden.

E.G., Mountain View, California

R.H., Montreal, Quebec

R.J., Urbana, Illinois

J.V., Hawthorne, New York

August 1994

Geleitwort von Grady Booch

Alle gut strukturierten objektorientierten Architekturen basieren auf Mustern, auf Englisch »Patterns« genannt. Ich persönlich bemesse die Qualität eines objektorientierten Systems im Prinzip daran, wie viel Sorgfalt die Entwickler der grundsätzlich möglichen Zusammenarbeit der einzelnen Objekte gewidmet haben. Konzentriert man sich bei der Systementwicklung vorrangig auf diese Mechanismen, dann gelangt man letztendlich zu Architekturen, die kompakter, schlichter und erheblich besser zu verstehen sind, als wenn diese Muster außer Acht gelassen werden.

Die Bedeutung von Patterns für die Erstellung komplexer Systeme wurde in anderen Disziplinen schon längst erkannt. Allen voran der Architekturtheoretiker Christopher Alexander und seine Mitarbeiter gehörten zu den Ersten, die die Etablierung und Instrumentalisierung einer »Pattern Language« (dt. »Mustersprache«) für die Konstruktion von Gebäuden und Städten vorschlugen. Seine Ideen und die Beiträge weiterer seiner Mitstreiter haben inzwischen auch in der objektorientierten Software Community Fuß gefasst. Kurzum: Das Konzept der Design Patterns verkörpert in der Programmentwicklung das Schlüsselement, um sich die Erfahrungen und das Wissen anderer talentierter (Software-)Architekten zunutze zu machen.

Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides erläutern in diesem Buch die Grundsätze der Design Patterns und stellen einen Katalog solcher Muster vor. Damit leistet dieses Nachschlagewerk zwei wichtige Beiträge: Zum einen veranschaulicht es nachdrücklich die Rolle, die Patterns bei der Entwicklung komplexer Systeme einnehmen können. Und zum anderen dient es als äußerst praktische Referenz für eine Sammlung wohldurchdachter Patterns, die praktizierende Entwickler bei der Gestaltung ihrer eigenen spezifischen Anwendungen nutzen können.

Ich bin stolz darauf, dass mir die Ehre zuteilwurde, mit einigen der Autoren dieses Buches unmittelbar an ihren architektonischen Designbemühungen zusammenarbeiten zu dürfen. Ich habe viel von ihnen gelernt und bin überzeugt, dass es Ihnen bei der Lektüre dieses Buches ebenso ergehen wird.

Grady Booch

Chief Scientist, Rational Software Corporation

Einleitung

Dieses Buch gliedert sich im Wesentlichen in zwei Teile. Der erste Teil umfasst die [Kapitel 1](#) und [2](#) und beschreibt, was Design Patterns sind und wie sie Ihnen bei der Entwicklung objektorientierter Software von Nutzen sein können. Zudem wird ihre praktische Anwendung anhand einer ausführlichen Fallstudie demonstriert. Im zweiten Teil des Buches (die [Kapitel 3](#), [4](#) und [5](#)) wird dann der eigentliche Katalog der einzelnen Design Patterns präsentiert.

Dieser Katalog macht den größten Teil des Buches aus. Die zugehörigen [Kapitel 3](#), [4](#) und [5](#) unterteilen die Design Patterns in drei Kategorien: *Erzeugungsmuster* (*Creational Patterns*), *Strukturmuster* (*Structural Patterns*) und *Verhaltensmuster* (*Behavioral Patterns*).

Sie können den Katalog auf verschiedene Art und Weise nutzen: Entweder lesen Sie ihn von Anfang bis Ende durch oder wechseln von Pattern zu Pattern. Oder aber Sie konzentrieren sich jeweils auf ein Kapitel, um zu ergründen, inwiefern sich die Patterns ein und derselben Kategorie voneinander unterscheiden.

Die in den einzelnen Kapiteln angegebenen Querverweise zwischen den Patterns bilden einen roten Faden durch den Katalog. Auf diese Weise können Sie sich bequem einen Überblick darüber verschaffen, in welcher Beziehung die Patterns zueinander stehen, wie sie kombiniert werden können und welche Design Patterns gut zusammenarbeiten. Die verwandtschaftlichen Beziehungen der einzelnen Design Patterns sind in [Abbildung 1.2](#) in einer Übersicht dargestellt.

Wenn Sie möchten, können Sie beim Lesen des Katalogs aber auch problemorientiert vorgehen. So könnten Sie z. B. gleich mit dem [Abschnitt 1.6](#) beginnen, um sich über diverse allgemeine Probleme hinsichtlich des Designs wiederverwendbarer objektorientierter Software zu informieren, und anschließend die Ausführungen zu den Design Patterns lesen, die für die jeweiligen Problemstellungen geeignet sind. Oder Sie arbeiten zuerst den gesamten Katalog durch und gehen danach problemorientiert vor, um die passenden Patterns in Ihren Projekten anzuwenden.

Wenn Sie auf dem Gebiet der objektorientierten Entwicklung noch nicht so viel Erfahrung haben, empfiehlt es sich, mit den einfachsten und geläufigsten Patterns anzufangen:

- *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#))
- *Adapter* (*Adapter*, siehe [Abschnitt 4.1](#))
- *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#))
- *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#))
- *Factory Method* (*Fabrikmethode*, siehe [Abschnitt 3.3](#))
- *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#))
- *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#))
- *Template Method* (*Schablonenmethode*, siehe [Abschnitt 5.10](#))

Kaum ein objektorientiertes System kommt ohne wenigstens ein paar Patterns aus – große Systeme nutzen sogar nahezu alle der hier genannten Exemplare. Diese Auswahl wird Ihnen helfen, die Design Patterns im Besonderen und das Konzept eines guten objektorientierten Designs im Allgemeinen wirklich zu verstehen.

Kapitel 1: Einführung

Das Entwickeln bzw. Entwerfen objektorientierter Software ist zweifellos kein leichtes Unterfangen – und das Designen *wiederverwendbarer* objektorientierter Software gestaltet sich sogar noch anspruchsvoller und komplexer: Neben der Bestimmung der relevanten Objekte und deren Abstrahierung zu Klassen in geeigneter Granularität müssen außerdem passende Schnittstellen und Vererbungshierarchien definiert sowie die zentralen Beziehungen zwischen den einzelnen Klassen bestimmt werden. Darüber hinaus sollte sich das Softwaredesign natürlich einerseits speziell an den jeweiligen Erfordernissen orientieren, andererseits aber gleichermaßen eine hinreichende Universalität aufweisen, um auch für zukünftige Problemstellungen und Anforderungen gewappnet zu sein. Und Designrevisionen sollten nach Möglichkeit ebenfalls vermieden oder zumindest auf ein Minimum reduziert werden.

Erfahrene objektorientierte Programmierer sind sich im Allgemeinen darüber im Klaren, dass es sehr schwierig, wenn nicht gar unmöglich ist, ein wiederverwendbares, flexibles Design gleich von Anfang an »richtig« hinzubekommen. Aus diesem Grund greifen sie häufig mehrfach auf ihre früheren Entwürfe zurück und versuchen zunächst, weitere Modifikationen zu implementieren, bevor sie ein Softwaredesign endgültig als »abgeschlossen« betrachten.

Das Erlernen und Verinnerlichen der maßgeblichen Aspekte, die ein gutes objektorientiertes Design ausmachen, erfordert viel Zeit und Geduld. Folglich fällt es erfahrenen objektorientierten Programmierern in der Regel leichter, gute Designs zu entwerfen, als weniger geübten Entwicklern, die sich nicht selten von den zahlreichen Designoptionen überfordert fühlen und daher oft doch lieber die nicht objektorientierten Techniken anwenden, die sie früher schon eingesetzt haben. Erfahrenen Programmierern ist also offenbar spezielles Wissen zu eigen, über das unerfahrene Entwicklern (noch) nicht verfügen. Doch worum genau handelt es sich dabei?

Nun, versierten Softwaredesignern ist beispielsweise bewusst, dass sie *nicht* ständig versuchen müssen, für jedes Problem eine völlig neue Lösung zu entwickeln. Stattdessen machen sie sich Lösungsmöglichkeiten zunutze, die sich nach ihrer persönlichen Erfahrung bereits bewährt haben – mit anderen Worten: Haben sie erst einmal gute Lösungsansätze gefunden, dann »recyclen« sie diese und wenden sie

immer wieder an. Dieser Erfahrungsvorsprung zeichnet die Experten im Bereich des Softwaredesigns aus – und erklärt auch, warum viele objektorientierte Systeme häufig wiederkehrende Klassenmuster und kommunizierende Objekte enthalten. Solche Muster (engl. *Patterns*) beheben bestimmte Designprobleme und gestalten objektorientierte Softwareentwürfe nicht nur flexibler und eleganter, sondern gestatten letztlich überhaupt erst ihre Wiederverwendbarkeit. Sie ermöglichen den Entwicklern, neue Designs auf gelungenen Entwürfen aufzusetzen und darauf aufzubauen. Kurz gesagt: Softwaredesigner, die mit Patterns vertraut sind, können diese unmittelbar auf die jeweils vorliegenden Problemstellungen anwenden, ohne erst »das Rad neu erfinden« zu müssen.

Ein vergleichbares Prinzip findet auch in anderen kreativen Prozessen Anwendung. So erarbeiten beispielsweise Roman- und Bühnenautoren die Handlungen ihrer Geschichten und Theaterstücke nur höchst selten von Grund auf – vielmehr bedienen sie sich in den meisten Fällen ebenfalls eines bewährten »Musters« bzw. Leitmotivs wie etwa »tragische Heldenfigur« (Macbeth, Hamlet etc.) oder »romantische Erzählung« (zahllose Liebesromane). In ähnlicher Weise setzen auch die objektorientierten Programmierer Entwurfsmuster (engl. *Design Patterns*) wie etwa »Zustände werden durch Objekte repräsentiert« oder »Objekte werden zwecks einfacher Ergänzung/Entfernung von Eigenschaften dekoriert« ein. Und steht das Muster erst einmal fest, ergeben sich viele weitere Designentscheidungen ganz automatisch.

Als Entwickler wissen wir alle den Wert der praktischen Designerfahrung zu schätzen. Haben Sie beim Betrachten eines Softwaredesigns nicht auch schon mal ein *Déjà-vu*-Erlebnis gehabt – das Gefühl, ein bestimmtes Problem in der Vergangenheit bereits bewältigt zu haben, ohne dass Sie sich daran erinnern könnten, wie genau oder in welchem Zusammenhang? Wenn Ihnen die exakte Problemstellung bzw. der seinerzeit eingeschlagene Lösungsweg wieder einfallen würde, könnten Sie auf diese Erfahrung zurückgreifen, statt ganz von vorn anfangen zu müssen. Leider werden die im Laufe eines objektorientierten Designprozesses gewonnenen Erkenntnisse in der Regel jedoch nicht so umfassend protokolliert, dass auch Dritte davon profitieren könnten.

Deshalb lautet die Zielsetzung dieses Buches, besagten Erkenntnisgewinn anhand von sogenannten **Design Patterns** (auch **Entwurfsmuster** oder kurz **Patterns** genannt) zu dokumentieren, damit er für jedermann zugänglich und effizient nutzbar wird. Zu diesem Zweck haben wir die wichtigsten Patterns in Katalogform zusammengefasst, wobei jedes Muster systematisch je ein bedeutsames wiederkehrendes Design benennt, beschreibt und evaluiert.

Design Patterns erleichtern nicht nur die Wiederverwendung erfolgreicher Softwaredesigns und -architekturen, sondern bieten Entwicklern zudem auch einen besseren, ungehinderten Zugang zu bewährten Techniken. Außerdem dienen sie als Entscheidungshilfe bei der Wahl möglicher Designalternativen zwecks Gewährleistung der Wiederverwendbarkeit eines Systems und verhindern gleichzeitig Alternativen, die einer Wiederverwendung entgegenstehen. Im Übrigen können Patterns ebenfalls dazu beitragen, die Dokumentation und Wartung bereits existenter Systeme zu verbessern, indem sie explizite Spezifikationen für die Klassen- und Objektinteraktionen sowie deren Intention bereitstellen. Unterm Strich heißt das: Design Patterns unterstützen Programmierer dabei, ihre Softwaredesigns schneller »richtig« hinzubekommen.

Keins der in diesem Buch beschriebenen Patterns ist vollkommen neu oder unerprobt. Im Gegenteil: Es werden ausschließlich solche Design Patterns verwendet, die schon mehrfach in verschiedenen Systemen eingesetzt wurden. Die meisten von ihnen wurden allerdings noch nie zuvor dokumentiert, sondern entstammen entweder dem gemeinschaftlichen Fundus der objektorientierten Community oder erfolgreichen objektorientierten Systemen – also zwei Bereichen, die sich unerfahrenen Programmierern nicht so leicht erschließen. Aber auch wenn die vorgestellten Patterns nicht brandneu sind, werden sie doch in einer neuartigen, leicht verständlichen Art und Weise vorgestellt: in Form eines Design-Pattern-Katalogs, der einer einheitlichen Präsentationskonvention folgt.

Aufgrund des naturgemäß begrenzten Platzangebots in diesem Buch repräsentieren die hier erläuterten Muster lediglich einen Bruchteil des Pattern-Fundus, der echten Programmierexperten zur Verfügung steht. So werden beispielsweise keine Design Patterns für die nebenläufige, die verteilte oder die Echtzeitprogrammierung oder für spezifische Anwendungsdomänen verwendet. Ebenso wenig werden Sie in diesem Buch Informationen zur Entwicklung von Benutzeroberflächen, zum Schreiben von Gerätetreibern oder zum Einsatz objektorientierter Datenbanken vorfinden. Für all diese Aufgabenbereiche existieren wiederum spezielle Design Patterns – die es jedoch sicherlich ebenfalls wert wären, eigens katalogisiert zu werden.

1.1 Was ist ein Design Pattern?

Der US-amerikanische Architekturtheoretiker Christopher Alexander erklärt in seinem Buch »*Eine Muster-Sprache*« [Löcker Verlag, Wien, 1995, Seite x]: »Jedes Muster beschreibt zunächst ein in unserer Umwelt immer wieder auftretendes

Problem, beschreibt dann den Kern der Lösung dieses Problems, und zwar so daß man diese Lösung millionenfach anwenden kann, ohne sich je zu wiederholen.« Natürlich bezieht sich Alexander in diesem Fall auf Muster, die sich in Gebäuden und Stadtplanungsentwürfen wiederfinden, dennoch trifft seine Definition auch auf objektorientierte Design Patterns zu – mit dem Unterschied, dass die Lösungen hier nicht in Form von Wänden und Türen, sondern von Objekten und Schnittstellen ausgedrückt werden. Prinzipiell repräsentieren jedoch beide Musterspezies Problemlösungen für bestimmte Situationen in bestimmten Kontexten.

Ein Design Pattern umfasst im Wesentlichen vier maßgebliche Elemente:

1. Der kurze, meist aus ein oder zwei Wörtern bestehende **Pattern-Name** dient zur Referenzierung des jeweiligen Designproblems sowie der zugehörigen Lösungen und deren Auswirkungen. Durch die Benennung eines Design Patterns erweitern wir unser designbezogenes Vokabular in der Art, dass ein Arbeiten auf einer höheren Abstraktionsebene möglich wird – denn dieses Vokabular erleichtert die Dokumentation des Softwaredesigns und vor allem auch die Kommunikation mit Kollegen und Dritten, z. B. zur Erörterung der Vor- und Nachteile neuer Ideen und Vorschläge, in erheblichem Maße. Wirklich geeignete Namen zu finden, ist allerdings keine leichte Aufgabe – das zeigt auch die Tatsache, dass die Vergabe passender Bezeichnungen (im Englischen wie im Deutschen) für die in diesem Buch vorgestellten Design Patterns bei der Entwicklung unseres Katalogs eine der größten Herausforderungen überhaupt darstellte.
2. Die **Problemstellung** beschreibt die Situation, in der das Design Pattern anzuwenden ist. Sie erläutert die jeweils vorliegende Problematik sowie deren Kontext. Dabei kann es sich um spezifische Designprobleme handeln, wie z. B. in welcher Form Algorithmen als Objekte abgebildet werden sollten, aber auch um für unflexible Designs symptomatische Klassen- oder Objektstrukturen. Mitunter werden im Rahmen der Problembeschreibung außerdem bestimmte Bedingungen aufgeführt, die erfüllt sein müssen, damit der Einsatz des Design Patterns überhaupt sinnvoll ist.
3. Die **Lösung** berücksichtigt neben den konkreten designbildenden Elementen auch deren Beziehungen zueinander, ihre Zuständigkeiten sowie ihre Interaktionen. Sie definiert allerdings kein bestimmtes Design oder eine konkrete Implementierung, denn ein Pattern entspricht eher einer Art Schablone, die in vielen verschiedenen Situationen anwendbar ist: Es skizziert eine abstrakte Beschreibung eines Designproblems und zeigt auf, wie dieses durch eine generelle Anordnung von Elementen (in unserem Fall Klassen und

Objekten) bewältigt werden kann.

4. Die **Konsequenzen** bilden eine Übersicht der Auswirkungen und Kompromisse ab, die sich aus der Anwendung des Patterns ergeben. Leider bleiben sie, obwohl sie für die Bewertung möglicher Designalternativen und das Abwägen ihrer Vor- und Nachteile von entscheidender Bedeutung sind, in der Argumentation für eine Designentscheidung häufig unerwähnt.

In der Softwareentwicklung stehen die Konsequenzen eines Pattern-Einsatzes oftmals in direktem Zusammenhang mit dem Speicherplatzbedarf und den Ausführungszeiten, sie können aber ebenso gut auch Sprach- und andere Implementierungsaspekte betreffen. Und da die Wiederverwendbarkeit in der objektorientierten Programmierung eine wichtige Rolle spielt, schließt dies selbstverständlich auch den Einfluss des infrage stehenden Design Patterns auf die Flexibilität, Erweiterbarkeit und Portabilität des Systems mit ein. Generell gilt also: Die ausdrückliche, sachliche Erwägung der zu erwartenden Konsequenzen ist für die lückenlose Evaluierung eines Patterns unverzichtbar.

Die Interpretation des tatsächlichen Nutzwertes eines Design Patterns hängt im Endeffekt immer auch von der Perspektive des Betrachters ab: Während ein Pattern für den einen Entwickler besonders hilfreich erscheinen mag, kann es für einen anderen Programmierer bloß einen primitiven Baustein darstellen. Bei der Zusammenstellung des Design-Pattern-Katalogs haben wir uns vordergründig auf eine bestimmte Abstraktionsebene konzentriert. Die Patterns verkörpern keine Entwürfe für verkettete Listen oder Hashtabellen, die als separate Klassen programmiert und dann im Ist-Zustand wiederverwendet werden können. Ebenso wenig stellen sie komplexe, domänen spezifische Designs für eine komplette Anwendung oder ein Teilsystem dar. Vielmehr handelt es sich bei den hier beschriebenen Design Patterns um *Darstellungen kommunizierender Objekte und Klassen, die auf die Lösung eines allgemeinen Designproblems in einem speziellen Kontext zugeschnitten sind.*

Ein Design Pattern benennt, abstrahiert und identifiziert die Schlüsselaspekte einer allgemeinen Designstruktur, durch die es sich für die Entwicklung eines wiederverwendbaren objektorientierten Designs auszeichnet. Es identifiziert die beteiligten Klassen und Instanzen, deren Rollen und Interaktionen sowie die ihnen zugedachte Aufgabenverteilung. Jedes Pattern eignet sich für eine ganz bestimmte objektorientierte Designproblematik und besitzt spezielle Merkmale, die Aufschluss darüber geben, wann seine Anwendung zweckmäßig ist, ob es trotz möglicher anderer designbedingten Einschränkungen überhaupt eingesetzt werden kann und welche Konsequenzen und Vor- und Nachteile es mit sich bringt. Und weil die

Design Patterns schlussendlich natürlich implementiert werden müssen, ist zu jedem von ihnen auch ein Codebeispiel in C++ und (mitunter) Smalltalk angegeben, das eine mögliche Implementierungsvariante aufzeigt.

Auch wenn Patterns prinzipiell objektorientierte Designs beschreiben, basieren sie dennoch auf praktischen Lösungen, die in etablierten objektorientierten Programmiersprachen wie Smalltalk und C++ statt in prozeduralen (Pascal, C, Ada) oder dynamischen objektorientierten Sprachen (CLOS, Dylan, Self) implementiert sind. In diesem Buch haben wir uns aus rein pragmatischen Gründen für die Verwendung von Smalltalk und C++ entschieden, weil wir selbst täglich mit diesen Programmiersprachen arbeiten.

Die Wahl der Programmiersprache spielt insofern eine gewichtige Rolle, als sie den eigenen Blickwinkel beeinflusst. Die in diesem Buch vorgestellten Patterns setzen Sprachfunktionen auf Smalltalk- bzw. C++-Niveau voraus, die somit auch bestimmen, was leicht bzw. was weniger leicht implementiert werden kann. Würden dagegen prozedurale Sprachen zugrunde gelegt, müsste der Katalog ggf. um weitere Design Patterns, z. B. der Art »Inheritance« (Vererbung), »Encapsulation« (Kapselung) oder »Polymorphism« (Polymorphie), ergänzt werden. Außerdem gilt für manche der weniger populären objektorientierten Sprachen, dass diese die Funktionalität einiger unserer Patterns bereits von Haus aus unterstützen. So sind in CLOS beispielsweise Multimethoden verfügbar, die den Bedarf für ein Design Pattern wie *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)) verringern. Aber auch Smalltalk und C++ unterscheiden sich in vielerlei Hinsicht so weit voneinander, dass sich manche Design Patterns in der einen Sprache einfacher ausdrücken lassen als in der anderen (siehe zum Beispiel *Iterator* (*Iterator*, [Abschnitt 5.4](#))).

1.2 Design Patterns im Smalltalk MVC

In Smalltalk-80 wird zur Entwicklung von Benutzeroberflächen das aus den drei Komponenten *Model* (Datenmodell), *View* (Präsentation) und *Controller* (Programmsteuerung) bestehende *MVC-Architektschema* eingesetzt. Zur Verdeutlichung des Konzepts der »Patterns« sollen die in MVC verwendeten Design Patterns an dieser Stelle einmal etwas eingehender betrachtet werden.

Das MVC-Architektschema besteht aus drei Objektarten bzw. -schichten: Das *Model*-Objekt entspricht dem Anwendungsobjekt, das *view*-Objekt repräsentiert dessen Bildschirmdarstellung und das *Controller*-Objekt steuert die Reaktionen der Benutzeroberfläche auf userseitige Eingaben. Vor der Einführung des MVC-

Schemas wurden diese Objekte in Benutzeroberflächendesigns tendenziell einfach in einem einzigen Objekt zusammengefasst. In MVC werden sie dagegen separat verwendet, wodurch eine bessere Flexibilität und Wiederverwendbarkeit gewährleistet ist.

Das MVC-Architekturmuster nutzt ein *Subscribe/Notify-Benachrichtigungsprotokoll*, um die View- und Model-Objekte zu entkoppeln und sicherzustellen, dass die View-Objekte stets den aktuellen Zustand ihres zugehörigen Model-Objekts abbilden. Zu diesem Zweck werden Änderungen an den Daten im Model-Objekt an die jeweils abhängigen view-Objekte kommuniziert, damit sie sich entsprechend aktualisieren können. Dieser Ansatz ermöglicht die Anbindung mehrerer view-Objekte an ein Model-Objekt, um verschiedene Präsentationen anzubieten, und gestattet zudem die Erzeugung neuer view-Objekte, ohne das Model-Objekt umschreiben zu müssen.

[Abbildung 1.1](#) zeigt ein Model-Objekt mit drei zugehörigen View-Objekten. (Der Einfachheit halber wurden die Controller-Objekte in diesem Beispiel weggelassen.) Das Model-Objekt enthält einige Daten, die mittels der View-Objekte in verschiedenen Ausgabeformaten wiedergegeben werden: als Tabelle, als Histogramm und als Tortendiagramm. Im Fall einer Datenänderung benachrichtigt das Model-Objekt die View-Objekte entsprechend, woraufhin diese auf die geänderten Daten zugreifen und die nötigen Anpassungen vornehmen.

Oberflächlich betrachtet, demonstriert dieses Pattern-Beispiel lediglich ein Design, bei dem die View-Objekte und das Model-Objekt entkoppelt werden. Auf den zweiten Blick lässt es jedoch ein weitaus umfassenderes Konzept erkennen: die Separierung von Objekten in der Form, dass sich Änderungen an einem Objekt auf eine beliebige Anzahl anderer Objekte auswirken können, ohne dass das geänderte Objekt die genaue Beschaffenheit der anderen Objekte berücksichtigen muss. Dieses allgemeinere Design wird durch das Design Pattern *Observer (Beobachter*, siehe [Abschnitt 5.7](#)) beschrieben.

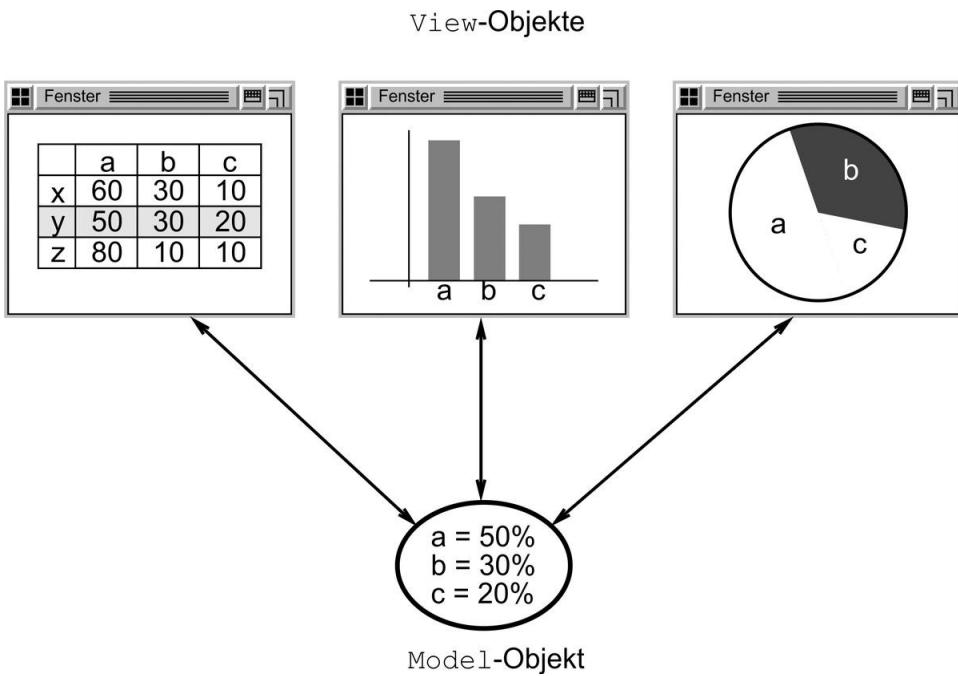


Abb. 1.1: MVC-Design-Pattern-Beispiel

Ein weiteres Merkmal des MVC-Architektschemas ist außerdem die Schachtelung von View-Objekten. So könnte beispielsweise ein mit Buttons bestückter Dialog als ein komplexes View-Objekt implementiert werden, das wiederum aus geschachtelten Button-View-Objekten besteht. Oder die Oberfläche eines Objektinspektors könnte aus geschachtelten View-Objekten bestehen, die sich im Debugger wiederverwenden lassen. Das MVC-Architektschema unterstützt geschachtelte View-Objekte durch die Klasse `CompositeView`, eine Unterklasse von `View`. `CompositeView`-Objekte verhalten sich genauso wie View-Objekte und können überall dort verwendet werden, wo auch Views einsetzbar sind – darüber hinaus enthalten und verwalten sie aber auch geschachtelte View-Objekte.

Nun könnte man sich darunter einfach ein Design vorstellen, das es ermöglicht, einen `CompositeView` genauso zu behandeln wie eine seiner Komponenten. Dieses Design begegnet jedoch zusätzlich noch einer allgemeineren Problematik, die eintritt, wenn man Objekte zu einer Gruppe zusammenfassen und diese Gruppe dann wie ein individuelles Objekt behandeln möchte. Ein derartiges universeller einsetzbares Design wird durch das Design Pattern *Composite (Kompositum, siehe Abschnitt 4.3)* beschrieben: Es ermöglicht die Erstellung einer Klassenhierarchie, in der einige Unterklassen primitive Objekte (wie z. B. Schaltflächen) definieren und andere Klassen wiederum die Composite-Objekte (`CompositeView`), die die primitiven Objekte zu komplexeren Objekten zusammenfügen.

Zudem gestattet das MVC-Architekturmuster auch die Steuerung der Reaktion eines

View-Objekts auf eine Benutzereingabe, ohne dessen visuelle Präsentation zu modifizieren. So ließe sich beispielsweise die Reaktion des View-Objekts auf Tastatureingaben verändern oder statt Tastenbefehlen ein entsprechendes Popup-Menü nutzen. MVC kapselt den Reaktionsmechanismus in sogenannten Controller-Objekten, die einer Klassenhierarchie unterliegen, welche die Erstellung neuer Variationen eines bereits vorhandenen Controller-Objekts erleichtert.

Zur Implementierung einer bestimmten Reaktionsstrategie nutzen View-Objekte eine Instanz einer Controller-Unterklasse. Soll also eine andere Strategie implementiert werden, wird die Instanz einfach durch ein anderes Controller-Objekt ersetzt. Es ist sogar möglich, den Controller eines View-Objekts zur Laufzeit zu ändern, um die Reaktion des Views auf User-Eingaben zu modifizieren. Wollte man beispielsweise erreichen, dass ein View-Objekt nicht mehr auf Eingaben reagiert, sprich deaktiviert wird, müsste man ihm lediglich ein Controller-Objekt zuweisen, das Eingabeereignisse ignoriert.

Die Beziehung zwischen View- und Controller-Objekt ist ein Beispiel für das Design Pattern *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#)). Ein Strategieobjekt ist ein Objekt, das einen Algorithmus repräsentiert. Es ist insbesondere dann nützlich, wenn ein Algorithmus strategisch oder dynamisch ersetzt werden soll, zahlreiche Varianten des Algorithmus vorliegen oder der Algorithmus komplexe Datenstrukturen enthält, die gekapselt werden sollen.

Darüber hinaus setzt das MVC-Architektschema auch andere Design Patterns ein, wie z. B. *Factory Method* (*Fabrikmethode*, siehe [Abschnitt 3.3](#)) zur Spezifikation der Standard-controller-Klasse eines View-Objekts oder auch *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#)), um ein View-Objekt mit Scrollfähigkeit auszustatten – die wichtigsten Beziehungen im MVC-Architektschema werden im Allgemeinen jedoch durch die Design Patterns *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)), *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) und *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#)) definiert.

1.3 Beschreibung der Design Patterns

Wie lassen sich Design Patterns beschreiben? Grafische Notationen, die zwar ebenso wichtig wie hilfreich sind, reichen hier nicht aus, denn sie erfassen das Endprodukt des Designprozesses lediglich als Beziehungen zwischen Klassen und Objekten. Zur Gewährleistung der Wiederverwendbarkeit des Softwaredesigns müssen jedoch auch die Entscheidungen, Alternativen und Erwägungen, die letztlich

zu ihm geführt haben, aufgezeichnet werden. Hilfreich sind in diesem Zusammenhang außerdem konkrete Beispiele, die die Arbeitsweise des Designs verdeutlichen.

Die Beschreibung der Design Patterns erfolgt in diesem Buch nach einem einheitlichen Schema, das jedes Pattern in bestimmte Eigenschaften gliedert. Auf diese Weise ist eine gleichmäßige Informationsstruktur gewährleistet, die das Erlernen, Vergleichen und Anwenden der Design Patterns erleichtert.

Pattern-Name und -Klassifizierung

Die Bezeichnung des Design Patterns charakterisiert zugleich dessen Hauptfunktion bzw. -eigenschaft. Ein aussagekräftiger, prägnanter Name spielt für den täglichen Sprachgebrauch in der Designarbeit eine ganz entscheidende Rolle, um eindeutigen Bezug auf das Pattern nehmen zu können. Die Klassifizierung des Design Patterns erfolgt nach dem in [Abschnitt 1.5](#) beschriebenen System.

Zweck

Die Zweckbeschreibung des Patterns fasst die Antworten auf folgende Fragen in knapper Form zusammen: Was bewirkt das Design Pattern? Welchem Grundprinzip folgt es und welche Intention steht dahinter? Auf welche spezifischen Designfragestellungen oder -probleme ist es ausgerichtet?

Auch bekannt als

Hier sind weitere Bezeichnungen angegeben, unter denen das Pattern ggf. auch bekannt ist.

Motivation

Zum besseren Verständnis der im weiteren Verlauf dieses Buches beschriebenen abstrakteren Eigenschaften des Design Patterns wird je ein Beispielszenario für eine zugrunde liegende Designproblematik angeführt, das aufzeigt, inwiefern die Klassen- und Objektstrukturen des Patterns das Problem lösen.

Anwendbarkeit

In welchen Situationen lässt sich das Design Pattern nutzbringend einsetzen? Welche Designprobleme lassen sich damit beheben? Wie erkennt man solche Situationen?

Struktur

Die grafische Darstellung der in dem Design Pattern enthaltenen Klassen basiert auf der *Object-Modeling Technique* (OMT)-Notation [RBP+91]. Darüber hinaus werden zur Veranschaulichung von Abfrage- und Interaktionsabfolgen zwischen Objekten auch diverse Interaktionsdiagramme [JCJO92, Boo94] dargestellt. Diese Notationen sind in [Anhang B](#) beschrieben.

Teilnehmer

Hier werden die an einem Design Pattern beteiligten Klassen und/oder Objekte sowie deren Zuständigkeiten erläutert.

Interaktionen

An dieser Stelle wird beschrieben, in welcher Weise die Teilnehmer zwecks Ausübung ihrer Zuständigkeiten interagieren.

Konsequenzen

In welcher Weise wird die Zielsetzung des Design Patterns unterstützt? Welche Vorteile bzw. Nachteile bzw. Resultate ergeben sich aus der Anwendung des Patterns? Welche Bereiche der Systemstruktur lassen sich unabhängig voneinander variieren?

Implementierung

Welche Stolperfallen, Tipps oder Techniken sollten bei der Implementierung des Design Patterns beachtet werden? Könnten sprachspezifische Probleme auftreten?

Codebeispiele

Die Codefragmente demonstrieren, wie sich das Design Pattern in C++ oder Smalltalk implementieren lässt.

Praxisbeispiele

Zur Demonstration des praktischen Einsatzes des Design Patterns in realen Systemen werden mindestens zwei Beispiele aus unterschiedlichen Anwendungsbereichen angeführt.

Verwandte Patterns

Welche anderen Design Patterns stehen in einem engeren Bezug zu dem aktuellen Pattern? Worin bestehen ihre wichtigsten Unterschiede? Welche Patterns lassen sich mit dem aktuellen Pattern kombinieren?

Weiterführende Hintergrundinformationen, die zum besseren Verständnis der Design Patterns beitragen, finden Sie in den Anhängen dieses Buches: Anhang A hält ein Glossar mit der gängigen Designterminologie bereit. Im bereits erwähnten [Anhang B](#) werden die verschiedenen Notationen erläutert, deren Kernaspekte in den nachfolgenden Kapiteln beschrieben werden. Und im [Anhang C](#) finden Sie den Quelltext für die Basisklassen, die in den Codebeispielen verwendet werden.

1.4 Der Design-Patterns-Katalog

Der in diesem Buch präsentierte Katalog umfasst insgesamt 23 Design Patterns, die im Folgenden in einer Kurzübersicht unter Angabe ihrer jeweiligen Namen sowie ihrer Zweckbestimmung vorgestellt werden. Eine ausführliche Beschreibung der einzelnen Patterns folgt ab [Kapitel 3](#).

Hinweis

Hinter dem englischen Pattern-Namen ist stets auch die deutsche Entsprechung sowie das Kapitel bzw. der Abschnitt angegeben, in dem es detailliert besprochen

wird. Diese Konvention wird im gesamten Buch durchgängig eingehalten.

Abstract Factory (Abstrakte Fabrik, siehe [Abschnitt 3.1](#))

Bereitstellung einer Schnittstelle zum Erzeugen verwandter oder voneinander abhängiger Objektfamilien ohne die Benennung ihrer konkreten Klassen.

Adapter (Adapter, siehe [Abschnitt 4.1](#))

Anpassung der Schnittstelle einer Klasse an ein anderes von den Clients erwartetes Interface. Das Design Pattern *Adapter (Adapter)* ermöglicht die Zusammenarbeit von Klassen, die ansonsten aufgrund der Inkompatibilität ihrer Schnittstellen nicht dazu in der Lage wären.

Bridge (Brücke, siehe [Abschnitt 4.2](#))

Entkopplung einer Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.

Builder (Erbauer, siehe [Abschnitt 3.2](#))

Getrennte Handhabung der Erzeugungs- und Darstellungsmechanismen komplexer Objekte zwecks Generierung verschiedener Repräsentationen in einem einzigen Erzeugungsprozess.

Chain of Responsibility (Zuständigkeitskette, siehe [Abschnitt 5.1](#))

Vermeidung der Kopplung eines Request-Auslösers mit seinem Empfänger, indem mehr als ein Objekt in die Lage versetzt wird, den Request zu bearbeiten. Die empfangenden Objekte werden miteinander verkettet und der Request wird dann so lange entlang dieser Kette weitergeleitet, bis er von einem Objekt angenommen und bearbeitet wird.

Command (Befehl, siehe [Abschnitt 5.2](#))

Kapselung eines Requests als Objekt, um so die Parametrisierung von Clients mit verschiedenen Requests, Warteschlangen- oder Logging-Operationen sowie das Rückgängigmachen von Operationen zu ermöglichen.

Composite (Kompositum, siehe [Abschnitt 4.3](#))

Komposition von Objekten in Baumstrukturen zur Abbildung von Teil-Ganzes-Hierarchien. Das Design Pattern *Composite (Kompositum)* gestattet den Clients einen einheitlichen Umgang sowohl mit individuellen Objekten als auch mit Objektkompositionen.

Decorator (Dekorierer, siehe [Abschnitt 4.4](#))

Dynamische Erweiterung der Funktionalität eines Objekts. Decorator-Objekte stellen hinsichtlich der Ergänzung einer Klasse um weitere Zuständigkeiten eine flexible Alternative zur Unterklassenbildung dar.

Facade (Fassade, siehe [Abschnitt 4.5](#))

Bereitstellung einer einheitlichen Schnittstelle zu einem Schnittstellensatz in einem Subsystem. Das Design Pattern *Facade (Fassade)* definiert eine Schnittstelle höherer Ebene, die die Nutzung des Subsystems vereinfacht.

Factory Method (Fabrikmethode, siehe [Abschnitt 3.3](#))

Definition einer Schnittstelle zur Objekterzeugung, wobei die Bestimmung der zu instanziierenden Klasse den Unterklassen überlassen bleibt. Das Design Pattern *Factory Method (Fabrikmethode)* gestattet einer Klasse, die Instanziierung an Unterklassen zu delegieren.

Flyweight (Fliegengewicht, siehe [Abschnitt 4.6](#))

Gemeinsame Nutzung feingranularer Objekte, um sie auch in großer Anzahl effizient nutzen zu können.

Interpreter (Interpreter, siehe [Abschnitt 5.3](#))

Definition einer Repräsentation der Grammatik einer gegebenen Sprache sowie Bereitstellung eines Interpreters, der diese Grammatik nutzt, um in der betreffenden Sprache verfasste Sätze zu interpretieren.

Iterator (Iterator, siehe [Abschnitt 5.4](#))

Bereitstellung eines sequenziellen Zugriffs auf die Elemente eines aggregierten Objekts, ohne dessen zugrunde liegende Struktur offenzulegen.

Mediator (Vermittler, siehe [Abschnitt 5.5](#))

Definition eines Objekts, das die Interaktionsweise eines Objektsatzes in sich kapselt. Das Design Pattern *Mediator (Vermittler)* begünstigt lose Kopplungen, indem es die explizite Referenzierung der Objekte untereinander unterbindet und so eine individuelle Steuerung ihrer Interaktionen ermöglicht.

Memento (Memento, siehe [Abschnitt 5.6](#))

Erfassung und Externalisierung des internen Zustands eines Objekts, ohne dessen Kapselung zu beeinträchtigen, so dass es später wieder in diesen Zustand zurückversetzt werden kann.

Observer (Beobachter, siehe [Abschnitt 5.7](#))

Definition einer 1-zu-n-Abhängigkeit zwischen Objekten, damit im Fall einer Zustandsänderung eines Objekts alle davon abhängigen Objekte entsprechend benachrichtigt und automatisch aktualisiert werden.

Prototype (Prototyp, siehe [Abschnitt 3.4](#))

Spezifikation der zu erzeugenden Objekttypen mittels einer prototypischen Instanz und Erzeugung neuer Objekte durch Kopieren dieses Prototyps.

Proxy (Proxy, siehe [Abschnitt 4.7](#))

Bereitstellung eines vorgelagerten Stellvertreterobjekts bzw. eines Platzhalters zwecks Zugriffssteuerung eines Objekts.

Singleton (Singleton, siehe [Abschnitt 3.5](#))

Sicherstellung der Existenz nur einer einzigen Klasseninstanz sowie Bereitstellung eines globalen Zugriffspunkts für diese Instanz.

State (Zustand, siehe [Abschnitt 5.8](#))

Anpassung der Verhaltensweise eines Objekts im Fall einer internen Zustandsänderung, so dass es den Anschein hat, als hätte es seine Klasse gewechselt.

Strategy (Strategie, siehe [Abschnitt 5.9](#))

Definition einer Familie von einzeln gekapselten, austauschbaren Algorithmen. Das Design Pattern *Strategy (Strategie)* ermöglicht eine variable und von den Clients unabhängige Nutzung des Algorithmus.

Template Method (Schablonenmethode, siehe [Abschnitt 5.10](#))

Definition der Grundstruktur eines Algorithmus in einer Operation sowie Delegation einiger Ablaufschritte an Unterklassen. Das Design Pattern *Template Method (Schablonenmethode)* ermöglicht den Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne dessen grundlegende Struktur zu verändern.

Visitor (Besucher, siehe [Abschnitt 5.11](#))

Darstellung einer auf die Elemente einer Objektstruktur anzuwendenden Operation. Das Design Pattern *Visitor (Besucher)* ermöglicht die Definition einer neuen Operation, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

1.5 Aufbau des Katalogs

Design Patterns unterscheiden sich sowohl in ihrer Granularität als auch in ihrem Abstraktionsgrad. Und weil es eine ganze Reihe von Patterns gibt, müssen sie irgendwie organisiert werden.

Bei der im Folgenden dargestellten Klassifizierungsmethode werden die Design Patterns des hier vorgestellten Katalogs nach verwandtschaftlichen Beziehungsgraden gruppiert. Diese Art der Gliederung fördert nicht nur ein besseres und schnelleres Verständnis der einzelnen Pattern-Konzepte, sondern erleichtert auch das Auffinden neuer Patterns.

Zweck			
	Erzeugungsmuster (Creational Patterns)	Strukturmuster (Structural Patterns)	Verhaltensmuster (Behavior Patterns)
Gültigkeitsbereich	Klasse	Factory Method (<i>Fabrikmethode, Abschnitt 3.3</i>)	Adapter (<i>Adapter, (klassenbasiert), Abschnitt 4.1</i>)
Objekt		Abstract Factory (<i>Abstrakte Fabrik, Abschnitt 3.1</i>)	Adapter (<i>Adapter, (objektbasiert), Abschnitt 4.2</i>)

	<i>Builder</i> (<i>Erbauer</i> , Abschnitt 3.2)	Abschnitt 4.1)	Abschnitt
	<i>Prototype</i> (<i>Prototyp</i> , Abschnitt 3.4)	<i>Bridge</i> (<i>Brücke</i> , Abschnitt 4.2)	<i>Command</i> Abschnitt
	<i>Singleton</i> (<i>Singleton</i> , Abschnitt 3.5)	<i>Composite</i> (<i>Kompositum</i> , Abschnitt 4.3)	<i>Iterator</i> (1 Abschnitt
		<i>Decorator</i> (<i>Dekorierer</i> , Abschnitt 4.4)	<i>Mediator</i> (<i>Vermittler</i> , Abschnitt
		<i>Facade</i> (<i>Fassade</i> , Abschnitt 4.5)	<i>Memento</i> Abschnitt
		<i>Flyweight</i> (<i>Fliegengewicht</i> , Abschnitt 4.6)	<i>Observer</i> (<i>Beobachter</i> , Abschnitt
		<i>Proxy</i> (<i>Proxy</i> , Abschnitt 4.7)	<i>State</i> (<i>Zustand</i> , Abschnitt
			<i>Strategy</i> (<i>Strategie</i> , Abschnitt
			<i>Visitor</i> (<i>Besucher</i> , Abschnitt

Tabelle 1.1: Klassifizierung der Design Patterns

In diesem Buch werden die Design Patterns nach zwei übergeordneten Kriterien klassifiziert (siehe [Tabelle 1.1](#)). Das erste Kriterium gibt den **Zweck** des Patterns an, also was es bewirkt. Design Patterns dienen entweder der **Objekterzeugung** oder sie sind **struktur- bzw. verhaltensorientiert**:

- *Erzeugungsmuster* beziehen sich auf den Erstellungsprozess von Objekten.
- *Strukturmuster* wirken sich auf die Zusammensetzung von Klassen und Objekten aus.
- *Verhaltensmuster* charakterisieren die Art und Weise der Interaktion von

Klassen und Objekten sowie die Verteilung der Zuständigkeiten.

Das zweite Kriterium, der **Gültigkeitsbereich**, spezifiziert, ob das Design Pattern vorwiegend auf Klassen oder auf Objekte angewendet wird.

- *Klassenbasierte* Design Patterns beeinflussen die Beziehungen der Klassen zu ihren Unterklassen, die durch Vererbung erstellt werden und damit statisch sind – d. h., sie werden schon beim Kompilieren festgelegt.
- *Objektbasierte* Design Patterns haben dagegen Auswirkungen auf die Beziehungen zwischen Objekten, die sich zur Laufzeit ändern können und somit dynamischer sind.

Weil fast alle Design Patterns ein gewisses Maß an Vererbung nutzen, werden prinzipiell nur diejenigen als »klassenbasiert« bezeichnet, die sich auch wirklich auf die Klassenbeziehungen konzentrieren. Im Allgemeinen sind die meisten Design Patterns jedoch objektbasiert.

Klassenbasierte Erzeugungsmuster delegieren die Objekterstellung teilweise an Unterklassen, *objektbasierte Erzeugungsmuster* dagegen an ein anderes Objekt. *Klassenbasierten Strukturmuster* stützen sich bei der Zusammenführung auf das Konzept der Vererbung, *objektbasierte Strukturmuster* bilden dagegen Wege ab, um Objekte zusammenzufügen. Und *klassenbasierte Verhaltensmuster* greifen wiederum ebenfalls auf die Vererbung zurück, um sowohl Algorithmen als auch den Programmablauf zu bestimmen, während *objektbasierte Verhaltensmuster* beschreiben, in welcher Form eine Gruppe von Objekten interagiert, um einen Task auszuführen, der nicht von einem einzelnen Objekt bewerkstelligt werden kann.

Es gibt aber noch andere Möglichkeiten, die Design Patterns zu klassifizieren. Manche Patterns werden beispielsweise häufig gemeinsam verwendet, wie etwa *Composite (Kompositum)* mit *Iterator (Iterator)* oder *Visitor (Besucher)*. Andere Patterns stellen hingegen Alternativen zueinander dar, wie z. B. im Fall von *Prototype (Prototyp)* und *Abstract Factory (Abstrakte Fabrik)*, die oftmals alternativ eingesetzt werden. Und schließlich gibt es auch noch solche Patterns, die im Endeffekt zu gleichartigen Designs führen, obwohl sie jeweils unterschiedliche Zwecke erfüllen. Ein Beispiel hierfür sind *Composite (Kompositum)* und *Decorator (Dekorierer)*, die sehr ähnliche Strukturdiagramme aufweisen.

Eine weitere Klassifizierungsvariante für Design Patterns basiert auf ihren verwandtschaftlichen Beziehungen zueinander, die in [Abbildung 1.2](#) in einer Übersicht dargestellt sind.

Insgesamt gibt es eine Vielzahl von hilfreichen Klassifizierungsarten für Design Patterns – und sie alle haben durchaus ihre Daseinsberechtigung, denn: Unterschiedliche Denkansätze im Hinblick auf die Klassifizierung führen in jedem Fall zu einem weitreichenderen, besseren Verständnis der Funktionsweise der einzelnen Design Patterns, bieten aufschlussreiche Vergleichsmöglichkeiten und zeigen auf, wann ihr Einsatz sinnvoll ist.

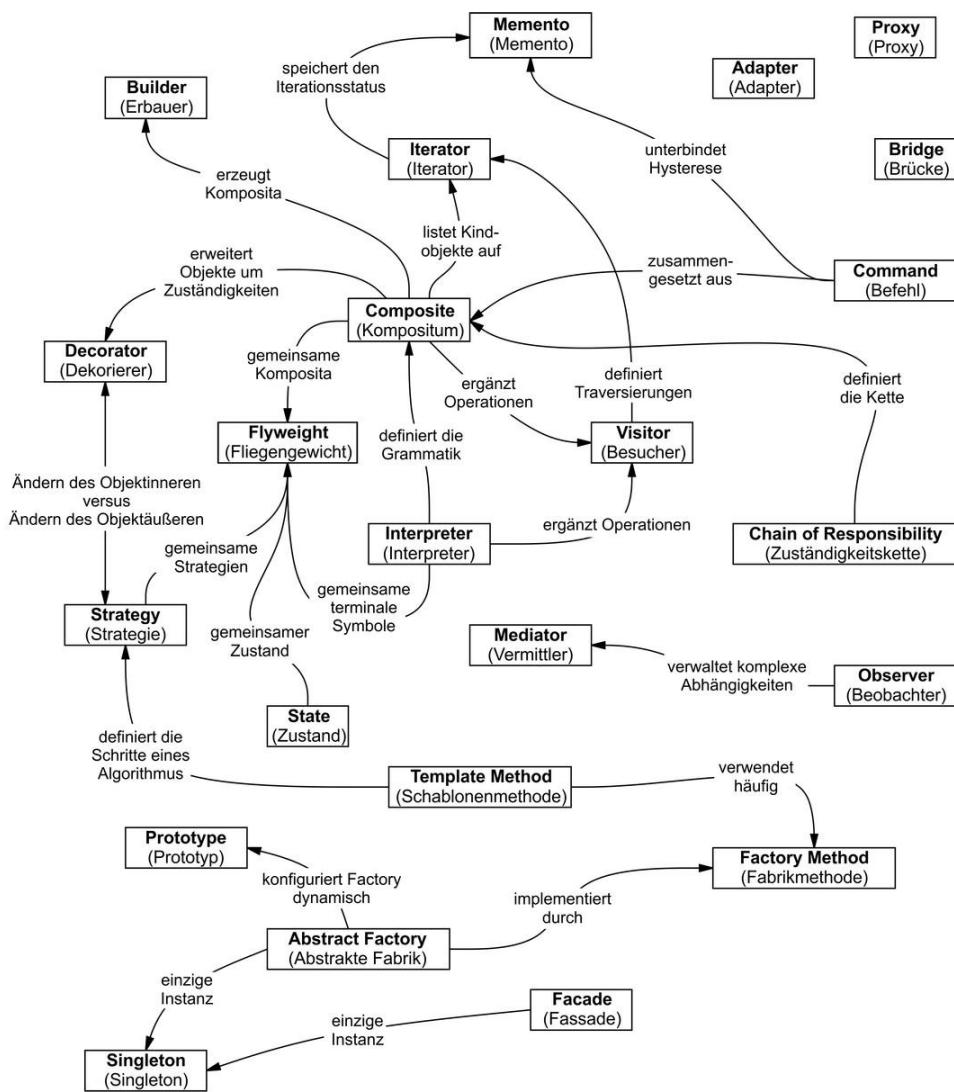


Abb. 1.2: Verwandtschaftliche Beziehungen der Design Patterns

1.6 Die Anwendung von Design Patterns zur Behebung von Designproblemen

Design Patterns bieten objektorientierten Softwareentwicklern die Gelegenheit,

zahlreichen Problemen, denen sie tagtäglich gegenüberstehen, auf sehr unterschiedliche Art und Weise zu begegnen. Die nachfolgenden Beispiele veranschaulichen einige solcher Problemfälle und demonstrieren, wie sie sich durch den Einsatz von Design Patterns lösen lassen.

1.6.1 Passende Objekte finden

Objektorientierte Programme setzen sich – wie der Name schon vermuten lässt – aus **Objekten** zusammen. Jedes Objekt umfasst neben dem reinen Datenbestand auch die prozeduralen Routinen – die sogenannten **Methoden** oder **Operationen** –, die auf diesen Daten basieren. Sobald ein Objekt einen **Request** (also eine **Anfrage** oder eine **Nachricht**) von einem **Client** erhält, führt es eine Operation aus.

Requests stellen die *einige* Möglichkeit dar, ein Objekt zur Durchführung einer Operation zu veranlassen. Und Operationen bieten ihrerseits die *einige* Möglichkeit, die internen Daten eines Objekts zu modifizieren. Dieses Konzept wird als **Kapselung** bezeichnet: Es ist kein direkter Zugriff auf den internen Zustand des Objekts möglich und er ist auch nicht nach außen hin sichtbar.

Eine besondere Herausforderung beim objektorientierten Design besteht in der Aufgliederung eines Systems in einzelne Objekte – denn dabei müssen zahlreiche Faktoren berücksichtigt werden, die sich nicht selten noch dazu in widersprüchlicher Art und Weise auswirken: die Kapselung, die Granularität, die Abhängigkeiten, die Flexibilität, das Laufzeitverhalten, die Evolution, die Wiederverwendbarkeit und vieles mehr.

Objektorientierte Designmethoden begünstigen in dieser Hinsicht viele verschiedene Ansätze. So könnte man beispielsweise die beherrschenden Substantiv und Verben zur Beschreibung der vorliegenden Problemstellung herausgreifen und dazu passende Klassen und Operationen erstellen. Oder man richtet sein Augenmerk mehr auf die Interaktionen und Zuständigkeiten in dem System. Oder man fertigt ein realitätsnahes Modell an, analysiert es und überträgt die dabei ermittelten Objekte in das Design. Welcher Ansatz letztlich der beste ist, ist immer auch ein wenig Geschmackssache.

In vielen Fällen entstammen die Objekte dem realitätsnahen Analysemodell, darüber hinaus werden in objektorientierten Designs häufig aber auch Klassen benutzt, für die es keine realen Entsprechungen gibt. Einige davon, wie z. B. Arrays, besitzen einen niedrigen Abstraktionsgrad, andere hingegen weisen einen deutlich höheren auf. So nutzt beispielsweise das Design Pattern *Composite (Kompositum)*, siehe

[Abschnitt 4.3](#)) eine Abstraktion zur einheitlichen Behandlung von Objekten, für die es kein physisches Gegenstück gibt. Eine strikt an der realen Welt orientierte Modellbildung hat ein System zur Folge, das zwar die gegenwärtigen Realitäten widerspiegelt, nicht notwendigerweise aber auch zukünftige – der Schlüssel für eine flexible Gestaltung des Designs sind die Abstraktionen, die aus dem Designprozess hervorgehen.

Patterns erleichtern die Identifizierung der weniger deutlich erkennbaren Abstraktionen sowie der Objekte, die diese erfassen können. So finden sich für Objekte, die einen Prozess oder einen Algorithmus repräsentieren, normalerweise keine natürlichen Entsprechungen, trotzdem sind sie ein wichtiger Bestandteil flexibler Designs. Das Design Pattern *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#)) beschreibt, wie sich austauschbare Algorithmusfamilien implementieren lassen. Und das Pattern *State* (*Zustand*, siehe [Abschnitt 5.8](#)) bildet jeden Zustand einer Entität in Form eines Objekts ab. Solche Objekte sind im Rahmen einer Analyse oder in einer frühen Phase des Designprozesses nur selten ausfindig zu machen, sondern treten meist erst später in Erscheinung, wenn es darum geht, das Design flexibler und wiederverwendbar auszugestalten.

1.6.2 Objektgranularität bestimmen

Sowohl die Größe als auch die Anzahl der verwendeten Objekte kann erheblich variieren. Außerdem können sie von der Hardware bis zur vollständigen Anwendung so ziemlich alles repräsentieren. Wie also lässt sich bestimmen, was genau als Objekt genutzt werden sollte?

Auch auf diese Frage liefern Design Patterns eine Antwort. Das Pattern *Facade* (*Fassade*, siehe [Abschnitt 4.5](#)) beschreibt beispielsweise, wie sich komplexe Subsysteme als Objekte abbilden lassen, und das Pattern *Flyweight* (*Fliegengewicht*, siehe [Abschnitt 4.6](#)) definiert die Unterstützung einer großen Zahl von Objekten geringster Granularität. Andere Design Patterns zeigen wiederum bestimmte Arten der Zergliederung eines Objekts in mehrere kleinere Objekte auf. *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) und *Builder* (*Erbauer*, siehe [Abschnitt 3.2](#)) befassen sich mit Objekten, die ausschließlich für die Erzeugung anderer Objekte zuständig sind. *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)) und *Command* (*Befehl*, siehe [Abschnitt 5.2](#)) zielen dagegen auf Objekte ab, deren einzige Aufgabe in der Implementierung eines Requests an ein anderes Objekt bzw. eine Objektgruppe besteht.

1.6.3 Objektschnittstellen spezifizieren

Jede von einem Objekt deklarierte Operation weist eine sogenannte **Signatur** auf, die den Namen, die als Parameter enthaltenen Objekte sowie den Rückgabewert der Operation spezifiziert. Die Menge aller durch die Operationen eines Objekts definierten Signaturen wird als **Schnittstelle** des Objekts bezeichnet. Sie gibt vor, welche Requests an das Objekt übermittelt werden können – d. h. jeder Request, der einer in der Objektschnittstelle erfassten Signatur entspricht.

Der **Typ** eines Objekts bezeichnet eine bestimmte Schnittstelle. Wenn ein Objekt beispielsweise alle Requests für die in seiner Schnittstelle `window` definierten Operationen akzeptiert, dann spricht man in diesem Fall davon, dass es vom Typ `window` ist. Grundsätzlich kann ein Objekt mehrere Typen haben und umgekehrt können sehr verschiedene Objekte einen gemeinsamen Typ besitzen. Ebenso können einzelne Teile einer Objektschnittstelle unterschiedlichen Typs sein, d. h., zwei Objekte desselben Typs haben gegebenenfalls lediglich individuelle Schnittstellenbestandteile gemeinsam. Darüber hinaus können Schnittstellen auch andere Schnittstellen als Untermengen enthalten. Wenn die Schnittstelle eines Typs die Schnittstelle seines übergeordneten **Supertyps** enthält, wird dieser Typ als **Subtyp** bezeichnet. Häufig spricht man in diesem Zusammenhang auch davon, dass ein Subtyp die Schnittstelle seines Supertyps *erbt*.

Schnittstellen sind ein grundlegender Bestandteil objektorientierter Systeme. Die Objekte sind ausschließlich über ihre Schnittstellen bekannt. Grundsätzlich gibt es keine andere Möglichkeit, etwas über ein Objekt in Erfahrung zu bringen oder es zu etwas zu veranlassen, als über die Schnittstelle. Sie sagt jedoch nichts über die Implementierung des Objekts aus: Verschiedene Objekte können Requests auf ganz unterschiedliche Art und Weise implementieren – d. h., zwei Objekte mit völlig verschiedenen Implementierungen können dennoch identische Schnittstellen besitzen.

Bei der Übermittlung eines Requests an ein Objekt ist die daraufhin auszuführende Operation *sowohl* von dem Request *als auch* von dem empfangenden Objekt abhängig. Unterschiedliche Objekte, die identische Requests unterstützen, können wiederum verschiedene Implementierungen der Operationen nutzen, die diese Requests ausführen. Die zur Laufzeit erfolgende Zuweisung eines Requests zu einem Objekt und einer seiner Operationen wird als **dynamische Bindung** bezeichnet.

Dank der dynamischen Bindung ist beim Absetzen eines Requests bis zum Zeitpunkt der Ausführung keine Festlegung auf eine spezifische Implementierung

erforderlich. Somit ist es problemlos möglich, Programme zu entwickeln, die Objekte mit bestimmten Schnittstellen erwarten, weil jedes Objekt, das die passende Schnittstelle aufweist, diesen Request akzeptieren wird. Zudem gestattet die dynamische Bindung auch die Substitution, sprich den Austausch von Objekten mit identischen Schnittstellen zur Laufzeit – dieses Schlüsselkonzept objektorientierter Systeme wird als **Polymorphie** bezeichnet. Dadurch braucht ein Client-Objekt außer in Bezug auf deren Unterstützung einer bestimmten Schnittstelle keine weiteren Annahmen über andere Objekte zu treffen. Die Polymorphie vereinfacht also die Client-Definition, entkoppelt Objekte voneinander und gestattet es ihnen außerdem, ihre wechselseitigen Beziehungen zur Laufzeit zu variieren.

Design Patterns erleichtern die Schnittstellendefinition, indem sie ihre Schlüsselemente sowie die Datentypen, die über eine Schnittstelle geleitet werden, identifizieren. Mitunter geben sie auch vor, was *nicht* in einer Schnittstelle enthalten sein sollte. So beschreibt zum Beispiel das Pattern *Memento* (*Memento*, siehe [Abschnitt 5.6](#)), wie der interne Zustand eines Objekts gekapselt und gespeichert sein muss, damit das Objekt zu einem späteren Zeitpunkt wieder in diesen Zustand zurückversetzt werden kann. Hier lautet die Vorgabe, dass *Memento*-Objekte zwei Schnittstellen definieren müssen: eine *eingeschränkte* Schnittstelle, die den Clients zum Verwahren und Kopieren der *Mementos* dient, und eine *privilegierte* Schnittstelle, die ausschließlich vom ursprünglichen Objekt genutzt werden kann, um seinen Zustand im *Memento* zu speichern und wiederherzustellen.

Darüber hinaus spezifizieren Design Patterns auch die Beziehungen zwischen den Schnittstellen. Insbesondere setzen sie häufig voraus, dass einige Klassen ähnliche Schnittstellen aufweisen oder sehen Einschränkungen für bestimmte Klassenschnittstellen vor. So erfordern sowohl *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#)) als auch *Proxy* (*Proxy*, siehe [Abschnitt 4.7](#)), dass die Schnittstellen der *Decorator*- und *Proxy*-Objekte mit denen der dekorierten bzw. stellvertretenen Objekten identisch sind. Und beim Pattern *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)) muss die *visitor*-Schnittstelle die Klassen aller Objekte abbilden, die der *visitor* besuchen kann.

1.6.4 Objektimplementierungen spezifizieren

Nachdem sie bislang lediglich am Rande erwähnt wurde, soll die konkrete Definition eines Objekts an dieser Stelle einmal ausführlicher betrachtet werden. Die Implementierung eines Objekts wird durch seine **Klasse** definiert. Sie spezifiziert die internen Daten sowie die Repräsentation des Objekts und bestimmt darüber

hinaus auch, welche Operationen es ausführen kann.

In der auf der *Object-Modeling Technique* (*OMT*, Objekt-Modellierungstechnik) basierenden Notation (siehe [Anhang B](#)) wird eine Klasse in einem Objektdiagramm wie folgt dargestellt: Zuoberst ist der Klassenname in Fettschrift angegeben, das nächste Segment enthält eine Auflistung der Operationen in Normalschrift, und zum Schluss sind die von der Klasse definierten Daten aufgeführt (siehe [Abbildung 1.3](#)).

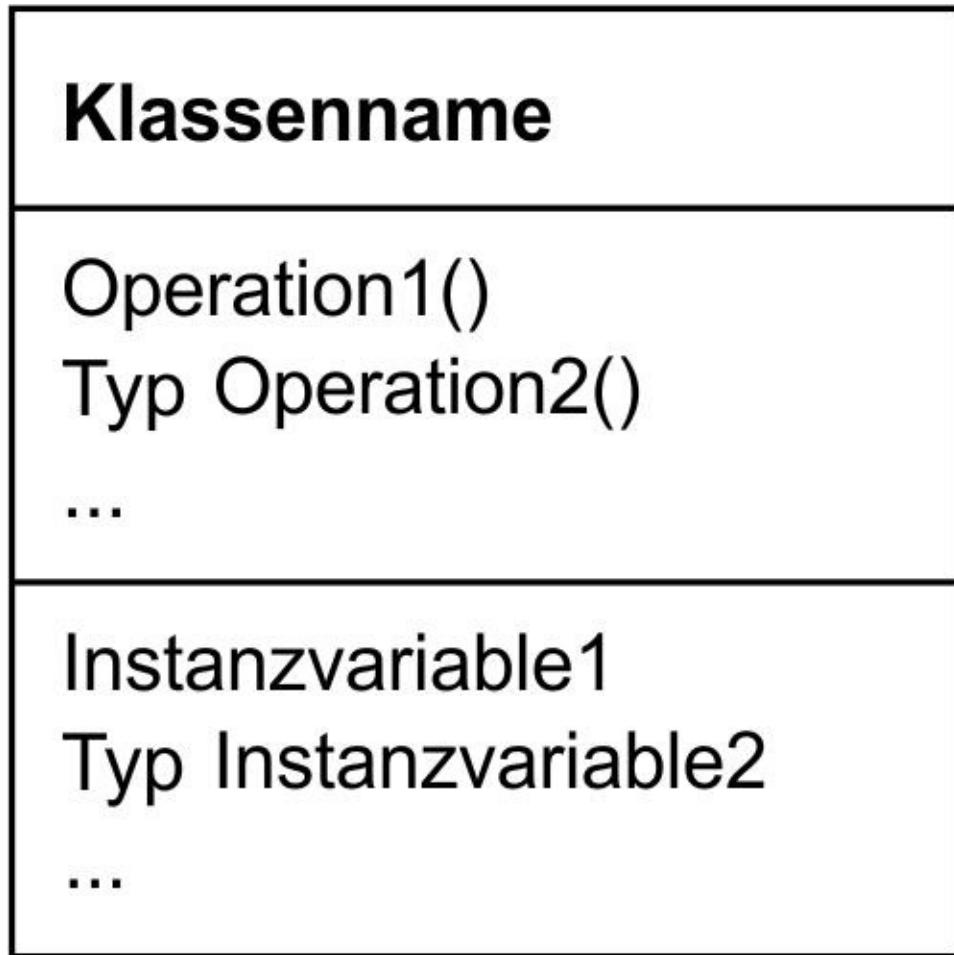


Abb. 1.3: Objektdiagramm einer Klasse

Da hier nicht von einer statisch typisierten Implementierungssprache ausgegangen wird, sind die Typen der Rückgabewerte und Instanzvariablen optional.

Objekte werden durch die **Instanziierung** einer Klasse erzeugt und dementsprechend als **Instanzen** der Klasse bezeichnet. Während des Instanziierungsvorgangs wird zum einen der Speicherbereich für die internen (aus den **Instanzvariablen** bestehenden) Daten des Objekts angelegt, und zum anderen werden die Operationen mit diesen Daten verknüpft. Durch die Instanziierung einer

Klasse können viele ähnliche Instanzen eines Objekts erzeugt werden.

Eine Klasse, die Objekte einer anderen Klasse instanziert, wird durch einen gestrichelten Pfeil gekennzeichnet. Die Pfeilspitze verweist dabei auf die Klasse der instanziierten Objekte:

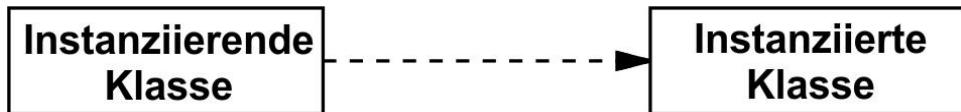


Abb. 1.4: Instanziierung von Objekten anderer Klassen

Die **Klassenvererbung** ermöglicht die Erstellung neuer Klassen, die auf bereits existierenden Klassen basieren. Und wenn eine **Unterklasse** (auch **abgeleitete** oder **Subklasse** genannt) von einer **Basisklasse** (auch als **Eltern-** oder **Superklasse** bezeichnet) erbt, beinhaltet sie auch die Definitionen aller zugehörigen Daten und Operationen, die in der Basisklasse definiert sind. Somit enthalten die Objekte, die Instanzen der Unterklasse sind, ebenfalls alle in der Unterklasse und ihren Basisklassen definierten Daten und sind in der Lage, sämtliche dort definierten Operationen auszuführen. Die Beziehung von Unterklasse und Basisklasse ist in [Abbildung 1.5](#) durch eine vertikale Linie mit aufgesetztem Dreiecksymbol gekennzeichnet:

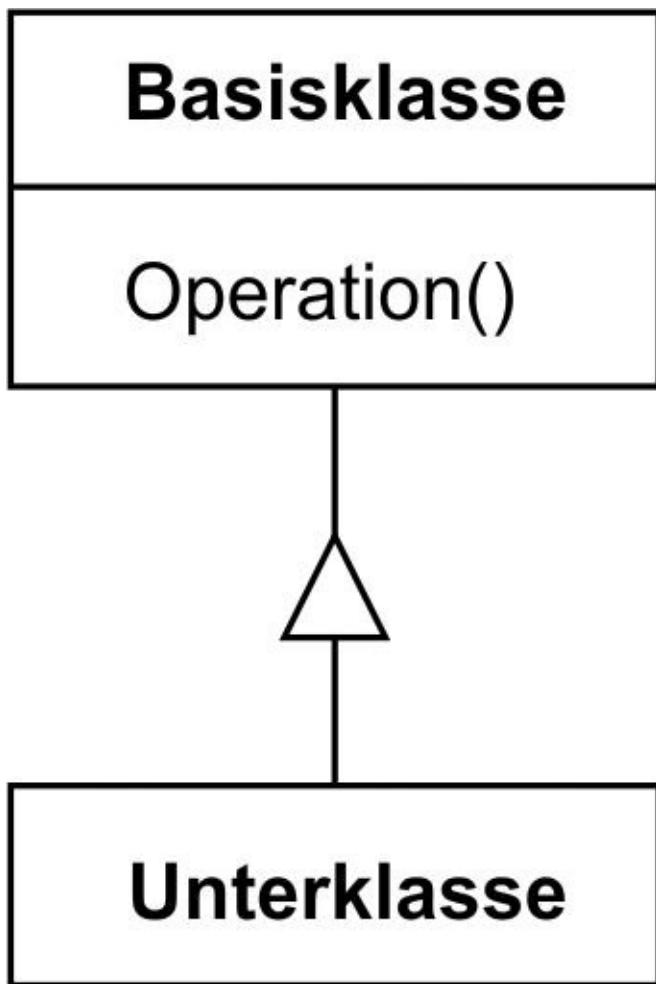


Abb. 1.5: Beziehung Basisklasse – Unterklasse

Der Hauptzweck einer **abstrakten Klasse** besteht in der Definition einer gemeinsamen Schnittstelle für alle ihre Unterklassen. Da solche Klassen ihre Implementierungen teilweise oder auch vollumfänglich an die in den Unterklassen definierten Operationen delegieren, können sie nicht instanziiert werden. Die von einer abstrakten Klasse deklarierten, aber nicht implementierten Operationen werden als **abstrakte Operationen** bezeichnet. Nicht abstrakte Klassen werden **konkrete Klassen** genannt.

Unterklassen können das Verhalten ihrer Basisklassen weiter verfeinern und auch umdefinieren. Genauer gesagt: Eine Klasse ist in der Lage, eine von ihrer Basisklasse definierte Operation zu **überschreiben**. Dadurch können Unterklassen stellvertretend für ihre Basisklassen die Bearbeitung von Requests übernehmen. Das Konzept der Klassenvererbung gestattet also nicht nur die Definition neuer Klassen durch die schlichte Erweiterung anderer Klassen, sondern erleichtert auch die Definition von Objektfamilien mit verwandter Funktionalität.

Zur besseren Unterscheidung von den konkreten Klassen sind die Namen von abstrakten Klassen – ebenso wie die abstrakten Operationen – in den nachfolgenden Diagrammen in Kursivschrift angegeben. Außerdem können die Diagramme auch Pseudocode für die Implementierung einer Operation ausweisen, der dann durch ein Eselsohr gekennzeichnet und über eine gestrichelte Linie mit der implementierten Operation verbunden ist (siehe [Abbildung 1.6](#)):

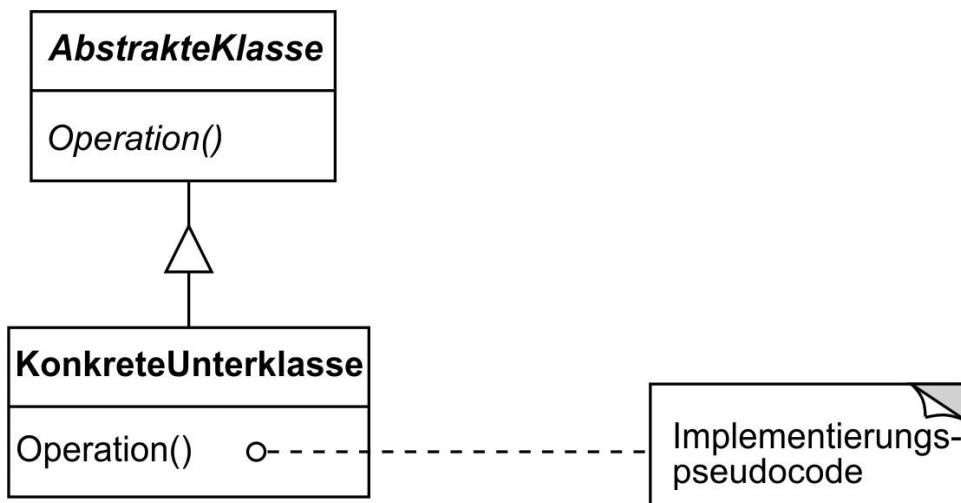


Abb. 1.6: Abstrakte und konkrete Klassen

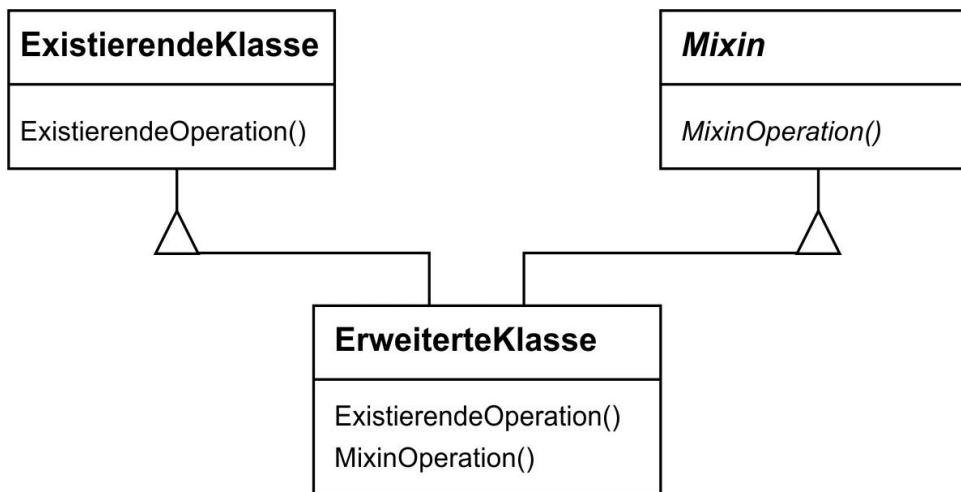


Abb. 1.7: Mehrfachvererbung der *Mixin*-Klasse

Eine **Mixin-Klasse** ist eine Klasse, die eine optionale Schnittstelle oder Funktionalität für andere Klassen bereitstellt. Mit einer abstrakten Klasse hat sie gemein, dass sie ebenfalls nicht instanziiert wird. **Mixin-Klassen** bedingen eine Mehrfachvererbung.

Klassenvererbung kontra Schnittstellenvererbung

Zwischen der **Klasse** und dem **Typ** eines Objekts besteht ein wesentlicher Unterschied: Die *Klasse* definiert, wie das Objekt implementiert ist. Sie beschreibt seinen internen Zustand und die Implementierung seiner Operationen. Im Gegensatz dazu bezieht sich der *Typ* eines Objekts ausschließlich auf seine Schnittstelle – die Gesamtheit der Requests, auf die es antworten kann. Ein Objekt kann viele Typen haben, andererseits können Objekte verschiedener Klassen aber auch denselben Typ besitzen.

Selbstverständlich stehen Klasse und Typ in einer engen Beziehung zueinander: Weil eine Klasse die Operationen definiert, die ein Objekt ausführen kann, bestimmt sie somit auch den Typ des Objekts. Die Feststellung, dass ein Objekt eine Instanz einer Klasse ist, bedeutet also im Grunde genommen, dass das Objekt die von der Klasse vorgegebene Schnittstelle unterstützt.

In Programmiersprachen wie C++ und Eiffel werden Klassen zur Spezifikation sowohl des Typs als auch der Implementierungen eines Objekts verwendet. Smalltalk-Programme deklarieren hingegen keine Typen für Variablen, und dementsprechend überprüft der Compiler auch nicht, ob die einer Variablen zugewiesenen Objekttypen Subtypen des Variablentyps sind. Das Versenden eines Requests erfordert zwar eine dahingehende Überprüfung, dass die Klasse des Empfängers den Request auch implementiert – ob der Empfänger eine Instanz einer bestimmten Klasse ist, muss jedoch nicht geprüft werden.

Ein weiterer wichtiger Aspekt, den es in diesem Zusammenhang zu beachten gilt, ist der Unterschied zwischen der **Klassenvererbung** und der **Schnittstellenvererbung** (auch als **Subtyping** bezeichnet). Bei der Klassenvererbung wird die Implementierung eines Objekts in Form der Implementierung eines anderen Objekts definiert. Es handelt sich also um einen Mechanismus zur Wiederverwertung bzw. Wiederverwendung von Code und Repräsentation. Bei der Schnittstellenvererbung geht es im Gegensatz dazu darum, wann ein Objekt *anstelle* eines anderen Objekts verwendet werden kann.

Diese beiden Konzepte sind leicht zu verwechseln, weil sie in vielen Sprachen nicht explizit unterschieden werden. In Programmiersprachen wie C++ und Eiffel bezieht sich das Konzept der Vererbung sowohl auf die Schnittstellen- als auch auf die Implementierungsvererbung. So wird die Vererbung einer Schnittstelle in C++ z. B. standardmäßig durch das öffentliche Erben von einer Klasse realisiert, die (rein) virtuelle Memberfunktionen aufweist. Eine Annäherung an eine reine Schnittstellenvererbung lässt sich in C++ durch das öffentliche Erben von rein

abstrakten Klassen erreichen. In Smalltalk erfolgt eine Vererbung grundsätzlich als Implementierungsvererbung. Hier können Variablen generell Instanzen beliebiger Klassen zugewiesen werden, solange diese die auf dem Wert der Variablen ausgeführte Operation unterstützen.

Auch wenn die meisten Programmiersprachen keine Unterscheidung zwischen Schnittstellen- und Implementierungsvererbung vornehmen, ist dies aufseiten der Entwickler durchaus gängige Praxis: Smalltalk-Programmierer behandeln Unterklassen in der Regel wie Subtypen (abgesehen von einigen bekannten Ausnahmen [Coo92]), und C++-Programmierer manipulieren Objekte mithilfe der von abstrakten Klassen definierten Typen.

Viele der in diesem Buch vorgestellten Design Patterns sind von dieser Unterscheidung abhängig. So müssen beispielsweise die Objekte im Pattern *Chain of Responsibility* (*Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) einen gemeinsamen Typ haben, weisen aber üblicherweise keine gemeinsame Implementierung auf. Im Pattern *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) definiert die Komponente zwar eine gemeinsame Schnittstelle, das Kompositum definiert jedoch häufig eine gemeinsame Implementierung. Und die Design Patterns *Command* (*Befehl*, siehe [Abschnitt 5.2](#)), *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)), *State* (*Zustand*, siehe [Abschnitt 5.8](#)) und *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#)) werden oftmals mit abstrakten Klassen implementiert, die reine Schnittstellen darstellen.

Gegen Schnittstellen statt gegen Implementierungen programmieren

Die Klassenvererbung stellt im Wesentlichen einen Mechanismus zur Erweiterung der Funktionalität einer Anwendung durch Wiederverwendung der in den Basisklassen enthaltenen Funktionen dar. Sie ermöglicht eine sehr schnelle, auf einem bereits existierenden Objekt basierende Definition eines neuen Objekts. Auf diese Weise lassen sich neue Implementierungen praktisch ohne Aufwand realisieren, weil ein Großteil dessen, was dafür nötig ist, von existierenden Klassen geerbt wird.

Die Wiederverwendung einer Implementierung ist allerdings erst »die halbe Miete« – denn auch die Fähigkeit, bei der Vererbung Objektfamilien mit *identischen* Schnittstellen definieren zu können (in der Regel durch das Erben von einer abstrakten Klasse), spielt hierbei eine bedeutende Rolle. Sie fragen sich warum? Weil die Polymorphie davon abhängig ist.

Eine sorgfältige (bzw. korrekte) Anwendung des Vererbungskonzepts bedingt, dass

alle von einer abstrakten Klasse abgeleiteten Klassen auch deren Schnittstelle nutzen. Und das bedeutet wiederum, dass eine Unterklasse Operationen lediglich ergänzt oder überschreibt, nicht aber die Operationen der Basisklasse verbirgt. So können alle Unterklassen auf die Requests in der Schnittstelle der betreffenden abstrakten Klasse reagieren und werden somit auch allesamt zu Subtypen der abstrakten Klasse.

Die einzig auf der durch die abstrakten Klassen definierten Schnittstelle basierende Manipulation von Objekten bringt zwei Vorteile mit sich:

1. Solange die Objekte die clientseitig erwartete Schnittstelle ansprechen, bleiben den Clients die spezifischen verwendeten Objekttypen verborgen.
2. Die Klassen, die diese Objekte implementieren, bleiben den Clients ebenfalls verborgen. Sie erkennen lediglich die abstrakten Klassen, die die Schnittstelle definieren.

Durch diese Verfahrensweise werden die Implementierungsabhängigkeiten zwischen den Subsystemen derart reduziert, dass folgender Grundsatz des wiederverwendbaren objektorientierten Designs greift:

Programmiere gegen Schnittstellen, nicht gegen Implementierungen.

Variablen dürfen also nicht als Instanzen bestimmter konkreter Klassen deklariert, sondern ausschließlich auf eine von einer abstrakten Klasse definierte Schnittstelle ausgerichtet sein. Und genau dieses Prinzip ist auch eins der Hauptmotive der Design Patterns in diesem Buch.

Natürlich müssen aber irgendwo im System auch konkrete Klassen instanziert bzw. eine bestimmte Implementierung spezifiziert werden – und eben dazu sind die **Erzeugungsmuster** (*Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)), *Builder* (*Erbauer*, siehe [Abschnitt 3.2](#)), *Factory Method* (*Fabrikmethode*, siehe [Abschnitt 3.3](#)), *Prototype* (*Prototyp*, siehe [Abschnitt 3.4](#)) und *Singleton* (*Singleton*, siehe [Abschnitt 3.5](#))) gedacht. Durch die Abstrahierung des Objekterstellungsprozesses bieten diese Patterns verschiedene Möglichkeiten, eine Schnittstelle bei der Instanziierung transparent mit ihrer Implementierung zu verknüpfen. Erzeugungsmuster gewährleisten, dass das System mit Blick auf die Schnittstellen und nicht auf die Implementierungen geschrieben wird.

1.6.5 Wiederverwendungsmechanismen einsetzen

Die meisten Entwickler sind sich über Konzepte wie Objekte, Schnittstellen, Klassen und Vererbung im Klaren. Die eigentliche Herausforderung besteht jedoch darin, sie bei der Entwicklung flexibler, wiederverwendbarer Software auch möglichst sinnvoll einzusetzen – und die Design Patterns zeigen auf, wie das funktioniert.

Vererbung kontra Komposition

Die zwei geläufigsten Techniken für die Wiederverwendung von Funktionalität in objektorientierten Systemen sind die **Klassenvererbung** und die

Objektkomposition. Wie zuvor bereits erwähnt, ermöglicht die Klassenvererbung die Definition der Implementierung einer Klasse auf der Grundlage einer anderen. Das Konzept der Wiederverwendung durch Unterklassenbildung wird häufig auch als **White-Box-Wiederverwendung** bezeichnet. Der Begriff »White-Box« bezieht sich dabei auf die Sichtbarkeit: Bei der Vererbung ist die interne Struktur der Basisklassen für die Unterklassen meist sichtbar.

Die Alternative zur Klassenvererbung ist die Objektkomposition. Bei dieser Technik wird eine neue, komplexere Funktionalität durch die Zusammensetzung, sprich die *Komposition* von Objekten erreicht. Voraussetzung ist hierbei, dass die zusammengeführten Objekte wohldefinierte Schnittstellen aufweisen. Diese Art der Wiederverwendung wird auch **Black-Box-Wiederverwendung** genannt, weil die interne Struktur der Objekte in diesem Fall nicht sichtbar ist – die Objekte treten lediglich als »Black-Boxes« in Erscheinung.

Sowohl die Vererbung als auch die Komposition haben jeweils ihre Vor- und Nachteile. Die Klassenvererbung wird schon beim Kompilieren statisch definiert und lässt sich unkompliziert anwenden, weil sie unmittelbar von der Programmiersprache unterstützt wird. Außerdem erleichtert sie die Modifikation der wiederverwendeten Implementierung. Wenn eine Unterklasse einige, aber nicht alle Operationen überschreibt, kann sie damit auch die geerbten Operationen beeinflussen, sofern diese die überschriebenen Operationen aufrufen.

Aber auch die Klassenvererbung hat einige Nachteile. Erstens können die von der Basisklasse geerbten Implementierungen nicht zur Laufzeit geändert werden, weil die Vererbung bereits beim Kompilieren definiert wird. Und zweitens, was meist schwerer wiegt, definieren die Basisklassen oftmals zumindest Teile der physischen Repräsentation ihrer Unterklassen. Da eine Unterklasse durch den Vererbungsprozess der detaillierten Implementierungsbeschaffenheit ihrer Basisklasse ausgesetzt wird, spricht man davon, dass die »Vererbung die Kapselung aufbricht« [Sny86]: Die Implementierung der Unterklasse wird in so hohem Maße

mit der Implementierung ihrer Basisklasse verknüpft, dass jede Modifikation an der Basisklassenimplementierung auch eine Änderung der UnterkLASSE erforderlich macht.

Solche Implementierungsabhängigkeiten können bei dem Versuch, eine UnterkLASSE wiederzuverwenden, zu Problemen führen: Sollte irgendein Aspekt der geerbten Implementierung nicht für die neuen Anwendungen geeignet sein, muss die Basisklasse umgeschrieben oder durch eine passendere Klasse ersetzt werden. Zudem schränken solche Abhängigkeiten nicht nur die Flexibilität, sondern letztlich auch die Wiederverwendbarkeit ein. Eine Möglichkeit, diesem Problem zu begegnen, ist das ausschließliche Erben von abstrakten Klassen, da diese in der Regel nur wenig bis gar keine Implementierung vorsehen.

Die Objektkomposition wird durch die Zuweisung von Referenzen auf andere Objekte dynamisch zur Laufzeit definiert. Hierfür ist es erforderlich, dass die Objekte ihre jeweiligen Schnittstellen anerkennen – was wiederum sorgfältig ausgestaltete Schnittstellen voraussetzt, die die Verwendung eines Objekts im Zusammenspiel mit vielen anderen Objekten nicht be- bzw. verhindern. Die Sache hat aber auch ihr Positives: Weil ausschließlich über ihre Schnittstellen auf die Objekte zugegriffen wird, wird die Kapselung nicht aufgebrochen. Jedes beliebige Objekt kann zur Laufzeit durch ein anderes ersetzt werden, solange dieses vom selben Typ ist. Und darüber hinaus ergeben sich bei der schnittstellenbasierten Implementierung eines Objekts auch erheblich weniger Implementierungsabhängigkeiten.

Zudem hat die Objektkomposition noch eine weitere Auswirkung auf das Systemdesign: Im Gegensatz zur Klassenvererbung erleichtert sie die Beibehaltung der Kapselung der Klassen sowie deren Ausrichtung auf eine Aufgabe. Die Klassen und Klassenhierarchien bleiben überschaubar und es ist unwahrscheinlicher, dass sie sich zu unkontrollierbaren Monstrositäten auswachsen. Andererseits weist ein auf der Objektkomposition basierendes Design mehr Objekte (wenn auch weniger Klassen) auf und das Systemverhalten ist von deren Wechselbeziehungen abhängig, statt in einer einzelnen Klasse definiert zu sein.

Und damit kommt nun auch der zweite Grundsatz des objektorientierten Designs zum Tragen:

Die Objektkomposition ist der Klassenvererbung vorzuziehen.

Idealerweise sollten zur Wiederverwendung vorhandener Klassen keine neuen Komponenten erzeugt werden müssen, sondern die gesamte benötigte Funktionalität

einfach durch Zusammenführen der verfügbaren Komponenten mittels Objektkomposition erzielt werden können. Allerdings gelingt das nur selten, weil die Menge der vorhandenen Komponenten in der Praxis meist schlicht nicht umfassend genug ist. Die Wiederverwendung durch Vererbung gestaltet es einfacher, neue Komponenten zu erzeugen, die mit älteren zusammengeführt werden können. Vererbung und Objektkomposition arbeiten also Hand in Hand.

Dennoch machen viele Entwickler erfahrungsgemäß zu ausgiebigen Gebrauch von der Vererbung als Wiederverwendungstechnik. Häufig lassen sich Designs durch eine verstärkte Anwendung der Objektkomposition einfacher und in höherem Maße wiederverwendbar gestalten – und dementsprechend kommt sie in den Design Patterns auch immer wieder zum Einsatz.

Delegation

Mithilfe der **Delegation** lässt sich die Objektkomposition als ebenso leistungsstarke Technik für die Wiederverwendung einsetzen wie die Vererbung [Lie86, JZ91]. Bei dieser Methode sind *zwei* Objekte an der Bearbeitung eines Requests beteiligt: Ein empfangendes Objekt delegiert Operationen an sein Delegationsobjekt, den sogenannten **Delegate**. Dieses Verfahren ist analog der Request-Weiterleitung der Unterklassen an die Basisklassen zu sehen, allerdings kann sich eine geerbte Operation bei der Vererbungstechnik immer auf das empfangende Objekt beziehen – in C++ über die `this`-Membervariable und in Smalltalk mittels `self`. Um mit der Delegation denselben Effekt zu erzielen, leitet der Empfänger eine Referenz auf sich selbst an den Delegate weiter, damit die weitergeleitete Operation auf den Empfänger verweist.

Statt beispielsweise die Klasse `Window` zu einer Unterklasse von `Rectangle` zu machen (weil Fenster nun mal rechteckig sind), könnte sie das Verhalten von `Rectangle` wiederverwenden, indem sie eine Instanzvariable `Rectangle` nutzt und das rechteckspezifische Verhalten an sie *delegiert*. Mit anderen Worten: Das Fenster wäre kein Rechteck, sondern würde ein Rechteck *enthalten*. In diesem Fall müsste die `Window`-Klasse Requests fortan explizit an ihre `Rectangle`-Instanz weiterleiten – im anderen Fall hätte sie diese Operationen geerbt.

[Abbildung 1.8](#) zeigt, wie die Klasse `Window` ihre Operation `Area()` an eine `Rectangle`-Instanz delegiert.

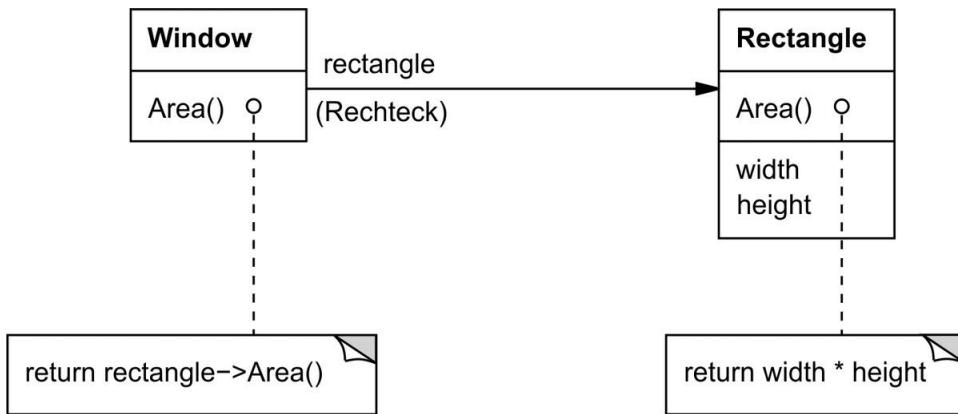


Abb. 1.8: Beispieldiagramm Delegation

Der Pfeil mit der durchgezogenen Linie symbolisiert, dass die Klasse eine Referenz auf eine Instanz einer anderen Klasse enthält. Diese Referenz kann optional benannt werden, hier beispielsweise mit `rectangle`.

Der wichtigste Vorteil der Delegation besteht darin, dass sie sowohl die Zusammenführung von Verhalten zur Laufzeit als auch die Änderung der Kompositionsstruktur erleichtert. So könnte man das Fenster in diesem Beispiel ohne Weiteres zur Laufzeit kreisförmig gestalten, indem man die `Rectangle`-Instanz durch eine `Circle`-Instanz ersetzt – Voraussetzung dafür wäre allerdings, dass `Rectangle` und `Circle` vom selben Typ sind.

Ebenso wie andere Techniken für eine flexiblere Softwaregestaltung, die sich die Objektkomposition zunutze machen, bringt aber auch die Delegation einen Nachteil mit sich: Dynamische, stark parametrisierte Software ist insgesamt schwieriger zu überblicken und zu verstehen als eher statisch orientierte Software. Abgesehen davon ist auch mit einigen Ineffizienzen hinsichtlich der Laufzeitperformance zu rechnen, wenngleich sich Ineffizienzen aufseiten des agierenden Menschen auf lange Sicht immer noch folgenschwerer auswirken. Die Anwendung des Delegationsprinzips ist immer dann eine gute Wahl, wenn dies zu einer Vereinfachung des Designs beiträgt, statt die Dinge zu verkomplizieren. Die Formulierung fester Regelvorgaben für den gezielten Einsatz der Delegation ist allerdings schwierig, denn wie effizient sie sein wird, hängt einerseits vom jeweiligen Kontext und andererseits von der Erfahrung des Entwicklers ab. Generell gilt jedoch, dass sie am besten funktioniert, wenn sie in ausgesprochen stilisierter Art und Weise angewendet wird – d. h. im Rahmen von Standard-Patterns.

Es gibt mehrere Design Patterns, die die Delegation standardmäßig nutzen. *State* (*Zustand*, siehe [Abschnitt 5.8](#)), *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#)) und *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)) basieren sogar vollständig auf diesem Konzept: Im

Design Pattern *State (Zustand)* leitet ein Objekt Requests an ein State-Objekt weiter, das den aktuellen Zustand des ursprünglichen Objekts repräsentiert. Beim Pattern *Strategy (Strategie)* delegiert ein Objekt einen spezifischen Request an ein anderes Objekt, das eine Strategie für die Ausführung des Requests bereithält – natürlich nimmt ein Objekt nur einen Zustand ein, trotzdem kann es mehrere Strategien für verschiedene Arten von Requests vorsehen. Der Zweck dieser beiden Design Patterns besteht darin, das Verhalten eines Objekts durch die Modifikation der Objekte zu manipulieren, an die es die Requests delegiert. Und im Fall des Patterns *Visitor (Besucher)* wird die Operation, die auf jedes Element der Objektstruktur angewendet wird, grundsätzlich an das visitor-Objekt delegiert.

Im Gegensatz zu den vorgenannten Beispielen machen andere Design Patterns weniger Gebrauch von der Delegation. So präsentiert das Pattern *Mediator (Vermittler*, siehe [Abschnitt 5.5](#)) ein Objekt zur Vermittlung der Kommunikation zwischen anderen Objekten. In manchen Fällen implementiert dieses Mediator-Objekt Operationen durch einfaches Weiterleiten an die übrigen Objekte, in anderen Fällen übergibt es allerdings zusätzlich noch eine Referenz auf sich selbst und wendet somit eine echte Form der Delegation an. Beim Pattern *Chain of Responsibility (Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) wird die Bearbeitung von Requests durch deren Weiterleitung innerhalb einer Objektkette von einem Objekt zum nächsten realisiert. Mitunter enthält ein solcher Request auch eine Referenz auf das Ausgangsobjekt, an das er ursprünglich gerichtet war – in diesem Fall nutzt das Pattern die Technik der Delegation. Und das Design Pattern *Bridge (Brücke*, siehe [Abschnitt 4.2](#)) entkoppelt schließlich eine Abstraktion von seiner Implementierung. Wenn die Abstraktion und eine bestimmte Implementierung eng aufeinander abgestimmt sind, kann die Abstraktion die Operationen einfach an diese Implementierung weiterleiten.

Die Delegation ist ein extremes Beispiel für die Objektkomposition, das zeigt, dass die Vererbung als Mechanismus für die Wiederverwendung von Code jederzeit durch die Objektkomposition ersetzt werden kann.

Vererbung kontra parametrisierte Typen

Eine weitere (nicht strikt objektorientierte) Technik zur Wiederverwendung von Funktionalität ist der Einsatz von **parametrisierten Typen**, auch als **Generics** (Ada, Eiffel) und **Templates** (C++) bekannt. Diese Verfahrensweise ermöglicht die Definition eines Typs ohne vorherige Spezifizierung sämtlicher anderen von ihm verwendeten Typen. Die unspezifizierten Typen werden zum Nutzungszeitpunkt als

Parameter bereitgestellt. So kann beispielsweise eine Klasse `List` durch den Typ der in ihr enthaltenen Elemente parametrisiert werden. Um eine Liste von Integerzahlen zu deklarieren, könnte dem parametrisierten Typ von `List` der Typ `Integer` als Parameter übergeben werden. Und zur Deklaration einer Liste mit `String`-Objekten würde der Typ `String` als Parameter verwendet. Die Sprachimplementierung erzeugt für jeden Elementtyp eine benutzerdefinierte Version des Templates der `List`-Klasse.

Parametisierte Typen stellen (neben der Klassenvererbung und der Objektkomposition) also eine dritte Methode für die Verhaltenskomposition in objektorientierten Systemen zur Verfügung. Viele Designs lassen sich mithilfe jeder dieser drei Techniken implementieren. Zur Parametrisierung einer Sortierroutine anhand der Operation zum Abgleichen der Elemente könnte man den Vergleich

1. als eine durch Unterklassen implementierte Operation definieren (unter Anwendung des Design Patterns *Template Method* (*Schablonenmethode*, siehe [Abschnitt 5.10](#))),
2. der Zuständigkeit eines Objekts zuweisen, das an die Sortierroutine übergeben wird (*Strategy* (*Strategie*, siehe [Abschnitt 5.9](#))) oder
3. als Argument eines C++-Templates oder Ada-Genericss deklarieren, das den Namen der Funktion zum Aufruf des Elementeabgleichs spezifiziert.

Diese Techniken unterscheiden sich in einigen wichtigen Punkten. Die Objektkomposition ermöglicht zwar die Änderung der Verhaltenskomposition zur Laufzeit, erfordert aber eine Dereferenzierung und kann weniger effizient sein. Die Vererbung gestattet neben der Bereitstellung von Standardimplementierungen für Operationen auch, dass diese von Unterklassen überschrieben werden. Und parametisierte Typen erlauben die Modifizierung der von einer Klasse nutzbaren Typen. Aber weder die Vererbung noch die parametrisierten Typen können zur Laufzeit verändert werden. Welcher Ansatz am besten geeignet ist, hängt von dem jeweiligen Design und den Einschränkungen in Bezug auf die Implementierung ab.

Auch wenn keins der in diesem Buch vorgestellten Design Patterns unmittelbar mit parametrisierten Typen zu tun hat, werden sie dennoch gelegentlich für die individuelle Anpassung der C++-Implementierung eines Design Patterns genutzt. In einer Programmiersprache wie Smalltalk, die beim Kompilieren keine Typüberprüfung vornimmt, werden die parametrisierten Typen allerdings überhaupt nicht benötigt.

1.6.6 Strukturen der Laufzeit und beim Kompilieren abstimmen

Die **Laufzeitstruktur** eines objektorientierten Programms lässt häufig kaum Ähnlichkeiten mit seiner **Codestruktur** erkennen. Die Codestruktur wird beim Kompilieren »eingefroren« – sie besteht aus Klassen in festen Vererbungsbeziehungen. Die Laufzeitstruktur eines Programms besteht hingegen aus einem Netzwerk miteinander kommunizierender Objekte, das sehr schnellen Änderungen unterworfen ist. Tatsächlich sind beide Strukturen größtenteils voneinander unabhängig. Der Versuch, die eine auf der Grundlage der anderen zu verstehen, kommt in etwa dem Versuch gleich, die Dynamik lebendiger Ökosysteme aus der statischen Pflanzen- und Tierartenbestimmung herzuleiten und umgekehrt.

Betrachten Sie einmal die Unterscheidung zwischen der **Aggregation** und der **Assoziation** von Objekten und wie verschieden sie sich beim Kompilieren bzw. zur Laufzeit darstellen. Aggregation bedeutet, dass ein Objekt ein anderes »besitzt« bzw. dafür verantwortlich ist. Im Allgemeinen spricht man davon, dass ein Objekt ein anderes Objekt *hat* bzw. Letzteres *Teil von ihm ist* (»*is-part-of*«-Beziehung). Dieses Prinzip bedingt auch, dass das aggregierte Objekt und dessen Besitzer identische Lebenszyklen haben.

Bei der Assoziation hat ein Objekt dagegen lediglich *Kenntnis* von einem anderen Objekt. Diese Beziehung wird oft auch als *Kennt-* oder *Benutzt-Beziehung* bezeichnet. Assoziierte Objekte können zwar wechselseitig Operationen abfragen, sind jedoch nicht füreinander verantwortlich. Die Assoziation repräsentiert gegenüber der Aggregation die schwächere Beziehung und beinhaltet eine wesentlich lockere Verknüpfung zwischen den Objekten.

Assoziationen werden in den Diagrammen in diesem Buch durch einen einfachen Pfeil mit durchgezogener Linie dargestellt. Aggregationen werden durch denselben Pfeil symbolisiert, der jedoch zusätzlich am Ausgangspunkt durch eine liegende Raute gekennzeichnet ist, wie in [Abbildung 1.9](#) zu sehen:



Abb. 1.9: Beispieldiagramm Aggregation

Aggregation und Assoziation sind leicht zu verwechseln, zumal sie häufig auf die gleiche Art und Weise implementiert werden. In Smalltalk referenzieren alle Variablen andere Objekte. Hier wird nicht zwischen Aggregation und Assoziation unterschieden. In C++ kann die Aggregation durch die Definition von

Membervariablen implementiert werden, die echte Objekte repräsentieren, in der Praxis wird sie jedoch meist eher durch Pointer (Zeiger) oder Referenzen auf Instanzen definiert. Die Assoziation wird ebenfalls durch Pointer und Referenzen implementiert.

Im Endeffekt werden diese beiden Konzepte in erster Linie anhand ihrer Zweckentsprechung und nicht anhand expliziter Sprachmechanismen unterschieden. Aber auch wenn der Unterschied zwischen Aggregation und Assoziation in der Struktur beim Kompilieren nur schwer zu erkennen sein mag, so ist er doch bedeutsam. Aggregationsbeziehungen treten in der Regel lediglich in geringer Zahl in Erscheinung und sind beständiger als Assoziationsbeziehungen. Diese werden hingegen häufiger erstellt, wieder aufgelöst und neu erstellt und existieren manchmal nur für die Dauer einer Operation. Außerdem sind Assoziationsbeziehungen dynamischer, wodurch sie im Quelltext schwieriger wahrzunehmen sind.

Angesichts dieser Ungleichheit zwischen den Strukturen zur Laufzeit und beim Kompilieren eines Programms wird deutlich, dass der Code keineswegs alles über die Funktions- bzw. Arbeitsweise eines Systems preisgibt. Die Laufzeitstruktur des Systems muss eher vom Entwickler statt von der Programmiersprache vorgegeben sein. Und auch die Beziehungen zwischen Objekten und deren Typen müssen sehr sorgfältig ausgestaltet werden, weil sie bestimmen, wie gut oder schlecht die Laufzeitstruktur ist.

Viele Design Patterns (insbesondere objektbasierte) treffen eine explizite Unterscheidung zwischen den Strukturen beim Kompilieren und zur Laufzeit. Die Patterns *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) und *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#)) dienen insbesondere der Errichtung komplexer Laufzeitstrukturen. Im *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)) finden sich Laufzeitstrukturen, die oftmals nur schwer zu verstehen sind, sofern man sich nicht ausreichend mit dem Pattern selbst auskennt. Und das Design Pattern *Chain of Responsibility* (*Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) fördert Kommunikationsmuster zutage, die im Rahmen der Vererbung nicht in Erscheinung treten. Generell gilt also, dass die Laufzeitstrukturen ohne eine umfassende Kenntnis des verwendeten Patterns nicht aus dem Quelltext ersichtlich sind.

1.6.7 Designänderungen berücksichtigen

Ein Höchstmaß an Wiederverwendbarkeit lässt sich nur durch das vorausschauende Einkalkulieren zukünftiger und/oder wechselnder Anforderungen erzielen, so dass

sich das Systemdesign bei Bedarf fortentwickeln kann.

Damit gewährleistet ist, dass ein System auch unter veränderten Bedingungen zuverlässig funktioniert, muss berücksichtigt werden, welche Anforderungen und Umstände im Laufe seines Lebenszyklus möglicherweise variieren könnten – denn ein Design, bei dem solche Überlegungen außer Acht gelassen werden, läuft stets Gefahr, relativ unvermittelt umfassend revidiert werden zu müssen. So könnten veränderte Rahmenbedingungen beispielsweise eine Neudefinition und Reimplementierung von Klassen, eine clientseitige Modifizierung oder auch erneute Testläufe erforderlich machen. Und die daraus resultierenden Designrevisionen betreffen im Allgemeinen viele Bestandteile eines Softwaresystems, so dass unerwartete Anpassungen an veränderte Gegebenheiten unweigerlich mit einem hohen (Kosten-)Aufwand verbunden sind.

Design Patterns tragen dazu bei, solche Situationen zu vermeiden, indem sie eine gewisse Anpassungsfähigkeit des Systems sicherstellen: Jedes Design Pattern verschafft einem Bestandteil der Systemstruktur mehr Spielraum für Variabilität, ohne dass andere Aspekte des Systems in Mitleidenschaft gezogen würden, und sorgt damit zugleich dafür, dass das System beim Eintreten entsprechender Veränderungen dennoch stabil bleibt.

Die folgenden Beispiele zeigen ein paar allgemeine Szenarien auf, die normalerweise eine Designrevision erforderlich machen würden, denen jedoch mit den jeweils genannten Design Patterns entgegengewirkt werden kann:

1. *Erzeugung eines Objekts durch explizite Klassenspezifizierung.* Die Spezifizierung eines Klassennamens während der Erstellung eines Objekts geht immer mit der Festlegung auf eine spezifische Implementierung statt auf eine spezifische Schnittstelle einher. Solche Verbindlichkeiten können im Fall zukünftiger Änderungen allerdings Komplikationen hervorrufen. Um dies zu vermeiden, sollte die Objekterzeugung deshalb lieber von vornherein indirekt erfolgen.

Design Patterns: *Abstract Factory* (*Abstrakte Fabrik*, s. [Abschnitt 3.1](#)), *Factory Method* (*Fabrikmethode*, s. [Abschnitt 3.3](#)), *Prototype* (*Prototyp*, s. [Abschnitt 3.4](#)).

2. *Abhängigkeit von spezifischen Operationen.* Bei der Spezifikation einer bestimmten Operation wird genau eine Möglichkeit festgelegt, einen Request zu beantworten. Die Vermeidung von hartkodierten Requests erleichtert hier die Beantwortung von Requests sowohl beim Kompilieren als auch zur Laufzeit.

Design Patterns: *Chain of Responsibility* (Zuständigkeitskette, s. [Abschnitt 5.1](#)), *Command* (Befehl, s. [Abschnitt 5.2](#)).

3. *Abhängigkeit von Hardware- und Softwareplattform.* Externe Betriebssystem- und Anwendungsprogrammierschnittstellen (APIs) funktionieren auf verschiedenen Hard- und Softwareplattformen jeweils unterschiedlich. Von einer bestimmten Plattform abhängige Software lässt sich nur schwer auf andere Plattformen portieren – mitunter bereitet schon die Aktualisierung auf der nativen Plattform Probleme. Deshalb sollte bereits beim Designentwurf darauf geachtet werden, dass ein System möglichst wenige Plattformabhängigkeiten aufweist.

Design Patterns: *Abstract Factory* (Abstrakte Fabrik, s. [Abschnitt 3.1](#)), *Bridge* (Brücke, s. [Abschnitt 4.2](#)).

4. *Abhängigkeit von Objektrepräsentationen oder -implementierungen.* Clients, denen die Art der Repräsentation, Speicherung, Lokalisierung und Implementierung eines Objekts bekannt ist, müssen im Fall einer Objektänderung gegebenenfalls entsprechend angepasst werden. Um eine kaskadenförmige Anhäufung der Änderungsmaßnahmen zu verhindern, sollten ihnen diese Informationen vorenthalten werden.

Design Patterns: *Abstract Factory* (Abstrakte Fabrik, s. [Abschnitt 3.1](#)), *Bridge* (Brücke, s. [Abschnitt 4.2](#)), *Memento* (Memento, s. [Abschnitt 5.6](#)), *Proxy* (Proxy, s. [Abschnitt 4.7](#)).

5. *Abhängigkeit von Algorithmen.* Algorithmen werden im Rahmen der Entwicklung und Wiederverwendung häufig erweitert, optimiert oder ersetzt. Und jede Änderung an einem Algorithmus macht auch eine Anpassung der von ihm abhängigen Objekte erforderlich. Aus diesem Grund sollten Algorithmen, die aller Wahrscheinlichkeit nach öfter modifiziert werden müssen, isoliert werden.

Design Patterns: *Builder* (Erbauer, s. [Abschnitt 3.2](#)), *Iterator* (Iterator, s. [Abschnitt 5.4](#)), *Strategy* (Strategie, s. [Abschnitt 5.9](#)), *Template Method* (Schablonenmethode, s. [Abschnitt 5.10](#)), *Visitor* (Besucher, s. [Abschnitt 5.11](#)).

6. *Enge Kopplung.* Eng miteinander verknüpfte Klassen lassen sich aufgrund ihrer gegenseitigen Abhängigkeit nur mit Mühe separat wiederverwenden. Solche engen Kopplungen haben monolithische Systeme zur Folge, in denen sich eine Klasse ohne genaue Kenntnis ihrer Beschaffenheit oder die gleichzeitige

Anpassung vieler weiterer Klassen nicht modifizieren oder entfernen lässt. Das System wird zu einem dichten Geflecht, das nur schwer zu durchschauen, zu portieren und zu warten ist.

Lose Kopplungen erhöhen dagegen die Wahrscheinlichkeit, dass eine Klasse eigenständig wiederverwendet und das System einfacher erlernt, portiert, modifiziert und erweitert werden kann. Design Patterns nutzen Techniken wie die abstrakte Kopplung und Schichtenmodelle, um lose verknüpfte Systeme zu fördern.

Design Patterns: *Abstract Factory* (Abstrakte Fabrik, s. [Abschnitt 3.1](#)), *Bridge* (Brücke, s. [Abschnitt 4.2](#)), *Chain of Responsibility* (Zuständigkeitskette, s. [Abschnitt 5.1](#)), *Command* (Befehl, s. [Abschnitt 5.2](#)), *Facade* (Fassade, s. [Abschnitt 4.5](#)), *Mediator* (Vermittler, s. [Abschnitt 5.5](#)), *Observer* (Beobachter, s. [Abschnitt 5.7](#)).

7. *Funktionalitätserweiterung durch Unterklassenbildung.* Die individuelle Objektgestaltung mithilfe der Unterklassenbildung ist oft nicht einfach. Jede neue Klasse geht mit einem fest vorgegebenen Implementierungsaufwand (Initialisierung, Finalisierung usw.) einher, und die Definition einer Unterklasse bedingt ein umfassendes Verständnis von der Beschaffenheit der Basisklasse. So könnte das Überschreiben einer Operation beispielweise das Überschreiben einer anderen erfordern. Oder eine überschriebene Operation muss eine geerbte Operation aufrufen. Außerdem kann die Unterklassenbildung zu einer regelrechten Klassenexplosion führen, weil selbst eine einfache Erweiterung die Erstellung zahlreicher neuer Unterklassen nach sich zieht.

Die Objektkomposition im Allgemeinen sowie die Delegation im Besonderen stellen in Bezug auf die Verhaltenskomposition flexible Alternativen zum Vererbungsprinzip dar. Um eine Anwendung mit neuer Funktionalität auszustatten, können statt der Definition neuer, auf bereits existierenden Klassen basierender Unterklassen auch neuartige Zusammenstellungen bereits vorhandener Objekte verwendet werden. Andererseits kann der intensive Einsatz der Objektkomposition jedoch dazu führen, dass das Design schwerer zu durchschauen ist. Viele Patterns begünstigen daher Softwaredesigns, die sich schlicht durch die Definition einer einzigen Unterklasse und deren Instanzenbildung auf der Grundlage existierender Objekte mit individueller Funktionalität ausstatten lassen.

Design Patterns: *Bridge* (Brücke, s. [Abschnitt 4.2](#)), *Chain of Responsibility*

(Zuständigkeitskette, s. [Abschnitt 5.1](#)), Composite (Kompositum, s. [Abschnitt 4.3](#)), Decorator (Dekorierer, s. [Abschnitt 4.4](#)), Observer (Beobachter, s. [Abschnitt 5.7](#)), Strategy (Strategie, s. [Abschnitt 5.9](#)).

8. *Unbequeme Klassenmodifikation.* Manche Klassen lassen sich nicht ganz so einfach modifizieren, etwa wenn dazu auf den Quelltext zurückgegriffen werden muss, der aber leider nicht verfügbar ist (was beispielsweise bei einer kommerziellen Klassenbibliothek der Fall sein kann). Oder wenn bei jeder Änderung an der Klasse gleichzeitig auch mehrere dazugehörige Unterklassen angepasst werden müssen. Design Patterns bieten hier komfortable Möglichkeiten, um die Modifikationen auch unter derartig widrigen Umständen bequem durchführen zu können.

Design Patterns: Adapter (*Adapter*, s. [Abschnitt 4.1](#)), Decorator (*Dekorierer*, s. [Abschnitt 4.4](#)), Visitor (*Besucher*, s. [Abschnitt 5.11](#)).

Die vorgenannten Fälle demonstrieren, wie sich die Softwareentwicklung mithilfe von Design Patterns flexibler gestalten lässt. Welchen Stellenwert diese höhere Flexibilität hat, hängt dabei natürlich von der Art der entwickelten Software ab. Im Folgenden wird die Rolle der Patterns in der Entwicklungsarbeit am Beispiel dreier sehr umfassender Softwaregattungen veranschaulicht: *Anwendungsprogramme*, *Toolkits* und *Frameworks*.

Anwendungsprogramme

Bei der Entwicklung eines **Anwendungsprogramms**, beispielsweise eines Texteditors oder einer Tabellenkalkulation, haben die *interne Wiederverwendbarkeit*, die *Wartbarkeit* sowie die *Erweiterbarkeit* hohe Priorität. Die interne Wiederverwendbarkeit gewährleistet, dass nicht mehr neu entwickelt und implementiert werden muss als nötig. Design Patterns, die eine Reduzierung der Abhängigkeiten sicherstellen, können die interne Wiederverwendung begünstigen. Und losere Kopplungen erhöhen die Wahrscheinlichkeit, dass eine Objektklasse mit mehreren anderen zusammenarbeiten kann. Werden beispielsweise die Abhängigkeiten von bestimmten Operationen durch die Isolierung und Kapselung jeder einzelnen Operation eliminiert, erleichtert dies die Wiederverwendung der Operation in unterschiedlichen Kontexten. Das Gleiche gilt für die Entfernung algorithmischer und repräsentativer Abhängigkeiten.

In ähnlicher Weise können Design Patterns auch zur Wartbarkeit einer Anwendung beitragen, wenn sie zur Einschränkung von Plattformabhängigkeiten sowie zur

Schichtenbildung im System genutzt werden. Ebenso verbessern sie die Erweiterbarkeit, indem sie aufzeigen, wie sich Klassenhierarchien ausdehnen lassen und in welcher Form die Objektkomposition instrumentalisiert werden kann. Und auch die reduzierte Kopplung ist der Erweiterbarkeit zuträglich, denn eine isolierte Klasse lässt sich einfacher erweitern, wenn sie nicht von zahlreichen anderen Klassen abhängig ist.

Toolkits

Oftmals werden in einer Anwendung Klassen verwendet, die aus einer oder mehreren Bibliotheken mit vordefinierten Klassen stammen, sogenannten **Toolkits**. Ein Toolkit besteht aus einem Satz verwandter und wiederverwendbarer Klassen, die nützliche, allgemein einsetzbare Funktionalität zur Verfügung stellen. Ein Beispiel für ein solches Toolkit wäre eine Reihe von Behälterklassen für Listen, assoziative Tabellen, Stacks und Ähnliches. Die **I/O-Streambibliothek für C++** ist ein weiteres Beispiel. Toolkits zwingen einer Anwendung kein bestimmtes Design auf, sondern stellen lediglich eine Funktionalität bereit, die der Anwendung bei der Erfüllung ihrer Aufgaben behilflich sein kann, so dass der Entwickler die grundsätzliche Funktionalität nicht jedes Mal erneut programmieren muss. Toolkits stellen die *Wiederverwendung von Code* in den Vordergrund. Sie sind das objektorientierte Gegenstück zu den Programmbibliotheken.

Das Toolkit-Design ist gegenüber dem Anwendungsdesign unter Umständen allerdings vergleichsweise schwieriger zu bewerkstelligen, weil Toolkits in vielen Anwendungen einsetzbar sein müssen, um tatsächlich von Nutzen zu sein. Erschwerend hinzu kommt auch, dass ein Toolkit-Autor nicht bereits im Voraus wissen kann, wie genau diese Anwendungen aussehen werden oder welche speziellen Anforderungen sie stellen. Deshalb ist es umso wichtiger, Annahmen und Abhängigkeiten, die die Flexibilität des Toolkits und somit auch dessen Anwendbarkeit und Effektivität einschränken könnten, zu vermeiden.

Frameworks

Ein **Framework** ist eine Sammlung kooperierender Klassen, die ein wiederverwendbares Design für eine spezifische Softwaregattung bilden [Deu89, JF88]. So kann ein Framework beispielsweise speziell für die Erstellung von Grafikeditoren für verschiedene Anwendungsbereiche wie das künstlerische Zeichnen, die Musikkomposition oder mechanisches CAD gedacht sein [VL90,

Joh92]. Ein anderes Framework könnte dagegen die Entwicklung von Compilern für verschiedene Programmiersprachen und Zielsysteme stützen [JML92]. Und wieder ein anderes Framework könnte auf die Erzeugung von Anwendungen für Finanzmodelle abzielen [BE93]. Die individuelle Ausrichtung eines Frameworks auf eine bestimmte Anwendung erfolgt durch die Bildung anwendungsspezifischer Unterklassen der abstrakten Framework-Klassen.

Das Framework diktiert die Architektur der Anwendung. Es definiert ihre übergeordnete Struktur, die Klassen- und Objektunterteilung, deren Hauptzuständigkeiten, die Art und Weise, wie die Klassen und Objekte zusammenarbeiten, sowie den Programmablauf. Im Prinzip liefert es also eine Vorabdefinition dieser Designparameter, so dass sich der Entwickler auf die detaillierte Ausgestaltung der Anwendung konzentrieren kann. Frameworks erfassen sämtliche für den Einsatzbereich der jeweiligen Anwendung allgemein gültigen Designentscheidungen – und legen damit mehr Gewicht auf die *Wiederverwendung des Designs* als auf die Wiederverwendung von Code, wenngleich sie generell auch konkrete Unterklassen enthalten, die unmittelbar genutzt werden können.

Die Wiederverwendung auf dieser Ebene führt zu einer Kontrollumkehr zwischen der Anwendung und der Software, auf der sie basiert. Beim Einsatz eines *Toolkits* (oder auch einer herkömmlichen Programmbibliothek) wird der Hauptteil der Anwendung individuell geschrieben und der wiederzuverwendende Code aufgerufen. Beim Einsatz eines *Frameworks* wird hingegen der Hauptteil wiederverwendet und der Code, den das *Framework* aufruft, selbst geschrieben. Es müssen also zwar Operationen mit bestimmten Bezeichnungen und Aufrufkonventionen erstellt werden, dennoch sind insgesamt weniger Designentscheidungen zu treffen.

Dadurch lassen sich Anwendungen nicht nur schneller erstellen, sondern weisen auch stets ähnliche Strukturen auf – was wiederum zu einer leichteren Wartbarkeit und einer größeren Konsistenz führt. Ein wenig kreative Freiheit geht bei dieser Verfahrensweise aber natürlich schon verloren, weil ja viele Designentscheidungen bereits im Voraus getroffen wurden.

Während die Entwicklung von Anwendungen schon schwierig und die von Toolkits noch schwieriger ist, ist die Erstellung von Frameworks am schwierigsten zu bewerkstelligen. Ein Framework-Entwickler setzt darauf, dass eine einzige Architektur für alle Applikationen eines bestimmten Anwendungsbereichs geeignet ist. Daraus folgt aber auch, dass jede tiefgreifende Änderung an einem Framework-Design dessen Vorteile erheblich mindert, weil der wichtigste Beitrag, den ein

Framework für die Anwendung leistet, nun mal die in ihm definierte Architektur ist. Aus diesem Grund ist eine möglichst flexible und erweiterbare Gestaltung des Frameworks ein Muss.

Weil das Anwendungsdesign in so hohem Maße von dem Framework abhängig ist, sind Modifikationen an den Framework-Schnittstellen ganz besonders heikel. Anwendungen müssen sich der Weiterentwicklung eines Frameworks anpassen und sich entsprechend mit fortentwickeln. In dieser Hinsicht kommt den losen Kopplungen eine besondere Bedeutung zu, denn ohne sie hätten schon lediglich geringfügige Modifikationen am Framework erhebliche Auswirkungen zur Folge.

Die vorstehend beschriebenen Designprobleme spielen für das Framework-Design eine äußerst wichtige Rolle. Ein Framework, in dem diese Faktoren durch den Einsatz entsprechender Design Patterns gebührend berücksichtigt werden, wird in Bezug auf die Design- und Codewiederverwendung in aller Regel ein deutlich höheres Niveau erreichen als eins, das dies nicht tut. Ausgereifte Frameworks machen sich normalerweise gleich mehrere Design Patterns zunutze, die die Eignung ihrer Architektur für möglichst viele verschiedene Anwendungen sicherstellen, ohne dass eine Designrevision erforderlich wird.

Ein zusätzlicher Vorteil kommt zum Tragen, wenn die verwendeten Design Patterns auch in der zum Framework gehörigen Dokumentation beschrieben sind [BJ94]. Dadurch erschließt sich die genaue Beschaffenheit des Frameworks deutlich leichter, so dass selbst Entwickler, denen die Patterns zuvor noch nicht geläufig waren, von der Struktur profitieren können, die sie der Framework-Dokumentation verleihen. Eine fortlaufende Verbesserung der zugehörigen Dokumentationen ist generell für alle Softwaregattungen von Bedeutung – ganz besonders gilt dies jedoch für Frameworks, da diese häufig mit einer recht steil ansteigenden Lernkurve einhergehen, die zunächst einmal bewältigt werden will, bevor eine sinnvolle Nutzung möglich ist. Natürlich sind auch Design Patterns nicht in der Lage, diese Lernkurve vollständig zu entschärfen, sie können sie aber durch die ausdrückliche Betonung der Schlüsselemente des Framework-Designs erheblich abflachen.

Angesichts der Tatsache, dass Design Patterns und Frameworks viele Ähnlichkeiten aufweisen, stellen sich Entwickler nicht selten die Frage, worin bzw. ob sie sich überhaupt unterscheiden. Grundsätzlich gibt es drei wesentliche Unterscheidungsmerkmale:

1. *Design Patterns sind abstrakter als Frameworks.* Während sich Frameworks als gebrauchsfertiger Code ausdrücken lassen, können Patterns lediglich in

Form von *Codebeispielen* dargestellt werden. Eine Stärke der Frameworks besteht darin, dass sie in den gewünschten Programmiersprachen geschrieben und somit nicht nur gelesen, sondern auch unmittelbar ausgeführt und wiederverwendet werden können. Die in diesem Buch vorgestellten Design Patterns haben im Gegensatz dazu hauptsächlich die Aufgabe, den Zweck, die Vor- und Nachteile sowie die Auswirkungen eines Designs zu veranschaulichen und müssen für jeden Einsatz individuell implementiert werden.

2. *Design Patterns sind kleinere architektonische Elemente als Frameworks.* Ein typisches Framework kann mehrere Design Patterns umfassen, der umgekehrte Fall ist jedoch nicht möglich.
3. *Design Patterns weisen eine geringere Spezialisierung auf als Frameworks.* Frameworks zielen immer auf einen bestimmten Anwendungsbereich ab. So könnte ein Framework für Grafikeditoren zwar *in* einer Fertigungssimulation eingesetzt werden, aber nicht *als* Simulationsframework. Im Gegensatz dazu können die Design Patterns aus diesem Katalog für nahezu alle Arten von Anwendungen genutzt werden. Natürlich gibt es darüber hinaus auch weitaus spezialisiertere Patterns (beispielsweise für verteilte Systeme oder parallele Programmierung), doch selbst diese schreiben die Anwendungsarchitektur nicht in dem Maße vor, wie es ein Framework tut.

Frameworks gewinnen mehr und mehr an Bedeutung und werden immer alltäglicher. Sie sind das Mittel der Wahl, wenn es darum geht, das größtmögliche Wiederverwendungspotenzial in objektorientierten Systemen auszuschöpfen. Umfassendere und aufwendigere objektorientierte Anwendungen werden zukünftig zunehmend aus mehreren miteinander kooperierenden Framework-Schichten bestehen – und auch deren Design und Code wird in weiten Teilen den verwendeten Frameworks entstammen oder zumindest von ihnen beeinflusst sein.

1.7 Auswahl eines Design Patterns

Da der in diesem Buch vorgestellte Katalog mehr als 20 Design Patterns enthält, kann es mitunter schwer fallen, genau das Pattern zu finden, das für ein bestimmtes Designproblem geeignet ist – dies gilt insbesondere für Entwickler, die den Katalog vorher noch nicht kannten und dementsprechend auch nicht mit ihm vertraut sind. Ein wenig Hilfestellung bieten diesbezüglich die nachfolgend beschriebenen Ansätze:

- Erwägen Sie, in welcher Art und Weise Design Patterns Designprobleme lösen. [Abschnitt 1.6](#) erläutert, inwiefern die Design Patterns Ihnen behilflich sein können, passende Objekte zu finden, die Objektgranularität zu bestimmen und Objektschnittstellen zu spezifizieren. Darüber hinaus werden aber auch diverse andere Möglichkeiten aufgezeigt, wie sie sich zur Lösung von Designproblemen anwenden lassen. Diese Informationen können sich bei Ihrer Suche nach dem richtigen Pattern als sehr hilfreich erweisen.
- Machen Sie sich mit dem Zweck der Design Patterns vertraut. [Abschnitt 1.4](#) enthält kompakte Beschreibungen der Zielsetzungen aller im Katalog enthaltenen Design Patterns, die Ihnen das Auffinden der für eine bestimmte Problemstellung relevanten Patterns erleichtern. Nutzen Sie das Klassifizierungsschema aus [Tabelle 1.1](#), um Ihre Suche einzuschränken.
- Beachten Sie, in welcher Beziehung die Patterns zueinander stehen. Die in [Abbildung 1.2](#) dargestellte Grafik verdeutlicht die verwandtschaftlichen Beziehungen zwischen den einzelnen Design Patterns und bietet wertvolle Hilfestellung, um die richtigen Patterns bzw. den richtigen Patterns-Satz für die jeweils vorliegende Problematik zu finden.
- Untersuchen Sie Design Patterns, die ähnliche Zwecke erfüllen. Der Katalog (siehe [Kapitel 3](#)) ist in drei Kategorien unterteilt: die **Erzeugungsmuster**, die **Strukturmuster** und die **Verhaltensmuster**. Jeder Kategorienteil beginnt mit einer Einleitung zu den behandelten Design Patterns und schließt mit einer vergleichenden Übersicht, aus der sowohl die Ähnlichkeiten als auch die Unterschiede zwischen den einzelnen Patterns ersichtlich sind.
- Ermitteln Sie die Gründe für eine Um- bzw. Neugestaltung des Designs. Vergleichen Sie das vorliegende Designproblem mit den im [Abschnitt 1.6.7](#) beschriebenen häufigen Ursachen für Designrevisionen und prüfen Sie, ob die hier vorgeschlagenen Design Patterns Ihnen bei der Vermeidung einer Um- bzw. Neugestaltung des Designs von Nutzen sein können.
- Prüfen Sie, welche Elemente in Ihrem Design variabel sein sollten. Bei diesem Ansatz wird der entgegengesetzte Weg zur Ermittlung der Gründe für eine Designrevision eingeschlagen: Statt zu berücksichtigen, wodurch eine Designänderung *erzwungen* werden könnte, steht hierbei die Überlegung im Vordergrund, was geändert werden kann, *ohne dass gleich das ganze Design revidiert werden muss*. In diesem Fall liegt der Fokus auf der *Kapselung des variierenden Konzepts*, einem zahlreichen Design Patterns zugrunde liegenden Leitmotiv. [Tabelle 1.2](#) enthält eine Auflistung der Designaspekte, die mithilfe

von Design Patterns unabhängig voneinander variiert und somit modifiziert werden können, ohne dass dafür eine Um- bzw. Neugestaltung des Designs erforderlich ist.

Zweck	Design Pattern	Variierbare Aspekte
Erzeugungsmuster (Creational Patterns)	<i>Abstract Factory</i> (<i>Abstrakte Fabrik</i> , siehe Abschnitt 3.1)	Produktobjektfamilien
	<i>Builder</i> (<i>Erbauer</i> , siehe Abschnitt 3.2)	Erzeugung eines zusammengesetzten Objekts
	<i>Factory Method</i> (<i>Fabrikmethode</i> , siehe Abschnitt 3.3)	Unterklasse eines instanzierten Objekts
	<i>Prototype</i> (<i>Prototyp</i> , siehe Abschnitt 3.4)	Klasse eines instanzierten Objekts
	<i>Singleton</i> (<i>Singleton</i> , siehe Abschnitt 3.5)	Einige Instanz einer Klasse
Strukturmuster (Structural Patterns)	<i>Adapter</i> (<i>Adapter</i> , siehe Abschnitt 4.1)	Schnittstelle zu einem Objekt
	<i>Bridge</i> (<i>Brücke</i> , siehe Abschnitt 4.2)	Implementierung eines Objekts
	<i>Composite</i> (<i>Kompositum</i> , siehe Abschnitt 4.3)	Struktur und Komposition eines Objekts

	<i>Decorator</i> (<i>Dekorierer</i> , siehe Abschnitt 4.4)	Zuständigkeiten eines Objekts ohne Unterklassenbildung
	<i>Facade</i> (<i>Fassade</i> , siehe Abschnitt 4.5)	Schnittstelle zu einem Subsystem
	<i>Flyweight</i> (<i>Fliegengewicht</i> , siehe Abschnitt 4.6)	Speicherkosten für Objekte
	<i>Proxy</i> (<i>Proxy</i> , siehe Abschnitt 4.7)	Zugriff auf ein Objekt sowie dessen Speicherort
Verhaltensmuster (Behavioral Patterns)	<i>Chain of Responsibility</i> (<i>Zuständigkeitskette</i> , siehe Abschnitt 5.1)	Objekt, das einen Request bearbeiten kann
	<i>Command</i> (<i>Befehl</i> , siehe Abschnitt 5.2)	Zeitpunkt und Art der Ausführung eines Requests
	<i>Interpreter</i> (<i>Interpreter</i> , siehe Abschnitt 5.3)	Grammatik und Interpretation einer Sprache
	<i>Iterator</i> (<i>Iterator</i> , siehe Abschnitt 5.4)	Zugriff auf und Traversierung von Elementen eines Aggregats
	<i>Mediator</i> (<i>Vermittler</i> , siehe Abschnitt 5.5)	Interagierende Objekte sowie Art und Weise der Interaktion

<i>Memento</i> (<i>Memento</i> , siehe Abschnitt 5.6)	Art und Zeitpunkt der Speicherung privater Informationen außerhalb eines Objekts
<i>Observer</i> (<i>Beobachter</i> , siehe Abschnitt 5.7)	Anzahl der von einem anderen Objekt abhängigen Objekte sowie Aktualisierungsart der abhängigen Objekte
<i>State</i> (<i>Zustand</i> , siehe Abschnitt 5.8)	Zustände eines Objekts
<i>Strategy</i> (<i>Strategie</i> , siehe Abschnitt 5.9)	Ein Algorithmus
<i>Template Method</i> (<i>Schablonenmethode</i> , siehe Abschnitt 5.10)	Schritte eines Algorithmus
<i>Visitor</i> (<i>Besucher</i> , siehe Abschnitt 5.11)	Ohne Klassenänderung auf Objekte anwendbare Operationen

Tabelle 1.2: Mithilfe von Design Patterns variierbare Designaspekte

1.8 Anwendung eines Design Patterns

Nach der Auswahl eines Patterns stellt sich die Frage, wie es benutzt werden kann. Die nachfolgende Schritt-für-Schritt-Anleitung beschreibt, wie Sie ein Design Pattern effizient anwenden können:

1. *Lesen Sie die Beschreibung des Design Patterns, um sich einen Gesamtüberblick zu verschaffen.* Achten Sie dabei insbesondere auf die Ausführungen zur »Anwendbarkeit« und den »Konsequenzen«, damit

gewährleistet ist, dass das Pattern auch tatsächlich für das Ihnen vorliegende Problem geeignet ist.

2. Überprüfen Sie die Angaben zur »Struktur«, den »Teilnehmern« und den »Interaktionen«. Machen Sie sich mit den Klassen und Objekten vertraut und ergründen Sie, in welcher Beziehung sie zueinander stehen.
3. Machen Sie sich den »Beispielcode« zunutze. Das unter dieser Überschrift präsentierte konkrete Beispiel leistet Ihnen Hilfestellung bei der Implementierung des gewählten Design Patterns im Quelltext.
4. Geben Sie den Patterns-Teilnehmern aussagefähige Namen mit einem Bezug zum Anwendungskontext. Die am Design Patterns beteiligten Objekte oder Klassen (Teilnehmer) sind normalerweise zu abstrakt benannt, als dass sie direkt in einer Anwendung verwendet werden könnten. Dennoch ist es sinnvoll, dass sich die Teilnehmernamen in den Bezeichnungen widerspiegeln, die auch in der Anwendung erscheinen, damit das Pattern in der Implementierung leichter zu erkennen ist. Wenn Sie also beispielsweise das Design Pattern *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#)) für einen Textformatierungsalgorithmus nutzen, könnten Sie die zugehörigen Klassen mit `SimpleLayoutStrategy` oder `TeXLayoutStrategy` benennen.
5. Definieren Sie die Klassen. Deklarieren Sie ihre Schnittstellen, richten Sie ihre Vererbungsbeziehungen ein und definieren Sie die Instanzvariablen, die die Daten- und Objektreferenzen repräsentieren. Identifizieren Sie die in Ihrer Anwendung existierenden Klassen, auf die sich das Pattern auswirken wird, und modifizieren Sie sie entsprechend.
6. Definieren Sie anwendungsspezifische Bezeichnungen für Operationen in dem Design Pattern. Auch in diesem Fall ist die Namensgebung im Prinzip von der Anwendung abhängig. Orientieren Sie sich dabei an den mit den einzelnen Operationen verknüpften Zuständigkeiten und Interaktionen. Achten Sie außerdem darauf, Ihre Namenskonventionen konsequent und konsistent anzuwenden, indem Sie zum Beispiel durchgängig das Präfix `Create` nutzen, um das Pattern *Factory Method* (*Fabrikmethode*, siehe [Abschnitt 3.3](#)) anzuzeigen.
7. Implementieren Sie die Operationen zur Umsetzung der Zuständigkeiten und Interaktionen im Design Pattern. Die Ausführungen zur »Implementierung« der einzelnen Design Patterns liefern Ihnen ebenso wie der jeweils zugehörige »Beispielcode« wertvolle Hinweise zur Durchführung des

Implementierungsvorgangs.

Die vorgenannten Richtlinien sollen Ihnen lediglich den Einstieg in die Arbeit mit den Design Patterns erleichtern, bis Sie im Laufe der Zeit Ihre eigene, individuelle Verfahrensweise entwickelt haben.

Aber natürlich wären unsere Erläuterungen zum Umgang mit den Design Patterns nicht vollständig, wenn wir Ihnen nicht auch ein paar warnende Worte dazu mit auf den Weg geben würden, wie Sie sie *nicht* einsetzen sollten. Generell gilt, dass Patterns nicht wahllos angewendet werden sollten. Ihre Flexibilität und Variabilität wird häufig erst durch die Ergänzung zusätzlicher Dereferenzierungen erreicht, die das Design um einiges komplexer machen oder zu Performanceeinbußen führen können. Deshalb empfiehlt es sich, ein Pattern wirklich nur dann anzuwenden, wenn die dadurch erreichte Flexibilität auch tatsächlich benötigt wird. Hilfestellung beim Abwägen der Vor- und Nachteile sowie zur Wirkweise eines Patterns finden Sie im jeweils zugehörigen »Konsequenzen«-Abschnitt.

Kapitel 2: Fallstudie: Erstellung eines Texteditors

Dieses Kapitel dokumentiert eine Fallstudie zur Designentwicklung eines »What-You-See-Is-What-You-Get« (oder kurz »WYSIWYG«)-Texteditors namens **Lexi**. Sie demonstriert verschiedene Problemstellungen, die sich im Design von Lexi und ähnlich gearteten Anwendungen ergeben können, und zeigt auf, wie sie sich mithilfe von Design Patterns beheben lassen. Insgesamt werden zu diesem Zweck acht verschiedene Patterns eingesetzt und anhand konkreter, nachvollziehbarer Beispiele ausführlich erläutert.

Hinweis

Das Design von Lexi basiert auf dem von Paul Calder entwickelten Texteditor »Doc« [CL92].

[Abbildung 2.1](#) zeigt die Benutzeroberfläche des Texteditors Lexi. Der zentrale rechteckige Anzeigebereich des Bildschirms ist für die WYSIWYG-Darstellung des Dokuments reserviert. Hier können die gewünschten Text- und Grafikelemente beliebig zusammengestellt und mit verschiedenen Formatierungsstilen ausgestattet werden. An den Bildschirmrändern befinden sich die üblichen Pulldown-Menüs und Scrollleisten sowie eine Reihe von Seitensteuerungssicons, die den gezielten Aufruf einer bestimmten Dokumentseite ermöglichen.

2.1 Designprobleme

Im Folgenden werden sieben Problemstellungen des Lexi-Designs betrachtet:

1. *Dokumentstruktur.* Die Wahl der internen Darstellung des Dokuments beeinflusst nahezu jeden Aspekt des Anwendungsdesigns. Sämtliche Bearbeitungs-, Formatierungs- und Anzeigekräfte sowie Textanalysen erfordern eine Traversierung, also das systematische Durchlaufen aller Strukturelemente der Oberflächendarstellung – insofern wirkt sich die Art und

Weise, wie diese Informationen organisiert sind, auch auf das Design der übrigen Bestandteile von Lexi aus.

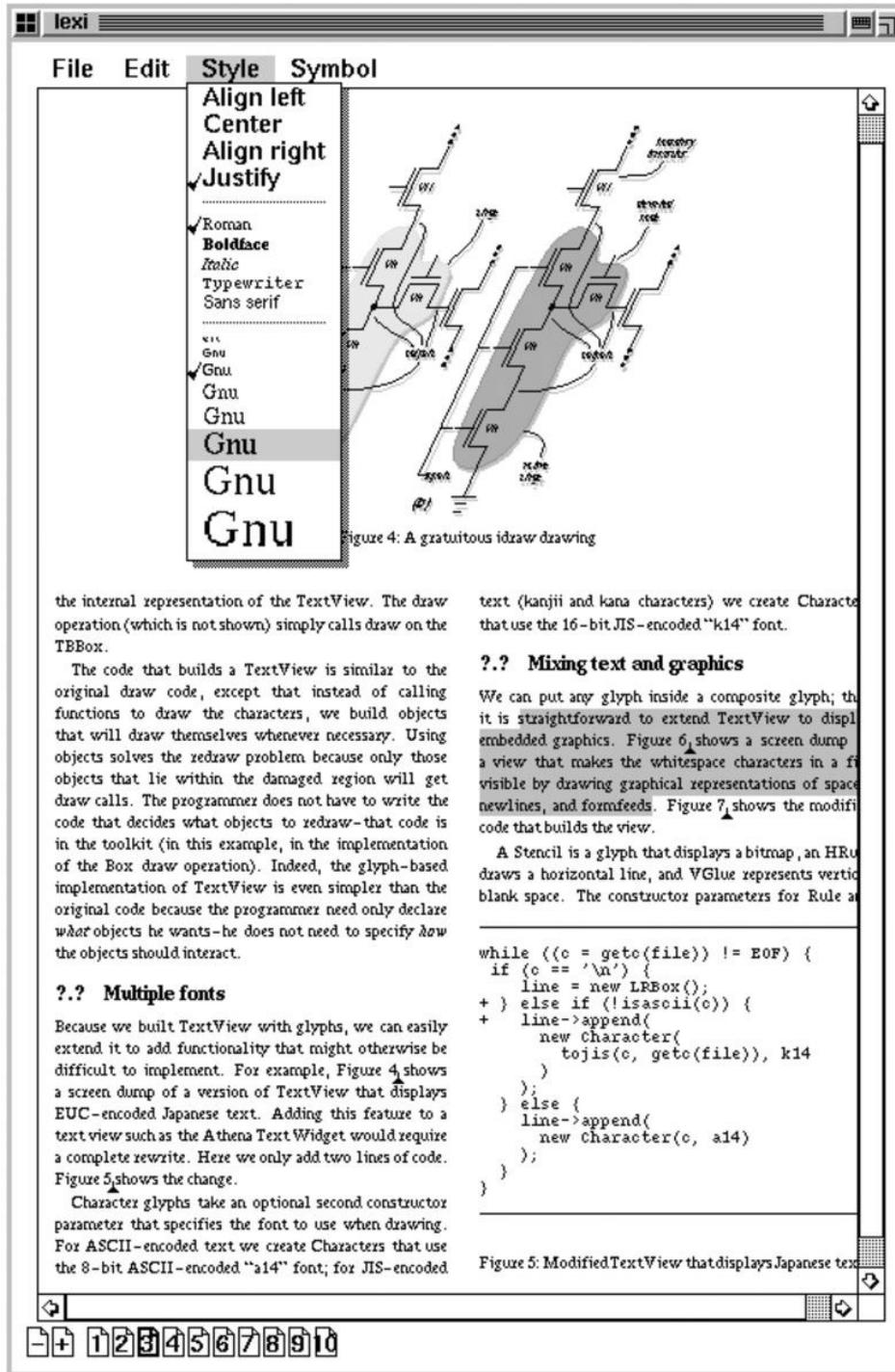


Abb. 2.1: Die Benutzeroberfläche des Texteditors »Lexi«

2. *Formatierung.* Wie genau ordnet Lexi die Texte und Grafiken eigentlich in Zeilen und Spalten an? Welche Objekte sind für die Umsetzung der diversen Formatierungsrichtlinien zuständig? Und wie interagieren diese Richtlinien mit

der internen Darstellung des Dokuments?

3. *Ausgestaltung der Benutzeroberfläche.* Die Benutzeroberfläche von Lexi umfasst Elemente wie Scrollleisten, Rahmen und Schlagschatten, die zur optischen Gestaltung der WYSIWYG-Oberfläche zur Verfügung stehen und während des Designprozesses in der Regel recht häufig variiert werden. Deshalb müssen sie so problemlos wie möglich hinzugefügt und entfernt werden können, ohne dass die übrigen Bestandteile der Anwendung davon beeinträchtigt werden.
4. *Unterstützung mehrerer Look-and-Feel-Standards.* Die Lexi-Benutzeroberfläche sollte sich möglichst leicht und ohne größere modifizierende Eingriffe an verschiedene Look-and-Feel-Standards wie Motif oder Presentation Manager (PM) anpassen lassen.
5. *Unterstützung verschiedener Fenstersysteme.* Unterschiedliche Look-and-Feel-Standards werden im Allgemeinen auf verschiedenen Fenstersystemen implementiert. Das Lexi-Design sollte daher möglichst neutral und weitgehend unabhängig von einem bestimmten Fenstersystem gestaltet sein.
6. *Userseitige Vorgänge.* Die User steuern Lexi mithilfe verschiedener Elemente, die auf der Benutzeroberfläche zur Verfügung stehen, wie z. B. Schaltflächen und Pulldown-Menüs, deren Funktionalität sich über die in der Anwendung verfügbaren Objekte verteilt. Die Herausforderung hierbei besteht in der Bereitstellung eines einheitlichen Mechanismus sowohl für den Zugriff auf diese verteilte Funktionalität als auch für das Rückgängigmachen ihrer Auswirkungen.
7. *Rechtschreibprüfung und Silbentrennung.* In welcher Form unterstützt Lexi analytische Operationen wie beispielsweise die Rechtschreibprüfung oder die Silbentrennung? Wie lässt sich die Anzahl der Klassen minimieren, die zur Ergänzung einer neuen analytischen Funktion geändert werden müssen?

Jede dieser Designfragen impliziert neben einer Reihe von Zielsetzungen auch feste Rahmenbedingungen für deren Realisierung. Deshalb werden diese beiden Faktoren im Folgenden zunächst eingehend analysiert und als Grundlage für den Entwurf eines geeigneten Lösungsvorschlags herangezogen. Anschließend werden dann die für die einzelnen Problemstellungen und deren Lösungen jeweils relevanten Design Patterns kurz vorgestellt.

2.2 Dokumentstruktur

Ein Dokument stellt im Prinzip nichts anderes dar, als eine spezifische Anordnung von grundlegenden grafischen Elementen wie Zeichen, Linien, Polygonen und anderen Formen. Diese Elemente spiegeln alle inhaltlichen Informationen des Dokuments wider. Autoren begreifen sie allerdings nicht als grafische Bausteine, sondern als physische Struktur des Dokuments – also als Zeilen, Spalten, Abbildungen, Tabellen und andere Substrukturen. Und diese Substrukturen besitzen wiederum eigene Substrukturen usw.

Hinweis

Die Verfasser eines Dokuments orientieren sich überwiegend an dessen *logischer* Struktur, d. h. an seiner Unterteilung in Sätze, Absätze, Abschnitte und Kapitel. Der Einfachheit halber werden in der internen Darstellung dieses Beispiels jedoch keine spezifischen Daten zur logischen Struktur gespeichert. Grundsätzlich ist die hier beschriebene Designlösung allerdings auch für die Repräsentation derartiger Informationen geeignet.

Im Fall von Lexi soll die Benutzeroberfläche den Usern die direkte Bearbeitung der Substrukturen gestatten. So sollen sie zum Beispiel in der Lage sein, ein Diagramm in seiner Gesamtheit und nicht als eine Ansammlung einzelner grafischer Primitive zu behandeln. Ebenso soll auch eine Tabelle als Ganzes referenziert werden können und nicht als eine unstrukturierte Ansammlung von Text und Grafiken – dadurch bleibt die Benutzeroberfläche überschaubar und intuitiv. Um die Lexi-Implementierung mit derartigen Eigenschaften auszustatten, wird in diesem Fallbeispiel eine interne Darstellung gewählt, die der physischen Struktur des Dokuments entspricht.

Insbesondere soll sie folgende Anforderungen erfüllen:

- Die physische Struktur des Dokuments, d. h. die Anordnung von Text und Grafiken in Zeilen, Spalten, Tabellen etc., bleibt erhalten.
- Das Dokument wird visuell generiert und präsentiert.
- Die Mapping-Koordinaten der einzelnen Elemente in der internen Darstellung werden am Bildschirm angezeigt. Sie gestatten Lexi zu bestimmen, worauf sich

der User bezieht, wenn er auf ein Bildschirmelement zeigt.

Darüber hinaus sind außerdem einige Rahmenbedingungen zu berücksichtigen. Zum einen soll ein einheitlicher Umgang sowohl mit Text als auch mit Grafiken möglich sein, damit die User Gelegenheit haben, Text beliebig in Grafiken einzubetten und umgekehrt. Grafiken sind also nicht als Spezialfall von Text und Text ist nicht als Spezialfall von Grafiken zu behandeln – denn das hätte redundante Formatierungs- und Manipulationsmechanismen zur Folge. Stattdessen soll hier lediglich ein einziger Mechanismus für Text *und* Grafiken genügen.

Und zum anderen soll die Implementierung in der internen Darstellung *nicht* zwischen einzelnen Elementen und Elementgruppen unterscheiden müssen. Vielmehr soll Lexi die einheitliche Behandlung von einfachen sowie von komplexen Elementen und somit beliebig vielschichtige Dokumente ermöglichen. So könnte beispielsweise das zehnte Element in Zeile 5 der Spalte 2 ein einzelnes Zeichen oder auch ein aufwendiges Diagramm mit vielen Unterelementen sein. Solange gewährleistet ist, dass sich dieses Element eigenständig zeichnen und seine Dimensionen selbst spezifizieren kann, hat seine Komplexität keinen Einfluss darauf, wie und wo es auf der Seite erscheinen wird.

Dieser zweiten Rahmenbedingung steht jedoch die Notwendigkeit entgegen, dass z. B. Textpassagen innerhalb des Dokuments auf Rechtschreibfehler, potenzielle Silbentrennstellen und Ähnliches hin überprüft werden müssen. Oftmals spielt es dabei keine Rolle, ob es sich bei dem untersuchten Element um ein einfaches oder ein komplexes Objekt handelt, in einigen Fällen sind derartige Analysen allerdings unmittelbar von der Beschaffenheit des Objekts abhängig. So wäre es beispielsweise nicht sinnvoll, ein Polygon auf Rechtschreibfehler oder Silbentrennstellen zu überprüfen. Deshalb sollten solche wie auch andere potenziell konflikträchtige Rahmenbedingungen stets von vornherein im Design der internen Darstellung berücksichtigt werden.

2.2.1 Rekursive Komposition

Eine allgemein gebräuchliche Technik für die Darstellung hierarchisch strukturierter Informationen ist die **rekursive Komposition**, die darauf basiert, dass zunehmend komplexe Elemente auf einfacheren Elementen aufbauen. Sie ermöglicht die Zusammenstellung eines Dokuments aus simplen grafischen Elementen. Dazu könnten im ersten Schritt zum Beispiel einige Zeichen- und Grafikkacheln von links nach rechts in einer Zeile im Dokument angeordnet werden, und anschließend werden dann mehrere Zeilen zu einer Spalte

zusammengefasst, mehrere Spalten zu einer Seite angeordnet usw. (siehe [Abbildung 2.2](#)).

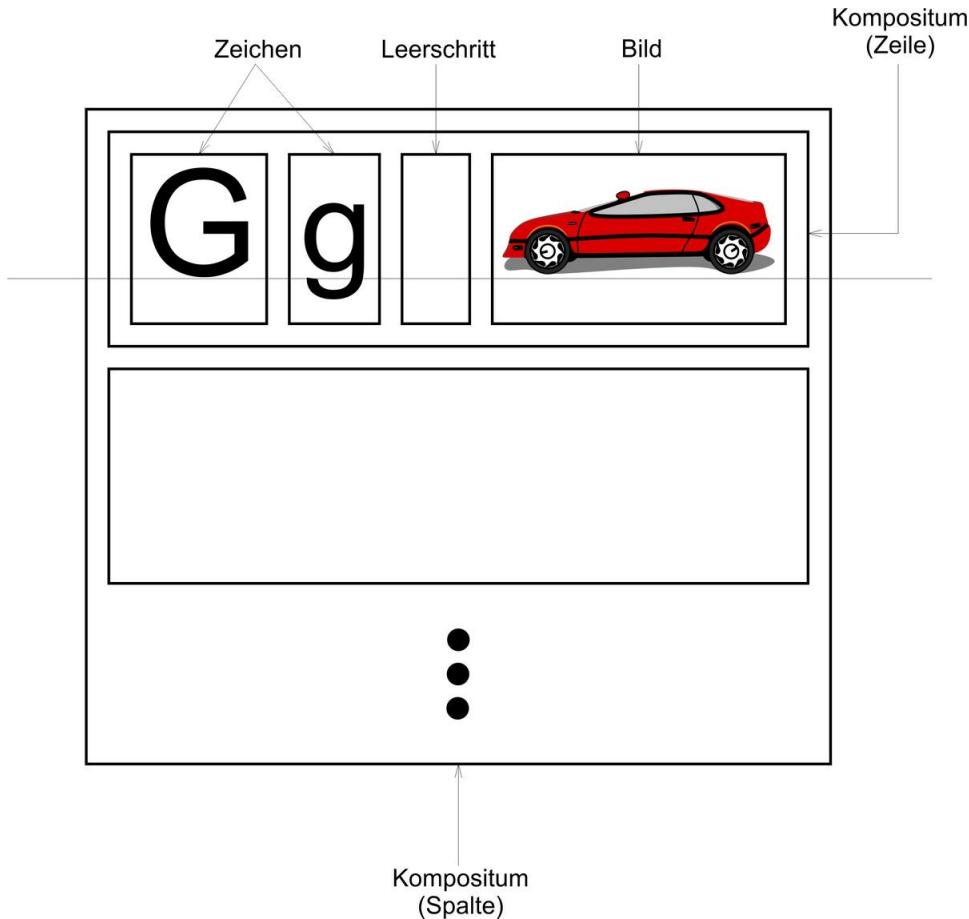


Abb. 2.2: Rekursive Komposition von Text und Grafik

Zur Bereitstellung der physischen Struktur wird jedem wichtigen Element je ein eigenes Objekt gewidmet, das nicht nur die sichtbaren Elemente wie die Zeichen und Grafiken, sondern auch die nicht sichtbaren strukturellen Elemente umfasst, also die Zeilen und Spalten. Auf diese Weise entsteht eine Objektstruktur, die z. B. der in [Abbildung 2.3](#) dargestellten ähneln könnte:

Die Verwendung eigener Objekte für jedes Zeichen- und Grafikelement im Dokument fördert die Flexibilität auf den feinkörnigsten Ebenen des Lexi-Designs. Sie gewährleistet den einheitlichen Umgang mit Text und Grafiken in Bezug auf deren Zeichnung, Formatierung sowie wechselseitige Einbettung und erleichtert außerdem die Ergänzung neuer Zeichensätze in Lexi, ohne dass die übrige Funktionalität beeinträchtigt wird. Die Objektstruktur des Texteditors gleicht einem Abbild der physischen Struktur des Dokuments.

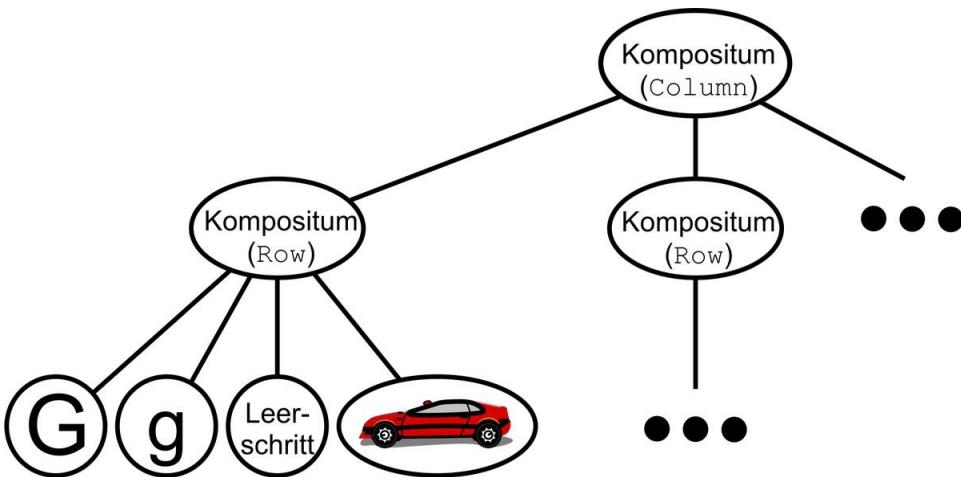


Abb. 2.3: Objektstruktur für die rekursive Komposition von Text und Grafik

Dieser Ansatz impliziert zwei wichtige Maßgaben: Zum einen müssen die Objekte korrespondierende Klassen besitzen. Und zum anderen müssen diese Klassen, was weniger offenkundig ist, kompatible Schnittstellen aufweisen, damit die Objekte einheitlich behandelt werden können. In einer Programmiersprache wie C++ wird eine solche Schnittstellenkompatibilität erreicht, indem die Klassen durch Vererbung in Beziehung zueinander gesetzt werden.

2.2.2 Glyphen

Der nächste Schritt besteht in der Definition einer abstrakten Klasse `Glyph` für alle Objekte, die in einer Dokumentstruktur in Erscheinung treten können. Ihre Unterklassen definieren sowohl primitive grafische Elemente (wie Zeichen und Bilder) als auch strukturelle Elemente (wie Zeilen und Spalten). [Abbildung 2.4](#) veranschaulicht dieses Prinzip am Beispiel eines repräsentativen Ausschnitts der `Glyph`-Klassenhierarchie.

Hinweis

Der Begriff »Glyph« (dt. »Glyphe«) wurde im Kontext der Designentwicklung von Benutzeroberflächen erstmals von Calder geprägt [CL90]. In den meisten modernen Texteditoren werden – mutmaßlich aus Effizienzgründen – keine eigenen Objekte für jedes einzelne Zeichen verwendet. Calder führt in seiner Dissertation jedoch den Nachweis, dass ein eben solcher Ansatz tragfähig ist [Cal93]. Die in diesem Buch vorgestellten Glyphen sind vergleichsweise simpel gehalten und auf strikte Hierarchien beschränkt. Calders Glyphen können

dagegen zwecks Einsparung von Speicherkapazität gemeinsam genutzt werden und bilden dadurch gerichtete azyklische Graphenstrukturen. Derselbe Effekt lässt sich auch durch die Anwendung des Design Patterns *Flyweight* (*Fliegengewicht*, siehe [Abschnitt 4.6](#)) erzielen – dies auszuprobieren, bleibt dem geneigten Leser an dieser Stelle allerdings als Übung überlassen.

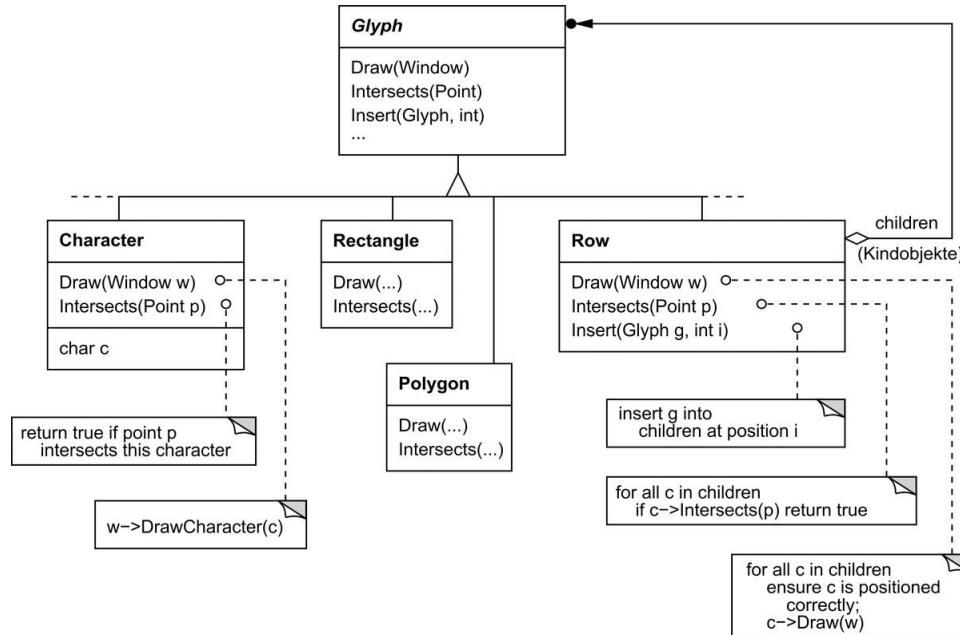


Abb. 2.4: Auszug aus der `Glyph`-Klassenhierarchie

Die nachfolgende [Tabelle 2.1](#) zeigt eine detaillierte Übersicht der `Glyph`-Klassenschnittstelle in der C++-Notation:

Zuständigkeit	Operationen
Darstellung	<code>virtual void Draw(Window*)</code> <code>virtual void Bounds(Rect&)</code>
Kollisionsabfrage	<code>virtual bool Intersects(const Point&)</code>
Struktur	<code>virtual void Insert(Glyph*, int)</code> <code>virtual void Remove(Glyph*)</code>

```
virtual Glyph* Child(int)  
virtual Glyph* Parent()
```

Tabelle 2.1: Grundlegende *Glyph*-Schnittstelle

Hinweis

Die Schnittstelle in diesem Beispiel ist der Einfachheit halber bewusst minimalistisch gehalten. Eine vollständige Schnittstelle würde auch Operationen zur Verwaltung grafischer Attribute wie Farbe, Schriftart und Koordinatentransformationen sowie für die erweiterte Verwaltung von Kindobjekten enthalten.

Jede Glyphe »weiß«,

1. wie sie sich selbst zu zeichnen hat,
2. wie viel Platz sie benötigt und
3. welche Eltern- und Kindobjekte sie besitzt.

Die `Glyph`-Unterklassen definieren die `Draw`-Operation neu, um sich selbst in einem Fenster darzustellen. Beim Aufruf der `Draw`-Operation wird ihnen dann eine Referenz auf ein `Window`-Objekt übergeben. Die Klasse `Window` definiert wiederum grafische Operationen zur Darstellung von Text und grundlegenden Formen in einem Bildschirmfenster. Eine `Rectangle`-Unterklasse von `Glyph` könnte `Draw` beispielsweise wie folgt umdefinieren:

```
void Rectangle::Draw(Window* w) {  
    w->DrawRect (_x0, _y0, _x1, _y1);  
}
```

wobei `_x0`, `_y0`, `_x1` und `_y1` Membervariablen von `Rectangle` sind, die zwei gegenüberliegende Ecken des Rechtecks spezifizieren, während `DrawRect` die `Window`-Operation darstellt, die das Rechteck auf den Bildschirm zeichnet.

Eine Elternglyphe muss in der Regel »wissen«, wie viel Platz eine Kindglyphe

belegt, um sie beispielsweise überlappungsfrei neben weiteren Glyphen einreihen zu können (siehe [Abbildung 2.2](#)). Die Bounds-Operation gibt die rechteckige Fläche zurück, die die Glyphe beansprucht, bzw. die gegenüberliegenden Ecken des kleinsten Rechtecks, in das die Glyphe noch hineinpasst. Glyph-Unterklassen definieren diese Operation so um, dass sie die rechteckige Fläche zurückgeben, in der gezeichnet wird.

Die Intersects-Operation gibt an, ob ein bestimmter Punkt die Glyphe überschneidet. Wann immer der User irgendwo in das Dokument klickt, ruft Lexi diese Operation auf, um zu ermitteln, welche Glyphe bzw. Glyph-Struktur sich unter dem Mauszeiger befindet. Die Rectangle-Klasse definiert diese Operation zur Berechnung der Überschneidung des Rechtecks und des übergebenen Punkts neu.

Da Glyphen Kindobjekte besitzen können, wird eine allgemeine Schnittstelle zum Hinzufügen, Entfernen sowie zum Zugriff darauf benötigt. Im Fall einer Glyphe Row sind z. B. alle Glyphen, die die Schnittstelle in einer Zeile anordnet, deren zugehörige Kindobjekte. Die Insert-Operation fügt eine Glyphe an einer durch einen Integerindex bezeichneten Position ein. Und die Remove-Operation entfernt eine spezifizierte Glyphe, sofern es sich dabei um ein Kindobjekt handelt.

Hinweis

Ein Integerindex ist – je nachdem, welche Datenstruktur die Glyphe verwendet – nicht immer unbedingt der beste Weg, um die Kindobjekte einer Glyphe zu spezifizieren. Sollten die Kindobjekte in einer verketteten Liste gespeichert sein, wäre ein Pointer auf ein Element dieser Liste effizienter. Eine bessere Lösung für das Indizierungsproblem wird in den Ausführungen zur Dokumentprüfung in [Abschnitt 2.8](#) vorgestellt.

Die child-Operation gibt das Kindobjekt (sofern vorhanden) an den betreffenden Index zurück. Glyphen wie Row, die Kindobjekte besitzen, sollten child intern verwenden, statt direkt auf die Datenstruktur des Kindobjekts zuzugreifen. Auf diese Weise erübrigen sich Modifikationen an Operationen wie Draw, die über die Kindobjekte iterieren, sobald die Datenstruktur beispielsweise von einem Array in eine verkettete Liste geändert wird. In ähnlicher Weise bietet Parent eine Standardschnittstelle zum Elternobjekt der Glyphe, sofern ein solches vorhanden ist. Im Fall von Lexi speichern die Glyphen eine Referenz auf ihr Elternobjekt, die dann einfach von ihrer Parent-Operation zurückgegeben wird.

2.2.3 Das Design Pattern *Composite (Kompositum)*

Neben der Strukturierung von Dokumentinhalten lässt sich das Konzept der rekursiven Komposition aber noch weitaus vielseitiger einsetzen – zum Beispiel zur Darstellung beliebig komplexer hierarchischer Strukturen. Besonders deutlich werden die maßgeblichen Aspekte der rekursiven Komposition im objektorientierten Sinn im Design Pattern *Composite (Kompositum*, siehe [Abschnitt 4.3](#)) – deshalb sei schon hier im Vorgriff auf [Kapitel 4](#) darauf hingewiesen, dass es sich durchaus lohnt, diesem Pattern etwas genauere Beachtung zu schenken und es zu Übungszwecken einmal auf das bisher geschilderte Szenario anzuwenden.

2.3 Formatierung

Nachdem im vorigen Abschnitt erläutert wurde, wie sich die physische Struktur eines Dokuments *darstellen* lässt, besteht der nächste Schritt darin zu ergründen, wie man eine *spezifische* physische Struktur realisieren kann – und zwar eine, die einem korrekt formatierten Dokument entspricht. *Darstellung* und *Formatierung* sind zwei verschiedene Dinge, denn die physische Struktur eines Dokuments abbilden zu können, bedeutet noch lange nicht, dass auch klar ist, wie am Ende eine spezifische Beschaffenheit dieser Struktur erreicht wird. Diese Aufgabe bleibt größtenteils der Anwendung überlassen – in diesem Fall hat also Lexi dafür Sorge zu tragen, die maßgeblichen Useranforderungen zu erfüllen und den Text entsprechend in Zeilen umzubrechen, die Zeilen in Spalten zu unterteilen usw. So könnte der User beispielsweise die Randbreiten, die Einzüge oder die Tabulatorpositionen verändern, einen einfachen oder einen doppelten Zeilenabstand vorgeben und noch viele weitere Formatierungseinstellungen variieren wollen. Deshalb müssen all diese Möglichkeiten im Umkehrschluss natürlich auch im Formatierungsalgorithmus von Lexi Berücksichtigung finden.

Hinweis

Erfahrungsgemäß wollen die User häufig sogar noch sehr viel weitreichendere Eingriffe in die *logische* Struktur des Dokuments vornehmen – die Sätze, die Absätze, die Kapitel etc. Im Vergleich dazu ist die *physische* Struktur lediglich von untergeordnetem Interesse. Den meisten Betrachtern ist es relativ egal, wo die Zeilenumbrüche in einem Absatz platziert sind, solange der Absatz insgesamt nur ordentlich formatiert ist. Das Gleiche gilt auch für die Spalten- und

Seitenformatierung. Insofern spezifizieren die User lediglich die übergeordneten Rahmenbedingungen für die physische Struktur und überlassen dann dem Texteditor Lexi die schwierige Aufgabe der Umsetzung.

Im Rahmen dieses Buches bezeichnet der Begriff »Formatierung« übrigens die Einteilung einer Ansammlung von Glyphen in Zeilen – die Begriffe »Formatierung« und »Zeilenumbruch« werden hier also synonym verwendet. Die vorgestellten Techniken sind gleichermaßen sowohl für das Aufbrechen von Zeilen in Spalten als auch das Aufbrechen von Spalten in Seiten geeignet.

2.3.1 Kapselung des Formatierungsalgorithmus

Eine Automatisierung des Formatierungsprozesses ist aufgrund der vielen damit einhergehenden Rahmenbedingungen und Details nicht so einfach. Es gibt zahlreiche Lösungsansätze für diese Problematik – und dementsprechend auch eine vielfältige Auswahl an Formatierungsalgorithmen mit individuellen Stärken und Schwächen. Da es sich im Fall von Lexi um einen WYSIWYG-Editor handelt, steht hier die richtige Balance zwischen Formatierungsqualität und Formatierungsgeschwindigkeit im Vordergrund: Generell sollte eine möglichst schnelle Antwortzeit des Editors gewährleistet sein, ohne dass dadurch Qualitätseinbußen am Erscheinungsbild des Dokuments entstehen. Diese Balancefindung hängt allerdings von vielen Faktoren ab, die nicht alle schon beim Kompilieren zweifelsfrei ermittelt werden können. So könnte beispielsweise eine geringfügig verzögerte Antwortzeit für die User durchaus akzeptabel sein, wenn im Gegenzug eine bessere Formatierung gewährleistet ist. Und damit wäre dann womöglich ein völlig anderer als der aktuell verwendete Formatierungsalgorithmus die bessere Alternative. Bei einem eher implementierungsorientierten Ansatz stünde dagegen eine gute Balance zwischen Formatierungsgeschwindigkeit und Speicherplatzanforderungen im Vordergrund – vielleicht ließe sich die Formatierungszeit durch die Zwischenspeicherung weiterer Daten reduzieren.

Da Formatierungsalgorithmen im Allgemeinen recht komplex sind, wäre es außerdem wünschenswert, sie angemessen zu kapseln oder – noch besser – völlig von der Dokumentstruktur abzukoppeln. Idealerweise sollte sich eine neuartige Glyph-Unterkategorie absolut unabhängig vom Formatierungsalgorithmus ergänzen lassen. Ebenso sollte auch die Verwendung eines neuen Algorithmus keine Modifikationen an bereits existierenden Glyphen erforderlich machen.

Die Erfüllung dieser Anforderungen bedingt, dass das Lexi-Design den problemlosen Austausch des Formatierungsalgorithmus zumindest beim Kompilieren, wenn nicht sogar auch zur Laufzeit gestatten muss. Dies lässt sich durch die Kapselung des Algorithmus in einem Objekt bewerkstelligen, die dafür sorgt, dass er isoliert und damit problemlos auszutauschen ist. Im Prinzip wird also eine separate Klassenhierarchie für Objekte erstellt, die Formatierungsalgorithmen kapseln. Die Wurzel dieser Hierarchie definiert eine Schnittstelle, die eine Vielzahl von Formatierungsalgorithmen unterstützt. Und diese Schnittstelle wird zwecks Ausführung eines bestimmten Algorithmus von jeder Unterkasse implementiert. Anschließend kann dann eine `Glyph`-Unterkasse angelegt werden, die ihre Kindobjekte unter Einsatz eines gegebenen Algorithmusobjekts automatisch strukturiert.

2.3.2 Die Unterklassen Compositor und Composition

Im Folgenden wird eine `compositor`-Klasse für Objekte definiert, die einen Formatierungsalgorithmus kapseln kann. Die Schnittstelle (siehe [Tabelle 2.2](#)) teilt dem `compositor` mit, welche Glyphen genau formatiert werden sollen und wann die Formatierung stattfinden soll. Grundsätzlich handelt es sich bei den zu formatierenden Glyphen um Kindobjekte einer speziellen `Glyph`-Unterkasse namens `Composition`.

Zuständigkeit	Operationen
Gegenstand der Formatierung	<code>void SetComposition(Composition*)</code>
Zeitpunkt der Formatierung	<code>virtual void Compose()</code>

Tabelle 2.2: Grundlegende `compositor`-Schnittstelle

Unmittelbar bei der Erstellung einer solchen `Composition`-Klasse wird ihr eine Instanz einer `Compositor`-Unterkasse zugewiesen, die auf einen bestimmten Zeilenumbruchalgorithmus spezialisiert ist und den `Compositor` anweist, bei Bedarf – beispielsweise wenn der User eine Änderung an einem Dokument vornimmt – eine `Compose`-Operation auf seine Glyphen auszuführen. [Abbildung 2.5](#) stellt die Beziehungen zwischen den `Composition`- und `Compositor`-Klassen in einer Übersicht

dar:

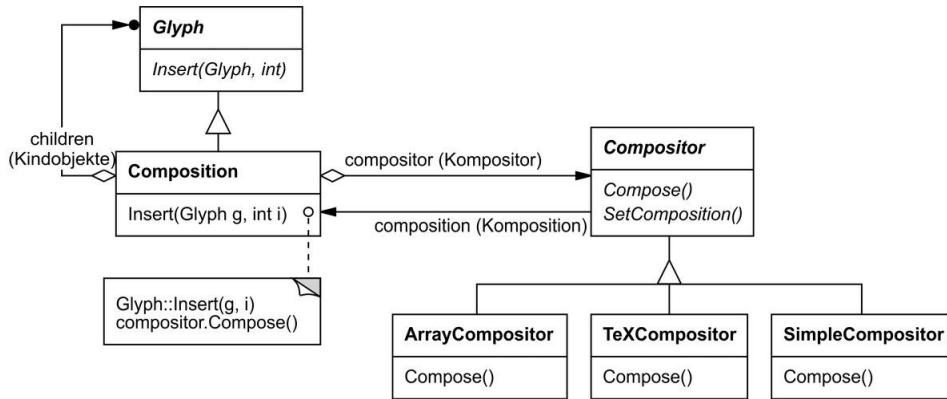


Abb. 2.5: Beziehungen zwischen *Composition*- und *Compositor*-Klassen

Ein unformatiertes **Composition**-Objekt enthält ausschließlich die sichtbaren Glyphen, die den grundlegenden Inhalt des Dokuments repräsentieren – nicht jedoch Glyphen, die zur Bestimmung der physischen Struktur des Dokuments beitragen, wie z. B. dessen Unterteilung in Zeilen und Spalten. Derartige Objekte werden nach ihrer Erzeugung mit den zu formatierenden Glyphen initialisiert und rufen dann zu dem Zeitpunkt, an dem die Komposition formatiert werden soll, die `Compose`-Operation seines **Compositor**-Objekts auf. Daraufhin iteriert der **Compositor** über die Kindobjekte der **Composition**-Klasse und fügt seinem Zeilenumbruchalgorithmus entsprechende neue Row- und column-Glyphen für die Zeilen- und Spaltengliederung ein.

Hinweis

Das **Compositor**-Objekt muss zur Berechnung der Zeilenumbrüche die Zeichencodes der **Character**-Glyphen abfragen. Wie sich diese Daten polymorph abrufen lassen, ohne dass die **Glyph**-Schnittstelle um eine zeichenspezifische Operation ergänzt werden muss, ist im [Abschnitt 2.8](#) beschrieben.

Die daraus resultierende Objektstruktur ist in [Abbildung 2.6](#) dargestellt. (Der besseren Übersicht halber sind die vom **Compositor** erzeugten und in die Objektstruktur eingefügten Glyphen hier durch einen grauen Hintergrund gekennzeichnet.)

Jede **Compositor**-Unterklasse kann einen anderen Zeilenumbruchalgorithmus

implementieren. So könnte beispielsweise eine Klasse `SimpleCompositor` die Komposition nur oberflächlich durchlaufen, ohne ästhetische Aspekte wie die »Farbe« des Dokuments zu berücksichtigen. (Eine stimmige Farbe bedeutet eine gleichmäßige Verteilung von Text und Leerraum.) Eine `TeXCompositor`-Klasse würde hingegen den vollständigen TeX-Algorithmus [Knu84] implementieren, der Eigenschaften wie die Farbe mit berücksichtigt, dafür jedoch längere Formatierungszeiten in Anspruch nimmt.

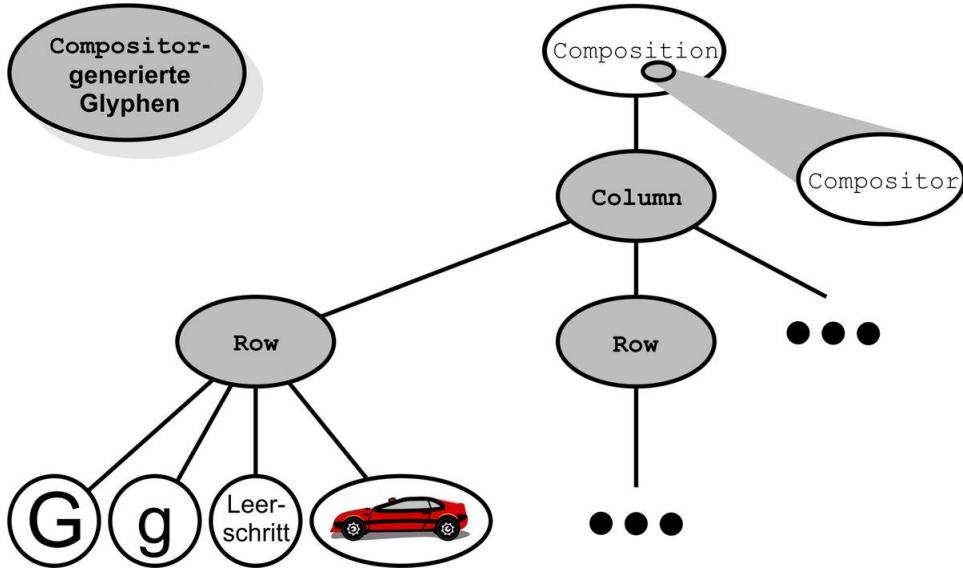


Abb. 2.6: Objektstruktur für *compositor*-gesteuerte Zeilenumbrüche

Die Separierung der `Compositor`- und `Composition`-Klassen gewährleistet eine klare Trennung zwischen dem Code, der die physische Struktur des Dokuments stützt, und dem Code, der für unterschiedliche Formatierungsalgorithmen zuständig ist. Neue `Compositor`-Unterklassen lassen sich somit vollkommen unabhängig von den `Glyph`-Klassen ergänzen und umgekehrt. Sogar die Zeilenumbruchalgorithmen können auf diese Weise zur Laufzeit modifiziert werden, indem eine einzelne `SetCompositor`-Operation zur grundlegenden `Glyph`-Schnittstelle der Komposition hinzugefügt wird.

2.3.3 Das Design Pattern *Strategy (Strategie)*

Der Zweck des Design Patterns *Strategy (Strategie)*, siehe [Abschnitt 5.9](#)) besteht in der Kapselung eines Algorithmus in einem Objekt. Die maßgeblichen Teilnehmer dieses Patterns sind die Strategieobjekte (die verschiedene Algorithmen kapseln) und der Kontext, in dem sie operieren. Compositors sind Strategien – sie kapseln verschiedene Formatierungsalgorithmen. Und das `Composition`-Objekt liefert den

Kontext für eine Compositor-Strategie.

Ein Kernaspekt für den Einsatz des Patterns *Strategy (Strategie)* ist die Erstellung von Schnittstellen, die zwar auf die Strategie und deren Kontext ausgerichtet, aber trotzdem noch universell genug sind, um eine gewisse Bandbreite von Algorithmen unterstützen zu können. Es sollte also nicht nötig sein, die Strategie- oder Kontextschnittstelle zwecks Unterstützung eines neuen Algorithmus modifizieren zu müssen. In dem hier verwendeten Beispiel ist die Unterstützung der grundlegenden Glyph-Schnittstelle für den Zugriff auf sowie das Hinzufügen und Entfernen von Kindobjekten hinreichend allgemeingültig, um den Compositor-Unterklassen die Änderung der physischen Dokumentstruktur zu gestatten – und zwar unabhängig von dem verwendeten Algorithmus. Ebenso versorgt die Compositor-Schnittstelle die Kompositionen mit allem, was sie zur Initialisierung der Formatierung benötigen.

2.4 Ausgestaltung der Benutzeroberfläche

Im Folgenden werden zwei Maßnahmen zur Ausgestaltung der Lexi-Benutzeroberfläche betrachtet. Als Erstes wird der für die Bearbeitung des Textinhalts zur Verfügung stehende Bereich mit einem Rahmen versehen, um die Dokumentseite kenntlich zu machen. Und im zweiten Fall werden Scrollleisten ergänzt, die es den Usern ermöglichen, verschiedene Bereiche der Seite anzusteuern bzw. einzusehen. Um das Hinzufügen und Entfernen dieser Gestaltungselemente zu erleichtern (insbesondere zur Laufzeit), sollten sie allerdings nicht mittels Vererbung in die Benutzeroberfläche implementiert werden. Die größtmögliche Flexibilität lässt sich hier vielmehr erreichen, wenn die anderen Elemente bzw. Objekte der Benutzeroberfläche erst gar keine Kenntnis von solchen gestalterischen Maßnahmen haben – denn dadurch können sie beliebig hinzugefügt und wieder entfernt werden, ohne dass andere Klassen modifiziert werden müssen.

2.4.1 Durchsichtige Umhüllung (Transparent Enclosure)

Aus programmiertechnischer Sicht wird die Ausgestaltung der Benutzeroberfläche durch die Erweiterung des bereits vorhandenen Codes um die gewünschten gestalterischen Elemente realisiert. Werden derartige Codeergänzungen allerdings mittels Vererbung hinzugefügt, ist eine Neuanordnung der Gestaltungselemente zur Laufzeit nicht möglich. Und auch der explosive zahlenmäßige Anstieg der genutzten Klassen stellt beim vererbungsbasierten Ansatz ein schwerwiegendes Problem dar.

So könnte die Klasse `Composition` beispielsweise um eine Unterklasse `BorderedComposition` erweitert werden, die die Benutzeroberfläche mit einem Rahmen versieht. In ähnlicher Weise ließe sich mit einer Unterklasse `ScrollableComposition` eine Scrollfunktion ergänzen. Alternativ könnte auch eine umfassendere Unterklasse `BorderedScrollableComposition` erstellt werden, die gleich beide Elemente hinzufügt – und immer so weiter. Im Extremfall kann somit für jedes gewünschte Ausgestaltungselement bzw. für jede Elementekombination eine eigene Klasse hinzukommen. Dadurch wird der Code jedoch schnell sehr unübersichtlich und ist kaum noch zu handhaben.

Die Objektkomposition bietet hier einen deutlich praktikableren und flexibleren Mechanismus zur Ausgestaltung der Benutzeroberfläche an. Aber welche Objekte sollten in einer Komposition zusammengefasst werden? Nun, da es sich in diesem Beispiel um eine Erweiterung einer bereits existierenden Glyphe handelt, könnte die gewünschte Ausgestaltungsmaßnahme als ein eigenes Objekt behandelt werden – beispielsweise in Form einer Instanz der Klasse `Border`. Damit stünden zwei mögliche Kandidaten für die Komposition zur Verfügung: die Glyphe und der Rahmen. Als Nächstes müsste dann entschieden werden, wie die Komposition erfolgen soll. Auch hier gibt es wieder zwei Möglichkeiten: Zum einen könnte der Rahmen die Glyphe enthalten, was insofern logisch wäre, als er sie auch auf dem Bildschirm umgibt. Zum anderen wäre aber auch der umgekehrte Fall denkbar, sprich dass die Glyphe den Rahmen enthält – dann müssten allerdings entsprechende Modifikationen an der `Glyph`-Unterklasse vorgenommen werden, damit diese überhaupt Kenntnis von dem Rahmen erhält. Bei der erstgenannten Möglichkeit, also der Einbindung der Glyphe in den Rahmen, verbliebe der Code zum Zeichnen des Rahmens dagegen vollständig in der `Border`-Klasse, ohne dass irgendwelche anderen Klassen davon betroffen wären.

Doch wie sieht die `Border`-Klasse eigentlich genau aus? Angesichts der Tatsache, dass ein Rahmen ein individuelles Aussehen hat, erscheint es naheliegend, dass er selbst eine Glyphe bzw. dass `Border` eine Unterklasse von `Glyph` sein sollte. Davon abgesehen gibt es aber noch einen triftigeren Grund, auf diese Weise zu verfahren: Die Clients sollten gar nicht erst zwischen Glyphen mit und ohne Rahmen unterscheiden müssen – sondern alle Glyphen gleich behandeln können. Wenn ein Client eine einfache, rahmenlose Glyphe anweist, sich selbst zu zeichnen, sollte sie dies ohne irgendwelche Ausgestaltungselemente tun. Und ist eine Glyphe von einem Rahmen umschlossen, dann sollten die Clients diesen Rahmen nicht in irgendeiner Form anders behandeln müssen, sondern genauso wie eine rahmenlose Glyphe ebenfalls einfach anweisen können, sich selbst zu zeichnen. Das bedeutet gleichzeitig aber auch, dass die `Border`-Schnittstelle der `Glyph`-Schnittstelle entsprechen muss – deshalb wird `Border` als Unterklasse von `Glyph` abgeleitet.

All diese Überlegungen führen letztendlich zum Konzept der **durchsichtigen Umhüllung** (*engl. Transparent Enclosure*), das zum einen die Technik der Komposition mit nur einem einzigen Kindobjekt (bzw. einer einzigen **Komponente**) und zum anderen das Prinzip der kompatiblen Schnittstellen miteinander kombiniert. Grundsätzlich können Clients nicht unterscheiden, ob sie es mit der Komponente an sich oder mit ihrer **Umhüllung** (d. h. dem Elternobjekt des Kindobjekts) zu tun haben – insbesondere dann nicht, wenn die Umhüllung einfach alle ihre Operationen an ihre Komponente weiterleitet. Andererseits kann die Umhüllung das Verhalten der Komponente jedoch auch *ausweiten*, indem sie vor und/oder nach dem Delegieren einer Operation selbst einige Arbeiten ausführt. Ebenso kann sie, wie im nächsten Abschnitt beschrieben, auch den Zustand der Komponente effektiv beeinflussen.

2.4.2 Die Unterkasse MonoGlyph

Das Konzept der durchsichtigen Umhüllung kann auf alle Glyphen angewendet werden, die zur Ausgestaltung anderer Glyphen genutzt werden. Zur konkreteren Darstellung dieses Sachverhalts wird im hier vorliegenden Fallbeispiel mit **MonoGlyph** eine Unterkasse von **Glyph** erstellt, die als eine abstrakte Klasse für »Ausgestaltungsglyphen« wie **Border** dienen soll (siehe [Abbildung 2.7](#)). **MonoGlyph** speichert eine Referenz auf eine Komponente und leitet dann alle Requests an sie weiter.

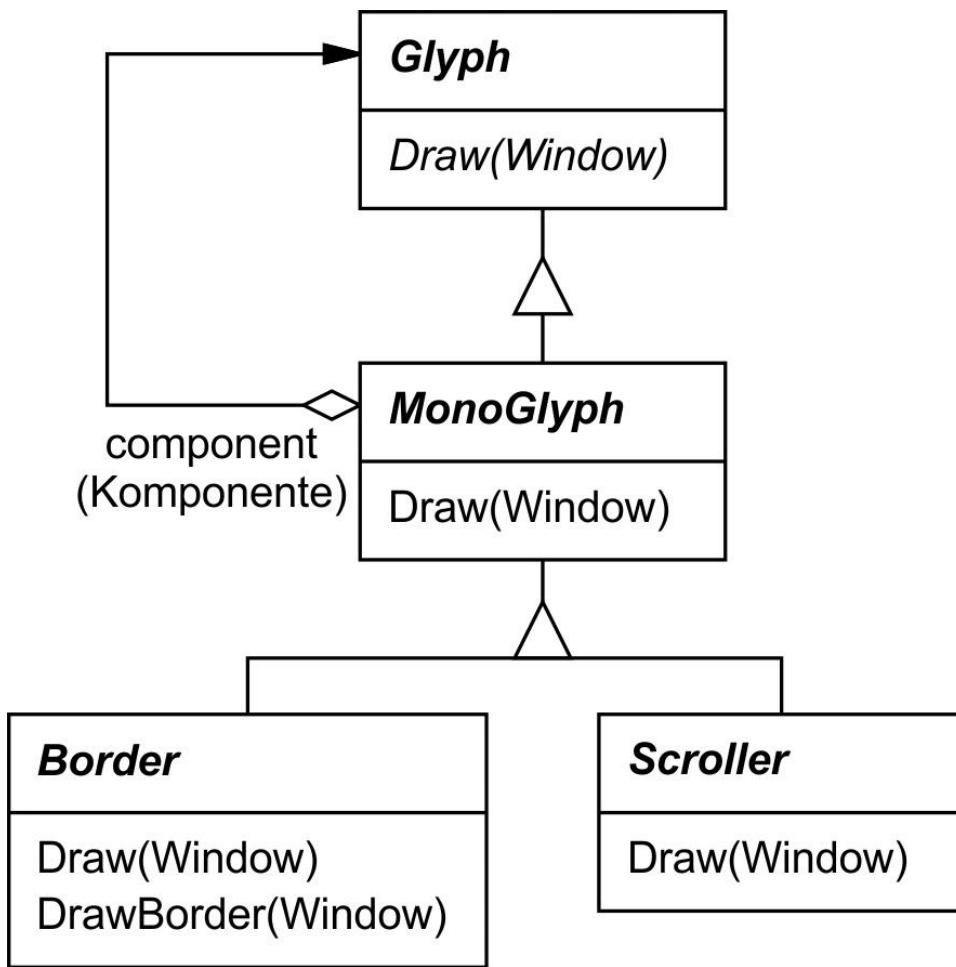


Abb. 2.7: Beziehungen der `MonoGlyph`-Klasse

Dadurch wird `MonoGlyph` für die Clients standardmäßig vollkommen transparent und implementiert die `Draw`-Operation beispielsweise wie folgt:

```

void MonoGlyph::Draw (Window* w) {
    _component->Draw(w);
}
  
```

Die Unterklassen von `MonoGlyph` reimplementieren mindestens eine dieser Weiterleitungsoperationen. Zum Beispiel ruft `Border::Draw` zunächst die Basisklassenoperation `MonoGlyph::Draw` für die Komponente auf, damit diese zu Werke gehen und, abgesehen von dem Rahmen, alles zeichnen kann. Danach erzeugt `Border::Draw` durch den Aufruf einer privaten Operation namens `DrawBorder`, die an dieser Stelle nicht weiter erläutert werden soll, den Rahmen:

```

void Border::Draw (Window* w) {
    MonoGlyph::Draw(w);
    DrawBorder(w);
}
  
```

Beachtenswert ist hier insbesondere, dass `Border` : :Draw die Basisklassenoperation zum Zeichnen des Rahmens effektiv *erweitert* – und nicht bloß die Basisklassenoperation *ersetzt*, wobei der Aufruf von `MonoGlyph` : :Draw entfallen wäre.

Eine weitere Unterklasse von `MonoGlyph` ist in [Abbildung 2.8](#) dargestellt. `Scroller` zeichnet ihre Komponente ausgehend von den Positionen der beiden Scrollleisten, die sie als gestalterische Elemente ergänzt, an verschiedenen Standorten. Bei der Durchführung des Zeichenvorgangs weist die Unterklasse `Scroller` das Grafiksystem an, die Ränder der Komponente abzuschneiden – dadurch wird verhindert, dass diejenigen Teile der Komponente, die aus dem sichtbaren Anzeigebereich herausgescrollt wurden, auf dem Bildschirm angezeigt werden.

Damit sind nun alle Elemente vorhanden, die nötig sind, um den für die Bearbeitung des Textinhalts zur Verfügung stehenden Bereich in Lexi mit einem Rahmen sowie einer Scrollfunktion auszustatten. Zur Ergänzung der Scrollfunktion wird die bereits vorhandene `Composition`-Instanz in eine `Scroller`-Instanz hineingelegt, und anschließend wird diese ganze Komposition dann wiederum in eine `Border`-Instanz eingebettet. Die daraus resultierende Objektstruktur ist in [Abbildung 2.8](#) zu sehen.

Man könnte die Reihenfolge auch umkehren und die umrahmte Komposition in die `Scroller`-Instanz legen – dann würde der Rahmen allerdings mit dem Text mitgescrollt werden, was normalerweise nicht erwünscht ist. Der maßgebliche Aspekt der durchsichtigen Umhüllung ist jedoch, dass sie das Experimentieren mit verschiedenen Alternativen sehr einfach gestaltet und den Clients zudem den Ballast des Ausgestaltungscodes erspart.

Bemerkenswert ist in diesem Zusammenhang auch, dass der Rahmen – anders als bei den bisher in diesem Buch definierten Kompositionen, in denen die Elternobjekte beliebig viele Kindobjekte haben durften – nur *eine* Gyphe enthält, und nicht etwa zwei oder mehr. Hier bedeutet das Umrahmen einer Gyphe, dass sie singulär ist, also allein steht. Möglicherweise ließe sich auch eine sinnvolle Ausgestaltung von mehr als einem Objekt auf einmal vornehmen, dazu müssten allerdings viele Kompositionsarten in das Konzept der Ausgestaltung eingemischt werden: die Zeilenausgestaltung, die Spaltenausgestaltung usw. Das wäre jedoch insofern nicht sonderlich hilfreich, als bereits Klassen existieren, die derartige Kompositionen ausführen. Deshalb sollten lieber die verfügbaren Klassen für die Komposition verwendet und dann zur Ausgestaltung des Ergebnisses neue Klassen ergänzt werden. Die Ausgestaltungsmaßnahmen insgesamt so unabhängig wie möglich von den anderen Kompositionsarten zu belassen, vereinfacht nicht nur die Ausgestaltungsklassen, sondern reduziert darüber hinaus auch ihre Anzahl –

außerdem wird so eine Replikation von bereits existierender Kompositionsfunktionalität unterbunden.

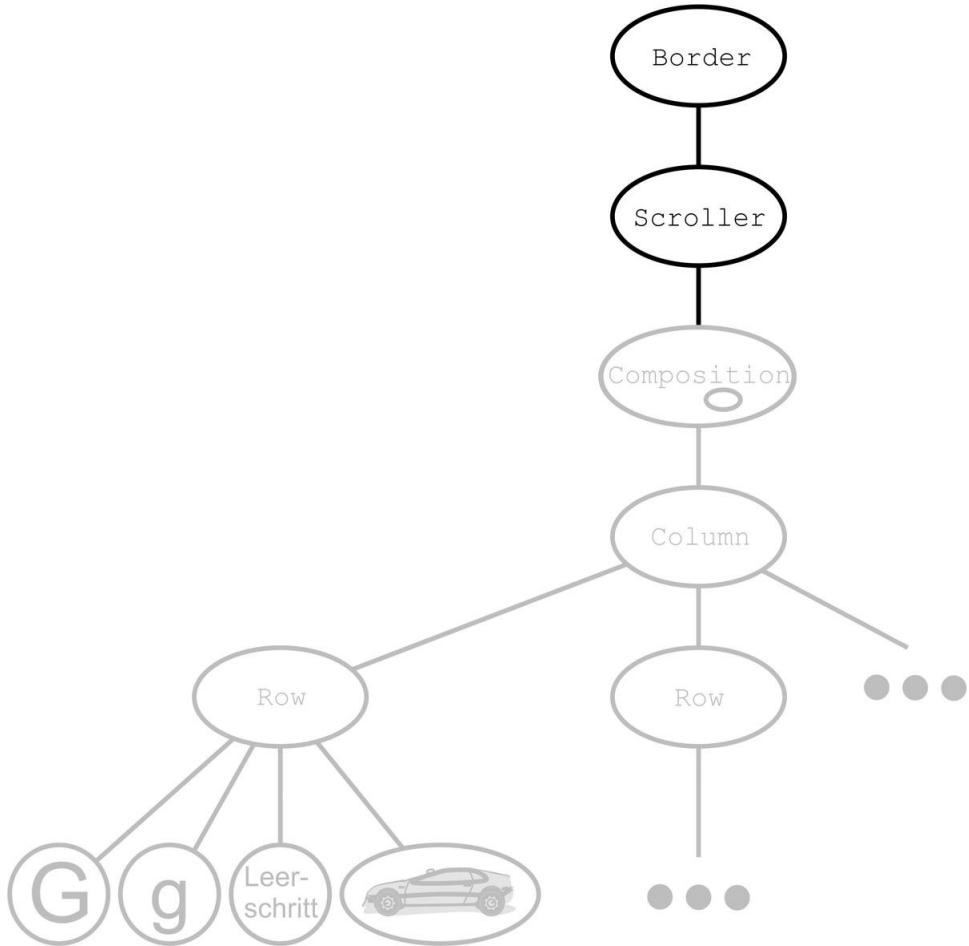


Abb. 2.8: Ausgestaltete Objektstruktur

2.4.3 Das Design Pattern *Decorator (Dekorierer)*

Das Pattern *Decorator (Dekorierer)*, siehe [Abschnitt 4.4](#)) erfasst die Klassen- und Objektbeziehungen, die eine Ausgestaltung mittels durchsichtiger Umhüllung unterstützen. Die Bedeutung des Begriffs »Ausgestaltung« geht hierbei allerdings deutlich über das hinaus, was in diesem Kapitel betrachtet wurde: Im Fall des Patterns *Decorator (Dekorierer)* ist damit alles gemeint, was ein Objekt mit weiteren Zuständigkeiten ausstattet – also beispielsweise einen abstrakten Syntaxbaum mit semantischen Aktionen, einen endlichen Automaten mit neuen Transitionen oder ein Netzwerk persistenter Objekte mit Attributbeschriftungen. Dieses Design Pattern pauschalisiert den Ansatz, der für Lexi verwendet wurde, damit er in höherem Maße anwendbar wird.

2.5 Unterstützung mehrerer Look-and-Feel-Standards

Eine der großen Herausforderungen im Systemdesign besteht im Erreichen einer Portabilität über Hardware- und Softwaregrenzen hinweg. Die Portierung von Lexi auf eine neue Plattform sollte keine umfassende Revision erforderlich machen – denn dann wäre sie gar nicht erst lohnenswert. Stattdessen sollte sie sich so einfach wie möglich bewerkstelligen lassen.

Eine Schwierigkeit bei der Gewährleistung der Portabilität besteht in der Vielfalt der *Look-and-Feel-Standards*, die für ein einheitliches Erscheinungsbild der Anwendungsversionen sorgen sollen. Diese Standards definieren Richtlinien in Bezug darauf, wie Anwendungen aussehen und auf Usereingaben reagieren sollen. Und obwohl sich die existierenden Standards gar nicht so sehr voneinander unterscheiden, sind sie doch unverwechselbar – so sehen beispielsweise Motif-Anwendungen ganz anders aus und »fühlen« sich auch anders an als ihre Gegenstücke auf anderen Plattformen und umgekehrt. Eine Anwendung, die auf mehr als einer Plattform läuft, muss den zugehörigen *Style Guides*, also den Stilrichtlinien für die Benutzeroberfläche, jeder dieser Plattformen entsprechen.

In diesem Fallbeispiel besteht das Designziel für die Lexi-Benutzeroberfläche darin, dass sie mehreren existierenden Look-and-Feel-Standards entspricht, gleichzeitig aber auch die problemlose Anpassung an neue Standards gewährleistet, die sich unweigerlich entwickeln werden. Außerdem soll das Design weiterhin maximale Flexibilität ermöglichen – in der Form, dass das Look-and-Feel zur Laufzeit geändert werden kann.

2.5.1 Abstrahierung der Objekterzeugung

Jedes Element, das auf der Benutzeroberfläche von Lexi zu sehen ist und mit dem interagiert werden kann, besteht jeweils aus einer Glyphe, die in anderen, nicht sichtbaren Glyphen wie Row und Column zusammengesetzt ist. Diese nicht sichtbaren Glyphen bilden sichtbare Glyphen wie Button und character und ordnen sie in geeigneter Weise an. Style Guides spielen für das Look-and-Feel sogenannter **Widgets** eine große Rolle. Der Begriff »Widget« ist eine andere Bezeichnung für sichtbare Glyphen wie Buttons, Scrollleisten und Menüs, die als Steuerungselemente der Benutzeroberfläche dienen. Sie verwenden zur Darstellung der Daten häufig einfachere Glyphen wie Zeichen, Kreise, Rechtecke und Polygone.

In diesem Beispiel sollen zwei Arten von Widget-Klassen betrachtet werden, mit

denen sich mehrere Look-and-Feel-Standards implementieren lassen:

1. Ein Satz abstrakter `Glyph`-Unterklassen für jede Kategorie der Widget-Glyphe. So erweitert zum Beispiel eine abstrakte Klasse `ScrollBar` die grundlegende `Glyph`-Schnittstelle um allgemeine Operationen zum Scrollen, während `Button` eine abstrakte Klasse zur Ergänzung schaltflächenorientierter Operationen darstellt usw.
2. Ein Satz konkreter Unterklassen für jede abstrakte Unterkategorie, die verschiedene Look-and-Feel-Standards implementieren. Zum Beispiel könnte `ScrollBar` die Unterklassen `MotifScrollBar` und `PMScrollBar` besitzen, die Scrollleisten im Stil von Motif bzw. Presentation Manager implementieren.

Lexi muss zwischen den Widget-Glyphen für verschiedene Look-and-Feel-Stile unterscheiden. Soll beispielsweise eine Schaltfläche auf der Benutzeroberfläche ergänzt werden, muss die Anwendung eine `Glyph`-Unterkategorie für den gewünschten Schaltflächenstil instanziieren (`MotifButton`, `PMBUTTON`, `MacButton` etc.).

Tatsächlich kann die Lexi-Implementierung dies nicht auf unmittelbare Weise erledigen, beispielsweise durch einen Konstruktoraufruf in C++, weil die Schaltfläche dann einem bestimmten Stil entsprechend hartkodiert werden müsste und die Auswahl eines Stils zur Laufzeit somit ausgeschlossen wäre. Außerdem müsste in diesem Fall jeder einzelne derartige Konstruktoraufruf ermittelt und geändert werden, damit Lexi auf eine andere Plattform portiert werden kann. Darüber hinaus repräsentieren Schaltflächen auch nur eine von vielen Widget-Varianten auf der Lexi-Benutzeroberfläche. Den Code mit Konstruktoraufrufen für spezifische Look-and-Feel-Klassen zu durchsetzen, wäre im Hinblick auf die Wartung ein wahrer Albtraum – denn wenn auch nur ein einziger übersehen wird, könnte am Ende mitten in einer Mac-Anwendung plötzlich ein Motif-Menü auftauchen.

Lexi muss also die für die Erzeugung der richtigen Widgets gewünschten Look-and-Feel-Standards selbst bestimmen können. Und das bedeutet, dass keine expliziten Konstruktoraufrufe verwendet werden dürfen und außerdem die Möglichkeit gegeben sein muss, einen ganzen Widget-Satz problemlos austauschen zu können. Beides lässt sich durch die *Abstrahierung des Objekterzeugungsprozesses* erreichen, wie das folgende Beispiel zeigt.

2.5.2 Factories und Produktklassen

Im Normalfall würde eine Instanz einer Motif-Scrollleistenglyphe mit folgendem C++-Code erzeugt:

```
ScrollBar* sb = new MotifScrollBar;
```

Das ist allerdings genau die Art von Syntax, die es zu vermeiden gilt, um die Look-and-Feel-Abhängigkeiten des Lexi-Texteditors so weit wie möglich zu reduzieren. Deshalb wird sb in diesem Fallbeispiel stattdessen wie folgt initialisiert:

```
ScrollBar* sb = guiFactory->CreateScrollBar();
```

wobei `guiFactory` eine Instanz einer `MotifFactory`-Klasse darstellt. `CreateScrollBar` gibt eine neue Instanz der passenden `ScrollBar`-Unterklasse für das gewünschte Look-and-Feel zurück, in diesem Fall Motif. Was die Clients angeht, wird hiermit derselbe Effekt erzielt wie beim direkten Aufruf des `MotifScrollBar`-Konstruktors – mit einem wichtigen Unterschied: Motif wird an keiner Stelle im Code namentlich referenziert. Das `guiFactory`-Objekt abstrahiert den Prozess also dahingehend, dass nicht nur Motif-Scrollleisten, sondern Scrollleisten für jeden *beliebigen* Look-and-Feel-Standard erzeugt werden. Zudem ist `guiFactory` nicht nur auf die Erzeugung von Scrollleisten beschränkt, sondern ermöglicht die Erstellung aller Arten von Widget-Glyphen, einschließlich Scrollleisten, Schaltflächen, Dialogfeldern, Menüs usw.

Dies alles ist möglich, weil `MotifFactory` eine Unterklasse von **GUIFactory** ist – einer abstrakten Klasse, die eine allgemeine Schnittstelle zur Erzeugung von Widget-Glyphen definiert und Operationen wie `CreateScrollBar` und `CreateButton` zur Instanziierung verschiedener Arten von Widget-Glyphen enthält. Die Unterklassen von `GUIFactory` implementieren diese Operationen in der Form, dass sie Glyphen wie `MotifScrollBar` und `PMBbutton` zurückgeben, die ein bestimmtes Look-and-Feel erzeugen. Die daraus resultierende Klassenhierarchie für `guiFactory`-Objekte ist in [Abbildung 2.9](#) dargestellt.

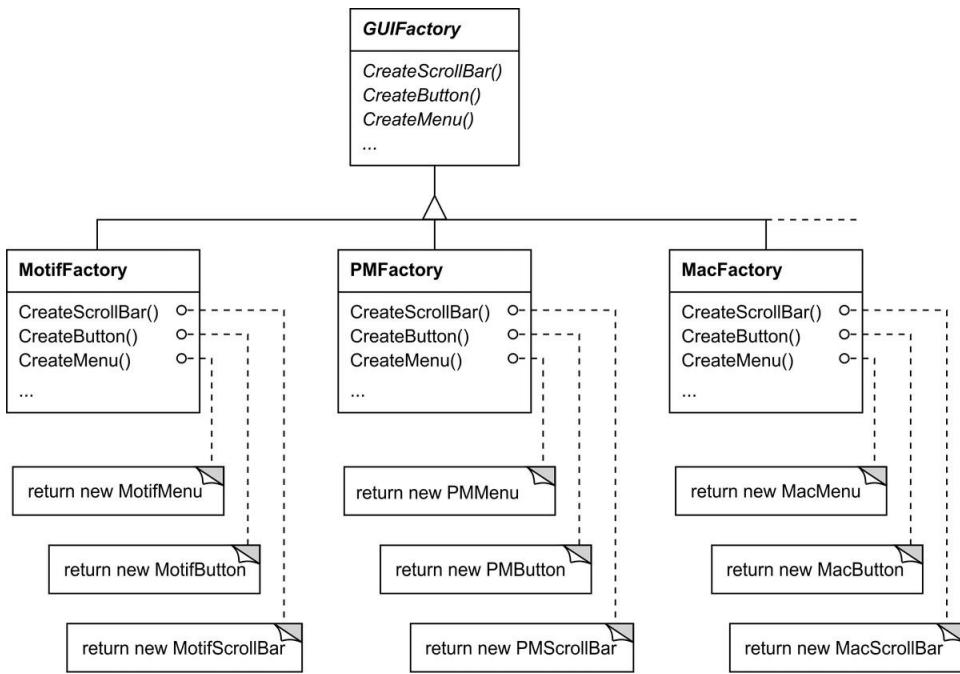


Abb. 2.9: *GUIFactory-Klassenhierarchie*

Man spricht hier davon, dass Factories **Produktobjekte** erzeugen. Außerdem stehen alle von einer Factory erzeugten Produkte in einer Beziehung zueinander – in diesem Fall handelt es sich bei den Produkten ausnahmslos um Widgets für dasselbe Look-and-Feel. [Abbildung 2.10](#) zeigt einige der Produktklassen, die für die Funktionsfähigkeit der Factories für Widget-Glyphen benötigt werden.

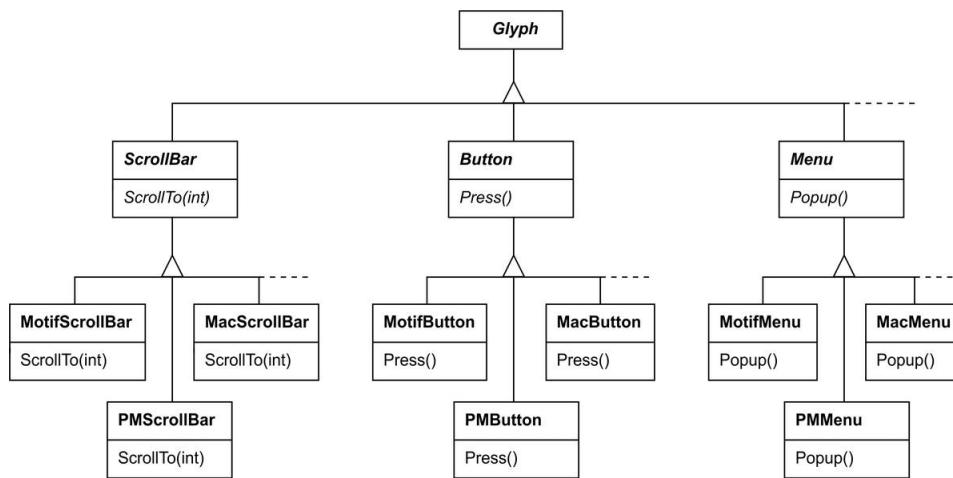


Abb. 2.10: *Abstrakte Produktklassen und konkrete Unterklassen*

Als Letztes stellt sich jetzt nur noch die Frage, woher die `GUIFactory`-Instanz eigentlich kommt. Nun, die Antwort darauf lautet: Sie kann von überall herstammen, wo es gerade passt. Die Variable `guiFactory` könnte eine globale statische Membervariable einer wohlbekannten Klasse sein oder auch eine lokale Variable,

wenn die gesamte Benutzeroberfläche in einer einzigen Klasse oder Funktion erzeugt wird. Das Design Pattern *Singleton* (*Singleton*, siehe [Abschnitt 3.5](#)) ist sogar speziell für die Verwaltung von bekannten, einzigartigen Objekten dieser Art gedacht. Entscheidend ist allerdings, dass `guiFactory` vor ihrer ersten Nutzung zur Widget-Erzeugung, aber *nach* der Festlegung des gewünschten Look-and-Feel im Programm initialisiert wird.

Sofern das Look-and-Feel schon beim Kompilieren bekannt ist, kann `guiFactory` durch die einfache Zuweisung einer neuen Factory-Instanz gleich zu Beginn des Programms initialisiert werden:

```
GUIFactory* guiFactory = new MotifFactory;
```

Hat der User die Gelegenheit, das Look-and-Feel mithilfe eines Strings während der Startphase zu spezifizieren, dann könnte der Code zum Erzeugen der Factory wie folgt aussehen:

```
GUIFactory* guiFactory;
const char* styleName = getenv("LOOK_AND_FEEL");
    // Wird beim Startup vom User oder der Umgebung geliefert

if (strcmp(styleName, "Motif") == 0) {
    guiFactory = new MotifFactory;

} else if (strcmp(styleName, "Presentation_Manager") == 0) {
    guiFactory = new PMFactory;

} else {
    guiFactory = new DefaultGUIFactory;
}
```

Es gibt aber natürlich auch noch raffiniertere Lösungen für die Auswahl der Factory zur Laufzeit. So könnte beispielsweise eine Registratur verwendet werden, die Strings auf Factory-Objekte abbildet. Dadurch lassen sich Instanzen neuer Factory-Unterklassen ohne irgendwelche Modifikationen am Code registrieren, wie dies bei der vorgenannten Herangehensweise erforderlich ist. Außerdem müssen auch nicht alle plattformspezifischen Factories in die Anwendung eingebunden werden – und das ist insofern von Bedeutung, als es gegebenenfalls nicht möglich ist, eine Klasse `MotifFactory` auf einer Plattform zu laden, die Motif nicht unterstützt.

Maßgeblich ist jedoch, dass das Look-and-Feel der Anwendung feststeht, sobald sie mit dem richtigen Factory-Objekt konfiguriert ist. Soll es dann zu einem späteren Zeitpunkt geändert werden, kann `guiFactory` mit einem Factory-Objekt für ein anderes Look-and-Feel neu initialisiert und die Benutzeroberfläche neu aufgebaut

werden. Unabhängig davon, wie und wann guiFactory initialisiert wird, kann die Anwendung auf diese Weise in jedem Fall das richtige Look-and-Feel erzeugen, ohne dass weitere Modifikationen nötig sind.

2.5.3 Das Design Pattern *Abstract Factory* (*Abstrakte Fabrik*)

Factories und Produkte sind die federführenden Teilnehmer des Design Patterns *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)). Es ermöglicht die Bildung von Familien verwandter Produktobjekte ohne direkte Klasseninstanziierungen. Die Verwendung dieses Patterns ist insbesondere dann angezeigt, wenn die Anzahl sowie die Arten der Produktobjekte konstant bleiben, die spezifischen Produktfamilien sich jedoch unterscheiden. Die Auswahl zwischen den Familien wird hier durch die Instanziierung einer bestimmten konkreten Factory getroffen, die anschließend durchgängig zur Produkterzeugung verwendet wird. Darüber hinaus lassen sich auch ganze Produktfamilien austauschen, indem die konkrete Factory durch eine Instanz einer anderen Factory ersetzt wird. Die Konzentration des Design Patterns *Abstract Factory* (*Abstrakte Fabrik*) auf »Produktfamilien« unterscheidet es von anderen Erzeugungsmustern, die sich lediglich auf eine Art von Produktobjekt beziehen.

2.6 Unterstützung mehrerer Fenstersysteme

Das Look-and-Feel ist nur einer von vielen Portabilitätsfaktoren. Ein anderer Faktor ist die Fensterumgebung, in der Lexi läuft. Das Fenstersystem einer Plattform erzeugt die Illusion mehrerer überlappender Fenster auf einem Bitmap-Display. Es verwaltet den auf dem Bildschirm für die Fenster zur Verfügung stehenden Platz und leitet Tastatur- und Mauseingaben an sie weiter. Mittlerweile gibt es eine ganze Reihe bedeutender und größtenteils inkompatibler Fenstersysteme (z. B. Macintosh, Presentation Manager, Microsoft Windows, X Window etc.) – und deshalb muss Lexi aus denselben Gründen, aus denen es auch mehrere Look-and-Feel-Standards unterstützen sollte, auf so vielen Fenstersystemen wie möglich lauffähig sein.

2.6.1 Eignung des Design Patterns *Abstract Factory* (*Abstrakte Fabrik*)

Auf den ersten Blick scheint es, als böte sich hier eine Gelegenheit für den Einsatz

des Design Patterns *Abstract Factory* (*Abstrakte Fabrik*) – allerdings gelten in Bezug auf die Portabilität von Fenstersystemen vollkommen andere Rahmenbedingungen für die Gewährleistung einer Unterstützung verschiedener Look-and-Feel-Standards.

Bei der im vorigen Abschnitt beschriebenen Anwendung des Patterns *Abstract Factory* (*Abstrakte Fabrik*) wurde davon ausgegangen, dass für jeden Look-and-Feel-Standard konkrete Widget-Klassen definiert werden. Das hieße, dass jedes konkrete Produkt für einen bestimmten Standard (z. B. `MotifScrollbar` und `MacScrollbar`) von einer abstrakten Produktklasse (z. B. `Scrollbar`) abgeleitet würde. Ginge man hingegen davon aus, dass bereits mehrere Klassenhierarchien von verschiedenen Herstellern vorhanden sind – je eine für jeden Look-and-Feel-Standard –, dann ist es natürlich sehr unwahrscheinlich, dass diese Hierarchien miteinander kompatibel sind. Dementsprechend stünde auch keine gemeinsame Produktklasse für jede Widget-Art (Scrollleiste, Schaltfläche, Menü etc.) zur Verfügung – das Pattern *Abstract Factory* (*Abstrakte Fabrik*) funktioniert aber ohne diese maßgeblichen Klassen nicht. Also müssen die verschiedenen Widget-Hierarchien auf einen gemeinsamen Satz abstrakter Produktschnittstellen angepasst werden, denn nur so können die `create...`-Operationen vernünftig in der Schnittstelle der abstrakten Factory deklariert werden.

Dieses Widget-Problem wurde in diesem Fallbeispiel durch die Erzeugung eigener abstrakter und konkreter Produktklassen gelöst. Bei dem Versuch, Lexi auf vorhandenen Fenstersystemen zu starten, ergibt sich nun aber noch ein ganz ähnliches Problem, denn: Die unterschiedlichen Fenstersysteme haben inkompatible Programmierschnittstellen (APIs). In diesem Fall wird es sogar noch ein wenig schwieriger, weil es kaum möglich sein dürfte, ein eigenes, nicht dem Standard entsprechendes Fenstersystem zu entwickeln.

Doch es ist Rettung in Sicht: Ebenso wie die Look-and-Feel-Standards unterscheiden sich auch die Schnittstellen von Fenstersystemen nicht radikal voneinander, weil immerhin alle Fenstersysteme in etwa dieselben Funktionen anbieten. Demzufolge wird ein einheitlicher Satz von Abstraktionen für Fenstersysteme benötigt, der es ermöglicht, unterschiedliche Implementierungen von Fenstersystemen hinter einer gemeinsamen Schnittstelle zu kapseln.

2.6.2 Kapselung von Implementierungsabhängigkeiten

Im [Abschnitt 2.2](#) wurde eine `Window`-Klasse zur Darstellung einer Glyphe oder Glyphenstruktur auf dem Display vorgestellt – in dem Fall wurde allerdings kein

Fenstersystem spezifiziert, mit dem dieses Objekt arbeitet, weil es sich nicht auf ein bestimmtes Fenstersystem bezieht. Die `window`-Klasse kapselt alle Aktivitäten, die sich üblicherweise mit den Fenstern *aller* Fenstersysteme ausführen lassen:

- In der Regel ermöglichen Fenster aller Art Operationen zum Zeichnen grundlegender geometrischer Formen.
- Sie können auf Symbolgröße minimiert und wieder in Normalgröße geöffnet werden.
- Sie können in ihren Dimensionen variiert werden.
- Sie können ihre Inhalte auf Verlangen neu zeichnen, beispielsweise wenn sie zunächst minimiert wurden und dann wieder in Normalgröße geöffnet werden oder wenn ein durch Überlappung verdeckter Teil ihres Anzeigebereichs wieder sichtbar gemacht wird.

Die `window`-Klasse muss also die gesamte Fensterfunktionalität verschiedener Fenstersysteme umfassen. Zu der Frage, wie dies erreicht werden kann, gibt es zwei recht extreme Philosophien:

1. *Schnittmenge der Funktionalität.* Die `window`-Klassenschnittstelle bietet lediglich diejenige Funktionalität, die alle Fenstersysteme *gemeinsam* haben. Das bedeutet aber auch, dass die `window`-Schnittstelle nur so leistungsfähig sein kann wie das einfachste der betreffenden Fenstersysteme. Alle darüber hinausgehenden Funktionen sind selbst dann, wenn sie von den meisten (aber nicht allen) Fenstersystemen angeboten werden, nicht nutzbar.
2. *Vereinigungsmenge der Funktionalität.* Bei dieser Variante wird eine Schnittstelle erzeugt, die die Fähigkeiten *aller* existierenden Systeme umfasst. Dabei besteht allerdings die Gefahr, dass die Schnittstelle schnell sehr weitschweifig und inkohärent wird. Außerdem muss sie (ebenso wie die von ihr abhängige Anwendung, in diesem Fall also Lexi) mit jeder herstellerseitigen Revision des Fenstersystems entsprechend angepasst werden.

Da im Prinzip keine dieser beiden extremen Maßnahmen eine sinnvolle Lösung darstellt, muss für das Lexi-Design ein praktikabler Mittelweg gefunden werden. Die `window`-Klasse wird eine komfortable Schnittstelle zur Unterstützung der beliebtesten Fensterfunktionen bieten. Und weil Lexi direkt mit dieser Klasse arbeiten wird, muss sie natürlich auch jene Elemente unterstützen, die der Texteditor verwendet – sprich die Glyphen. Das bedeutet wiederum, dass die Schnittstelle von `window` einen grundlegenden Satz von Grafikoperationen enthalten muss, die es den

Glyphen ermöglichen, sich selbst im Fenster zu zeichnen. [Tabelle 2.3](#) liefert einige Beispiele für die Operationen in der Window-Klassenschnittstelle:

Zuständigkeit	Operationen
Fensterverwaltung	<code>virtual void Redraw()</code> <code>virtual void Raise()</code> <code>virtual void Lower()</code> <code>virtual void Iconify()</code> <code>virtual void Deiconify()</code> ...
Grafiken	<code>virtual void DrawLine(...)</code> <code>virtual void DrawRect(...)</code> <code>virtual void DrawPolygon(...)</code> <code>virtual void DrawText(...)</code> ...

Tabelle 2.3: Window-Klassenschnittstelle

Window ist eine abstrakte Klasse. Konkrete Unterklassen von Window unterstützen verschiedene Arten von Fenstern, mit denen die User interagieren. Beispielsweise sind Anwendungsfenster, Icons und Warnmeldungen allesamt Fenster, die sich jedoch jeweils etwas unterschiedlich verhalten. Um diese Unterschiede zu erfassen, können Unterklassen wie ApplicationWindow, IconWindow und DialogWindow definiert werden. Die daraus resultierende Klassenhierarchie verleiht Anwendungen wie Lexi eine einheitliche und intuitive Fensterabstraktion, die nicht vom Fenstersystem eines bestimmten Herstellers abhängig ist (siehe [Abbildung 2.11](#)).

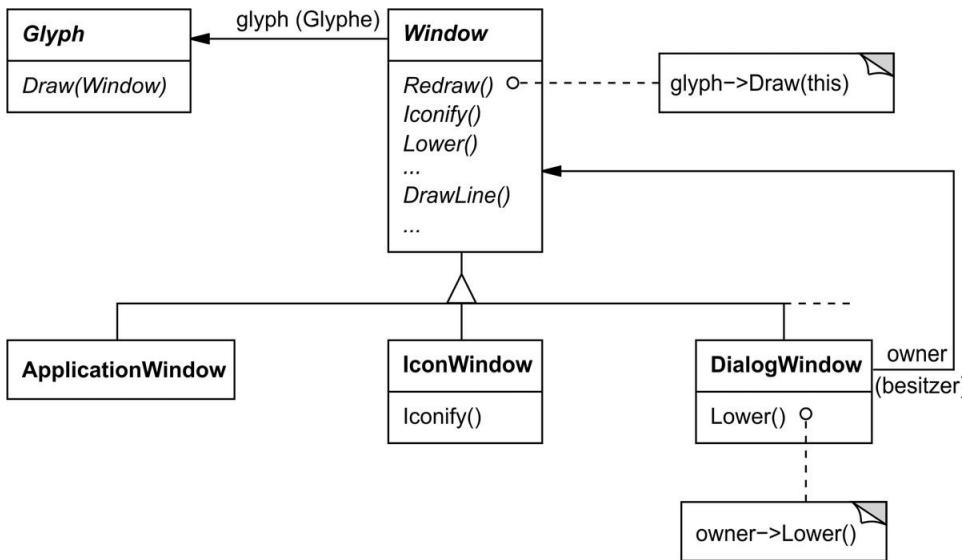


Abb. 2.11: Systemunabhängige Klassenhierarchie

Nachdem nun eine `Window`-Schnittstelle für Lexi definiert ist, stellt sich die Frage, wann das eigentliche plattformspezifische Fenster relevant wird. Wenn kein eigenes Fenstersystem implementiert wird, dann muss die Fensterabstraktion an irgendeinem Punkt auf der Grundlage dessen implementiert werden, was das Zielfenstersystem bietet. Aber wo befindet sich diese Implementierung?

Eine Möglichkeit besteht in der Implementierung mehrerer Versionen der `Window`-Klasse und ihrer Unterklassen, und zwar je einer Version für jede Fenstersystemplattform. In dem Fall müsste die zu nutzende Version während der Ausrichtung von Lexi für eine gegebene Plattform ausgewählt werden. Das könnte allerdings die Wartbarkeit spürbar erschweren, weil dann zahlreiche Klassen überwacht werden müssten, die zwar alle mit `Window` benannt, aber jeweils für verschiedene Fenstersysteme implementiert sind. Alternativ könnten auch implementierungsspezifische Unterklassen jeder Klasse in der `Window`-Hierarchie erzeugt werden – das würde aber wiederum zu einem ähnlich explosiven zahlenmäßigen Anstieg der verwendeten Klassen führen, wie schon zuvor bei den Ausgestaltungsmaßnahmen erwähnt. Außerdem bringen beide Alternativen noch einen weiteren Nachteil mit sich: Keine dieser beiden Optionen bietet die nötige Flexibilität, um nach der Kompilierung des Programms einen Wechsel des Fenstersystems zu ermöglichen. Dementsprechend müssen zusätzlich diverse ausführbare Programme verwaltet werden.

Aber gibt es denn keine andere Lösung? Doch, im Prinzip dieselbe, die sich schon bei der Formatierung und der Ausgestaltung bewährt hat, sprich die Kapselung des variierenden Konzepts, wobei das variierende Konzept in diesem Fall die Implementierung des Fenstersystems ist. Wird die Funktionalität eines

Fenstersystems in einem Objekt gekapselt, lassen sich die `Window`-Klasse und deren Unterklassen auf der Grundlage der Schnittstelle dieses Objekts implementieren. Und wenn diese Schnittstelle für alle gewünschten Fenstersysteme geeignet ist, dann muss darüber hinaus zur Unterstützung verschiedener Fenstersysteme weder die `Window`-Klasse noch irgendeine ihrer Unterklassen modifiziert werden. Stattdessen können die `Window`-Objekte für das gewünschte Fenstersystem einfach durch die Übergabe des passenden Kapselungsobjekts für eben dieses System konfiguriert werden.

2.6.3 Die Klassenhierarchien `Window` und `WindowImp`

Im Folgenden wird eine eigene `WindowImp`-Klassenhierarchie zur Unterbringung der diversen Fenstersystemimplementierungen definiert. `WindowImp` ist eine abstrakte Klasse für Objekte, die fenstersystemabhängigen Code kapseln. Um Lexi auf einem bestimmten Fenstersystem starten zu können, wird jedes `Window`-Objekt mit einer Instanz einer `WindowImp`-Unterklasse für das betreffende System konfiguriert. [Abbildung 2.12](#) demonstriert die Beziehungen zwischen den `Window`- und `WindowImp`-Hierarchien:

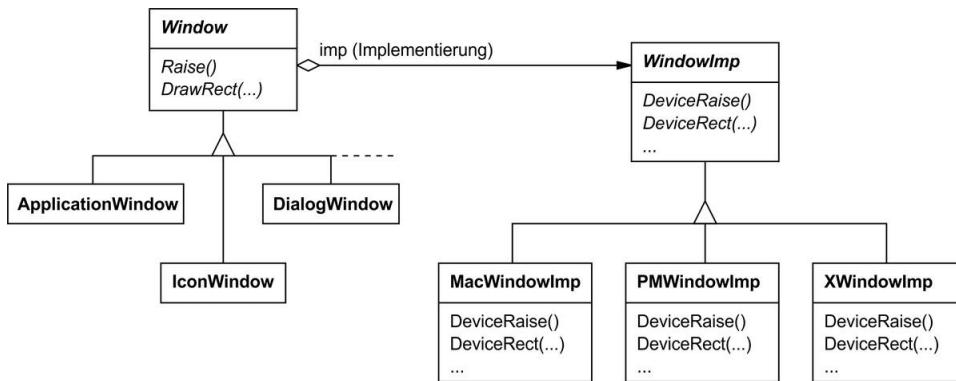


Abb. 2.12: Beziehungen zwischen den Klassenhierarchien `Window` und `WindowImp`

Durch das Verbergen der Implementierungen in den `WindowImp`-Klassen werden die `Window`-Klassen nicht mit Fensterabhängigkeiten belastet, so dass die `Window`-Klassenhierarchie überschaubar und stabil bleibt. Zudem ist bei diesem Verfahren eine problemlose Erweiterbarkeit der Implementierungshierarchie zur Unterstützung neuer Fenstersysteme gewährleistet.

WindowImp-Unterklassen

Die Unterklassen von `WindowImp` konvertieren Requests in fenstersystemspezifische Operationen. Erinnern Sie sich noch an das Beispiel aus [Abschnitt 2.2](#)? Dort wurde `Rectangle::Draw` im Sinne der `DrawRect`-Operation in der `Window`-Instanz definiert:

```
void Rectangle::Draw(Window* w) {
    w->DrawRect (_x0, _y0, _x1, _y1);
}
```

Die Standardimplementierung von `DrawRect` verwendet die abstrakte Operation zum Zeichnen der durch `WindowImp` deklarierten Rechtecke:

```
void Window::DrawRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    _imp->DeviceRect(x0, y0, x1, y1);
}
```

`_imp` ist eine Membervariable von `Window`, die das `WindowImp`-Objekt zur Konfiguration des Fensters speichert. Die Fensterimplementierung ist durch die Instanz der `WindowImp`-Unterklasse definiert, auf die `_imp` verweist. Für eine Unterkorrekte `XWindowImp` (also eine `WindowImp`-Unterklasse für das X-Window-System) könnte die Implementierung von `DeviceRect` wie folgt aussehen:

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dp, _winid, _gc, x, y, w, h);
}
```

Diese Art der `DeviceRect`-Definition basiert darauf, dass `XDrawRectangle` (die X-Window-Schnittstelle zum Zeichnen eines Rechtecks) ein Rechteck anhand seiner unteren linken Ecke, seiner Breite und seiner Höhe definiert – `DeviceRect` muss diese Daten also aus den bereitgestellten Werten berechnen. Dabei wird zuerst die untere linke Ecke ermittelt (da (x_0, y_0) eine beliebige Ecke des Rechtecks sein könnte) und anschließend werden die Breiten- und Höhenmaße errechnet.

`PMWindowImp` (eine `WindowImp`-Unterklasse für den Presentation Manager) würde `DeviceRect` anders definieren:

```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
```

```

Coord left = minfx0, x1);
Coord right = max(x0, x1);
Coord bottom = min(y0, y1);
Coord top = max(y0, y1);

PPOINTL point[4];

point[0].x = left; point[0].y = top;
point[1].x = right; point[1].y = top;
point[2].x = right; point[2].y = bottom;
point[3].x = left; point[3].y = bottom;

if (
    (GpiBeginPath(_hps, 1L) == false) ||
    (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
    (GpiPolyLine(_hps, 4L, point) == GPI_ERROR) ||
    (GpiEndPath(_hps) == false)
) {
    // Fehlermeldung
} else {
    GpiStrokePath(_hps, 1L, 0L);
}
}

```

Aber warum unterscheidet sich diese Implementierung so deutlich von der X-Window-Variante? Nun, anders als X Window verfügt der Presentation Manager nicht über eine explizite Operation zum Zeichnen von Rechtecken. Stattdessen bietet er eine allgemeinere Schnittstelle zur Spezifikation von Knotenpunkten mehrfach segmentierter Formen (sogenannter **paths**) sowie zur Umrahmung bzw. zum Füllen der von diesen Formen umschlossenen Flächen.

Der offenkundige Unterschied zwischen der Presentation-Manager- und der X-Window-Implementierung von `DeviceRect` spielt hier allerdings keine Rolle. `WindowImp` verbirgt Variationen der Fenstersystemschnittstellen hinter einer potenziell umfangreichen, aber stabilen Schnittstelle. Dadurch können sich die Entwickler von `Window`-Unterklassen ganz auf die Fensterabstraktion konzentrieren und müssen sich nicht mit den Feinheiten des Fenstersystems auseinandersetzen. Darüber hinaus begünstigt die Schnittstelle auch die Unterstützung neuer bzw. anderer Fenstersysteme, ohne dass die `window`-Klassen dadurch beeinträchtigt würden.

Fensterkonfiguration mit `WindowImp`

Ein wichtiger Punkt, der bislang noch nicht angesprochen wurde, ist die Frage, wie

ein Fenster eigentlich genau mit der richtigen `WindowImp`-Unterklasse konfiguriert wird. Oder, anders ausgedrückt: Wann wird `_imp` initialisiert und »wer« weiß, welches Fenstersystem (bzw. welche `WindowImp`-Unterklasse) verwendet wird? Das Fenster wird irgendein `WindowImp`-Objekt brauchen, bevor es überhaupt etwas Interessantes bewirken kann.

Hier bieten sich gleich mehrere Möglichkeiten an, an dieser Stelle wird aber lediglich eine davon betrachtet – und zwar diejenige, bei der das Design Pattern *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) zum Einsatz kommt. Im Folgenden wird zunächst eine Abstract-Factory-Klasse namens `WindowSystemFactory` definiert, die eine Schnittstelle zur Erzeugung verschiedener Arten von fenstersystemabhängigen Implementierungsobjekten bereitstellt:

```
class WindowSystemFactory {
public:
    virtual WindowImp* CreateWindowImp() = 0;
    virtual ColorImp* CreateColorImp() = 0;
    virtual FontImp* CreateFontImp() = 0;

    // Je eine "Create..."-Operation für alle Fenstersystemressourcen
};
```

Anschließend wird eine konkrete Factory für jedes Fenstersystem definiert:

```
class PMwindowSystemFactory : public WindowSystemFactory {
    virtual WindowImp* CreateWindowImp()
        { return new PMwindowImp; }
    // ...
};

class XwindowSystemFactory : public WindowSystemFactory {
    virtual WindowImp* CreateWindowImp()
        { return new XwindowImp; }
    // ...
};
```

Der Konstruktor der `Window`-Basisklasse kann die `WindowSystemFactory`-Schnittstelle zur Initialisierung der `_imp`-Membervariablen mit dem für das Fenstersystem passenden `WindowImp`-Objekt nutzen:

```
Window::Window() {
    _imp = windowSystemFactory->CreateWindowImp();
}
```

Bei der Variablen `windowSystemFactory` handelt es sich um eine gebräuchliche Instanz einer `WindowSystemFactory`-Unterklasse, die der wohlbekannten

guiFactory-Variablen zur Definition des Look-and-Feel ähnelt und auf dieselbe Art initialisiert werden kann.

2.6.4 Das Design Pattern *Bridge* (*Brücke*)

Obwohl die `WindowImp`-Klasse eine Schnittstelle für die allgemeine Fenstersystemfunktionalität definiert, unterliegt ihr Design dennoch anderen Rahmenbedingungen als das der `Window`-Schnittstelle. Die Anwendungsentwickler benutzen die `WindowImp`-Schnittstelle nicht direkt, sondern arbeiten lediglich mit den `Window`-Objekten. Insofern muss die `WindowImp`-Schnittstelle auch nicht der Denkart des Entwicklers entsprechen – die beim Design der `Window`-Klassenhierarchie und -Schnittstelle im Vordergrund steht –, sondern kann sich näher an der tatsächlichen Funktionalität der Fenstersysteme sowie deren Stärken und Schwächen orientieren. Sie kann also entweder auf die Schnittmenge oder die Vereinigungsmenge der Funktionalität ausgerichtet werden – je nachdem, was für das Zielfenstersystem besser geeignet ist.

Ausschlaggebend ist hierbei, dass sich die `Window`-Schnittstelle nach den Ansprüchen des Anwendungsentwicklers richtet, während die `WindowImp`-Schnittstelle auf die Fenstersysteme zugeschnitten ist. Durch die separate Handhabung der Fensterfunktionalität in `Window`- und `WindowImp`-Hierarchien ist eine unabhängige Implementierung und Spezialisierung der Schnittstellen möglich. Und die Zusammenarbeit der Objekte dieser Hierarchien gewährleistet die Lauffähigkeit von Lexi auf mehreren Fenstersystemen, ohne dass irgendwelche Modifizierungen vorgenommen werden müssen.

Ein Beispiel für die Beziehung zwischen `Window` und `WindowImp` liefert das Design Pattern *Bridge* (*Brücke*, siehe [Abschnitt 4.2](#)). Es dient dem Zweck, die Zusammenarbeit separater, sich unabhängig voneinander weiterentwickelnder Klassenhierarchien zu ermöglichen. Aufgrund der hier zugrundeliegenden Designkriterien wurden zwei separate Klassenhierarchien erstellt: Die eine unterstützt das logische Konzept der Fenster und die andere konzentriert sich auf die Erfassung verschiedener Fensterimplementierungen. Das *Bridge* (*Brücke*)-Pattern ermöglicht die Wartung und Erweiterung der logischen Fensterabstraktionen, ohne dass der fenstersystemabhängige Code angetastet werden muss und umgekehrt.

2.7 Userseitige Operationen

Ein Teil der Lexi-Funktionalität wird durch die WYSIWYG-Präsentation des Dokuments bereitgestellt. So erfolgen beispielsweise Texteingaben und -löschungen, das Verschieben der Einfügemarkierung sowie die Auswahl von Textbereichen durch Zeigen und Klicken sowie direktes Tippen im Dokument. Andere Lexi-Funktionen werden indirekt durch userseitige Operationen ausgelöst, etwa durch die Auswahl eines Menüeintrags, das Anklicken einer Schaltfläche oder die Aktivierung eines Tastenbefehls. Zu dieser Funktionalität gehören unter anderem auch folgende Operationen:

- Das Erstellen eines neuen Dokuments,
- das Öffnen, Speichern und Drucken eines vorhandenen Dokuments,
- das Ausschneiden und Einfügen eines markierten Textbereichs,
- die Änderung der Schriftart und des Schriftstils des markierten Textes,
- das Ändern der Textformatierung, wie z. B. der Textausrichtung und -einrückung,
- das Beenden der Anwendung
- und vieles mehr.

Lexi bietet zur Durchführung dieser Operationen zwar verschiedene Benutzeroberflächen an, allerdings soll es keine Verknüpfungen mit einer bestimmten Oberfläche geben, weil mitunter mehrere Ausführungsoptionen (z. B. durch Betätigung einer Schaltfläche, Aufruf eines Menübefehls etc.) für ein und dieselbe Operation (beispielsweise das Blättern im Dokument) zur Verfügung gestellt werden sollen. Außerdem könnte die Benutzeroberfläche nachträglich noch verändert werden.

Darüber hinaus werden solche Operationen in vielen verschiedenen Klassen implementiert – deshalb sollten die Entwickler auf deren Funktionalität zugreifen können, ohne allzu viele Abhängigkeiten zwischen den Klassen der Implementierung und der Benutzeroberfläche zu erzeugen – damit eine eng gekoppelte Implementierung, die schwieriger zu verstehen, zu erweitern und zu warten ist, vermieden werden kann.

Eine weitere Schwierigkeit dieses Fallbeispiels ist, dass Lexi für die meisten, *aber nicht alle* Funktionen UNDO- (RÜCKGÄNGIG) und REDO (WIEDERHOLEN)-Aktionen erlauben soll. Insbesondere Änderungsoperationen, wie z. B. das Löschen von Text,

mit denen der User unabsichtlich große Datenmengen des Dokuments zerstören kann, sollen rückgängig gemacht werden können. Andere Operationen wie das Speichern einer Zeichnung oder das Beenden der Anwendung können dagegen ohne eine solche Funktionalität auskommen. Außerdem soll eine uneingeschränkte Ebenentiefe hinsichtlich der wiederholbaren und rückgängig zu machenden Schritte gewährleistet sein.

Selbstverständlich erstreckt sich die Unterstützung userseitiger Operationen über die gesamte Anwendung. Die Herausforderung dabei besteht darin, einen einfachen und erweiterbaren Mechanismus zu etablieren, der alle damit verbundenen Anforderungen erfüllt.

2.7.1 Kapselung eines Requests

Aus der Perspektive des Designers betrachtet, ist ein Pulldown-Menü nur eine andere Art von Glyphen, die wiederum andere Glyphen enthält. Was die Pulldown-Menüs von herkömmlichen Glyphen mit Kindobjekten unterscheidet, ist, dass die meisten »Menüglyphen« ihre Arbeit erst nach dem Anklicken eines Elements aufnehmen.

Angenommen, diese Glyphen seien Instanzen einer `Glyph`-Unterklasse namens `MenuItem` und würden ausschließlich auf vom Client übermittelte Requests reagieren. Die Ausführung des Requests könnte durch eine Operation auf einem Objekt oder auch durch mehrere Operationen auf mehreren Objekten oder irgendetwas dazwischen bewerkstelligt werden.

Hinweis

Konzeptionell gesehen, ist der Lexi-User der Client – praktisch ist der Client jedoch ein anderes Objekt (z. B. ein Event Dispatcher (Ereignisverteiler)), der die benutzerseitigen Eingaben verwaltet.

Nun könnte man natürlich für jede userseitige Operation je eine Unterklasse von `MenuItem` definieren und jede von ihnen per Hartkodierung anweisen, den Request auszuführen. Das wäre aber nicht unbedingt hilfreich, denn ebenso wenig wie für jeden Textstring in einem Pulldown-Menü eine Unterklasse benötigt wird, ist für jeden Request eine Unterklasse von `MenuItem` erforderlich. Darüber hinaus würde

der Request bei diesem Ansatz an eine bestimmte Benutzeroberfläche gekoppelt, so dass es schwierig wäre, ihn über eine andere Benutzeroberfläche zu erfüllen.

Stellen Sie sich zum besseren Verständnis einmal vor, es wäre möglich, sowohl durch die Auswahl eines entsprechenden Eintrags in einem Pulldown-Menü *als auch* durch Betätigung eines Seitensymbols am unteren Rand der Lexi-Benutzeroberfläche (was bei kürzeren Dokumenten bequemer sein dürfte) zum Ende des Dokuments bzw. zur letzten Dokumentseite vorzublättern. Wird der Request durch Vererbung mit einem `MenuItem` verknüpft, dann muss dasselbe auch für das Seitensymbol und jede andere Art von Widget erfolgen, das einen solchen Request auslösen könnte. Dadurch kann die Anzahl der Klassen allerdings in dem Maße ansteigen, dass sie annähernd dem Produkt der Anzahl der Widget-Typen und der Anzahl der Requests entsprechen würde.

Was hier fehlt, ist ein Mechanismus, der eine Parametrisierung der Menüeinträge anhand des zu erfüllenden Requests ermöglicht. So würde einerseits eine Ausweitung der Unterklassen verhindert und andererseits eine größere Flexibilität zur Laufzeit gewährleistet. Würde `MenuItem` jedoch mit einer aufzurufenden Funktion parametrisiert, wäre das aus mindestens drei Gründen keine vollkommene Lösung:

1. Das UNDO/REDO-Problem bliebe unangetastet.
2. Es wäre schwierig, einen Zustand mit einer Funktion zu verknüpfen, weil beispielsweise eine Funktion zum Ändern der Schriftart auch wissen muss, welche Schriftart zu ändern ist.
3. Ebenso problematisch wären auch die Erweiterung von Funktionen sowie deren teilweise Wiederverwendung.

Schon allein diese Argumente lassen den Schluss zu, dass `MenuItem`-Objekte mit einem Objekt, nicht aber mit einer Funktion parametrisiert werden sollten. Damit ließe sich die Implementierung des Requests mittels Vererbung erweitern und wiederverwenden. Außerdem stünde dadurch auch ein Ort zum Speichern des Zustands und zur Implementierung der UNDO/REDO-Funktionalität zur Verfügung. Hierbei handelt es sich um ein weiteres Beispiel für die Kapselung des variierenden Konzepts, in diesem Fall eines Requests: Jeder Request wird in einem **Befehlsobjekt** gekapselt.

2.7.2 Die Command-Klasse und ihre Unterklassen

Als Erstes wird in diesem Fallbeispiel eine abstrakte Command-Klasse zur Bereitstellung einer Schnittstelle für die Ausgabe eines Requests definiert. Die grundlegende Schnittstelle besteht aus einer einzelnen abstrakten Operation namens Execute. Die Command-Unterklassen implementieren Execute zur Ausführung verschiedener Requests auf unterschiedliche Art und Weise: Manche delegieren die gesamte Arbeit oder Teile davon an andere Objekte, andere sind dagegen möglicherweise in der Lage, den Request vollkommen selbstständig zu erfüllen (siehe Abbildung 2.13). Für den Auslöser des Requests ist ein Command-Objekt in jedem Fall einfach ein Command-Objekt – und daher wird es auch stets einheitlich behandelt.

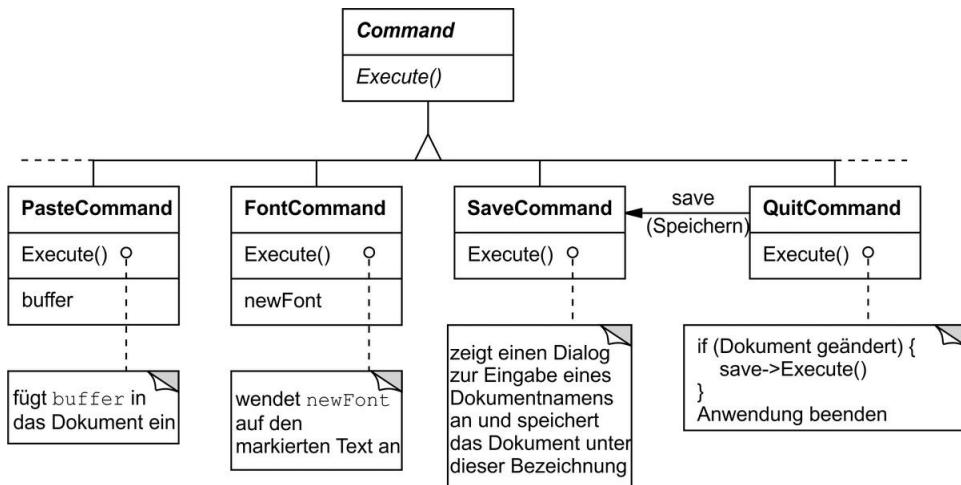


Abb. 2.13: Auszug der *Command*-Klassenhierarchie

Nun kann MenuItem also ein Command-Objekt speichern, das einen Request kapselt (siehe Abbildung 2.14). Als Nächstes wird jedem MenuItem-Objekt eine Instanz der Command-Unterklasse, die zu diesem Menüeintrag passt sowie ein entsprechender Text zugewiesen. Wenn dann ein User einen bestimmten Menüeintrag auswählt, ruft MenuItem einfach die Execute-Operation auf seinem Command-Objekt auf, um den Request auszuführen. Schaltflächen und andere Widgets können Command-Objekte auf dieselbe Weise nutzen.

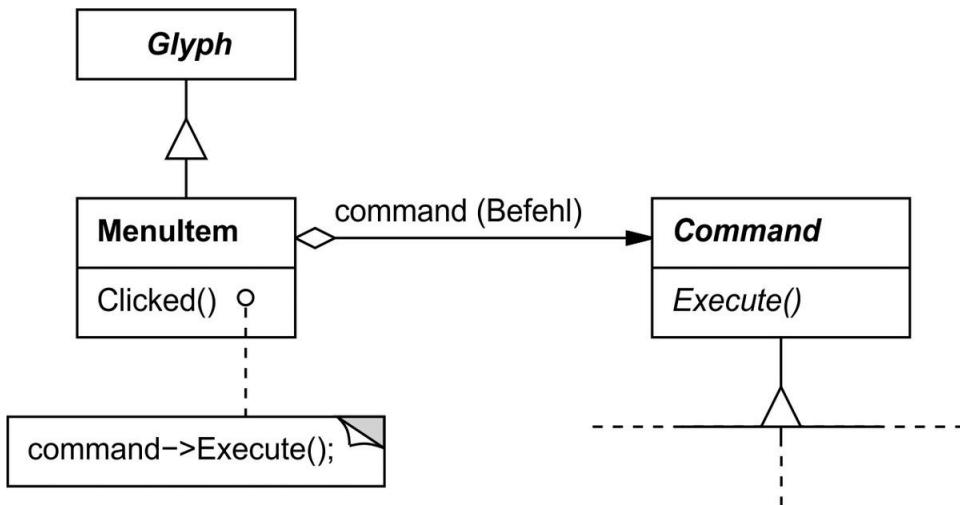


Abb. 2.14: Beziehung zwischen *MenuItem* und *Command*

2.7.3 Die Funktion Undo (Rückgängig)

Die Funktionen UNDO (RÜCKGÄNGIG) und REDO (WIEDERHOLEN) sind in interaktiven Anwendungen geradezu unverzichtbar. Um durchgeführte Aktionen rückgängig machen (UNDO) und wiederholen (REDO) zu können, muss eine Unexecute-Operation in der Command-Schnittstelle ergänzt werden, die die Auswirkung der vorangegangenen Execute-Operation zurücknimmt und durch die vor der Ausführung des Befehls gespeicherten Daten ersetzt. Im Fall einer Schriftartänderung mittels FontCommand wären beispielsweise der von der Änderung betroffene Textbereich sowie die ursprünglich verwendete/n Schriftart/en in der Execute-Operation gespeichert, so dass die Unexecute-Operation den Text wieder in seinen Ursprungszustand zurückversetzen würde.

Mitunter muss die Entscheidung, ob eine Rückgängigmachung gestattet sein soll, zur Laufzeit getroffen werden. Sollte der Text bereits in der durch einen Request geforderten Schriftart formatiert sein, bleibt der markierte Textbereich unverändert – der Request ist wirkungslos. Denn wenn ein User einen Text markiert und versucht, einen Wechsel in eine Schriftart vorzunehmen, die ohnehin schon vorhanden ist, welches Ergebnis sollte ein darauffolgender UNDO-Request dann liefern? Wäre es sinnvoll, wenn eine wirkungslose Änderungsanfrage zu einer ebenso wirkungslosen UNDO-Operation führt? Natürlich nicht. Ebenso unergiebig wäre es auch, wenn der User einen derartigen nicht durchführbaren Request möglicherweise sogar mehrfach stellt und dann exakt dieselbe Anzahl an UNDO-Operationen durchführen müsste, bis er schließlich wieder bei der zuletzt durchgeführten sinnvollen Operation angelangt wäre. Mit anderen Worten: Wenn

die Ausführung eines Befehls ohne Wirkung bleibt, ist auch kein entsprechender UNDO-Request erforderlich.

Um festzustellen, ob die Rückgängigmachung eines Befehls möglich sein soll, muss die Command-Schnittstelle um eine abstrakte Reversible-Operation ergänzt werden, die einen booleschen Wert zurückgibt. Unterklassen können diese Operation in der Art neu definieren, dass sie ihren Rückgabewert (`true` oder `false`) anhand von Laufzeitkriterien ermittelt.

2.7.4 Befehlshistorie

Der letzte Schritt zur Bereitstellung einer UNDO- bzw. REDO-Funktion mit beliebiger Ebenentiefe ist die Definition einer **Befehlshistorie**, d. h. einer Liste der ausgeführten bzw. zuvor bereits rückgängig gemachten Befehle. Prinzipiell sieht der Aufbau einer Befehlshistorie etwa wie in [Abbildung 2.15](#) dargestellt aus:

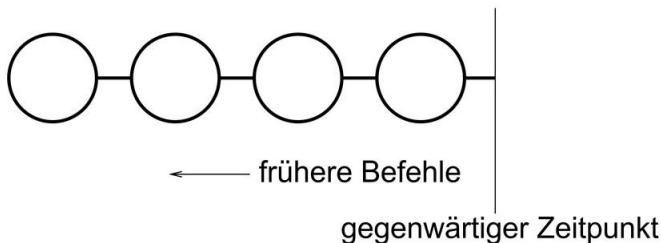


Abb. 2.15: Grundaufbau der Befehlshistorie

Jeder Kreis repräsentiert ein Command-Objekt. In diesem Beispiel hat der User vier Befehle aufgerufen: Als Erstes wurde das äußerst linke Command-Objekt ausgeführt, danach das zweite von links und so weiter bis hin zum zuletzt aktivierten Befehl, der durch den Kreis ganz rechts symbolisiert wird. Die mit »gegenwärtiger Zeitpunkt« beschriftete Kennzeichnungslinie markiert den Zeitpunkt zwischen dem zuletzt ausgeführten und dem nächsten (noch nicht ausgeführten) Command-Objekt.

Um den letzten Befehl rückgängig zu machen, wird die `Unexecute`-Operation auf das jüngste Command-Objekt angewendet (siehe [Abbildung 2.16](#)).

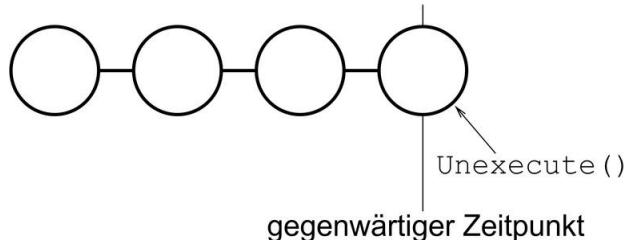


Abb. 2.16: Rückgängigmachen des zuletzt ausgeführten Befehls

Nachdem der Befehl rückgängig gemacht wurde, wird die Kennzeichnungslinie »gegenwärtiger Zeitpunkt« um ein Command-Objekt nach links bewegt. Erteilt der User nun einen weiteren UNDO-Befehl, wird der nächstjüngere ausgeführte Befehl auf die gleiche Weise rückgängig gemacht. Der damit erreichte Zustand der Befehlshistorie ist in [Abbildung 2.17](#) dargestellt:

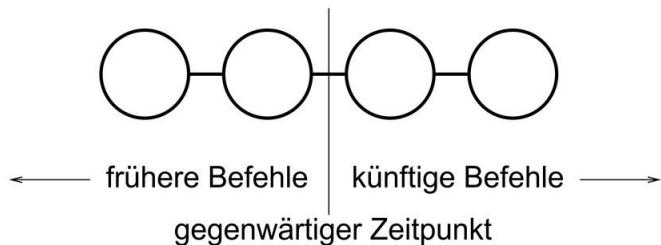


Abb. 2.17: Zustand der Befehlshistorie nach einer Rückgängigmachung

Wie hier zu sehen ist, führt jede weitere Anwendung der UNDO-Funktion jeweils zum nächstjüngeren zuvor ausgeführten Befehl. Die Anzahl der bearbeitbaren Befehle ist dabei nur durch die Länge der Befehlshistorie begrenzt.

Zur Wiederholung eines gerade erst rückgängig gemachten Befehls wird dasselbe Vorgang in umgekehrter Reihenfolge ausgeführt. Die Command-Objekte, die sich rechts der Kennzeichnungslinie »gegenwärtiger Zeitpunkt« befinden, können zu einem zukünftigen Zeitpunkt wiederholt werden. Um den letzten rückgängig gemachten Befehl erneut auszuführen bzw. zu wiederholen, wird die Execute-Operation auf das rechts der Kennzeichnungslinie befindliche Command-Objekt angewendet:

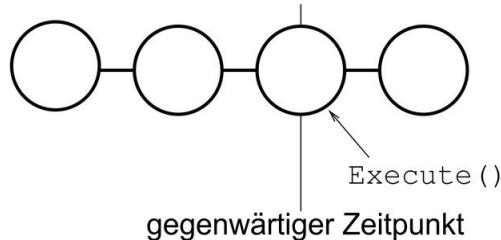


Abb. 2.18: Wiederholen des zuletzt rückgängig gemachten Befehls

Dadurch wird die Kennzeichnungsstruktur »gegenwärtiger Zeitpunkt« um ein Command-Objekt vorwärtsbewegt, so dass bei einer anschließenden Wiederholung der REDO-Befehl auf das nächste Command-Objekt angewendet wird.



Abb. 2.19: Zustand der Befehlshistorie nach einer Wiederholung

Handelt es sich bei der anschließenden Operation nicht um einen REDO-, sondern um einen UNDO-Befehl, dann wird das Command-Objekt links der Kennzeichnungsstruktur rückgängig gemacht. Auf diese Weise kann der User bei Bedarf effizient in der Befehlshistorie vor- und zurückgehen, um fälschlicherweise erteilte Befehle zu korrigieren.

2.7.5 Das Design Pattern **Command (Befehl)**

Die Befehlsobjekte in Lexi entsprechen der Anwendung des Design Patterns **Command (Befehl)**, siehe [Abschnitt 5.2](#)), das die Kapselung einer Operation beschreibt. Es bestimmt eine einheitliche Schnittstelle zur Ausführung von Requests, die die Konfiguration von Clients zur Handhabung verschiedenster Requests ermöglicht. Die Schnittstelle schirmt die Clients von der Implementierung des Requests ab. Ein Command-Objekt kann gegebenenfalls die gesamte Implementierung des Requests, Teile davon oder gar nichts an andere Objekte delegieren. Diese Vorgehensweise ist für Anwendungen wie Lexi, die einen zentralisierten Zugriff auf über die gesamte Anwendung verteilte Funktionalität bieten müssen, ideal geeignet.

Darüber hinaus berücksichtigt das Pattern *Command (Befehl)* auch UNDO- und REDO-Mechanismen, die auf der zugrundeliegenden Command-Schnittstelle aufbauen.

2.8 Rechtschreibprüfung und Silbentrennung

Die letzte Designproblemstellung des Lexi-Fallbeispiels betrifft die Textanalyse – genauer gesagt, die Überprüfung des Dokuments auf Rechtschreibfehler und einzufügende Silbentrennungen, wo diese zu einem ansprechenderen Erscheinungsbild des Textes beitragen.

Hier gelten ähnliche Rahmenbedingungen wie im [Abschnitt 2.3](#) zum Thema »Formatierung« beschrieben. Ebenso wie für die Zeilenumbruchstrategien stehen auch für die Implementierung der Rechtschreibprüfung und das Ermitteln der Silbentrennstellen mehrere Möglichkeiten zur Verfügung. Deshalb sollen abermals verschiedene Algorithmen unterstützt werden, die jeweils individuelle Vor- und Nachteile hinsichtlich des Speicherplatzbedarfs, der Ausführungsgeschwindigkeit und der Qualität mit sich bringen. Ebenso sollen auch neue Algorithmen möglichst problemlos ergänzt werden können.

Eine weitere Zielsetzung lautet, eine direkte Verknüpfung der Funktionalität mit der Dokumentstruktur zu vermeiden. Diesem Aspekt kommt hier eine noch größere Bedeutung zu als bei der Formatierung, weil die Rechtschreibprüfung und die Silbentrennung lediglich zwei von vielen Analysearten repräsentieren, die die Anwendung ermöglichen soll. Eine nachträgliche Erweiterung der analytischen Funktionalität von Lexi, beispielsweise in Form einer Suchfunktion, einer Wortzählung, einer Kalkulationsfunktion für Tabellenwerte, einer Grammatikprüfung und noch vielem mehr, wird unvermeidlich sein – allerdings sollte dafür selbstverständlich nicht jedes Mal die `Glyph`-Klasse samt all ihrer Unterklassen modifiziert werden müssen.

In diesem Fall stehen also zwei Dinge im Vordergrund: zum einen der Zugriff auf die zu analysierenden Daten, die über alle Glyphen in der Dokumentstruktur verteilt sind, und zum anderen die eigentliche Durchführung der Textanalyse. Im Folgenden werden diese beiden Aufgaben genauer betrachtet.

2.8.1 Zugriff auf verteilte Informationen

Viele Analysemaßnahmen erfordern eine zeichenweise Überprüfung des

Dokuments. Der in diesem Beispiel zu prüfende Text ist über eine hierarchische Struktur von `Glyph`-Objekten verteilt. Für die Untersuchung von Texten innerhalb einer solchen Struktur wird ein Zugriffsmechanismus benötigt, dem die Datenstrukturen, in denen die Objekte gespeichert sind, bekannt sind. Manche Glyphen speichern ihre Kindobjekte in verketteten Listen, andere nutzen möglicherweise Arrays, und wieder andere verwenden vielleicht sogar noch komplexere Datenstrukturen – deshalb soll der hier zu verwendende Zugriffsmechanismus in der Lage sein, mit all diesen Eventualitäten umzugehen.

Ebenfalls zu berücksichtigen ist, dass die einzelnen Analysen auf unterschiedliche Art und Weise auf die Daten zugreifen. In den *meisten* Fällen wird der Text vom Anfang bis zum Ende durchforstet – einige Analysevorgänge verlaufen aber genau umgekehrt, beispielsweise arbeitet sich eine Rückwärtssuche von hinten nach vorne vor. Zudem könnte die Überprüfung algebraischer Ausdrücke eine Inorder-Traversierung des Textes erforderlich machen.

Im Endeffekt bedeutet das: Der Zugriffsmechanismus muss mit verschiedenen Datenstrukturen zureckkommen und diverse Traversierungsarten wie Preorder, Postorder und Inorder unterstützen.

2.8.2 Kapselung von Zugriff und Traversierung

Aktuell verwendet die `Glyph`-Schnittstelle einen Integerindex, um den Clients den Zugriff auf die Kindobjekte zu ermöglichen. Das mag für `Glyph`-Klassen, die ihre Kindobjekte in einem Array speichern, auch sinnvoll sein – für Glyphen, die eine verkettete Liste nutzen, wäre diese Lösung jedoch ineffizient. Eine wichtige Aufgabe der Glyphenabstraktion besteht darin, die Datenstruktur, in der die Kindobjekte abgelegt sind, zu verbergen. Dadurch lassen sich Änderungen an der von einer `Glyph`-Klasse verwendeten Datenstruktur vornehmen, ohne dass andere Klassen in Mitleidenschaft gezogen werden.

Aus diesem Grund darf nur die Glyphe die von ihr verwendete Datenstruktur kennen – und dementsprechend sollte die `Glyph`-Schnittstelle nicht auf eine bestimmte Datenstruktur ausgerichtet und sich auch nicht, wie es gegenwärtig der Fall ist, für Arrays besser eignen als für verkettete Listen.

Dieser Problematik lässt sich jedoch durchaus begegnen – und das sogar bei gleichzeitiger Sicherstellung der Unterstützung verschiedener Traversierungsarten. Dazu werden der mehrfache Zugriff sowie die Traversierungsoptionen direkt in die `Glyph`-Klassen integriert. Um die Auswahl einer Traversierungsart zu ermöglichen,

wird den verschiedenen Varianten eine Aufzählungskonstante als Parameter mitgegeben. Die Klassen geben diesen Parameter dann im Rahmen der Traversierung weiter, damit gewährleistet ist, dass sie alle dieselbe Traversierungsart ausführen. Außerdem müssen die Klassen alle während der Traversierung gesammelten Daten weiterreichen.

Zur Unterstützung dieses Ansatzes könnte beispielsweise die `Glyph`-Schnittstelle um die folgenden abstrakten Operationen ergänzt werden:

```
void First(Traversierungsart)
void Next()
bool IsDone()
Glyph* GetCurrent()
void Insert(Glyph*)
```

Die Operationen `First`, `Next` und `IsDone` dienen der Steuerung der Traversierung. `First` initialisiert die Traversierung. Sie nimmt die gewünschte Traversierungsart als Parameter des Typs `Traversal` entgegen – einer Aufzählungskonstanten mit Werten wie `CHILDREN` (zur ausschließlichen Traversierung der direkten Kindobjekte der Glyphe), `PREORDER` (zur Traversierung der gesamten Preorder-Struktur), `POSTORDER` und `INORDER`. Die Operation `Next` setzt die Traversierung mit der nächsten Glyphe fort, und `IsDone` meldet, ob die Traversierung abgeschlossen ist oder nicht. `GetCurrent` ersetzt die `Child`-Operation und greift auf die aktuelle Glyphe in der Traversierung zu. `Insert` ersetzt die alte Operation und fügt die jeweilige Glyphe an der aktuellen Position ein.

So könnte eine Analyse folgenden C++-Code nutzen, um eine Preorder-Traversierung einer Glyphenstruktur mit der Wurzel `g` auszuführen:

```
Glyph* g;

for (g->First(PREORDER); !g->IsDone(); g->Next()) {
    Glyph* current = g->GetCurrent();

    // Führt die Analyse aus
}
```

Beachtenswert ist hier, dass der Integerindex aus der `Glyph`-Schnittstelle verbannt wurde. Die Oberfläche ist nun nicht länger mit irgendeiner spezifischen Behälterart oder Ähnlichem verknüpft. Außerdem bleibt es den Clients dadurch erspart, gängige Traversierungsarten selbst implementieren zu müssen.

Dennoch bringt dieser Ansatz immer noch einige Probleme mit sich. Zum einen unterstützt er keine neuen Traversierungen, ohne dass entweder die Werte der

Aufzählungstypen erweitert oder neue Operationen ergänzt werden. Angenommen, es soll eine Variante der Preorder-Traversierung verwendet werden, die nicht textbezogene Glyphen automatisch überspringt. Dann müsste der Aufzählungstyp `Traversal` dahingehend modifiziert werden, dass er beispielsweise einen Wert wie `TEXTUAL_PREORDER` umfasst.

Außerdem sollten bereits vorhandene Deklarationen nach Möglichkeit nicht geändert werden müssen. Die Einbettung des gesamten Traversierungsmechanismus in die `Glyph`-Klassenhierarchie gestaltet es allerdings schwierig, Modifizierungen oder Erweiterungen vorzunehmen, ohne gleichzeitig auch eine Menge Klassen anzupassen. Die Wiederverwendung des Mechanismus zur Traversierung anderer Objektstrukturarten ist ebenfalls problematisch. Und es kann immer nur eine Strukturtraversierung nach der anderen durchgeführt werden.

Auch hier gilt wieder, dass die Kapselung des variierenden Konzepts eine bessere Lösung darstellt, wobei das variierende Konzept in diesem Fall aus den Zugriffs- und Traversierungsmechanismen besteht. Damit kann eine Klasse von sogenannten **Iterator-Objekten** erstellt werden, deren einzige Aufgabe in der Definition verschiedener Sätze dieser Mechanismen besteht. Zudem kann das Vererbungsprinzip angewendet werden, um einen einheitlichen Zugriff auf unterschiedliche Datenstrukturen bei gleichzeitiger Unterstützung neuer Traversierungsarten sicherzustellen – und dazu müssen weder die `Glyph`-Schnittstellen geändert noch vorhandene `Glyph`-Implementierungen angepasst werden.

2.8.3 Die Iterator-Klasse und ihre Unterklassen

In diesem Fallbeispiel wird zur Definition einer allgemeinen Schnittstelle für den Zugriff und die Traversierung eine abstrakte Klasse namens **Iterator** verwendet. Dabei wird die Schnittstelle mittels konkreter Unterklassen wie **ArrayIterator** und **ListIterator** für den Zugriff auf Arrays und Listen implementiert, während **PreorderIterator**, **PostorderIterator** und vergleichbare Iteratoren für die Implementierung diverser Traversierungen auf spezifischen Strukturen sorgen. Jede Iterator-Unterklasse besitzt eine Referenz auf die Struktur, die sie traversiert. Die Instanzen der Unterklassen werden bei ihrer Erstellung mit dieser Referenz initialisiert. [Abbildung 2.20](#) illustriert die Iterator-Klasse mit einigen ihrer Unterklassen. Beachten Sie bitte, dass die Schnittstelle der `Glyph`-Klasse hier zur Unterstützung der Iteratoren um eine abstrakte Operation `CreateIterator` erweitert wurde.

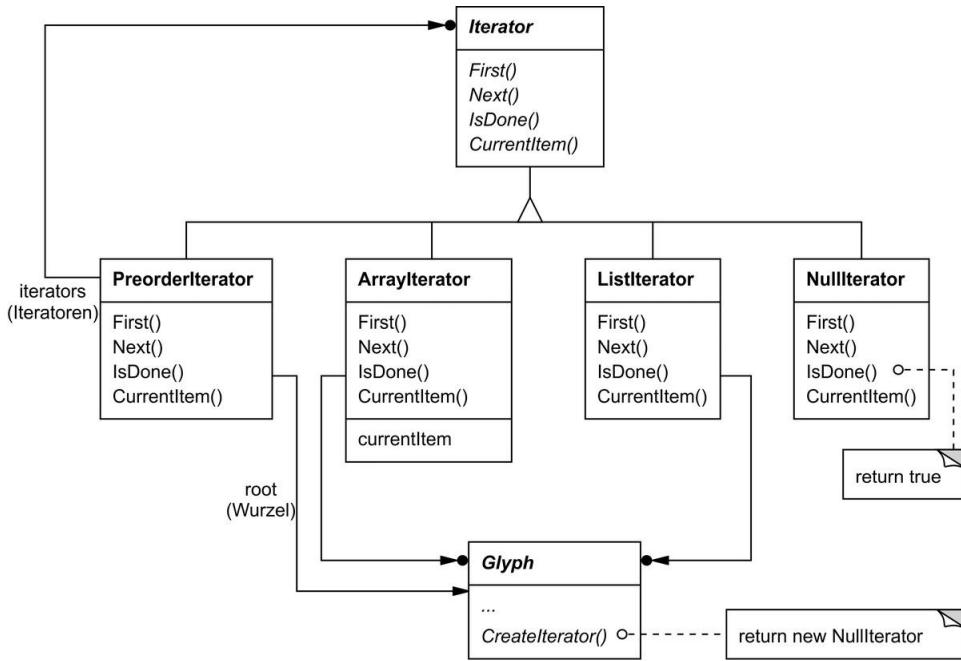


Abb. 2.20: Die *Iterator*-Klasse mit Unterklassen

Zur Steuerung der Traversierung stellt die `Iterator`-Schnittstelle die Operationen `First`, `Next` und `IsDone` bereit. Die Klasse `ListIterator` implementiert `First` in der Form, dass sie auf das erste Element der Liste zeigt, während `Next` den Iterator zum nächsten Element der Liste vorrückt. `IsDone` meldet, ob der Pointer hinter das letzte Element in der Liste zeigt. Und `currentItem` dereferenziert den Iterator, so dass er die `Glyphe` zurückgibt, auf die er zeigt. Eine `ArrayIterator`-Klasse würde – allerdings in Bezug auf ein Array der `Glyphen` – in ähnlicher Weise funktionieren.

Nun kann ohne Kenntnis von deren Repräsentation auf die Kindobjekte einer `Glyph`-Struktur zugegriffen werden:

```

Glyph* g;
Iterator<Glyph*>* i = g->CreateIterator();

for (i->First(); !i->IsDone ( ); i->Next()) {
    Glyph* child = i->CurrentItem();

    // Führt die Operation mit dem aktuellen Kindobjekt durch
}

```

`CreateIterator` gibt standardmäßig eine `NullIterator`-Instanz zurück. Ein `NullIterator` ist ein degenerierter Iterator für `Glyphen`, die keine Kindobjekte besitzen (auch »Blattglyphen« genannt, weil sie in einer Baumstruktur die Blätter an den Ästen darstellen). Die `IsDone`-Operation von `NullIterator` gibt immer `true` zurück.

Eine `Glyph`-Unterklasse mit Kindobjekten wird `CreateIterator` zwecks Rückgabe einer Instanz einer anderen `Iterator`-Unterklasse überschreiben. Welche Unterklasse das sein wird, ist von der Struktur abhängig, in der die Kindobjekte gespeichert sind. Sollte die `Row`-Unterklasse von `Glyph` deren Kindobjekte in einer Liste `_children` speichern, dann würde die `CreateIterator`-Operation folgendermaßen aussehen:

```
Iterator<Glyph*>* Row::CreateIterator() {
    return new ListIterator<Glyph*>(_children);
}
```

Iteratoren für Preorder- und Inorder-Traversierungen implementieren diese anhand von glyphspezifischen Iteratoren. Iteratoren für diese Traversierungen werden mit der Root-Glyphe der Struktur ausgestattet, die sie traversieren. Sie rufen `CreateIterator` auf den Glyphen in der Struktur auf und verwenden zur Überwachung der resultierenden Iteratoren einen Stack.

Die Klasse `PreorderIterator` bezieht den Iterator beispielsweise von der Root-Glyphe, initialisiert ihn so, dass er auf das erste Element zeigt, und legt ihn dann auf den Stack:

```
void PreorderIterator::First() {
    Iterator<Glyph*>* i = _root->CreateIterator();

    if (i) {
        i->First();
        _iterators.RemoveAll();
        _iterators.Push (i);
    }
}
```

`CurrentItem` ruft einfach `CurrentItem` desjenigen Iterators auf, der ganz oben auf dem Stack liegt:

```
Glyph* PreorderIterator::CurrentItem() const {
    return
        _iterators.Size() > 0 ?
            _iterators.Top()->CurrentItem() : 0;
}
```

Die `Next`-Operation nimmt den obersten Iterator vom Stack und fordert einen neu zu erzeugenden Iterator von seinem aktuellen Element an, um so weit wie möglich in der `Glyph`-Struktur hinabzusteigen (immerhin handelt es sich hier um eine Preorder-Traversierung). Sie setzt den neuen Iterator auf das erste Element in der Traversierung und schiebt ihn auf den Stack. Dann testet die `Next`-Operation den

neuesten Iterator. Wenn dessen `IsDone`-Operation `true` zurückgibt, ist die Traversierung des aktuellen Teilbaums (oder Blatts) beendet. In diesem Fall nimmt `Next` den obersten Iterator vom Stack und wiederholt den Vorgang, bis sie die nächste unvollständige Traversierung findet, sofern vorhanden. Andernfalls ist die Traversierung der Struktur damit abgeschlossen.

```
void PreorderIterator::Next() {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();

    i->First();
    _iterators.Push(i);
    while (
        _iterators.Size() > 0 && _iterators.Top()->IsDone()
    ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

Interessant ist hier, wie die `Iterator`-Klassenhierarchie die Ergänzung neuer Traversierungsarten ohne jegliche Modifikation der `Glyph`-Klassen ermöglicht: Es wird einfach eine Unterklasse von `Iterator` erzeugt und eine neue Traversierung hinzugefügt, wie dies auch schon mit `PreorderIterator` durchexerziert wurde. `Glyph`-Unterklassen verwenden dieselbe Schnittstelle, um den Clients Zugriff auf ihre Kindobjekte zu gewähren, ohne die zugrundeliegende verborgene Datenstruktur, in der sie gespeichert sind, preiszugeben. Da Iteratoren eine eigene Kopie vom Zustand einer Traversierung speichern, können gleich mehrere Traversierungen parallel ausgeführt werden – sogar auf derselben Struktur. Und auch wenn sich die Traversierungen in diesem Beispiel auf `Glyph`-Strukturen beziehen, spricht nichts dagegen, eine Klasse wie `PreorderIterator` auch anhand des Objekttyps in der Struktur zu parametrisieren. In C++ würden zu diesem Zweck Templates eingesetzt und die Mechanismen der Klasse `PreorderIterator` zur Traversierung anderer Strukturen wiederverwendet.

2.8.4 Das Design Pattern *Iterator (Iterator)*

Das Pattern *Iterator (Iterator*, siehe [Abschnitt 5.4](#)) macht sich die vorgenannten Techniken zur Unterstützung des Zugriffs auf und die Traversierung von Objektstrukturen zunutze. Es lässt sich nicht nur auf zusammengesetzte Strukturen, sondern auch auf Behälter anwenden. Zudem abstrahiert es den Traversierungsalgorithmus und schirmt die Clients von der internen Struktur der

traversierten Objekte ab. Auch dieses Pattern demonstriert einmal mehr, welchen Beitrag die Kapselung des variierenden Konzepts zur Flexibilität und Wiederverwendbarkeit leistet – und erfasst darüber hinaus noch zahlreiche weitere Aspekte in Bezug auf die überraschend tiefgreifende Problematik der Iteration, die in diesem Kapitel gar nicht angesprochen wurden.

2.8.5 Traversierung kontra Traversierungsaktionen

Nachdem nun die Möglichkeit besteht, die `Glyph`-Struktur zu traversieren, kann es mit der Rechtschreibprüfung und der Silbentrennung weitergehen. Für beide Analysevorgänge müssen während der Traversierung Daten gesammelt werden.

Als Erstes muss jedoch die Zuständigkeit für die Analyse zugewiesen werden. Diese könnte beispielsweise den `Iterator`-Klassen übertragen werden, wodurch der Analysevorgang selbst zu einem integralen Bestandteil der Traversierung würde. Mehr Flexibilität und ein höheres Potenzial für Wiederverwendbarkeit lässt sich allerdings durch eine separate Handhabung der Traversierung und der dabei durchgeführten Aktionen erzielen – weil unterschiedliche Analysen häufig auf derselben Traversierungsart basieren und daher derselbe Iteratorenansatz für verschiedene Analysevarianten wiederverwendet werden kann. Ein Beispiel dafür ist die Preorder-Traversierung, die in vielen Analysen zur Anwendung kommt, einschließlich der Rechtschreibprüfung, der Silbentrennung, der Vorwärtssuche und der Wortzählung.

Gut, die Analyse und die Traversierung sollten also möglichst separiert werden. Aber wie könnte man die Zuständigkeit für die Analyse noch zuweisen? In jedem Analyseprozess werden zu verschiedenen Zeitpunkten der Traversierung unterschiedliche Aktionen ausgeführt. Je nach Art der Analyse stehen dabei bestimmte Glyphen mehr im Vordergrund als andere: Bei der Rechtschreibprüfung oder Silbentrennung werden nur die Zeichenglyphen, nicht aber die grafischen Glyphen wie Linien und Pixelgrafiken berücksichtigt. Zur Untersuchung der »Farbe« eines Dokuments, d. h. der Verteilung der sichtbaren Objekte, werden nur die sichtbaren Glyphen, nicht aber die nicht sichtbaren überprüft. Mit anderen Worten: Jede Analysevariante betrifft in der Regel verschiedene Glyphen.

Deshalb muss die Analyse selbst natürlich in der Lage sein, die diversen Glyphenarten voneinander zu unterscheiden. Ein Ansatz, der sich hierfür anbietet, wäre, die `Glyph`-Klassen selbst unmittelbar mit den analytischen Fähigkeiten auszustatten. So könnten beispielsweise für jeden Analysevorgang ein oder mehrere abstrakte Operationen in der `Glyph`-Klasse ergänzt werden, während die

Unterklassen diese entsprechend der Rolle implementieren, die sie in der Analyse jeweils einnehmen.

Das Problem dabei ist allerdings, dass dann mit jeder neu hinzukommenden Analyseart auch jede einzelne `Glyph`-Klasse angepasst werden muss. In manchen Fällen kann man dem jedoch entgegenwirken: Wenn nur einige wenige Klassen an der Analyse beteiligt sind oder wenn die meisten Klassen den Analyseprozess auf dieselbe Art und Weise ausführen, kann eine Standardimplementierung für die abstrakte Operation in der `Glyph`-Klasse eingerichtet werden – und diese Standardoperation würde dann die allgemeinen Fälle abdecken. Dadurch wären die vorzunehmenden Anpassungen nur auf die `Glyph`-Klasse und jene Unterklassen beschränkt, die von der Norm abweichen.

Aber auch wenn eine Standardimplementierung die Anzahl der erforderlichen Anpassungen reduziert, bleibt immer noch ein Problem übrig: Die `Glyph`-Schnittstelle wird mit jeder neuen analytischen Fähigkeit umfangreicher. Und mit der Zeit werden die analytischen Operationen die grundlegende `Glyph`-Schnittstelle immer unübersichtlicher machen, so dass nur noch schwer zu erkennen ist, dass ihr Hauptzweck in der Definition und Strukturierung von Objekten mit einem bestimmten optischen Erscheinungsbild bzw. einer bestimmten Form haben – diese Schnittstelle wird schlichtweg verschleiert.

2.8.6 Kapselung der Analyse

Angesichts all dieser Störfaktoren sollte die Analyse in einem separaten Objekt gekapselt werden, wie dies auch an anderer Stelle schon mehrfach geschehen ist. Dazu könnte die Implementierung der betreffenden Analyse in einer eigenen Klasse untergebracht und anschließend eine Instanz dieser Klasse in Verbindung mit einem passenden Iterator genutzt werden, der wiederum diese Instanz in jede Glyphe der Struktur befördert. Auf diese Weise könnte das Analyseobjekt zu jedem Zeitpunkt der Traversierung einen Teil der Analysearbeit ausführen. Das analysierende Objekt, der sogenannte **Analyzer**, sammelt während des Traversierungsvorgangs die jeweils relevanten Daten (in diesem Fall Zeichen):

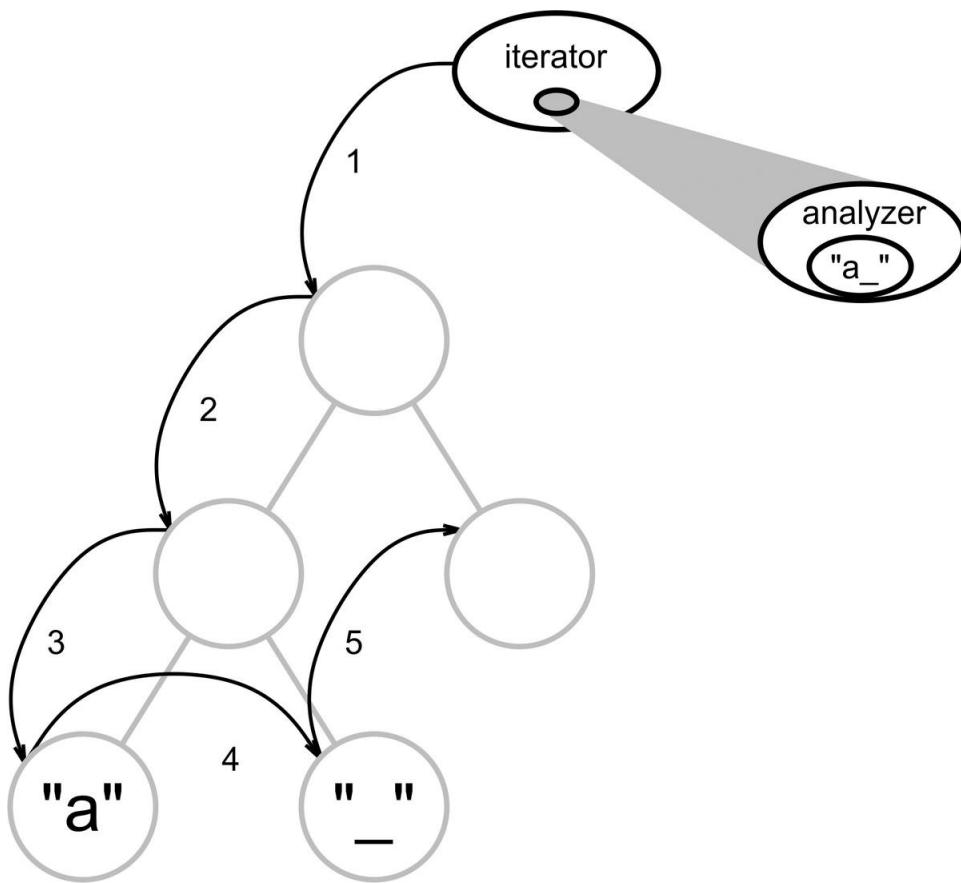


Abb. 2.21: Kapselung einer Analyse

Die grundlegende Frage, die sich bei diesem Ansatz stellt, lautet: Wie unterscheidet das Analyseobjekt die verschiedenen Glyphenarten, ohne spezielle Typprüfungen oder Downcasts vorzunehmen? Ein unschöner (Pseudo-)Code wie der folgende sollte jedenfalls nicht in einer Klasse SpellingChecker enthalten sein:

```
void SpellingChecker::Check (Glyph* glyph) {
    Character* c;
    Row* r;
    Image* i;

    if (c = dynamic_cast<Character*>(glyph)) {
        // Analysiert Zeichen

    } else if (r = dynamic_cast<Row*>(glyph)) {
        // Bereitet die Analyse der Kindobjekte von r vor

    } else if (i = dynamic_cast<Image*>(glyph)) {
        // Tut nichts
    }
}
```

Dieses unansehnliche Codefragment stützt sich nicht nur auf relativ unsichere

Fähigkeiten wie typsichere Downcasts, sondern ist darüber hinaus auch schwer zu erweitern: Der Body der Funktion muss bei jeder Änderung innerhalb der `Glyph`-Klassenhierarchie unweigerlich angepasst werden – und prinzipiell ist das genau die Art von Code, die in objektorientierten Programmiersprachen unerwünscht ist.

Ein derartiger Brute-Force-Ansatz kommt also nicht infrage – aber wie lässt sich das Problem auf andere Weise lösen? Man könnte die `Glyph`-Klasse beispielsweise um folgende abstrakte Operation ergänzen:

```
void CheckMe(SpellingChecker&)
```

und `CheckMe` in jeder `Glyph`-UnterkLASSE wie folgt definieren:

```
void GlyphSubclass::CheckMe (SpellingChecker& checker) {
    checker.CheckGlyphSubclass(this);
}
```

wobei `GlyphSubclass` durch den Namen der `Glyph`-UnterkLASSE zu ersetzen wäre. Zu beachten ist dabei, dass die spezifische `Glyph`-UnterkLASSE beim Aufruf von `CheckMe` bekannt ist – immerhin findet das Ganze ja in einer ihrer Operationen statt. Und die Klassenschnittstelle `SpellingChecker` wiederum enthält für jede `Glyph`-UnterkLASSE eine Operation wie `CheckGlyphSubclass`:

```
class SpellingChecker {
public:
    SpellingChecker();

    virtual void CheckCharacter(Character* );
    virtual void CheckRow(Row* );
    virtual void CheckImage(Image* );

    // ... und so weiter

    List<char*>& GetMisspellings();

protected:
    virtual bool IsMisspelled(const char* );

private:
    char _currentWord[MAX_WORD_SIZE];
    List<char*> _misspellings;
};
```

Hinweis

Da sich diese Memberfunktionen schon durch ihre Parameter unterscheiden, könnte man ihnen im Rahmen der Funktionsüberladung in C++ auch identische Namen zuweisen. Zur Verdeutlichung der Unterschiede – insbesondere beim Aufruf – werden in diesem Beispiel jedoch individuelle Bezeichnungen verwendet.

Die Prüfoperation von SpellingChecker für Character-Glyphen könnte wie folgt aussehen:

```
void SpellingChecker::CheckCharacter (Character* c) {
    const char ch = c->GetCharCode();

    if (isalpha(ch)) {
        // Hängt ein alphabetisches Zeichen an _currentWord an

    } else {
        // Findet ein nicht-alphabetisches Zeichen

        if (IsMisspelled(_currentWord)) {
            // Fügt _currentWord zu _misspellings hinzu
            _misspellings.Append(strdup(_currentWord));
        }

        _currentWord[0] = ' \0 ';
        // Setzt _currentWord auf check next word zurück
    }
}
```

Das Besondere an diesem Code ist, dass hier eine spezielle, ausschließlich auf die Character-Klasse bezogene GetCharCode-Operation definiert wurde. Somit ist der SpellingChecker in der Lage, unterklassenspezifische Operationen zu bewältigen, ohne auf Typprüfungen oder Downcasts zurückgreifen zu müssen – und dadurch ist eine individuelle Behandlung der Objekte möglich.

CheckCharacter sammelt die alphabetischen Zeichen im _currentWord-Puffer. Beim Auffinden eines nicht-alphabetischen Zeichens, beispielsweise eines Unterstrichs, kommt die IsMisspelled-Operation zum Einsatz, um die korrekte Rechtschreibung des in _currentWord abgelegten Wortes zu überprüfen. Ist der Begriff falsch geschrieben, fügt CheckCharacter ihn zu der Liste der falsch geschriebenen Wörter hinzu. Anschließend muss der _currentWord-Puffer in Vorbereitung auf die nächste Wortprüfung zunächst gelöscht werden. Nach Beendigung der Traversierung lässt sich die Liste der falsch geschriebenen Wörter mithilfe der GetMisspellings-Operation abrufen.

Hinweis

`IsMisspelled` implementiert einen Algorithmus zur Rechtschreibprüfung, der an dieser Stelle allerdings nicht näher erläutert werden soll, da er nicht zwingend an das Lexi-Design gebunden ist. Generell gilt jedoch: Die Unterstützung mehrerer verschiedener Algorithmen kann durch die Bildung von `SpellingChecker`-Unterklassen oder (wie schon bei der Formatierung, siehe [Abschnitt 2.3](#)) alternativ auch durch die Anwendung des Design Patterns *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#)) gewährleistet werden.

```
SpellingChecker spellingChecker;
Composition* c;

// ...

Glyph* g;
PreorderIterator i(c);
for (i.First(); !i.IsDone(); i.Next()) {
    g = i.CurrentItem();
    g->CheckMe(spellingChecker);
}
```

Das in [Abbildung 2.22](#) dargestellte Interaktionsdiagramm veranschaulicht das Zusammenwirken der Character-Glyphen und des `SpellingChecker`:

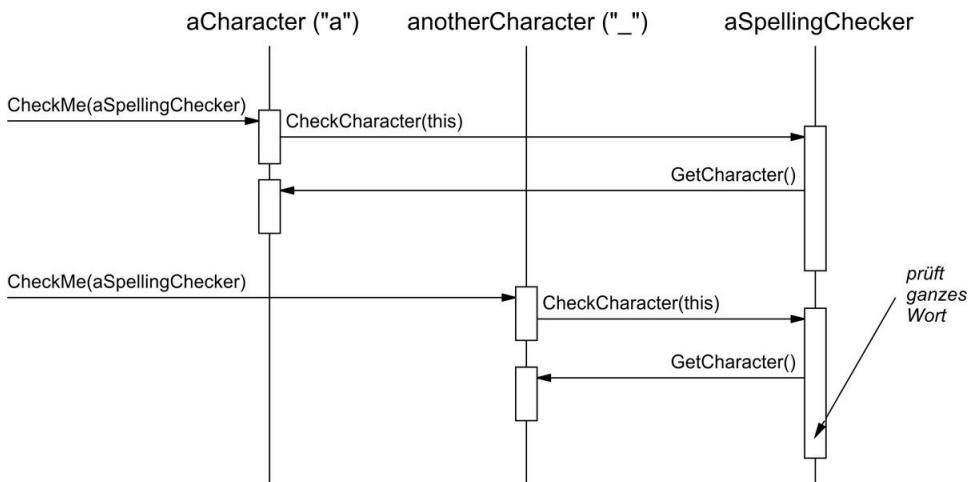


Abb. 2.22: Interaktion zwischen *Character-Glyphen* und *SpellingChecker*

Zum Aufspüren von Rechtschreibfehlern ist dieser Ansatz also geeignet – aber inwiefern soll er zur Unterstützung verschiedener Analysearten beitragen? Allem

Anschein nach muss doch immerhin für jede neue Analyseart eine Operation wie `CheckMe(SpellingChecker&)` zur `Glyph`-Klasse und ihren Unterklassen hinzugefügt werden. Das ist richtig, aber nur dann, wenn tatsächlich für jede Analyse eine *unabhängige* Klasse gefordert ist. Ansonsten spricht jedoch nichts dagegen, *allen* Analyseklassen dieselbe Schnittstelle mitzugeben – wodurch sie sich polymorph verwenden lassen. Und das wiederum bedeutet, dass analysespezifische Operationen wie `CheckMe(SpellingChecker&)` durch eine analyseunabhängige Operation ersetzt werden können, die mit einem allgemeineren Parameter auskommt.

2.8.7 Die Visitor-Klasse und ihre Unterklassen

Die im Folgenden verwendete Bezeichnung **Visitor (Besucher)** bezieht sich allgemein auf Klassen, deren Objekte im Rahmen einer Traversierung zu irgendeinem gerade vorliegenden Zweck andere Objekte »aufsuchen«. In diesem Fallbeispiel wird eine `Visitor`-Klasse definiert, die ihrerseits eine abstrakte Schnittstelle zum »Aufsuchen« von Glyphen in einer Struktur definiert.

Hinweis

Mit »Aufsuchen« bzw. mit der Bezeichnung »Visitor« (dt. »Besucher«) ist hier im Prinzip eine allgemein gefasstere Form des »Analysierens« gemeint. Damit wird an dieser Stelle bereits auf die Terminologie des im nächsten Abschnitt vorgestellten Design Patterns vorgegriffen.

```
class Visitor {
public:
    virtual void VisitCharacter(Character*) { }
    virtual void VisitRow(Row*) { }
    virtual void VisitImage(Image*) { }

    // ... und so weiter
};
```

Die konkreten Unterklassen von `Visitor` führen unterschiedliche Analysen aus. So könnte beispielsweise eine Unterklasse `SpellingCheckingVisitor` für die Rechtschreibprüfung und eine Unterklasse `HyphenationVisitor` für die Silbentrennung genutzt werden. Dabei würde `SpellingCheckingVisitor` vom Prinzip her genauso implementiert werden wie zuvor der `SpellingChecker`, mit

dem Unterschied, dass die Operationsnamen auf die allgemeinere `Visitor`-Schnittstelle Bezug nehmen und `CheckCharacter` daher in diesem Fall mit `VisitCharacter` benannt werden würde.

Da ein Name wie `CheckMe` für `Visitors`, die gar keine Überprüfung vornehmen, nicht angebracht wäre, wird hier eine neutralere Bezeichnung gewählt: `Accept`. Außerdem wird der Argumenttyp dieser Unterkategorie in `Visitor`& geändert, damit deutlich wird, dass sie jeden beliebigen `Visitor` akzeptiert. Die Ergänzung einer neuen Analyseart erfordert also lediglich die Definition einer neuen Unterkategorie von `Visitor` – ansonsten bleiben die `Glyph`-Klassen unangetastet. Zur Unterstützung aller zukünftigen Analysen muss nur diese eine Operation zu `Glyph` und ihren Unterklassen hinzugefügt werden.

Wie die Rechtschreibprüfung funktioniert, wurde ja bereits beschrieben. Was die Silbentrennung angeht, wird in der Unterkategorie `HyphenationVisitor` ein ganz ähnlicher Ansatz verfolgt, um den zu überprüfenden Text »einzusammeln« – hat deren Operation `VisitCharacter` dann aber erst einmal ein vollständiges Wort beisammen, geht es etwas anders weiter: Statt den Begriff auf Rechtschreibfehler zu untersuchen, wird hier ein Silbentrennungsalgorithmus angewendet, um die potenziellen Silbentrennstellen in dem Wort zu finden, sofern solche vorhanden sind. Danach wird an jeder Trennstelle eine `Glyphe` mit einem **bedingten Trennstrich** in die Komposition eingefügt. Bedingte Trennstriche sind Instanzen der Klasse `Discretionary`, einer Unterkategorie von `Glyph`.

Ein bedingter Trennstrich besitzt – je nachdem, ob er das letzte Zeichen in einer Zeile ist oder nicht – zwei Darstellungsformen: Ist er das letzte Zeichen, wird er als Bindestrich angezeigt, andernfalls bleibt er unsichtbar. Zunächst fragt das `Discretionary`-Objekt bei seinem Elternobjekt (einem `Row`-Objekt) ab, ob es das letzte seiner Kindobjekte ist – diese Abfrage erfolgt immer dann, wenn es aufgefordert wird, sich selbst zu zeichnen oder seine Ausmaße zu berechnen. Die Formatierungsstrategie behandelt `Discretionary`-Objekte wie Leerraum, weshalb sie auch als letztes Zeichen einer Zeile infrage kommen können. [Abbildung 2.23](#) zeigt, wie ein bedingter Trennstrich dargestellt werden kann:

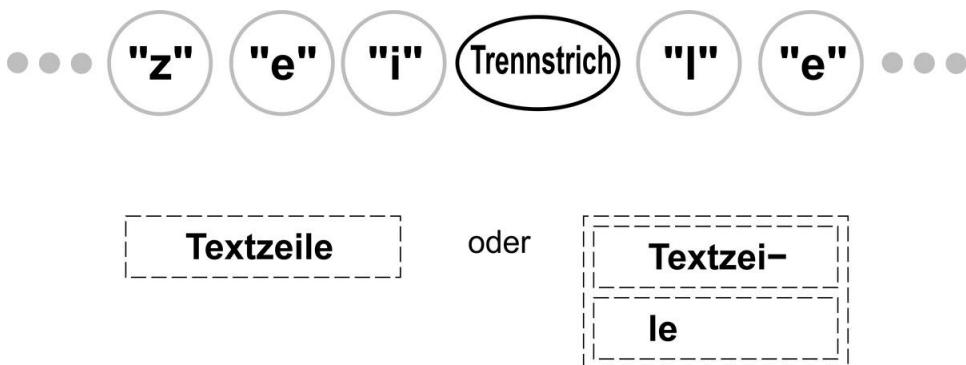


Abb. 2.23: Darstellungsformen des bedingten Trennstrichs

2.8.8 Das Design Pattern *Visitor* (*Besucher*)

Die in diesem Abschnitt beschriebenen Vorgänge entsprechen der Anwendung des Design Patterns *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)), deren zentrale Teilnehmer die *Visitor*-Klasse und ihre Unterklassen sind. Das *Visitor*-Pattern erfasst die in diesem Beispiel verwendeten Techniken zur Ermöglichung einer unbeschränkten Anzahl von Analysearten für Glyphenstrukturen, ohne dass eine Änderung der *Glyph*-Klassen erforderlich wird. Eine weitere nette Eigenschaft der *Visitor*-Objekte ist, dass sie nicht nur auf Kompositionen wie die hier vorgestellten *Glyph*-Strukturen, sondern auf *jede beliebige* Objektstruktur angewendet werden können – unter anderem also auch auf Objektsätze, Listen und sogar gerichtete azyklische Graphen. Zudem brauchen die Klassen, die ein *Visitor* aufsuchen kann, nicht durch eine gemeinsame Basisklasse in einer verwandschaftlichen Beziehung zueinander zu stehen, d. h., *visitor*-Objekte können über Klassenhierarchien hinweg arbeiten.

Eine wichtige Frage, die man sich vor der Anwendung des Design Patterns *Visitor* (*Besucher*) stellen sollte, lautet: Welche Klassenhierarchien ändern sich am häufigsten? Ideal geeignet ist dieses Pattern dann, wenn die Objekte einer stabilen Klassenstruktur für viele verschiedene Zwecke und Aufgaben vorgesehen sind. Die Ergänzung einer neuen *Visitor*-Variante bedingt keinerlei Änderung der betreffenden Klassenstruktur – ein sehr bedeutsames Kriterium insbesondere für umfangreiche Klassenstrukturen. Wird allerdings eine Unterklasse zu der Struktur hinzugefügt, müssen auch alle *visitor*-Schnittstellen eine *visit...*-Operation für diese Unterklasse erhalten. Auf dieses Fallbeispiel bezogen, würde das Hinzufügen einer neuen *Glyph*-Unterklasse namens *Foo* also dazu führen, dass die *Visitor*-Klasse und alle ihre Unterklassen um eine Operation *visitFoo* erweitert werden müssten. Angesichts der für das Lexi-Design geltenden Rahmenbedingungen ist es allerdings wahrscheinlicher, dass die Anwendung um eine neue Analyseart erweitert

wird, als um eine neue Glyphenart – und insofern ist das Design Pattern *Visitor* (*Besucher*) hier prima geeignet.

2.9 Zusammenfassung

In diesem Fallbeispiel wurden acht verschiedene Patterns auf das Lexi-Design angewendet:

1. *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) zur Darstellung der physischen Dokumentstruktur,
2. *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#)) zur Unterstützung verschiedener Formatierungsalgorithmen,
3. *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#)) zur Ausgestaltung der Benutzeroberfläche,
4. *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) zur Unterstützung mehrerer Look-and-Feel-Standards,
5. *Bridge* (*Brücke*, siehe [Abschnitt 4.2](#)) zur Unterstützung verschiedener Fensterplattformen,
6. *Command* (*Befehl*, siehe [Abschnitt 5.2](#)) zur Rückgängigmachung userseitiger Operationen,
7. *Iterator* (*Iterator*, siehe [Abschnitt 5.4](#)) für den Zugriff auf und die Traversierung von Objektstrukturen und
8. *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)), um eine unbeschränkte Anzahl von Analysearten zu ermöglichen, ohne die Implementierung der Dokumentstruktur zu verkomplizieren.

Keine der in diesem Kapitel betrachteten Designproblemstellungen beschränkt sich ausschließlich auf Texteditoren wie Lexi. Tatsächlich bieten sich im Allgemeinen selbst in den einfachsten Anwendungen Einsatzmöglichkeiten für viele der genannten Patterns, wenn auch möglicherweise mit anderen Zielsetzungen: In einer Software für Finanzanalysen könnte zum Beispiel das Design Pattern *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) angewendet werden, um aus Unterportfolios und verschiedenartigen Depots bestehende Investment-Portfolios zu definieren. Ein

Compiler könnte dagegen das Pattern *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#)) verwenden, um mehrere Registerzuweisungsschemata für diverse Zielmaschinen zu ermöglichen. Und Anwendungen mit einer grafischen Benutzeroberfläche – wie auch dieses Fallbeispiel – werden sich wiederum vermutlich zumindest die Design Patterns *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#)) und *Command* (*Befehl*, siehe [Abschnitt 5.2](#)) zunutze machen.

Obgleich in diesem Kapitel bereits einige wichtige Problemstellungen im Lexi-Design aufgezeigt wurden, gibt es darüber hinaus natürlich noch viele weitere Designfragen, auf die an dieser Stelle nicht eingegangen werden konnte. Im weiteren Verlauf dieses Buches werden allerdings neben den acht in diesem Fallbeispiel verwendeten noch zahlreiche andere Design Patterns ausführlich vorgestellt. Denken Sie also bei der weiteren Lektüre ruhig auch einmal darüber nach, wie sich jedes einzelne dieser Patterns in Lexi einsetzen ließe – oder besser noch, wie Sie sie nutzbringend in Ihren eigenen Designs anwenden könnten!

Design-Patterns-Katalog

- **Kapitel 3**

Erzeugungsmuster (Creational Patterns)

- **Kapitel 4**

Strukturmuster (Structural Patterns)

- **Kapitel 5**

Verhaltensmuster (Behavioral Patterns)

Kapitel 3: Erzeugungsmuster (**Creational Patterns**)

Die als **Erzeugungsmuster (Creational Patterns)** klassifizierten Design Patterns abstrahieren den Instanziierungsprozess. Sie sorgen dafür, dass ein System unabhängig von der Generierung, Komposition und Darstellung seiner Objekte funktioniert. Dabei bedienen sich *klassenbasierte* Erzeugungsmuster des Vererbungsprinzips, um die Klassen der zu erzeugenden Objekte zu variieren, während *objektbasierte* Erzeugungsmuster die Instanziierung an andere Objekte delegieren.

Der Einsatz von Erzeugungsmustern gewinnt vor allem dann an Bedeutung, wenn sich die Systeme zunehmend auf die Objektkomposition statt auf die Vererbung stützen: Hier verlagert sich die Akzentuierung von einem innerhalb des Codes fest vorgegebenen Verhalten hin zur Definition einer kleineren Auswahl grundlegender Verhaltensvorgaben, die zu einer beliebigen Anzahl komplexerer Verhaltensweisen zusammengesetzt werden können. Insofern bedarf es für die Erzeugung von Objekten, die ein spezifisches Verhalten aufweisen sollen, schon etwas mehr als nur der Instanziierung einer Klasse.

Erzeugungsmuster folgen zwei stets wiederkehrenden Leitmotiven: Zum einen kapseln sie die Informationen zu den vom System verwendeten konkreten Klassen, und zum anderen verbergen sie die Art und Weise, wie die Instanzen dieser Klassen erzeugt und zusammengeführt werden. Im Wesentlichen sind dem System die Schnittstellen der Objekte lediglich in der Form bekannt, wie sie von den abstrakten Klassen definiert sind. Dadurch bieten die Erzeugungsmuster viel Flexibilität in Bezug darauf, *was* erzeugt wird, *wer* es erzeugt sowie *wie* und *wann* es erzeugt wird: Sie ermöglichen die Konfiguration eines Systems mithilfe von »Produktobjekten«, die in Struktur und Funktionalität stark variieren können – und die Konfiguration kann statisch (also beim Kompilieren) oder auch dynamisch (sprich zur Laufzeit) erfolgen.

Mitunter können Erzeugungsmuster einen konkurrierenden Stellenwert haben, wie etwa in Situationen, in denen z. B. sowohl das Pattern *Prototype* (*Prototyp*, siehe [Abschnitt 3.4](#)) als auch *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) nutzbringend angewendet werden könnte. In anderen Fällen lassen sie sich aber auch komplementär nutzen, beispielsweise wenn sich das Pattern *Builder* (*Erbauer*, siehe

[Abschnitt 3.2](#)) zur gezielten Zusammenführung von Komponenten eines anderen Design Patterns bedient oder das Design Pattern *Prototype* (*Prototyp*, siehe [Abschnitt 3.4](#)) das Pattern *Singleton* (*Singleton*, siehe [Abschnitt 3.5](#)) für seine Implementierung verwendet.

Da die Erzeugungsmuster alle in einer engen verwandschaftlichen Beziehung zueinander stehen, werden im Folgenden die Gemeinsamkeiten und Unterschiede aller fünf Design Patterns dieser Kategorie nacheinander betrachtet. Außerdem wird die Implementierung jedes einzelnen Patterns jeweils anhand des klassischen Beispiels der Entwicklung eines Labyrinth-Spiels demonstriert, wobei sich die Irrgärten bzw. der Spielaufbau von Pattern zu Pattern leicht unterscheiden: In einigen Fällen besteht das Spielkonzept schlicht darin, einen Weg aus dem Labyrinth zu finden, d. h., der Spieler hat gegebenenfalls lediglich eine standortgebundene Sicht auf das Labyrinth. Die Irrgärten anderer Programmbeispiele sind dagegen zusätzlich mit diversen Hindernissen und Gefahrenquellen ausgestattet, so dass hier möglicherweise eine Kartenansicht des bereits erkundeten Labyrinth-Bereichs zur Verfügung gestellt wird.

Generell wird die detaillierte Ausgestaltung der Irrgärten ebenso wie die Frage, ob das Spiel im Single- oder Multiplayer-Modus zur Verfügung steht, in den folgenden Beispielen weitestgehend außer Acht gelassen. Hier soll es in erster Linie um den eigentlichen Erzeugungsprozess der Labyrinthe gehen, die als individuelle Ansammlungen von Räumen definiert sind, wobei jeder Raum Kenntnis von seinen Nachbarobjekten hat, sprich einem weiteren Raum, einer Wand oder einer Tür.

Die Komponenten der Labyrinthe sind in allen Beispielen durch die Klassen Room, Door und wall definiert, allerdings werden immer nur die für die Erzeugung des betreffenden Irrgartens relevanten Teile dieser Klassen definiert. Sowohl die Spieler als auch die Operationen für die Darstellung und Fortbewegung in einem Labyrinth sowie andere maßgebliche Funktionen, die prinzipiell nicht mit der Erzeugung des Labyrinths zusammenhängen, werden ignoriert.

Das in [Abbildung 3.1](#) dargestellte Diagramm veranschaulicht die Beziehungen zwischen den diversen Klassen:

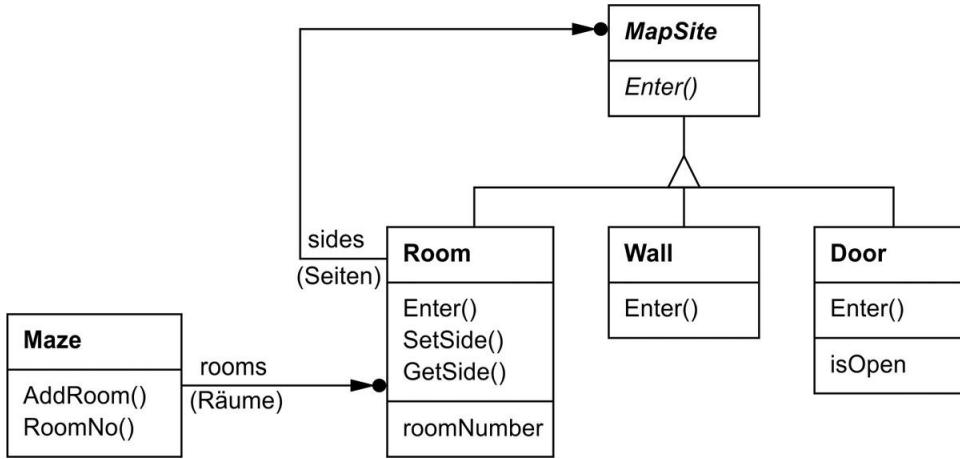


Abb. 3.1: Beziehungen zwischen den für die Labyrinthe verwendeten Klassen

Jeder Raum hat vier Seiten. In C++-Implementierungen erfolgt die Spezifizierung der Nord-, Süd-, Ost- und Westseiten eines Raums mithilfe des Aufzählungstyps **Direction**:

```
enum Direction {North, South, East, West};
```

In Smalltalk-Implementierungen werden zu diesem Zweck entsprechende Symbole verwendet.

MapSite ist die gemeinsame abstrakte Klasse für alle Komponenten eines Labyrinths. Der Einfachheit halber definiert **MapSite** in diesem Beispiel lediglich eine einzige Operation, **Enter**, deren Verhalten davon abhängt, wohin sich der Spieler bewegt: Betritt er einen Raum, ändern sich seine Standortkoordinaten. Erreicht er hingegen eine Tür, gibt es zwei Möglichkeiten: Ist die Tür unverschlossen, gelangt er in den nächsten Raum – ist sie jedoch verschlossen, stößt er sich bloß die Nase.

```
class MapSite {
public:
    virtual void Enter() = 0;
};
```

Enter bildet eine einfache Grundlage für komplexere Spieloperationen. Befindet sich der Spieler beispielsweise in einem Raum und erteilt den Befehl **GO EAST (GEHE NACH OSTEN)**, kann das Spiel ohne Probleme ermitteln, welche **MapSite** unmittelbar östlich seiner aktuellen Position liegt und **Enter** dafür auslösen – und die unterklassenspezifische **Enter**-Operation überprüft daraufhin, ob sich die Position des Spielers geändert oder er sich nur die Nase gestoßen hat. In einem richtigen Spiel könnte **Enter** auch das fortzubewegende Spielerobjekt als Argument erhalten.

`Room` ist die konkrete Unterklasse von `MapSite`, die die Schlüsselbeziehungen zwischen den Labyrinth-Komponenten definiert. Sie enthält Referenzen auf andere `MapSite`-Objekte und speichert eine Raumnummer, die den betreffenden Raum im Labyrinth identifiziert:

```
class Room : public MapSite {  
public:  
    Room(int roomNo);  
  
    MapSite* GetSide(Direction) const;  
    void SetSide(Direction, MapSite*);  
  
    virtual void Enter();  
  
private:  
    MapSite* _sides[4];  
    int _roomNumber;  
};
```

Die folgenden Klassen repräsentieren die Wände oder Türen auf jeder Seite eines Raums:

```
class Wall : public MapSite {  
public:  
    Wall();  
  
    virtual void Enter();  
};  
  
class Door : public MapSite {  
public:  
    Door(Room* = 0, Room* = 0);  
  
    virtual void Enter();  
    Room* OtherSideFrom(Room*);  
  
private:  
    Room* _room1;  
    Room* _room2;  
    bool _isOpen;  
};
```

Neben den Komponenten des Labyrinths müssen aber natürlich auch noch andere Faktoren bestimmt werden. Deshalb wird nun eine Klasse `Maze` definiert, die eine Ansammlung von Räumen repräsentiert. Diese Klasse ist dank ihrer `RoomNo`-Operation ebenfalls in der Lage, einen spezifischen Raum anhand seiner Raumnummer zu bestimmen:

```

class Maze {
public:
    Maze();

    void AddRoom(Room* );
    Room* RoomNo(int) const;

private:
    // ...
};

```

RoomNo könnte den Raum z. B. mittels einer linearen Suche, einer Hashtabelle oder auch eines einfachen Arrays ausfindig machen – solche Details spielen für die in diesem Kapitel verwendeten Beispiele aber keine Rolle. Hier soll es stattdessen vorrangig darum gehen, wie sich die Komponenten eines Labyrinth-Objekts spezifizieren lassen.

Als Nächstes wird die Klasse `MazeGame` definiert, die das Labyrinth erzeugt. Dies lässt sich auf direktem Wege durch eine einfache Abfolge von Operationen bewerkstelligen, die einzelne Komponenten zu dem Labyrinth hinzufügen und sie dann miteinander verbinden. Die folgende Memberfunktion generiert beispielsweise ein Labyrinth mit zwei Räumen, die durch eine Tür miteinander verbunden sind:

```

Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room (2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}

```

Angesichts der Tatsache, dass hier einfach nur ein Labyrinth mit zwei Räumen erzeugt wird, erscheint diese Funktion ziemlich komplex – es geht aber auch einfacher. Beispielsweise könnte der `Room`-Konstruktor die Seiten mit den Wänden

bereits im Voraus initialisieren. Dadurch würde der zugehörige Code jedoch nur ausgelagert bzw. an eine andere Stelle verschoben. Das eigentliche Problem bei dieser Memberfunktion ist allerdings nicht ihre Größe, sondern ihre *mangelnde Flexibilität*: Weil das Layout des Labyrinths hartkodiert ist, bedingt jede Veränderung daran unweigerlich auch entsprechende Anpassungen an der Memberfunktion – sei es durch Überschreiben bzw. komplette Reimplementierung oder durch die Änderung einzelner Codefragmente, wobei Letzteres eine nicht nur fehleranfällige, sondern auch wenig wiederverwendungsfreundliche Lösung darstellt.

Hier weisen die Erzeugungsmuster den Weg zu einem *flexibleren*, wenn auch nicht unbedingt weniger umfangreichen Design – vor allem in Bezug auf eine einfachere Anpassung der Klassen, die die Komponenten des Labyrinths definieren.

Angenommen, für die Entwicklung eines neuen Spiels soll ein bereits existierendes Labyrinth-Layout wiederverwendet und um zusätzliche Komponenten erweitert werden – etwa `Door` `NeedingSpell` (eine Tür, die sich nur mithilfe eines Zauberspruchs öffnen und verschließen lässt) und `EnchantedRoom` (ein Raum, in dem sich besondere Gegenstände wie magische Schlüssel oder Zaubersprüche befinden). Wie ließe sich die Klasse `CreateMaze` auf einfache Art und Weise so modifizieren, dass sie Labyrinthe mit diesen neuen Objektklassen erzeugt?

Der größte Problemfaktor ist in diesem Fall die Hartkodierung der zu instanziiierenden Klassen – erfreulicherweise stellen die Erzeugungsmuster jedoch diverse Möglichkeiten bereit, um explizite Referenzen auf konkrete Klassen aus dem Instanziierungscode zu entfernen:

- Wenn `CreateMaze` zur Erzeugung der benötigten Räume, Wände und Türen anstelle der Konstruktoren virtuelle Funktionen aufruft, dann lassen sich die zu instanziiierenden Klassen durch Anlegen einer Unterklasse von `MazeGame` und Neudefinition dieser virtuellen Funktionen modifizieren. Dieser Ansatz wird z. B. mit dem Design Pattern *Factory Method (Fabrikmethode*, siehe [Abschnitt 3.3](#)) verfolgt.
- Wenn `CreateMaze` zur Erzeugung von Räumen, Wänden und Türen ein Objekt als Parameter übergeben wird, dann ist eine Anpassung der jeweils zugehörigen Klassen einfach durch die Übergabe eines anderen Parameters möglich. Dieser Ansatz wird z. B. mit dem Design Pattern *Abstract Factory (Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) verfolgt.
- Wenn `CreateMaze` ein Objekt übergeben wird, das mithilfe von Operationen

zum Hinzufügen von Räumen, Türen und Wänden ein komplettes neues Labyrinth erzeugt, dann können Teile des Labyrinths oder dessen Komposition mittels Vererbung verändert werden. Dieser Ansatz wird z. B. mit dem Design Pattern *Builder* (*Erbauer*, siehe [Abschnitt 3.2](#)) verfolgt.

- Wenn `CreateMaze` mit verschiedenen prototypischen Raum-, Tür- und Wandobjekten parametrisiert wird, die kopiert und zum Labyrinth ergänzt werden, dann brauchen diese Objekte zur Anpassung der Labyrinth-Komposition lediglich durch andere ersetzt zu werden. Dieser Ansatz wird z. B. mit dem Design Pattern *Prototype* (*Prototyp*, siehe [Abschnitt 3.4](#)) verfolgt.

Und das fünfte Erzeugungsmuster, *Singleton* (*Singleton*, siehe [Abschnitt 3.5](#)), kann schließlich sicherstellen, dass immer nur ein Labyrinth pro Spiel existiert und alle Spielobjekte ohne weitere Umwege über globale Variablen oder Funktionen direkten Zugriff darauf haben. Des Weiteren ermöglicht dieses Design Pattern auch die Erweiterung oder sogar den Austausch des gesamten Labyrinths, ohne dass Änderungen am existierenden Code vorgenommen werden müssen.

3.1 Abstract Factory (Abstrakte Fabrik)

Objektbasiertes Erzeugungsmuster

Zweck

Bereitstellung einer Schnittstelle zum Erzeugen verwandter oder voneinander abhängiger Objektfamilien ohne die Benennung ihrer konkreten Klassen.

Auch bekannt als

Kit

Motivation

In einem Benutzeroberflächen-Toolkit, das mehrere Look-and-Feel-Standards wie Motif oder den Presentation Manager unterstützt, wird das individuelle Aussehen und Verhalten der Widgets, sprich Scrollleisten, Fenster und Schaltflächen etc., von dem jeweils zugehörigen Standard definiert. Das bedeutet im Umkehrschluss, dass die Portierbarkeit zwischen den verschiedenen Standards nur dann gewährleistet sein kann, wenn keine Look-and-Feel-spezifischen Widgets in der Anwendung hartkodiert sind – weil die Instanziierung spezifischer Widget-Klassen innerhalb der Anwendung die nachträgliche Änderung des Look-and-Feels erschweren würde.

Dieser Problematik lässt sich durch die Definition einer abstrakten `WidgetFactory`-Klasse begegnen, die eine Schnittstelle zur Erzeugung aller grundlegenden Arten von Widgets deklariert. Des Weiteren existiert für jeden Widget-Typ jeweils eine eigene abstrakte Klasse und die Look-and-Feel-spezifischen Widgets werden über konkrete Unterklassen implementiert. Die `WidgetFactory`-Schnittstelle stellt für jede abstrakte Widget-Klasse eine Operation bereit, die ein neues Widget-Objekt zurückgibt. Diese Operationen werden dann zur Generierung der Widget-Instanzen von den Clients aufgerufen, ohne dass ihnen deren konkrete Klassen bekannt sind. Auf diese Weise bleibt die Unabhängigkeit der Clients vom aktuellen Look-and-Feel gewahrt.

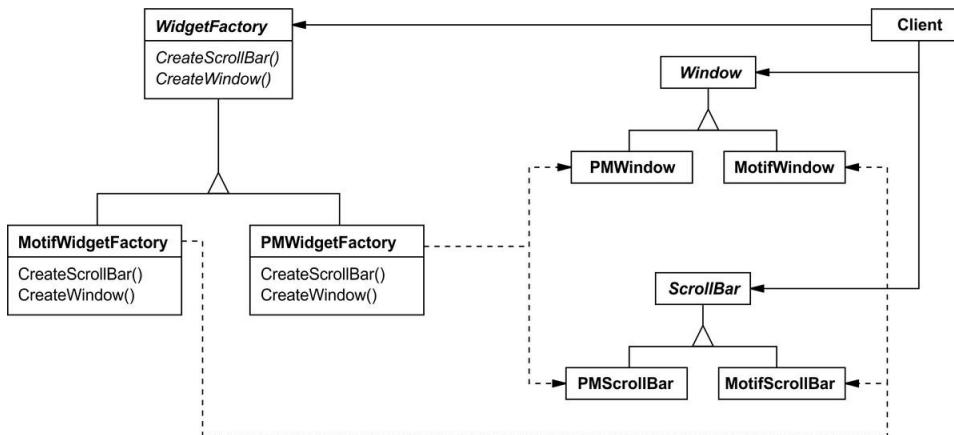


Abb. 3.2: Schnittstelle zur Erzeugung der grundlegenden Widget-Arten

Jeder Look-and-Feel-Standard verfügt über eine eigene konkrete UnterkLASSE von `WidgetFactory`, die wiederum die Operation zur Erzeugung des Look-and-Feel-spezifischen Widgets implementiert. Beispielsweise instanziert die `CreateScrollBar`-Operation der UnterkLASSE `MotifWidgetFactory` eine Motif-Scrollleiste und gibt diese zurück, während die entsprechende Operation der UnterkLASSE `PMWidgetFactory` eine Scrollleiste für den Presentation Manager zurückliefert. Die clientseitige Erzeugung der Widgets erfolgt ausschließlich über die `WidgetFactory`-Schnittstelle – und zwar ohne jegliche Kenntnis von den Klassen,

die Widgets für ein bestimmtes Look-and-Feel implementieren. Mit anderen Worten: Die Clients orientieren sich ausschließlich an der von einer abstrakten Klasse definierten Schnittstelle, nicht jedoch an einer bestimmten konkreten Klasse.

Darüber hinaus erzwingt `WidgetFactory` Abhängigkeiten zwischen den konkreten Widget-Klassen: Eine Motif-Scrollleiste sollte immer nur mit einer Motif-Schaltfläche und einem Motif-Texteditor verwendet werden – und genau diese Art von Vorgabe wird durch den Einsatz einer `MotifWidgetFactory` auch umgesetzt.

Anwendbarkeit

Die Verwendung des Design Patterns *Abstract Factory* (*Abstrakte Fabrik*) empfiehlt sich, wenn

- ein System unabhängig von der Art der Erzeugung, Komposition und Darstellung seiner Produkte arbeiten soll.
- ein System mit einer von mehreren Produktfamilien konfiguriert werden soll.
- eine Familie verwandter Produktobjekte für die gemeinsame Verwendung vorgesehen ist und diese Vorgabe auch zwingend eingehalten werden soll.
- eine Klassenbibliothek von Produkten bereitgestellt werden soll, aber nur deren Schnittstellen, nicht jedoch ihre Implementierungen preisgegeben werden sollen.

Hinweis

Mit »Produkten« sind in diesem Zusammenhang die vom System erzeugten Objekte gemeint.

Struktur

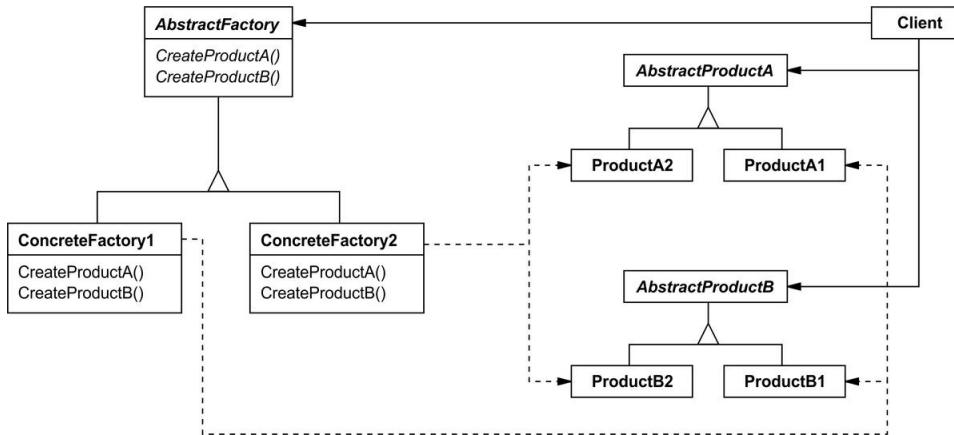


Abb. 3.3: Die Struktur des Design Patterns Abstract Factory (Abstrakte Fabrik)

Teilnehmer

- **AbstractFactory** (`WidgetFactory`)
 - Deklariert eine Schnittstelle für Operationen, die abstrakte Produktobjekte erzeugen.
- **ConcreteFactory** (`MotifWidgetFactory`, `PMWidgetFactory`)
 - Implementiert die Operationen zur Erzeugung konkreter Produktobjekte.
- **AbstractProduct** (`Window`, `ScrollBar`)
 - Deklariert eine Schnittstelle für einen bestimmten Produktobjekttyp.
- **ConcreteProduct** (`MotifWindow`, `MotifScrollBar`)
 - Definiert ein von der zugehörigen konkreten Factory zu erzeugendes Produktobjekt.
 - Implementiert die `AbstractProduct`-Schnittstelle.
- **Client**
 - Verwendet ausschließlich von `AbstractFactory`- und `AbstractProduct`-Klassen deklarierte Schnittstellen.

Interaktionen

- Normalerweise wird eine einzelne Instanz einer `ConcreteFactory`-Klasse zur Laufzeit erzeugt. Diese konkrete Factory generiert wiederum Produktobjekte mit spezifischen Implementierungen. Zur Erzeugung verschiedener Produktobjekte sollten die Clients eine andere konkrete Factory nutzen.
- Eine `AbstractFactory` delegiert die Erzeugung von Produktobjekten an ihre `ConcreteFactory`-Unterklassen.

Konsequenzen

Das Design Pattern *Abstract Factory (Abstrakte Fabrik)* begünstigt bzw. bewirkt Folgendes:

1. *Isolierung konkreter Klassen.* Es gestattet die Steuerung der von einer Anwendung erzeugten Objektklassen. Eine Factory ist nicht nur für den Erzeugungsprozess der Produktobjekte zuständig, sondern kapselt ihn auch – und unterbindet somit den Zugang der Clients zu den Implementierungsklassen. Die clientseitige Manipulation der Instanzen erfolgt über deren abstrakte Schnittstellen. Auf diese Weise sind die Namen der Produktklassen in der Implementierung der konkreten Factory isoliert und treten im Clientcode nicht in Erscheinung.
2. *Einfacher Austausch von Produktfamilien.* Die Klasse einer konkreten Factory tritt nur ein einziges Mal in der Anwendung in Erscheinung – und zwar bei der Instanziierung. Dadurch lässt sich diese Factory problemlos austauschen, um der Anwendung die Nutzung unterschiedlicher Produktkonfigurationen zu ermöglichen. Und weil das Design Pattern *Abstract Factory (Abstrakte Fabrik)* ohnehin eine komplette Produktfamilie erzeugt, findet in diesem Fall auch gleich ein vollständiger Austausch statt. Auf die Benutzeroberfläche dieses Beispiels bezogen, ließe sich ein Wechsel von den Motif-Widgets zu den Presentation-Manager-Widgets somit einfach durch das Ersetzen der entsprechenden Factory-Objekte und die anschließende Neuinitialisierung der Schnittstelle vollziehen.
3. *Produktkonsistenz.* Wenn die Produktobjekte einer Familie für eine gemeinsame Nutzung vorgesehen sind, ist es wichtig, dass die Anwendung auch jeweils immer nur die Objekte einer Familie verwendet – und genau dafür sorgt die `AbstractFactory`-Schnittstelle.

4. *Unterstützung neuer Produktarten.* Da die Menge der erzeugbaren Produkte von der `AbstractFactory`-Schnittstelle bestimmt wird, ist eine Erweiterung der abstrakten Factories zur Generierung neuer Produktarten in der Regel nicht so einfach durchzuführen, denn: Die Unterstützung neuer Produktarten bedingt eine Erweiterung der `Factory`-Schnittstelle, was die Modifizierung der `AbstractFactory`-Klasse sowie all ihrer Unterklassen mit einschließt. Wie sich dieses Problem dennoch lösen lässt, erfahren Sie im nachfolgenden Abschnitt »**Implementierung**«.

Implementierung

Für die Implementierung des Design Patterns *Abstract Factory* (*Abstrakte Fabrik*) stehen gleich mehrere nützliche Techniken zur Verfügung:

1. *Factories als Singletons.* Eine Anwendung benötigt normalerweise nur eine Instanz einer `ConcreteFactory` pro Produktfamilie. Dementsprechend ist es in der Regel am sinnvollsten, sie als *Singleton* (siehe [Abschnitt 3.5](#)) zu implementieren.
2. *Produkterzeugung.* `AbstractFactory` deklariert lediglich eine *Schnittstelle* zur Erzeugung von Produkten. Für den eigentlichen Erzeugungsvorgang sind die `ConcreteProduct`-Unterklassen zuständig. Üblicherweise wird zu diesem Zweck eine *Factory Method (Fabrikmethode)* (siehe [Abschnitt 3.3](#)) für jedes Produkt definiert. Eine konkrete Factory spezifiziert ihre Produkte durch das Überschreiben der Fabrikmethode für jedes Produkt. Diese Implementierung ist zwar simpel, erfordert aber dennoch eine neue konkrete Factory-Unterklasse für jede Produktfamilie, selbst wenn sich die Familien nur geringfügig unterscheiden.

Sollten viele Produktfamilien möglich sein, kann die konkrete Factory mithilfe des Design Patterns *Prototype (Prototyp)* (siehe [Abschnitt 3.4](#)) implementiert werden. Die konkrete Factory wird mit einer prototypischen Instanz jedes Produkts aus der Familie initialisiert und erzeugt neue Produkte durch Klonen ihres Prototyps. Bei diesem prototypbasierten Ansatz entfällt die Notwendigkeit einer neuen konkreten Factory-Klasse für jede neue Produktfamilie.

Das nachfolgende Beispiel zeigt eine Implementierungsmöglichkeit für eine prototypbasierte Factory in Smalltalk auf. Die konkrete Factory speichert die zu klonenden Prototypen in einem Dictionary namens `partCatalog`. Die

Methode `make:` fragt den Prototyp ab und klonen ihn:

```
make: partName
  ^ (partCatalog at: partName) copy
```

Die konkrete Factory besitzt eine Methode, um Teile zum Katalog zu ergänzen:

```
addPart: partTemplate named: partName
  partCatalog at: partName put: partTemplate
```

Prototypen werden beim Hinzufügen zur Factory durch ein Symbol identifiziert:

```
aFactory addPart: aPrototype named: #ACMEWidget
```

Eine Variation des prototypbasierten Ansatzes ist in solchen Programmiersprachen möglich, die Klassen als Objekte erster Ordnung behandeln (z. B. Smalltalk und Objective-C). In diesen Sprachen wird eine Klasse als eine degenerierte Factory aufgefasst, die lediglich eine einzige Produktart erzeugt. *Klassen*, die – ähnlich wie Prototypen – die verschiedenen konkreten Produkte in Variablen erzeugen, können in einer konkreten Factory gespeichert werden und legen auf Veranlassung der konkreten Factory neue Instanzen an. Eine neue Factory wird durch die Initialisierung einer Instanz einer konkreten Factory mit *Produktklassen* statt durch Unterklassenbildung definiert. Bei diesem Ansatz werden die spezifischen Eigenschaften der Programmiersprache ausgenutzt, während die rein prototypbasierte Herangehensweise sprachunabhängig ist.

Wie die soeben beschriebene prototypbasierte Factory in Smalltalk besitzt auch die klassenbasierte Version eine einzige Instanzvariable namens `partCatalog` – ein Dictionary, dessen Schlüsselement der Name der Klasse ist. Anstelle der zu klonenden Prototypen speichert `partCatalog` die Klassen der Produkte. Die Methode `make:` sieht damit folgendermaßen aus:

```
make: partName
  ^ (partCatalog at: partName) new
```

3. *Definition erweiterbarer Factories.* `AbstractFactory` definiert im Allgemeinen verschiedene Operationen für jede Produktart, die sie erzeugen kann. Die Produktarten werden in den Operationssignaturen kodiert. Zur Ergänzung einer neuen Produktart müssen die `AbstractFactory`-Schnittstelle und alle davon abhängigen Klassen geändert werden.

In einem flexibleren, aber auch weniger sicheren Design würde den Operationen, die Objekte erzeugen, hingegen ein Parameter mitgegeben, der die Art des zu erzeugenden Objekts identifiziert. Dabei könnte es sich um einen Klassenidentifizierer, einen String, einen Integer- oder irgendeinen anderen Wert handeln, der die Art des Produkts kennzeichnet. Bei diesem Ansatz benötigt `AbstractFactory` lediglich eine einzige `make`-Operation mit einem Parameter, der den Typ des zu erstellenden Objekts angibt. Diese Technik wird auch in den prototyp- und klassenbasierten abstrakten Factories verwendet, die unter Punkt 2 besprochen wurden.

Die hier beschriebene Variante ist in einer dynamisch typisierten Programmiersprache wie Smalltalk leichter zu verwenden als in einer statisch typisierten Sprache wie C++. In C++ lässt sie sich nur dann einsetzen, wenn alle Objekte dieselbe abstrakte Basisklasse besitzen oder der Client, der sie angefordert hat, die Produktobjekte sicher in den richtigen Typ konvertieren kann. Wie sich solche parametrisierten Operationen in C++ implementieren lassen, ist im Abschnitt »Implementierung« des Design Patterns *Factory Method (Fabrikmethode*, siehe [Abschnitt 3.3](#)) beschrieben.

Aber selbst wenn keine Typkonvertierung erforderlich ist, bleibt ein inhärentes Problem bestehen: Alle Produkte werden mit *derselben* abstrakten, durch den Rückgabetyp vorgegebenen Schnittstelle an den Client zurückgegeben. Der Client ist nicht in der Lage, die Klassen der Produkte voneinander zu unterscheiden oder sichere Annahmen darüber zu machen. Wenn ein Client unterklassenspezifische Operationen durchführen muss, sind diese nicht über die abstrakte Schnittstelle zugänglich. Und obwohl er einen Downcast ausführen könnte (in C++ z. B. mit `dynamic_cast`), ist das nicht immer sinnvoll oder sicher, weil der Downcast fehlschlagen kann – der klassische Nachteil einer äußerst flexiblen und erweiterbaren Schnittstelle.

Beispielcode

Im folgenden Beispiel wird das Design Pattern *Abstract Factory (Abstrakte Fabrik)* zur Erzeugung eines der zu Beginn dieses Kapitels bereits angesprochenen Labyrinth-Spiele angewendet.

Mit der Klasse `MazeFactory` lassen sich Komponenten der Labyrinthe erzeugen, also z. B. Räume, Wände und Türen. Sie kann in einem Programm verwendet werden, das die Entwürfe der Labyrinthe aus einer Datei ausliest und diese dann entsprechend erstellt. Oder auch in einem Programm, das zufallsbasierte Labyrinthe

erstellt. Programmcode, der ein Labyrinth erzeugt, nutzt MazeFactory als Argument und gestattet dem Programmierer damit, die Klassen der zu errichtenden Räume, Wände und Türen zu spezifizieren.

```
class MazeFactory {
public:
    MazeFactory();

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Wall* Makewall() const
        { return new Wall; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Wie eingangs dieses Kapitels bereits erwähnt, erzeugt die Memberfunktion CreateMaze (siehe [hier](#)) ein kleines Labyrinth mit zwei durch eine Tür verbundenen Räumen. Da sie die Klassennamen jedoch hartkodiert, ist es schwierig, Labyrinthe mit anderen Komponenten zu erzeugen.

Die nachfolgende Version von CreateMaze behebt dieses Problem mithilfe des Parameters MazeFactory:

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, factory.Makewall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.Makewall());
    r1->SetSide(West, factory.Makewall());

    r2->SetSide(North, factory.Makewall());
    r2->SetSide(East, factory.Makewall());
    r2->SetSide(South, factory.Makewall());
    r2->SetSide(West, aDoor);
    return aMaze;
}
```

Durch die Bildung einer Unterklasse MazeFactory kann nun eine

EnchantedMazeFactory für verzauberte Labyrinthe erstellt werden. Diese Factory überschreibt die diversen Memberfunktionen und gibt verschiedene Unterklassen von Room, Wall etc. zurück:

```
class EnchantedMazeFactory : public MazeFactory {  
public:  
    EnchantedMazeFactory();  
  
    virtual Room* MakeRoom(int n) const  
    { return new EnchantedRoom(n, CastSpell()); }  
    virtual Door* MakeDoor(Room* r1, Room* r2) const  
    { return new DoorNeedingSpell(r1, r2); }  
  
protected:  
    Spell* CastSpell() const;  
};
```

Angenommen, als Nächstes soll ein Labyrinth erzeugt werden, in dessen Räumen sich Bomben platzieren lassen, die im Fall einer Detonation zumindest Schäden an den Wänden verursachen. Dazu könnte eine Unterklasse von Room angelegt werden, die feststellt, ob eine Bombe in dem Raum vorhanden ist und zudem auch angibt, ob sie explodiert ist. Außerdem wird eine Unterklasse von Wall benötigt, um das Ausmaß des Schadens an der Wand aufzuzeichnen. Diese Klassen werden mit RoomWithABomb und BombedWall bezeichnet.

Die letzte zu definierende Klasse heißt BombedMazeFactory. Hierbei handelt es sich um eine Unterklasse von MazeFactory, die sicherstellt, dass die Wände Instanzen der Klasse BombedWall und die Räume Instanzen der Klasse RoomWithABomb sind. BombedMazeFactory muss lediglich zwei Funktionen überschreiben:

```
Wall* BombedMazeFactory::MakeWall () const {  
    return new BombedWall;  
}  
  
Room* BombedMazeFactory::MakeRoom(int n) const {  
    return new RoomWithABomb(n);  
}
```

Zur Erzeugung eines schlichten Labyrinths, in dem Bomben platziert werden können, wird CreateMaze einfach mit dem Parameter BombedMazeFactory aufgerufen:

```
MazeGame game;  
BombedMazeFactory factory;  
  
game.CreateMaze (factory)
```

Alternativ könnte `CreateMaze` zur Generierung verzauberter Labyrinthe auch eine Instanz von `EnchantedMazeFactory` nutzen.

Beachtenswert ist in diesem Zusammenhang, dass `MazeFactory` lediglich eine Sammlung von `Factory`-Methoden verkörpert und als solche die häufigste Implementierungsform des Design Patterns *Abstract Factory (Abstrakte Fabrik)* darstellt. Ebenfalls interessant ist auch, dass `MazeFactory` keine abstrakte Klasse ist und somit sowohl als `AbstractFactory` als auch als `ConcreteFactory` fungiert – auch dies ist eine gängige Implementierungsart für einfache Anwendungen des Patterns *Abstract Factory (Abstrakte Fabrik)*. Und da es sich bei `MazeFactory` um eine konkrete Klasse handelt, die vollständig aus `Factory`-Methoden besteht, lässt sich zudem durch das Anlegen einer Unterklasse und Überschreiben der zu modifizierenden Operationen problemlos eine neue `MazeFactory` erstellen.

`CreateMaze` verwendet in diesem Beispiel die `SetSide`-Operation, um die Seiten der Räume zu spezifizieren. Werden die Räume mit `BombedMazeFactory` erzeugt, besteht das Labyrinth aus `RoomWithABomb`-Objekten mit `BombedWall`-Seiten. Müsste die Klasse `RoomWithABomb` dagegen auf ein unterklassenspezifisches Member von `BombedWall` zugreifen, dann müsste sie die Referenzen auf ihre Wände von `Wall*` in `BombedWall*` umwandeln. Ein derartiges Downcasting ist allerdings nur so lange sicher, wie das Argument tatsächlich eine `BombedWall` ist – was wiederum nur genau dann garantiert ist, wenn alle Wände ausschließlich mit `BombedMazeFactory` erzeugt werden.

Dynamisch typisierte Programmiersprachen wie Smalltalk erfordern natürlich kein Downcasting, allerdings können sie Laufzeitfehler produzieren, wenn sie dort, wo sie eine Unterklasse von `Wall` erwarten, auf `Wall` selbst treffen. Der Einsatz des Patterns *Abstract Factory (Abstrakte Fabrik)* zur Generierung von Wänden verhindert solche Laufzeitfehler, weil dabei sichergestellt ist, dass nur bestimmte Arten von Wänden erzeugt werden können.

Das nächste Beispiel beschreibt eine Smalltalk-Version von `MazeFactory` mit einer einzelnen `make`-Operation, die die Art des zu erzeugenden Objekts als Parameter erhält. Darüber hinaus speichert die konkrete Factory auch die Klassen der Produkte, die sie erzeugt.

Dazu wird als Erstes eine Smalltalk-Entsprechung von `CreateMaze` geschrieben:

```
CreateMaze: aFactory
| room1 room2 aDoor |
room1 := (aFactory make: #room) number: 1.
room2 := (aFactory make: #room) number: 2.
```

```

aDoor := (aFactory make: #door) from: room1 to: room2.
room1 atSide: #north put: (aFactory make: #wall).
room1 atSide: #east put: aDoor.
room1 atSide: #south put: (aFactory make: #wall).
room1 atSide: #west put: (aFactory make: #wall).
room2 atSide: #north put: (aFactory make: #wall).
room2 atSide: #east put: (aFactory make: #wall).
room2 atSide: #south put: (aFactory make: #wall).
room2 atSide: #west put: aDoor.
^ Maze new addRoom: room1; addRoom: room2; yourself

```

Wie schon im Abschnitt »Implementierung« beschrieben, benötigt `MazeFactory` nur eine einzelne Instanzvariable, `partCatalog`, um ein Dictionary bereitzustellen, dessen Schlüssel die Klasse der betreffenden Komponente ist. Erinnern Sie sich noch an die Implementierung der `make:-Operation`?

```

make: partName
^ (partCatalog at: partName) new

```

Nun kann eine `MazeFactory` erzeugt und für die Implementierung von `createMaze` genutzt werden. Dazu wird die Factory mithilfe einer Methode `createMazeFactory` der Klasse `MazeGame` erstellt:

```

createMazeFactory
^ (MazeFactory new
  addPart: Wall named: #wall;
  addPart: Room named: #room;
  addPart: Door named: #door;
  yourself)

```

Anschließend wird durch die Verknüpfung verschiedener Klassen mit den jeweiligen Schlüsseln eine `BombedMazeFactory` oder `EnchantedMazeFactory` erzeugt, Letztere beispielsweise so:

```

createMazeFactory
^ (MazeFactory new
  addPart: Wall named: #wall;
  addPart: EnchantedRoom named: #room;
  addPart: DoorNeedingSpell named: #door;
  yourself)

```

Praxisbeispiele

Das von Mark Linton entwickelte C++-Toolkit **InterViews** für X Window kennzeichnet AbstractFactory-Klassen mit dem Suffix `Kit` [Lin92]. Es definiert die

abstrakten Klassen `WidgetKit` und `DialogKit` zur Generierung Look-and-Feel-spezifischer Objekte auf der Benutzeroberfläche. Darüber hinaus enthält `InterViews` auch ein `LayoutKit`, das je nach gewünschtem Layout unterschiedliche Kompositionsobjekte erzeugt – wenn beispielsweise für ein grundsätzlich horizontal konzipiertes Layout je nach Ausrichtung des Dokuments im Hoch- oder Querformat verschiedene Objekte benötigt werden.

Die C++-Klassenbibliothek `ET++` [WGM88] nutzt das Design Pattern *Abstract Factory (Abstrakte Fabrik)* zur Sicherstellung der Portabilität zwischen verschiedenen Fenstersystemen (z. B. X Window und SunView). Die abstrakte Basisklasse `WindowSystem` definiert die Schnittstelle zur Erzeugung von Objekten, die die Ressourcen (z. B. `MakeWindow`, `SetFont`, `SetColor`) der Fenstersysteme repräsentieren. Konkrete Unterklassen implementieren die Schnittstellen für ein spezifisches Fenstersystem. `ET++` legt zur Laufzeit eine Instanz einer konkreten `WindowSystem`-Unterkelas an, die konkrete Objekte für Systemressourcen erzeugt.

Verwandte Patterns

Die `AbstractFactory`-Klassen werden häufig mit dem Design Pattern *Factory Method (Fabrikmethode*, siehe [Abschnitt 3.3](#)), mitunter aber auch mit dem Pattern *Prototype (Prototyp*, siehe [Abschnitt 3.4](#)) implementiert.

Eine konkrete Factory basiert dagegen oftmals auf dem Design Pattern *Singleton (Singleton*, siehe [Abschnitt 3.5](#)).

3.2 Builder (Erbauer)

Objektbasiertes Erzeugungsmuster

Zweck

Getrennte Handhabung der Erzeugungs- und Darstellungsmechanismen komplexer Objekte zwecks Generierung verschiedener Repräsentationen in einem einzigen Erzeugungsprozess.

Motivation

Grundsätzlich sollte ein RTF (*Rich Text Format*) Reader RTF-Dateien in viele andere Textformate konvertieren können, z. B. in reinen ASCII-Text oder in ein interaktiv editierbares Text-Widget. Das Problem dabei ist jedoch, dass es eine schier endlose Zahl möglicher Konvertierungsformate gibt. Deshalb sollte eine problemlose Ergänzung neuer Konvertierungen gewährleistet sein, ohne dass irgendwelche Modifikationen am Reader selbst vorgenommen werden müssen.

Eine Möglichkeit wäre, die `RTFReader`-Klasse mit einem `TextConverter`-Objekt zu konfigurieren, das die RTF-Datei in eine andere Textdarstellung konvertiert. Dazu macht sich `RTFReader` beim Parsen des RTF-Dokuments das `TextConverter`-Objekt zunutze und sendet ihm bei jeder Erkennung eines RTF-Tokens (sei es reiner Text oder ein RTF-Steuerwort) einen Konvertierungsrequest. Ein `TextConverter`-Objekt ist sowohl für die Durchführung der Datenkonvertierung als auch für die Darstellung des Tokens in einem bestimmten Format zuständig.

Die Unterklassen von `TextConverter` sind auf unterschiedliche Konvertierungen und Formate spezialisiert. `ASCIIConverter` ignoriert beispielsweise jegliche Konvertierungsrequests, sofern es sich dabei nicht um reinen Text handelt. `TeXConverter` implementiert dagegen Operationen für alle Requests zur Erzeugung einer TeX-Darstellung, die alle stilistischen Daten im Text erfasst. Und `TextWidgetConverter` erstellt wiederum ein komplexes Steuerungselement auf der Benutzeroberfläche, das es dem User ermöglicht, den Text zu betrachten und zu bearbeiten.

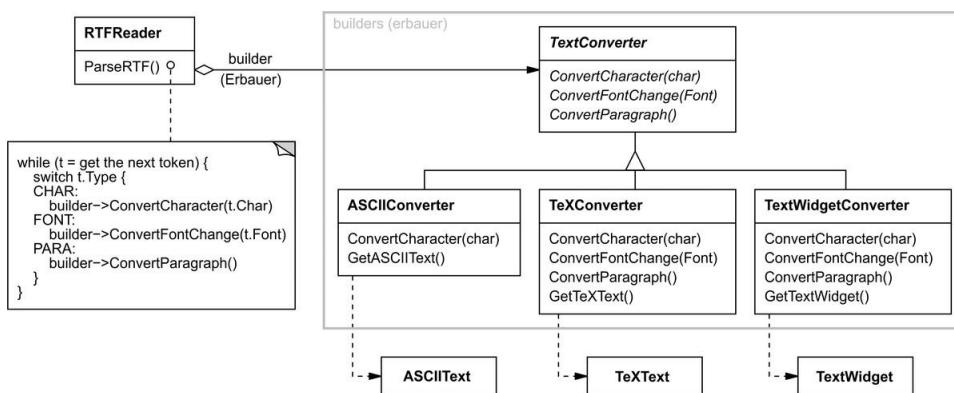


Abb. 3.4: Konfiguration der Klasse `RTFReader` mit einem `TextConverter`-Objekt

Alle `Converter`-Klassen verbergen den Mechanismus zur Erzeugung und Komposition eines komplexen Objekts hinter einer abstrakten Schnittstelle. Der Konvertierer arbeitet völlig unabhängig von dem Reader, der für das Parsen eines

RTF-Dokuments zuständig ist.

All diese Beziehungen und Zusammenhänge werden im Design Pattern *Builder* (*Erbauer*) berücksichtigt. Die Converter-Klassen werden im Rahmen dieses Patterns als **Builder (Erbauer)**, der Reader als **Director (Direktor)** bezeichnet. In dem hier angeführten Beispiel sorgt das Pattern *Builder* (*Erbauer*) somit für die getrennte Handhabung des Algorithmus zur Interpretation eines Textformats (d. h. den Parser für RTF-Dokumente) auf der einen Seite und des Prozesses zur Erzeugung und Darstellung eines konvertierten Formats auf der anderen Seite. Dadurch ist die Wiederverwendung des Parsing-Algorithmus der Klasse RTFReader zur Generierung verschiedener Textdarstellungen von RTF-Dokumenten möglich – einfach durch die Konfiguration von RTFReader mit verschiedenen TextConverter-Unterklassen.

Anwendbarkeit

Der Einsatz des Design Patterns *Builder* (*Erbauer*) empfiehlt sich, wenn

- die Unabhängigkeit des Algorithmus zur Erzeugung komplexer Objekte von deren Bestandteilen und Komposition gewährleistet sein soll,
- der Erzeugungsprozess verschiedene Darstellungsformen des zu generierenden Objekts zulassen soll.

Struktur

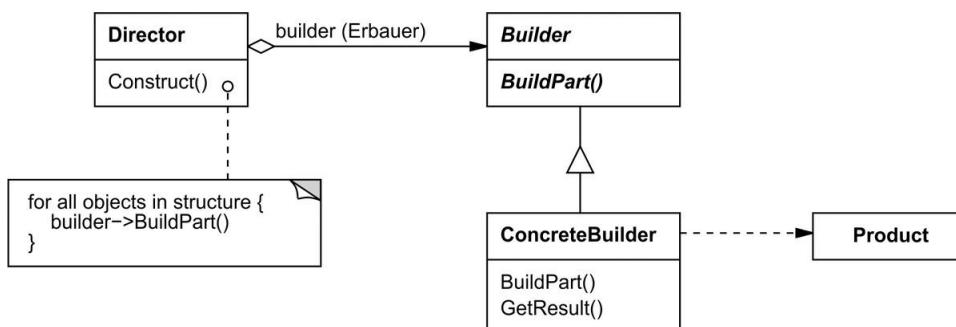


Abb. 3.5: Die Struktur des Design Patterns *Builder* (*Erbauer*)

Teilnehmer

- **Builder** (`TextConverter`)
 - Spezifiziert eine abstrakte Schnittstelle zur Erzeugung der Bestandteile eines Produktobjekts.
- **ConcreteBuilder** (`ASCIIConverter`, `TeXConverter`, `TextWidgetConverter`)
 - Implementiert zur Erzeugung und Komposition des Produkts die Builder-Schnittstelle.
 - Definiert und überwacht die generierte Darstellung.
 - Stellt eine Schnittstelle zum Abrufen des Produkts bereit (z. B. `GetASCIIText`, `GetTextWidget`).
- **Director** (`RTFReader`)
 - Nutzt die Builder-Schnittstelle zur Erzeugung eines Objekts.
- **Product** (`ASCIIText`, `TeXText`, `TextWidget`)
 - Repräsentiert das gegenwärtig erstellte komplexe Objekt. `ConcreteBuilder` definiert die interne Darstellung des Produkts sowie den dazugehörigen Kompositionsprozess.
 - Enthält die Klassen der einzelnen Bestandteile, einschließlich der Schnittstellen für die endgültige Komposition der Teile.

Interaktionen

- Der Client generiert ein `Director`-Objekt und konfiguriert es mit dem gewünschten `Builder`-Objekt.
- Der Director teilt dem Builder mit, wann ein Bestandteil des Produkts erzeugt werden soll.
- Der Builder bearbeitet die vom Director eingehenden Requests und fügt dem Produkt die angefragten Teile hinzu.
- Der Client ruft das erzeugte Produkt vom Builder ab.

Das in [Abbildung 3.6](#) dargestellte Interaktionsdiagramm veranschaulicht die Zusammenarbeit zwischen Builder und Director:

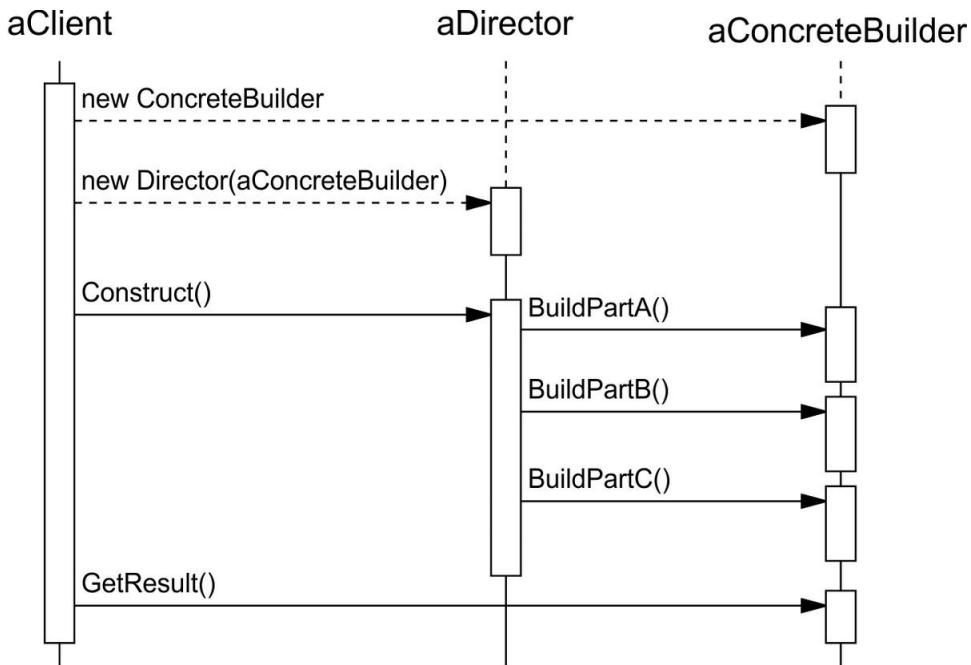


Abb. 3.6: Interaktionen zwischen Builder und Director

Konsequenzen

Der Einsatz des Design Patterns *Builder (Erbauer)* hat folgende Auswirkungen:

1. *Variable interne Darstellung eines Produkts.* Der Builder stellt dem Director eine abstrakte Schnittstelle zur Erzeugung des Produkts zur Verfügung. Diese Schnittstelle gestattet es dem Builder, die Darstellung und interne Struktur ebenso wie den Erzeugungsprozess des Produkts zu verbergen. Weil das Produkt mithilfe einer abstrakten Schnittstelle erzeugt wird, muss für eine Variation der internen Darstellung des Produkts lediglich ein neuer Builder-Typ definiert werden.
2. *Isolierung des erzeugungs- und darstellungsrelevanten Codes.* Das Pattern *Builder (Erbauer)* kapselt sowohl den Erzeugungsprozess als auch die Darstellung des Produkts und verbessert damit dessen Modularität. Da die Clients keine Kenntnis von den Klassen haben müssen, die die interne Struktur des Produkts definieren, treten diese auch nicht in der Builder-Schnittstelle in Erscheinung.

Jeder `ConcreteBuilder` enthält sämtlichen für die Erzeugung und Komposition eines bestimmten Produkttyps relevanten Code. Dieser Code wird nur ein einziges Mal geschrieben und kann dann von verschiedenen `Director`-Objekten wiederverwendet werden, um aus ein und demselben Teilebestand diverse Produktvariationen zu erstellen. Würde also in dem eingangs dieses Kapitels beschriebenen Szenario statt des RTF-Readers beispielsweise ein SGML-Reader definiert, könnte dieser weiterhin auf dieselben `TextConverter` zugreifen, um `ASCIIText`-, `TeXText`- und `TextWidget`-Darstellungen von SGML-Dokumenten zu erzeugen.

3. *Exaktere Überwachung des Erzeugungsprozesses.* Anders als diejenigen Erzeugungsmuster, die Produkte in einem Durchgang generieren, wird die Produkterzeugung mit dem Design Pattern *Builder (Erbauer)* Schritt für Schritt vom Director kontrolliert. Erst wenn das Produkt fertiggestellt ist, ruft er es vom Builder ab. Die `Builder`-Schnittstelle spiegelt den Erzeugungsprozess des Produkts somit in umfassenderer Weise wider als die anderen Erzeugungsmuster und gestattet dadurch eine genauere Überwachung sowohl des Prozesses selbst als auch der internen Struktur des resultierenden Produkts.

Implementierung

In der Regel existiert eine abstrakte `Builder`-Klasse, die jeweils eine Operation für jede Komponente definiert, deren Erzeugung ein Director anfordern könnte. Diese Operationen werden standardmäßig leer implementiert und dann von einer `ConcreteBuilder`-Klasse, die die Erzeugung einer Komponente anfordert, überschrieben.

Darüber hinaus sollten aber auch noch andere Implementierungsaspekte berücksichtigt werden:

1. *Konstruktionsschnittstelle.* Da die `Builder` ihre Produkte schrittweise konstruieren, muss die Schnittstelle der `Builder`-Klasse universell genug gehalten sein, dass die Produkterzeugung für alle Arten von konkreten Buildern gewährleistet ist.

Ein wesentlicher Designaspekt ist hierbei das Modell für den Erzeugungs- und Kompositionsprozess des Produkts. Normalerweise reicht ein Modell aus, bei dem die aus den Erzeugungsrequests resultierenden Ergebnisse einfach an das Produkt angehängt werden. Im Fall des RTF-Beispiels konvertiert der `Builder` das nächste Token und hängt es an den bis zu diesem Zeitpunkt konvertierten

Text an.

Manchmal muss allerdings auch auf die Teile des zuvor generierten Produkts zurückgegriffen werden. So ermöglicht das im Abschnitt »Beispielcode« angeführte Labyrinth-Spiel das Hinzufügen einer Tür zwischen zwei bereits vorhandenen Räumen über die `MazeBuilder`-Schnittstelle. Ein weiteres Beispiel sind auch Baumstrukturen wie Parse-Bäume, die nach dem Bottom-up-Prinzip (von unten nach oben) aufgebaut sind. Hier würde der Builder Kindobjektknoten an den `Director` zurückgeben, der sie daraufhin zur Erstellung der Elternknoten wieder an den Builder zurückleitet.

2. *Keine abstrakte Klasse für Produkte.* Im Normalfall unterscheiden sich die von den konkreten Buildern erzeugten Produkte in ihrer Darstellung so weit voneinander, dass die Zuweisung einer gemeinsamen Basisklasse an verschiedene Produkte kaum einen Nutzen bringen würde. Auf das RTF-Beispiel bezogen, ist es eher unwahrscheinlich, dass die `ASCIIText`- und `TextWidget`-Objekte eine gemeinsame Schnittstelle haben – und die brauchen sie auch gar nicht. Da der Client den Director in der Regel mit dem passenden konkreten Builder konfiguriert, ist er auch in der Lage, die gerade verwendete konkrete Unterklassie zu erkennen und seine Produkte entsprechend zu handhaben.
3. *Leere Methoden als Standardimplementierung in der Builder-Klasse.* In C++ sind die `Build`-Operationen bewusst nicht als rein virtuelle Memberfunktionen deklariert, sondern als leere Operationen – damit die Clients nur diejenigen Operationen überschreiben müssen, die sie auch tatsächlich benötigen.

Beispielcode

Für dieses Labyrinth-Spiel wird eine Variante der `CreateMaze`-Memberfunktion (siehe [Kapitel 2](#)) definiert, die einen Builder der Klasse `MazeBuilder` als Argument erhält.

`MazeBuilder` definiert zur Generierung des Labyrinths folgende Schnittstelle:

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }
```

```
    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

Sie hat die Aufgabe, drei Dinge zu erzeugen: 1. das Labyrinth, 2. Räume mit bestimmten Raumnummern und 3. Türen zwischen den nummerierten Räumen. Die GetMaze-Operation übergibt das Labyrinth an den Client, und die Unterklassen von MazeBuilder überschreiben dann wiederum diese Operation und liefern das so erzeugte Labyrinth zurück.

Alle von MazeBuilder zur Errichtung des Labyrinths durchgeföhrten Operationen sind standardmäßig leer implementiert und nicht rein virtuell deklariert, damit die abgeleiteten Klassen nur die für sie relevanten Methoden überschreiben.

Mithilfe der MazeBuilder-Schnittstelle lässt sich die Memberfunktion CreateMaze dahingehend modifizieren, dass sie diesen Builder als Parameter annimmt:

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

Vergleicht man diese modifizierte mit der ursprünglichen Version von CreateMaze, wird deutlich, inwieweit der Builder die interne Darstellung des Labyrinths verbirgt – sprich die Klassen, die die Räume, Türen und Wände definieren – und wie diese Bestandteile zum endgültigen Labyrinth zusammengefügt werden. Es steht zwar zu vermuten, dass es Klassen zur Darstellung der Räume und Türen gibt, auf eine Klasse für die Wände findet sich allerdings kein Hinweis. Dadurch lassen sich Änderungen an der Labyrinth-Darstellung um einiges einfacher vornehmen, weil keinerlei Anpassungen an den Clients von MazeBuilder erforderlich sind.

Wie die anderen Erzeugungsmuster, kapselt auch das Design Pattern *Builder (Erbauer)* den Erzeugungsprozess der Objekte – in diesem Fall durch die von MazeBuilder definierte Schnittstelle. Somit ist eine Wiederverwendung von MazeBuilder zur Errichtung anderer Labyrinth-Varianten möglich. Die Operation CreateComplexMaze liefert dazu ein passendes Beispiel:

```
Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
```

```

// ...
builder.BuildRoom(1001);

return builder.GetMaze();
}

```

Hierbei ist zu beachten, dass `MazeBuilder` die Labyrinthe nicht selbst generiert, sondern in der Hauptsache nur eine Schnittstelle für deren Erzeugung definiert. Die Definition der leeren Implementierungen von Operationen dient in erster Linie der Bequemlichkeit, die eigentliche Arbeit wird jedoch von den `MazeBuilder`-Unterklassen verrichtet.

Bei der Unterkasse `StandardMazeBuilder` handelt es sich um eine Implementierung zur Erzeugung einfacher Labyrinthe. Sie überwacht das aktuell generierte Labyrinth in der Variablen `_currentMaze`:

```

class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);

    virtual Maze* GetMaze ();

private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};

```

Die Hilfsoperation `CommonWall` legt die Ausrichtung der Trennwand zwischen zwei Räumen fest.

Der `StandardMazeBuilder`-Konstruktor initialisiert einfach bloß `_currentMaze`:

```

StandardMazeBuilder::StandardMazeBuilder () {
    _currentMaze = 0;
}

```

`BuildMaze` instanziert ein Labyrinth (`Maze`), das wiederum von anderen Operationen zusammengesetzt und schließlich an den Client zurückgegeben wird (mittels `GetMaze`):

```

void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}

```

```
Maze* StandardMazeBuilder::GetMaze () {
    return _currentMaze;
}
```

Die Operation `BuildRoom` generiert einen Raum und erzeugt die ihn umgebenden Wände:

```
void StandardMazeBuilder::BuildRoom (int n) {
    if (!_currentMaze->RoomNo(n)) {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);

        room->SetSide(North, new Wall);
        room->SetSide(South, new Wall);
        room->SetSide(East, new Wall);
        room->SetSide(West, new Wall);
    }
}
```

Um eine Verbindungstür zwischen zwei Räumen zu platzieren, lokalisiert `StandardMazeBuilder` zunächst die betreffenden Räume im Labyrinth und ermittelt dann ihre gemeinsame Wand:

```
void StandardMazeBuilder::rBuildDoor (int n1, int n2) {
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);
    Door* d = new Door(r1, r2);

    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
}
```

Anschließend können die Clients `CreateMaze` zusammen mit `StandardMazeBuilder` zur Erzeugung der Labyrinthe verwenden:

```
Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();
```

Man hätte auch alle `StandardMazeBuilder`-Operationen in `Maze` aufnehmen und jedes Labyrinth einzeln erstellen lassen können – generell gilt jedoch: Je kleiner die Schnittstelle `Maze` ist, desto einfacher ist es auch, sie zu verstehen und zu variieren. Zudem ist `StandardMazeBuilder` ohnehin problemlos von `Maze` zu trennen. Vor allem ermöglicht die Separierung dieser beiden Klassen aber eine Vielfalt von

MazeBuilder-Klassen, die jeweils verschiedene Klassen für die Räume, Wände und Türen nutzen können.

Eine exotischere MazeBuilder-Variante ist dagegen CountingMazeBuilder. Dieser Builder hat allerdings überhaupt nichts mit der Labyrinth-Erzeugung an sich zu tun, sondern zählt lediglich die verschiedenen Komponententypen, die in einem »normalen« MazeBuilder erstellt worden wären:

```
class CountingMazeBuilder : public MazeBuilder {  
public:  
    CountingMazeBuilder();  
  
    virtual void BuildMaze();  
    virtual void BuildRoom(int);  
    virtual void BuildDoor(int, int);  
    virtual void AddWall(int, Direction);  
  
    void GetCounts(int&, int&) const;  
  
private:  
    int _doors;  
    int _rooms;  
};
```

Die Zähler werden vom Konstruktor initialisiert und die überschriebenen MazeBuilder-Operationen erhöhen sie dann entsprechend:

```
CountingMazeBuilder::CountingMazeBuilder () {  
    _rooms = _doors = 0;  
}  
  
void CountingMazeBuilder::BuildRoom (int) {  
    _rooms++;  
}  
  
void CountingMazeBuilder::BuildDoor (int, int) {  
    _doors++;  
}  
  
void CountingMazeBuilder::GetCounts (  
    int& rooms, int& doors  
) const {  
    rooms = _rooms;  
    doors = _doors;  
}  
  
int rooms, doors;  
MazeGame game;  
CountingMazeBuilder builder;
```

```

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);

count << "The maze has "
    << rooms << " rooms and "
    << doors << " doors" << endl;

```

Praxisbeispiele

Der RTF-Konvertierer stammt aus **ET++** [WGM88] und setzt in seinem Textbausteinmodul einen Builder ein, um im RTF-Format vorliegende Texte bearbeiten zu können.

In **Smalltalk-80** ist das Pattern *Builder (Erbauer)* recht gebräuchlich [Par90]:

- Die `Parser`-Klasse im Subsystem des Compilers ist ein Director, der ein `ProgramNodeBuilder`-Objekt als Argument entgegennimmt. Ein `Parser`-Objekt meldet seinem `ProgramNodeBuilder`-Objekt jede Erkennung eines syntaktischen Konstrukts. Sobald der `Parser` fertig ist, fragt er den errichteten Parse-Baum vom `Builder` ab und gibt ihn an den Client zurück.
- `ClassBuilder` wird von den Klassen zum Anlegen eigener Unterklassen verwendet. In diesem Fall ist `Class` sowohl der Director als auch das Produkt.
- Der `Builder` `ByteCodeStream` erzeugt eine kompilierte Methode als `Bytearray` – und repräsentiert damit keine standardmäßige Verwendung des Design Patterns *Builder (Erbauer)*, weil das erzeugte Objekt hier nicht als normales Smalltalk-Objekt kodiert ist, sondern eben als `Bytearray`. Bei der Schnittstelle von `ByteCodeStream` handelt es sich jedoch um einen herkömmlichen `Builder`, d. h., `ByteCodeStream` könnte auch problemlos durch eine andere Klasse ersetzt werden, die Programme als zusammengesetzte Objekte darstellt.

Das **Service Configurator Framework von Adaptive Communications Environment (ACE)** nutzt einen Builder zur Erzeugung von Komponenten für Netzwerkdienste, die zur Laufzeit in einen Server eingebunden werden [SS94]. Diese Komponenten werden mithilfe einer Konfigurationssprache beschrieben, die von einem LALR(1)-Parser geparsst wird. Die semantischen Aktionen des Parsers führen Operationen auf dem `Builder` aus, die Daten zu der Dienstkomponente ergänzen. In diesem Fall fungiert der Parser als Director.

Verwandte Patterns

Das Design Pattern *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) ähnelt dem Pattern *Builder* (*Erbauer*) insofern, als dass es ebenfalls die Erzeugung komplexer Objekte ermöglicht. Im Wesentlichen unterscheiden sie sich dadurch, dass der Schwerpunkt des Patterns *Builder* (*Erbauer*) auf der schrittweisen Generierung eines komplexen Objekts liegt, während sich das Pattern *Abstract Factory* (*Abstrakte Fabrik*) auf Familien von Produktobjekten (sowohl einfache als auch komplexe) konzentriert. Und im Gegensatz zum Pattern *Builder* (*Erbauer*), das die Rückgabe des Produkts als letzten Schritt vollzieht, wird das Produkt beim Pattern *Abstract Factory* (*Abstrakte Fabrik*) sofort zurückgeliefert.

Builder erzeugen häufig die im Fokus des Design Patterns *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) stehenden Komposita.

3.3 Factory Method (Fabrikmethode)

Objektbasiertes Erzeugungsmuster

Zweck

Definition einer Schnittstelle zur Objekterzeugung, wobei die Bestimmung der zu instanzierenden Klasse den Unterklassen überlassen bleibt. Das Design Pattern *Factory Method* (*Fabrikmethode*) gestattet einer Klasse, die Instanziierung an Unterklassen zu delegieren.

Auch bekannt als

Virtual Constructor (Virtueller Konstruktor)

Motivation

Frameworks machen sich zur Definition und Verwaltung der Beziehungen zwischen

Objekten abstrakte Klassen zunutze. Oftmals ist ein Framework auch für die Erzeugung dieser Objekte zuständig.

Nehmen Sie als Beispiel einmal an, es läge ein Framework für Anwendungen vor, die dem User mehrere Dokumente auf einmal präsentieren können. Zwei wichtige Abstraktionen in diesem Framework sind die Klassen `Application` und `Document`. Beide sind abstrakte Klassen und die Clients müssen zur Umsetzung ihrer anwendungsspezifischen Implementierungen Unterklassen von ihnen bilden. Zur Erstellung eines Zeichenprogramms würden beispielsweise die Klassen `DrawingApplication` und `DrawingDocument` definiert. Die Klasse `Application` ist für die Verwaltung der Dokumente zuständig und legt sie auf Anforderung an – also z. B. sobald der User die Menübefehle OPEN (ÖFFNEN) oder NEW (NEU) auswählt.

Weil die jeweils zu instanzierende `Document`-Unterklasse anwendungsspezifisch ist, kann die Klasse `Application` sie nicht vorherbestimmen – sie weiß nur, wann ein neues Dokument angelegt werden soll, nicht aber welche Art von Dokument zu erstellen ist. Und das führt zu einem Problem: Das Framework muss zwar Klassen instanziieren, hat aber ausschließlich von den abstrakten Klassen Kenntnis, die es nicht instanziieren kann.

Hier stellt das Design Pattern *Factory Method (Fabrikmethode)* eine passende Lösung bereit: Es kapselt die Informationen über die zu erzeugende `Document`-Unterklasse und leitet sie aus dem Framework heraus:

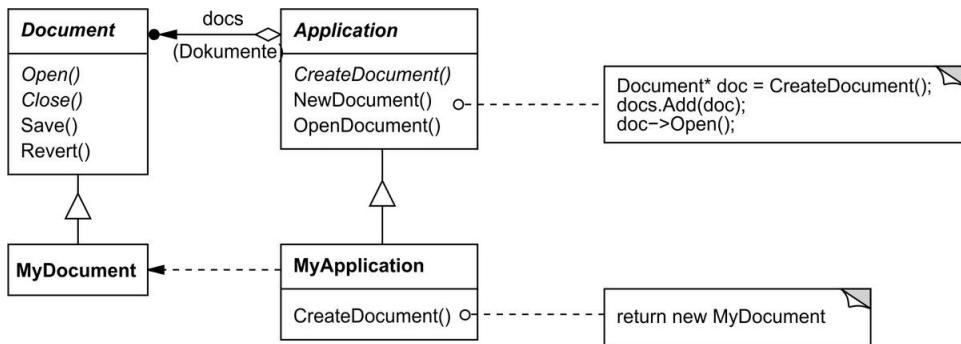


Abb. 3.7: Weiterleitung der Informationen über die zu erzeugende `Document`-Unterklasse aus dem Framework

Die `Application`-Unterklassen nehmen eine dahingehende Neudeinition einer `CreateDocument`-Operation in `Application` vor, dass sie die passende `Document`-Unterklasse zurückgibt. Nach ihrer Instanziierung kann eine `Application`-Unterklasse selbst anwendungsspezifische Dokumente instanziieren, ohne deren zugehörige Klassen zu kennen. Solche Operation wie hier `CreateDocument`, die quasi für die »Herstellung« eines Objekts zuständig sind, werden als **Factory**

Methods bzw. **Fabrikmethoden** bezeichnet.

Anwendbarkeit

Der Einsatz des Design Patterns *Factory Method (Fabrikmethode)* ist dann sinnvoll, wenn

- eine Klasse die Klassen der Objekte, die sie erzeugen muss, nicht im Vorhinein bestimmen kann,
- eine Klasse von ihren Unterklassen eine Spezifizierung der erzeugten Objekte erwartet,
- Klassen Zuständigkeiten an eine von mehreren Hilfsunterklassen delegieren und dann festgestellt werden soll, welche Hilfsunterklasse der Delegate ist.

Struktur

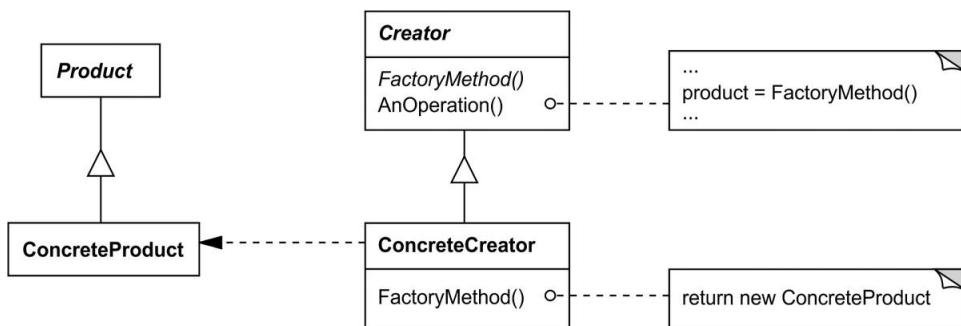


Abb. 3.8: Die Struktur des Design Patterns *Factory Method (Fabrikmethode)*

Teilnehmer

- **Product** (*Document*)
 - Definiert die von der Factory Method erzeugte Objektschnittstelle.
- **ConcreteProduct** (*MyDocument*)
 - Implementiert die Product-Schnittstelle.

- **Creator** (Application)
 - Deklariert die Factory Method, die ein Objekt des Typs Product zurückgibt.
 - Kann die Factory Method zur Erzeugung eines Product-Objekts aufrufen.
- **ConcreteCreator** (MyApplication)
 - Überschreibt die Factory Method, um eine Instanz von ConcreteProduct zurückzugeben.

Interaktionen

- Der creator (Erzeuger) überlässt die Definition der Factory Method, die eine Instanz des passenden ConcreteProduct zurückgeben soll, seinen Unterklassen.

Konsequenzen

Durch die Nutzung von Factory Methods entfällt die Notwendigkeit der Einbindung anwendungsspezifischer Klassen in den Code – dieser befasst sich ausschließlich mit der Product-Schnittstelle und kann daher mit jeder beliebigen benutzerdefinierten ConcreteProduct-Klasse arbeiten.

Ein potenzieller Nachteil der Factory Methods ist jedoch, dass die Clients speziell für die Erzeugung eines bestimmten ConcreteProduct-Objekts gegebenenfalls Unterklassen von der Creator-Klasse bilden müssen. Wenn der Client ohnehin Unterklassen vom Creator anlegen muss, ist dagegen auch nichts einzuwenden – andernfalls ist er allerdings gezwungen, ausschließlich wegen der Factory Method einen zusätzlichen Evolutionszweig zu verwalten.

Darüber hinaus stellt das Design Pattern *Factory Method (Fabrikmethode)* auch folgende Möglichkeiten zur Verfügung:

1. *Spezialisierungsoptionen für Unterklassen.* Die Objekterzeugung innerhalb einer Klasse mittels einer Factory Method ist generell eine flexiblere Lösung als die direkte Erstellung des Objekts – weil sie es den Unterklassen ermöglicht, eine erweiterte Version eines Objekts bereitzustellen.

So könnte die Document-Klasse in dem vorliegenden Beispiel eine Factory Method namens `CreateFileDialog` definieren, die einen Standard-Dateidialog zum Öffnen eines existierenden Dokuments erzeugt. Und eine Document-Unterklasse könnte durch Überschreiben dieser Factory Method wiederum einen anwendungsspezifischen Dateidialog definieren. In diesem Fall wäre die Factory Method nicht abstrakt, sondern eine sinnvolle Standardimplementierung.

2. *Verbindung paralleler Klassenhierarchien.* In den bisher erwähnten Beispielen wurde die Factory Method ausschließlich von Creator-Klassen aufgerufen. Das muss aber nicht so sein. Factory Methods können auch für Clients von Nutzen sein, insbesondere wenn sie parallele Klassenhierarchien verwalten.

Parallele Klassenhierarchien entstehen dann, wenn eine Klasse einen Teil ihrer Zuständigkeiten an eine separate, andere Klasse delegiert. Ein Beispiel dafür sind interaktiv manipulierbare grafische Objekte (`Figure`), die per Maussteuerung gestreckt, bewegt oder gedreht werden können. Solche Interaktionen sind nicht immer leicht zu implementieren: Häufig müssen die Daten, die den Status der Manipulation zu einem bestimmten Zeitpunkt aufzeichnen, immer wieder gespeichert und aktualisiert werden. Diese Angaben werden allerdings nur während der Manipulation benötigt – deshalb brauchen sie nicht in dem grafischen Objekt selbst hinterlegt zu werden. Erschwerend hinzu kommt jedoch, dass die verschiedenen grafischen Objekte bei einer userseitigen Manipulation in der Regel unterschiedliche Verhalten aufweisen. Beispielsweise lässt sich das Strecken einer Linie möglicherweise recht unverfänglich durch das Fortbewegen eines ihrer Endpunkte bewerkstelligen, das Strecken eines Textobjekts könnte dagegen womöglich auch eine Veränderung des Zeilenabstands zur Folge haben.

Angesichts dieser Besonderheiten ist es in der Regel besser, ein separates `Manipulator`-Objekt zu verwenden, das die Interaktion implementiert und jeden benötigten manipulationsspezifischen Zustand überwacht. Dabei nutzen die einzelnen Objekte auch unterschiedliche `Manipulator`-Unterklassen, um bestimmte Interaktionen zu handhaben. Die daraus resultierende `Manipulator`-Klassenhierarchie ist (zumindest teilweise) parallel zur `Figure`-Klassenhierarchie strukturiert:

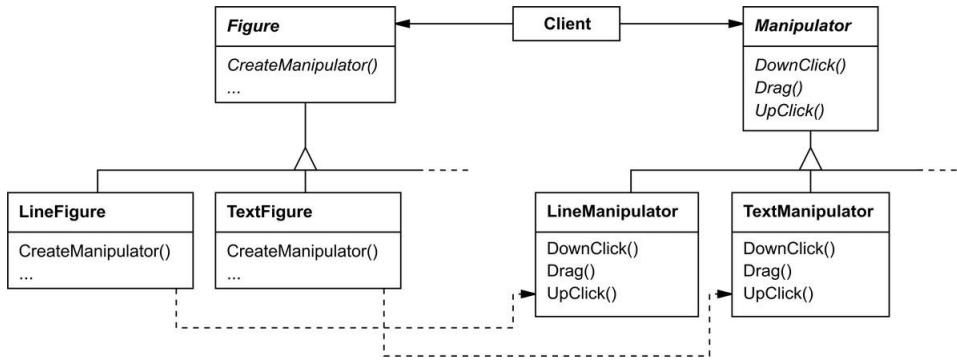


Abb. 3.9: Die Klassenhierarchien *Figure* und *Manipulator*

Die *Figure*-Klasse stellt eine Factory Method `CreateManipulator` bereit, die es den Clients gestattet, eine dem grafischen Objekt (*Figure*) entsprechende *Manipulator*-Unterklasse zu erzeugen. Die *Figure*-Unterklassen überschreiben diese Methode dann so, dass sie eine Instanz der ihnen zugehörigen *Manipulator*-Unterklasse zurückgeben. Alternativ kann die *Figure*-Klasse die Factory Method `CreateManipulator` auch so implementieren, dass sie eine standardmäßige *Manipulator*-Instanz zurückgibt und die *Figure*-Unterklassen diesen Standard dann einfach erben. *Figure*-Klassen, die so vorgehen, benötigen keine entsprechende *Manipulator*-Unterklasse – insofern sind die Hierarchien nur teilweise parallel strukturiert.

Beachtenswert ist hier auch, wie die Factory Method die Verbindung zwischen diesen beiden Klassenhierarchien definiert – indem sie die Zusammenghörigkeit der einzelnen Klassen ermittelt.

Implementierung

Beim Einsatz des Design Patterns *Factory Method (Fabrikmethode)* sollten folgende Aspekte bedacht werden:

1. *Zwei wichtige Pattern-Varianten.* Im Wesentlichen gibt es zwei maßgebliche Varianten des Patterns *Factory Method (Fabrikmethode)*: Im ersten Fall ist die *Creator*-Klasse eine abstrakte Klasse und bietet keine Implementierung für die von ihr deklarierte Factory Method, und im anderen Fall ist *Creator* eine konkrete Klasse und stellt eine Standardimplementierung für die Factory Method bereit. Darüber hinaus ist es ebenfalls möglich, wenn auch nicht sehr gebräuchlich, eine abstrakte Klasse für die Definition einer Standardimplementierung zu nutzen.

Bei der ersten Variante muss *zwingend* eine Implementierung durch die Unterklassen definiert sein, weil kein sinnvoller Standard existiert. Auf diese Weise wird die Problematik der Instanziierung nicht vorherbestimmbarer Klassen umgangen. Im zweiten Fall nutzt der konkrete Creator die Factory Method hauptsächlich aus Gründen der Flexibilität und folgt damit der Regel: »Objekte werden in einer separaten Operation erzeugt, so dass die Unterklassen ihren Erzeugungscode überschreiben können.« Die Einhaltung dieser Regel gewährleistet, dass die Entwickler der Unterklassen die von deren Basisklassen instanziierten Objektklassen bei Bedarf ändern können.

2. *Parametrisierte Factory Methods.* Eine weitere Variante dieses Design Patterns gestattet der Factory Method, gleich *mehrere* Produktarten zu erzeugen. Zu diesem Zweck erhält sie einen Parameter, der den Typ des zu erzeugenden Objekts identifiziert. Alle von der Factory Method erstellten Objekte haben eine gemeinsame Product-Schnittstelle. Würde die Anwendung aus dem im Abschnitt »Motivation« angeführten Framework-Beispiel unterschiedliche Arten von Dokumenten unterstützen, dann müsste CreateDocument zur Bestimmung des zu erzeugenden Dokumenttyps ein zusätzlicher Parameter übergeben werden.

Das Unidraw Application Framework für grafische Editoren [VL90] nutzt diesen Ansatz zur Rekonstruktion von auf der Festplatte gespeicherten Objekten. Unidraw definiert eine Klasse `Creator` mit einer Factory Method `Create`, die einen Klassenidentifizierer (engl. *Class Identifier*) als Argument erhält. Dieser spezifiziert die zu instanzierende Klasse. Wenn Unidraw ein Objekt auf der Festplatte speichert, schreibt es zuerst den Klassenidentifizierer und dann die Instanzvariablen. Ebenso wird auch bei der Rekonstruktion eines Objekts von der Festplatte als Erstes der Klassenidentifizierer ausgelesen.

Nach dem Auslesen des Klassenidentifizierers ruft das Framework die Factory Method `Create` auf und übergibt dabei gleichzeitig den Identifizierer als Parameter. `Create` ermittelt daraufhin den Konstruktor der entsprechenden Klasse und verwendet ihn zur Instanziierung des Objekts. Und als Letztes löst `Create` schließlich die Read-Operation des Objekts aus, die die übrigen Daten von der Festplatte liest und die Instanzvariablen des Objekts initialisiert.

Eine parametrisierbare Factory Method hat im Allgemeinen die folgende Syntax, wobei `MyProduct` und `YourProduct` Unterklassen von `Product` sind:

```
class Creator {  
public:  
    virtual Product* Create(ProductId);
```

```

};

Product* Creator::Create (ProductId id) {
    if (id == MINE) return new MyProduct;
    if (id == YOURS) return new YourProduct;
    // Wird für die verbleibenden Produkte wiederholt...

    return 0;
}

```

Durch das Überschreiben einer parametrisierten Factory Method lassen sich die von einer Creator-Klasse erzeugten Produkte einfach und gezielt erweitern oder ändern. Für neue Produkttypen können entweder neue Identifizierer eingerichtet oder bereits existierende mit verschiedenen Produkten verbunden werden.

Beispielsweise könnte eine Unterklasse MyCreator die Unterklassen MyProduct und YourProduct vertauschen und eine neue Unterklassse TheirProduct unterstützen:

```

Product* MyCreator::Create (ProductId id) {
    if (id == YOURS) return new MyProduct;
    if (id == MINE) return new YourProduct;
    // YOURS und MINE wurden vertauscht

    if (id == THEIRS) return new TheirProduct;

    return Creator::Create(id);
    // Wird aufgerufen, wenn alles andere fehlschlägt
}

```

Beachtenswert ist hier insbesondere, dass als letzter Schritt in dieser Operation ein Aufruf der Create-Methode der Basisklasse ausgeführt wird. Der Grund dafür ist, dass MyCreator::Create ausschließlich die Klassen YOURS, MINE und THEIRS anders als die Basisklasse behandelt – und an anderen Klassen nicht interessiert ist. Das heißt, MyCreator erweitert die erzeugten Produkttypen und delegiert die Zuständigkeit für die Produkterzeugung bis auf wenige Ausnahmen an ihre Basisklasse.

3. *Sprachspezifische Varianten und Problemstellungen.* Die diversen Programmiersprachen bedingen von Natur aus weitere interessante Varianten und Herausforderungen in Bezug auf den Einsatz der Factory Methods.

Smalltalk-Programme nutzen häufig eine Methode, die die Klasse des zu instanziiierenden Objekts zurückliefert. Creator kann diesen Wert zur

Produkterzeugung verwenden und die Factory Method `ConcreteCreator` kann ihn speichern oder sogar selbst berechnen. Auf diese Weise lässt sich die Typbindung des zu instanzierenden `ConcreteProduct` noch weiter hinauszögern.

Eine Smalltalk-Version des Framework-Beispiels könnte gegebenenfalls eine `documentClass`-Methode der Klasse `Application` definieren, die die passende `Document`-Klasse zur Instanziierung von Dokumenten zurückgibt. Die Implementierung von `documentClass` in `MyApplication` gibt die `MyDocument`-Klasse zurück. Somit sähe die Klasse `Application` wie folgt aus:

```
clientMethod
    document := self documentClass new.

documentClass
    self subclassResponsibility
```

Die Klasse `MyApplication` hätte dann diese Syntax:

```
documentClass
    ^ MyDocument
```

und würde die in `Application` zu instanzierende Klasse `MyDocument` zurückliefern.

Ein noch flexiblerer Ansatz, der in etwa den parametrisierten Factory Methods ähnelt, ist das Speichern der zu erzeugenden Klasse als Klassenvariable von `Application`. Dadurch erübrigtsich die Unterklassenbildung von `Application` zwecks Variierung des Produkts.

In C++ sind Factory Methods ausnahmslos virtuelle, häufig sogar rein virtuelle Funktionen. Hier muss allerdings darauf geachtet werden, dass die Factory Methods nicht aus dem Konstruktor des Creators heraus aufgerufen werden – die Factory Method in der Unterklasse `ConcreteCreator` steht zu diesem Zeitpunkt noch nicht zur Verfügung.

Diese Problematik lässt sich jedoch umgehen, indem der Zugriff auf die Produkte sicherheitshalber nur über Zugriffsoperationen erfolgt, die das Produkt auf Anforderung erzeugen – das Produkt wird also nicht unmittelbar im Konstruktor generiert, sondern lediglich mit 0 initialisiert. Die Rückgabe des Produkts erfolgt dann über die Zugriffsoperation, die jedoch zunächst überprüft, ob das Produkt existiert und es gegebenenfalls erst erzeugt. Diese Technik wird auch als **Lazy Initialization (verzögerte Initialisierung)**

bezeichnet. Der folgende Code demonstriert eine typische Implementierung:

```
class Creator {  
public:  
    Product* GetProduct();  
  
protected:  
    virtual Product* CreateProduct();  
  
private:  
    Product* _product;  
};  
  
Product* Creator::GetProduct () {  
    if (_product == 0) {  
        _product = CreateProduct();  
    }  
    return _product;  
}
```

4. *Templates zur Vermeidung der Unterklassenbildung.* Wie bereits im Abschnitt »Konsequenzen« erwähnt, bringen die Factory Methods das potenzielle Problem mit sich, dass nur zum Zweck der Erzeugung passender Produktobjekte zwangsweise Unterklassen angelegt werden müssen. C++ bietet hier eine weitere Lösung in Form der Bereitstellung einer Template-basierten Unterklasse von der Klasse Creator, die durch die Product-Klasse parametrisiert wird:

```
class Creator {  
public:  
    virtual Product* CreateProduct() = 0;  
};  
  
template <class TheProduct>  
class StandardCreator: public Creator {  
public:  
    virtual Product* CreateProduct();  
};  
  
template <class TheProduct>  
Product* StandardCreator<TheProduct>::CreateProduct () {  
    return new TheProduct;  
}
```

Mithilfe dieses Templates liefert der Client nur die Produktklasse zurück – es sind keine Unterklassen von Creator erforderlich.

```
class MyProduct : public Product {  
public:
```

```

        MyProduct();
        // ...
};

StandardCreator<MyProduct> myCreator;

```

5. *Namenskonventionen.* In der Praxis haben sich überwiegend Namenskonventionen etabliert, die die Verwendung von Factory Methods eindeutig zu erkennen geben. Beispielsweise deklariert das MacApp Macintosh Application Framework [App89] die abstrakte Operation, die die Factory Method definiert, grundsätzlich als `Class* DoMakeclass()`, wobei `Class` die Produktklasse angibt.

Beispielcode

Die Memberfunktion `CreateMaze` (siehe [hier](#)) erzeugt ein Labyrinth und gibt es zurück. Ein Problem, das sich hierbei stellt, ist allerdings, dass sie die Klassen des Labyrinths, der Räume, der Türen und der Wände hartkodiert. Deshalb werden im nachfolgenden Beispiel zunächst einmal einige Factory Methods erstellt, die es den Unterklassen ermöglichen, diese Komponenten auszuwählen.

Die Factory Methods zur Erzeugung der Labyrinth-, Raum-, Wand- und Tür-Objekte in `MazeGame` sind folgendermaßen definiert:

```

class MazeGame {
public:
    Maze* CreateMaze();

// Factory Methods:

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};

```

Jede Factory Method gibt eine Labyrinth-Komponente des jeweiligen Typs zurück. `MazeGame` stellt Standardimplementierungen zur Verfügung, die die einfachsten Arten von Labyrinthen, Räumen, Wänden und Türen zurückliefern.

Als Nächstes kann die Klasse CreateMaze nun so umgeschrieben werden, dass sie diese Factory Methods auch aktiv nutzt:

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze ();

    Room* r1 = MakeRoom (1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, Makewall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, Makewall());
    r1->SetSide(West, Makewall());

    r2->SetSide(North, Makewall());
    r2->SetSide(East, Makewall());
    r2->SetSide(South, Makewall());
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Für die Erzeugung mehrerer verschiedener Spiele können außerdem Unterklassen von MazeGame angelegt und einige oder alle von den Factory Methods erzeugten Produkte variiert werden, um einzelne Bestandteile des Labyrinths mit speziellen Eigenschaften auszustatten. So könnte eine Unterklasse BombedMazeGame beispielsweise die Produkte Room und wall neu definieren, so dass sie in der »Bomben-Version« zurückgegeben werden:

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* Makewall() const
        { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

Eine »Zauber-Version« ließe sich in Form der Unterklasse EnchantedMazeGame wie folgt definieren:

```
class EnchantedMazeGame : public MazeGame {
public:
```

```

EnchantedMazeGame();

virtual Room* MakeRoom(int n) const
{ return new EnchantedRoom(n, CastSpell()); }

virtual Door* MakeDoor(Room* r1, Room* r2) const
{ return new DoorNeedingSpell(r1, r2); }

protected:
    Spell* CastSpell() const;
};

```

Praxisbeispiele

Factory Methods sind in Toolkits und Frameworks allgemein gebräuchlich. Das hier vorgestellte Framework-Beispiel demonstriert ihre typische Anwendung in **MacApp** und **ET++** [WGM88]. Das Manipulator-Beispiel stammt dagegen aus **Unidraw**.

Die Klasse **View** im **Smalltalk-80 MVC-Framework** verfügt über eine Methode namens **defaultController**, die ein Controller-Objekt erzeugt, das die Methode wie eine Factory Method aussehen lassen kann [Par90]. Dabei spezifizieren die Unterklassen von **View** die Klasse ihres Standard-Controllers allerdings durch die Definition der Methode **defaultControllerClass**, die die von **defaultController** zu instanzierende Klasse zurückgibt. Somit ist **defaultControllerClass** die eigentliche Factory Method bzw. die Methode, die von den Unterklassen überschrieben werden sollte.

Ein ausgefalleneres Beispiel aus der Smalltalk-80-Welt ist die von der Klasse **Behavior** (der Basisklasse aller Objekte, die Klassen repräsentieren) definierte Factory Method **parserClass**. Sie gestattet einer Klasse, einen benutzerdefinierten Parser für ihren Quelltext zu verwenden. So kann ein Client beispielsweise eine Klasse **SQLParser** definieren, um den Quelltext einer anderen Klasse mit eingebetteten SQL-Anweisungen zu analysieren. Die **Behavior**-Klasse implementiert **parserClass** in der Form, dass die standardmäßige Smalltalk-Parserklasse zurückgegeben wird. Eine Klasse, die eingebettete SQL-Anweisungen enthält, überschreibt diese Methode (als eine Klassenmethode) und gibt die Klasse **SQLParser** zurück.

Das **Orbix-ORB-System** von IONA Technologies [ION94] verwendet das Design Pattern **Factory Method (Fabrikmethode)** zur Generierung eines passenden **Proxy**-Objekts (siehe Design Pattern **Proxy (Proxy)**, [Abschnitt 4.7](#)), wenn ein Objekt eine Referenz auf ein Remote-Objekt anfragt. Das Design Pattern **Factory Method**

(*Fabrikmethode*) erleichtert das Ersetzen des Standard-Proxys durch ein anderes Proxy-Objekt, das z. B. clientseitiges Caching nutzt.

Verwandte Patterns

Das Design Pattern *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) wird häufig mithilfe von Factory Methods implementiert, wie auch das im Abschnitt »Motivation« dieses Patterns (siehe [hier](#)) vorgestellte Beispiel demonstriert.

Im Pattern *Template Method* (*Schablonenmethode*, siehe [Abschnitt 5.10](#)) werden üblicherweise ebenfalls Factory Methods eingesetzt. In dem zuvor beschriebenen Framework-Beispiel repräsentiert NewDocument eine Template Method (Schablonenmethode).

Beim Einsatz des Design Patterns *Prototype* (*Prototyp*, siehe [Abschnitt 3.4](#)) ist keine Unterklassenbildung der Klasse Creator erforderlich. In diesem Fall werden die Objekte mithilfe der Operation Initialize initialisiert, die im Pattern *Factory Method* (*Fabrikmethode*) nicht benötigt wird.

3.4 Prototype (Prototyp)

Objektbasiertes Erzeugungsmuster

Zweck

Spezifikation der zu erzeugenden Objekttypen mittels einer prototypischen Instanz und Erzeugung neuer Objekte durch Kopieren dieses Prototyps.

Motivation

Zur Veranschaulichung des Design Patterns *Prototype* (*Prototyp*) wird im Folgenden ein Editor für Musikpartituren verwendet, der auf einem entsprechend angepassten allgemeinen Framework basiert und um neue Objekte erweitert wurde, die die

Noten, Pausen und Notenlinien repräsentieren. Dieses Editor-Framework könnte auch eine Werkzeugpalette umfassen, die Tools zum Hinzufügen, Auswählen, Verschieben etc. der Musikobjekte in der Partitur bereitstellt. Der User kann also z. B. durch Anklicken des Viertelnoten-Tools Viertelnoten in die Partitur einfügen oder mit einem Klick auf das Verschieben-Tool die Position einer Note auf den Notenlinien und somit die Tonhöhe ändern.

Dieses Beispiel-Framework enthält eine abstrakte Klasse `Graphic` für grafische Komponenten wie Noten und Notenlinien. Außerdem stellt es eine abstrakte Klasse `Tool` zur Definition von Werkzeugen wie denen in der Palette zur Verfügung. Und schließlich definiert das Framework auch eine Unterklasse `GraphicTool` für Tools, die Instanzen von grafischen Objekten erzeugen und zu dem Notenblatt-Dokument hinzufügen.

Doch eben diese Klasse `GraphicTool` stellt den Framework-Designer vor ein Problem, denn: Die Klassen für die Noten und Notenlinien sind anwendungsspezifisch, die Klasse `GraphicTool` gehört jedoch zum Framework. Das heißt, `GraphicTool` besitzt keine Kenntnis darüber, wie die Instanzen der Musikklassen, die zu der Partitur hinzugefügt werden sollen, zu erzeugen sind. Nun könnte man für jede Art von Musikobjekt eine eigene Unterklasse von `GraphicTool` anlegen, allerdings würde das zu einer unüberschaubaren Menge an Unterklassen führen, die sich zudem nur durch die Art des zu instanzierenden Musikobjekts unterscheiden. Aber bekanntermaßen stellt ja die Objektkomposition eine flexible Alternative zur Unterklassenbildung dar – und insofern drängt sich hier die Frage auf, wie das Framework sie zur Parametrisierung von `GraphicTool`-Instanzen mit der Klasse der zu erzeugenden `Graphic`-Objekte nutzen könnte.

Die Lösung lautet, `GraphicTool` neue `Graphic`-Objekte durch Kopieren oder »Klonen« einer Instanz einer `Graphic`-Unterklassie erzeugen zu lassen. Eine solche Instanz wird als **Prototyp** bezeichnet. `GraphicTool` wird mit dem Prototyp parametrisiert, den es klonen und zu dem Dokument hinzufügen soll. Wenn alle `Graphic`-Unterklassen eine `Clone`-Operation unterstützen, kann `GraphicTool` jede Art von `Graphic`-Objekt klonen.

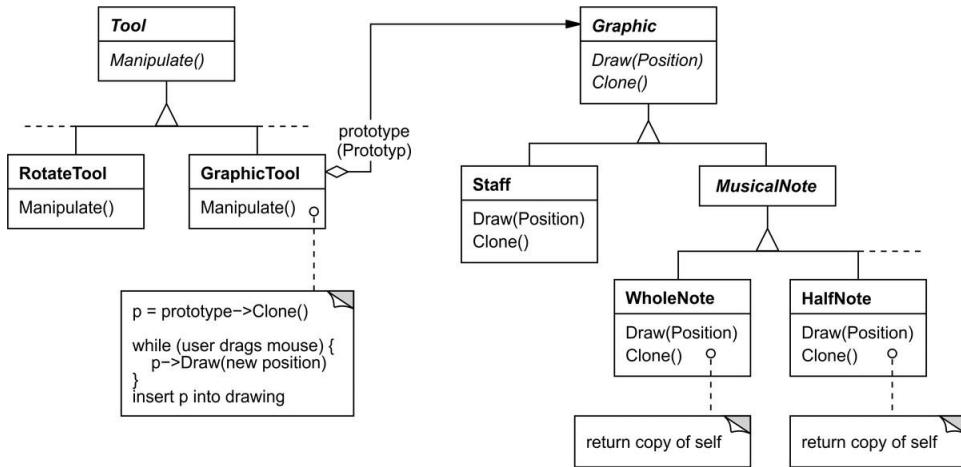


Abb. 3.10: Prototyperstellung durch Klonen einer *Graphic*-Unterklasseninstanz

Damit ist in diesem Musikeditor also jedes Werkzeug zum Erzeugen eines Musikobjekts eine Instanz von **GraphicTool**, die mit einem anderen Prototyp initialisiert wird. Jede **GraphicTool**-Instanz erzeugt durch Klonen ihres Prototyps und Hinzufügen des Klons zu der Partitur jeweils ein Musikobjekt.

Mithilfe des Design Patterns *Prototype (Prototyp)* lässt sich die Anzahl der Klassen sogar noch weiter reduzieren. In diesem Beispiel finden sich separate Klassen für ganze (**WholeNote**) und halbe Noten (**HalfNote**) – das ist allerdings gar nicht unbedingt nötig. Stattdessen könnte man auch mit verschiedenen Bitmaps und Tonlängen initialisierte Instanzen derselben Klasse verwenden. Damit würde ein Werkzeug zum Generieren ganzer Noten zu einem **GraphicTool**-Objekt, dessen Prototyp eine Klasse **MusicalNote** ist, die als ganze Note initialisiert wurde. Auf diese Weise lässt sich die Anzahl der im System befindlichen Klassen dramatisch reduzieren – und die Ergänzung einer neuen Notenart zum Musikeditor ist zudem ebenfalls einfacher zu bewerkstelligen.

Anwendbarkeit

Die Verwendung des Design Patterns *Prototype (Prototyp)* ist dann sinnvoll, wenn die Unabhängigkeit eines Systems von der Art der Erzeugung, Komposition und Darstellung seiner Produkte gewährleistet sein soll *und*

- wenn die zu instanziierenden Klassen zur Laufzeit spezifiziert werden, beispielsweise durch dynamisches Laden, *oder*
- wenn die Errichtung einer Klassenhierarchie von Factories, die parallel zur

Klassenhierarchie der Produkte strukturiert ist, vermieden werden soll, *oder*

- wenn Instanzen einer Klasse nur eine einzige von lediglich wenigen verschiedenen Zustandskombinationen aufweisen können. Es kann gegebenenfalls praktischer sein, eine entsprechende Anzahl von Prototypen zu installieren und diese zu klonen, anstatt die Klasse jedes Mal manuell mit dem richtigen Zustand zu instanzieren.

Struktur

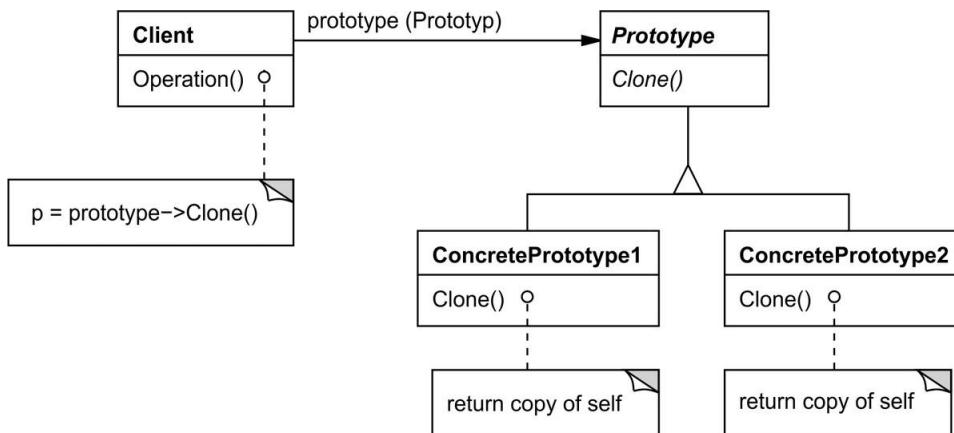


Abb. 3.11: Die Struktur des Design Patterns Prototype (Prototyp)

Teilnehmer

- **Prototype (Graphic)**
 - Deklariert eine Schnittstelle, um sich selbst zu klonen.
- **ConcretePrototype (Staff, WholeNote, HalfNote)**
 - Implementiert eine Operation, um sich selbst zu klonen.
- **Client (GraphicTool)**
 - Erzeugt ein neues Objekt, indem er einen Prototyp anweist, sich selbst zu klonen.

Interaktionen

- Ein Client weist einen Prototyp an, sich selbst zu klonen.

Konsequenzen

Die Auswirkungen des Design Patterns *Prototype* (*Prototyp*) ähneln in vielerlei Hinsicht denen der Patterns *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) und *Builder* (*Erbauer*, siehe [Abschnitt 3.2](#)): Die konkreten Produktklassen bleiben vor dem Client verborgen, so dass sich die Anzahl der ihm bekannten Klassennamen reduziert. Außerdem kann der Client mit anwendungsspezifischen Klassen arbeiten, ohne irgendwelche Modifikationen vornehmen zu müssen.

Darüber hinaus stellt das Pattern *Prototype* (*Prototyp*) aber auch noch folgende Möglichkeiten zur Verfügung:

1. *Produktergänzung und -entfernung zur Laufzeit*. Prototypen gestatten die Einbindung einer neuen Produktklasse in ein System per Registrierung einer prototypischen Instanz am Client. Damit bietet das Pattern *Prototype* (*Prototyp*) etwas mehr Flexibilität als die anderen Erzeugungsmuster, weil ein Client Prototypen zur Laufzeit installieren und entfernen kann.
2. *Spezifikation neuer Objekte mittels Wertevariation*. Hochdynamische Systeme unterstützen die Definition eines neuen Verhaltens per Objektkomposition, beispielsweise durch die Spezifikation von Werten für die Variablen eines Objekts – und nicht durch die Definition neuer Klassen. Neue Objektarten werden effektiv durch die Instanziierung bereits vorhandener Klassen und die Registrierung dieser Instanzen als Prototypen von Client-Objekten definiert. Und durch die Delegierung der Zuständigkeit an den Prototyp kann ein Client ein neues Verhalten annehmen.

Diese Art von Design gestattet den Usern die Definition neuer »Klassen« ohne jegliche Programmierfähigkeit. Im Prinzip ähnelt das Klonen eines Prototyps der Instanziierung einer Klasse. Das Design Pattern *Prototype* (*Prototyp*) kann die Anzahl der im System benötigten Klassen erheblich reduzieren. Im Fall des Musikeditor-Beispiels wird damit erreicht, dass nur eine einzige `GraphicTool`-Klasse eine unbegrenzte Vielfalt von Musikobjekten erzeugen kann.

3. *Spezifikation neuer Objekte mittels Strukturvariation*. Viele Anwendungen erzeugen Objekte auf der Grundlage von Teilen und Subteilen. Beispielsweise

basieren Editoren für Schaltkreise in der Regel auf Teilschaltkreisen und bauen dann darauf auf. (In solchen Anwendungen kommen häufig auch die Design Patterns *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) und *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#)) zum Einsatz.) Komfortablerweise gestatten derartige Anwendungen meist auch die Instanziierung komplexer benutzerdefinierter Strukturen, beispielsweise um einen spezifischen Teilschaltkreis immer wieder verwenden zu können.

Diese Methodik wird im Design Pattern *Prototype* (*Prototyp*) ebenfalls unterstützt: Der Teilschaltkreis wird einfach als Prototyp zu der Palette der verfügbaren Schaltkreiselemente hinzugefügt – und solange das zusammengesetzte Schaltkreisobjekt `Clone` als tiefen Kopie (engl. *Deep Copy*) implementiert, können selbst Schaltkreise mit unterschiedlichen Strukturen als Prototypen verwendet werden.

4. *Reduzierte Unterklassenbildung.* Das Design Pattern *Factory Method* (*Fabrikmethode*, siehe [Abschnitt 3.3](#)) erzeugt oftmals eine Hierarchie von *Creator*-Klassen, die eine gewisse Parallelität mit der Hierarchie der Produktklassen aufweist. Das Pattern *Prototype* (*Prototyp*) ermöglicht es hingegen, einen Prototyp zu klonen, statt eine *Factory Method* anzugeben, ein neues Objekt zu erzeugen – infolgedessen wird hier keine *Creator*-Klassenhierarchie benötigt. Dieser Vorteil wirkt sich vor allem in Sprachen wie C++ aus, die Klassen nicht als Objekte erster Ordnung behandeln. Weniger zur Geltung kommt er hingegen in Programmiersprachen wie Smalltalk und Objective-C, weil Klassenobjekte in diesen Fällen immer als *Creator* verwendet werden können – d. h., hier fungieren die Klassenobjekte ohnehin bereits als Prototypen.
5. *Dynamische Anwendungskonfiguration mit Klassen.* Manche Laufzeitumgebungen gestatten das dynamische Laden von Klassen in eine Anwendung. Und in Sprachen wie C++ ist das Design Pattern *Prototype* (*Prototyp*) das Mittel der Wahl, um den größtmöglichen Nutzen aus diesem Vorteil ziehen zu können.

Eine Anwendung, in der Instanzen einer dynamisch geladenen Klasse zu erstellen sind, wird deren Konstruktor nicht statisch referenzieren können. Stattdessen erzeugt die Laufzeitumgebung beim Laden jeder Klasse automatisch eine Instanz und registriert diese an einem *Prototyp-Manager* (siehe Abschnitt »Implementierung«). Anschließend kann die Anwendung die Instanzen von neu geladenen Klassen – Klassen, die ursprünglich gar nicht mit dem Programm verknüpft waren – beim *Prototyp-Manager* abfragen. Diese

Technik wird beispielsweise vom Laufzeitsystem des ET++ Application Frameworks [WGM88] angewendet.

Das wichtigste Erfordernis für den Einsatz des Design Patterns *Prototype* (*Prototyp*) ist, dass jede Unterklasse von *Prototype* die `clone`-Operation implementieren muss – was mitunter schwierig sein kann, beispielsweise wenn die infrage kommenden Klassen bereits existieren oder ihre internen Darstellungen Objekte enthalten, die ein Kopieren nicht zulassen oder zirkuläre Referenzen aufweisen.

Implementierung

Besonders hilfreich ist das Design Pattern *Prototype* (*Prototyp*) in Programmiersprachen wie C++, in denen Klassen nicht wie Objekte behandelt werden und nur wenige oder gar keine Typinformationen zur Laufzeit verfügbar sind. Weniger Bedeutung kommt ihm dagegen in Sprachen wie Smalltalk oder Objective-C zu, die mit Prototypen vergleichbare Objekte (sprich Klassenobjekte) für die Erzeugung von Instanzen jeder Klasse bieten. In Prototyp-basierten Sprachen wie Self [US87], in denen die Objekterzeugung grundsätzlich durch das Klonen eines Prototyps erfolgt, ist das Pattern *Prototype* (*Prototyp*) schon von vornherein integriert.

Bei der Implementierung dieses Patterns sollten zudem folgende Möglichkeiten berücksichtigt werden:

1. *Einsatz eines Prototyp-Managers*. Wenn keine feste Prototyp-Anzahl in einem System vorgegeben ist (d. h., sie können dynamisch erzeugt und entfernt werden), sollten die verfügbaren Prototypen in einer speziellen Registratur, dem sogenannten **Prototyp-Manager** hinterlegt sein. Dabei verwalten die Clients die Prototypen nicht selbst, sondern speichern sie in der Registratur und rufen sie bei Bedarf wieder von ihr ab – das gilt auch für Prototypen, die geklont werden sollen.

Ein Prototyp-Manager ist ein assoziativer Speicher, der Prototypen zurückgibt, die jeweils einem bestimmten Schlüssel entsprechen. Zu diesem Zweck enthält er Operationen, die die Registrierung eines Prototyps mit einem zugehörigen Schlüssel sowie den Ausschluss eines Prototyps von der Registrierung ermöglichen. Die Clients können die Registratur zur Laufzeit ändern oder auch durchsuchen und haben damit die Möglichkeit, das System zu erweitern und sich eine Übersicht zu verschaffen, ohne Code schreiben zu müssen.

2. *Implementierung der Clone-Operation.* Die schwierigste Aufgabe im Umgang mit dem Design Pattern *Prototype* (*Prototyp*) ist die korrekte Implementierung der `clone`-Operation – die sich besonders knifflig gestaltet, wenn die Objektstrukturen zirkuläre Referenzen enthalten.

Die meisten Programmiersprachen unterstützen das Klonen von Objekten zumindest teilweise. Smalltalk stellt z. B. eine Implementierung von `copy` zur Verfügung, die von allen `Object`-Unterklassen geerbt wird. Und C++ verfügt über einen Copy-Konstruktor. Dennoch bieten auch diese Optionen keine Lösung für das »Flache Kopie (engl. *Shallow Copy*) kontra tiefe Kopie (engl. *Deep Copy*)«-Problem [GR83]. Aber bedeutet das nun, dass das Klonen eines Objekts das Klonen seiner Instanzvariablen zur Folge hat oder teilen sich der Klon und das Originalobjekt nur dieselben Variablen?

Eine flache Kopie ist zwar simpel, reicht aber häufig völlig aus und wird daher in Smalltalk standardmäßig angeboten. Der standardmäßige Copy-Konstruktor in C++ erstellt eine Kopie der Membervariablen, d. h., die Kopie und das Original teilen sich die Pointer. Allerdings ist für das Klonen von Prototypen mit komplexen Strukturen normalerweise eine tiefe Kopie erforderlich, weil Klon und Original unabhängig voneinander sein müssen. Deshalb muss sichergestellt werden, dass die Komponenten des Klons wiederum Klone der Komponenten des Prototyps sind. Mit anderen Worten: Das Klonen an sich erzwingt eine Entscheidung hinsichtlich dessen, was genau – wenn überhaupt – gemeinsam genutzt werden soll.

Wenn Objekte in dem System Speicher- und Ladeoperationen zur Verfügung stellen, können diese für eine Standardimplementierung der `clone`-Operation genutzt werden, indem das betreffende Objekt gespeichert und dann sofort wieder geladen wird: `Save` (Speichern) speichert das Objekt in einen Zwischenspeicher und `Load` (Laden) rekonstruiert das Objekt in Form eines Duplikats.

3. *Initialisierung von Klonen.* Während manche Clients den Klon problemlos akzeptieren, wie er ist, verlangen andere Clients die teilweise oder vollständige Initialisierung seines internen Status mit von ihnen vorgegebenen Werten. Und im Allgemeinen können diese Werte nicht unmittelbar in einer `clone`-Operation übergeben werden, weil ihre Anzahl je nach Klasse des Prototyps variiert: Einige Prototypen erfordern möglicherweise mehrere Initialisierungsparameter, andere brauchen dagegen überhaupt keine. Die Übergabe von Parametern im Rahmen der `Clone`-Operation schließt eine einheitliche Klonschnittstelle aus.

Es kann auch vorkommen, dass die Klassen der Prototypen bereits Operationen zum (Zurück-)Setzen wichtiger Zustandsfaktoren definieren. In diesem Fall könnten die Clients diese Operationen unmittelbar nach dem Klonen verwenden. Ansonsten muss gegebenenfalls eine `Initialize`-Operation erstellt werden (siehe Abschnitt »Beispielcode«), die die Initialisierungsparameter als Argumente entgegennimmt und den internen Zustand des Klons entsprechend anpasst. Besondere Vorsicht ist bei `clone`-Operationen geboten, die tiefe Kopien erzeugen, denn hier müssen die Kopien vor der erneuten Initialisierung gegebenenfalls gelöscht werden (entweder explizit oder innerhalb der `Initialize`-Operation).

Beispielcode

Im Folgenden wird zunächst eine Unterklasse `MazePrototypeFactory` von `MazeFactory` (siehe [hier](#)) definiert. Diese Unterklasse wird mit den Prototypen der Objekte initialisiert, die sie erzeugen wird, damit für die Änderung der Klassen der von ihr generierten Wände und Räume keine weitere Unterklassenbildung erforderlich ist.

`MazePrototypeFactory` erweitert die `MazeFactory`-Schnittstelle um einen Konstruktor, der die Prototypen als Argument erhält:

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory (Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

Der neue Konstruktor initialisiert einfach nur seine Prototypen:

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
    _prototypeMaze = m;
```

```

    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
}

```

Die Memberfunktionen zur Erzeugung der Wände, Räume und Türen sind im Prinzip alle sehr ähnlich aufgebaut: Jede dieser Funktionen klonen einen Prototyp und initialisiert ihn anschließend. `MakeWall` und `MakeDoor` sind beispielsweise wie folgt definiert:

```

Wall* MazePrototypeFactory::MakeWall () const {
    return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
    Door* door = _prototypeDoor->Clone();
    door->Initialize(r1, r2);
    return door;
}

```

Die Unterklasse `MazePrototypeFactory` kann nun zur Erzeugung eines prototypischen oder standardmäßigen Labyrinths verwendet und mit den Prototypen der grundlegenden Labyrinth-Bestandteile initialisiert werden:

```

MazeGame game;
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);

```

Zur Modifizierung des Labyrinth-Aufbaus wird `MazePrototypeFactory` mit einem anderen Satz Prototypen initialisiert, wie z. B. mit folgendem Aufruf, der ein Labyrinth mit den Komponenten `BombedDoor` und `RoomWithABomb` erzeugt:

```

MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);

```

Ein als Prototyp zu verwendendes Objekt, beispielsweise eine Instanz von `Wall`, muss die `Clone`-Operation unterstützen und zudem über einen Copy-Konstruktor zum Klonen verfügen. Gegebenenfalls benötigt es auch eine separate Operation zur Neuinitialisierung des internen Zustands. Um eine clientseitige Initialisierung der geklonnten Räume zu ermöglichen, wird die Klasse `Door` in diesem Beispiel um die `Initialize`-Operation ergänzt.

Abweichend von ihrer ursprünglichen Fassung (siehe [hier](#)) ist die Klasse Door nun folgendermaßen definiert:

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    virtual void Initialize(Room*, Room*);
    virtual Door* Clone() const;
    virtual void Enter();
    Room* OtherSideFrom(Room*);

private:
    Room* _room1;
    Room* _room2;
};

Door::Door (const Door& other) {
    _room1 = other._room1;
    _room2 = other._room2;
}

void Door::Initialize (Room* r1, Room* r2) {
    _room1 = r1;
    _room2 = r2;
}

Door* Door::Clone () const {
    return new Door(*this);
}
```

Die Unterklasse BombedWall muss `Clone` überschreiben und einen entsprechenden Copy-Konstruktor implementieren:

```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();

private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}
```

```
Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}
```

`BombedWall::Clone` gibt zwar `Wall*` zurück, ihre Implementierung liefert jedoch einen Pointer auf eine neue Instanz einer UnterkLASSE, sprich `BombedWall*`. Durch diese Definitionsform der `Clone`-Operation in der BasiskLASSE ist sichergestellt, dass die Clients, die den Prototyp klonen, keine Kenntnis von dessen konkreten Unterklassen haben müssen. Generell sollten die Clients niemals gezwungen sein, einen Downcast auf den Rückgabewert der `Clone`-Operation zum gewünschten Typ auszuführen.

Smalltalk gestattet die Wiederverwendung der von `Object` geerbten Standardmethode `copy` zum Klonen jeder beliebigen `MapSite`-UnterkLASSE. Und mit `MazeFactory` können alle benötigten Prototypen generiert werden – so ließe sich beispielsweise durch die Übergabe des Bezeichners `#room` ein Raum erzeugen. Außerdem verfügt die Klasse `MazeFactory` über ein Dictionary, das den jeweiligen Prototypen mithilfe der `make`-Operation Namen zuweist:

```
make: partName
  ^ (partCatalog at: partName) copy
```

Unter der Voraussetzung, dass passende Methoden zur Initialisierung der `MazeFactory` mit Prototypen vorhanden sind, würde folgender Code ein einfaches Labyrinth generieren:

```
CreateMaze
on: (MazeFactory new
    with: Door new named: #door;
    with: Wall new named: #wall;
    with: Room new named: #room;
    yourself)
```

wobei die Definition der `on`-Klassenmethode für `CreateMaze` so aussehen würde:

```
on: aFactory
| room1 room2 |
room1 := (aFactory make: #room) location: 1@1.
room2 := (aFactory make: #room) location: 2@1.
door := (aFactory make: #door) from: room1 to: room2.

room1
    atSide: #north put: (aFactory make: #wall);
    atSide: #east put: door;
    atSide: #south put: (aFactory make: #wall);
```

```

    atSide: #west put: (aFactory make: #wall).
room2
    atSide: #north put: (aFactory make: #wall);
    atSide: #east put: (aFactory make: #wall);
    atSide: #south put: (aFactory make: #wall);
    atSide: #west put: door.
^ Maze new
addRoom: room1;
addRoom: room2;
yourself

```

Praxisbeispiele

Eins der frühesten Beispiele für den Einsatz des Design Patterns *Prototype* (*Prototyp*) findet sich in Ivan Sutherlands **Sketchpad-System** [Sut63]. Einem breiteren Publikum bekannt wurde es jedoch erst durch seine Verwendung in einer objektorientierten Sprache – und zwar in der grafischen Programmierumgebung **ThingLab**, die es den Usern gestattet, Kompositobjekte zu gestalten und diese durch anschließende Installation in einer Library wiederverwendbarer Objekte als Prototyp zu deklarieren [Bor81]. Die Nutzung von Prototypen als Pattern findet auch bei **Goldberg und Robson** Erwähnung [GR83], allerdings liefert **Coplien** [Cop92] hier eine deutlich umfassendere Erläuterung: Er beschreibt mit dem Design Pattern *Prototype* (*Prototyp*) verwandte C++-Idiome anhand zahlreicher Beispiele und Variationen.

Das auf ET++ basierte Debugger-Frontend **etgdb** verfügt über eine Point-and-click-Schnittstelle für diverse zeilenorientierte Debugger, von denen jeder eine zugehörige **DebuggerAdaptor**-Unterklasse besitzt. Beispielsweise passt **GdbAdaptor** **etgdb** an die Befehlssyntax des GNU-Debuggers **gdb** an, während **SunDbxAdaptor** in ähnlicher Weise eine Anpassung für den **dbx** debugger von Sun vornimmt. **Etgdb** enthält keinen hartkodierten **DebuggerAdaptor**-Klassensatz, sondern liest den Namen des zu verwendenden Adapters aus einer Umgebungsvariablen aus, macht den in einer globalen Tabelle hinterlegten Prototyp mit derselben Bezeichnung ausfindig und klont diesen dann. Zur Ergänzung neuer Debugger wird **etgdb** mit dem passenden **DebuggerAdaptor** verknüpft.

Die »Interaction Technique Library« (Bibliothek für Interaktionstechniken) im **Mode Composer** speichert Prototypen von Objekten, die verschiedene Interaktionstechniken unterstützen [Sha90]. Jede vom Mode Composer erzeugte Interaktionstechnik kann durch Einfügen in diese Bibliothek als Prototyp verwendet werden. Das Design Pattern *Prototype* (*Prototyp*) ermöglicht dem Mode Composer die Unterstützung einer unbegrenzten Anzahl von Interaktionstechniken.

Das im Abschnitt »Motivation« vorgestellte Musikeditor-Beispiel basiert auf dem **Unidraw Application Framework** für grafische Editoren [VL90].

Verwandte Patterns

Wie am Ende dieses Kapitels noch näher ausgeführt wird, stehen die Design Patterns *Prototype (Prototyp)* und *Abstract Factory (Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) in mancher Hinsicht in Konkurrenz zueinander. Andererseits können sie aber auch gemeinsam genutzt werden. Beispielsweise kann eine abstrakte Factory Prototypen speichern, die geklont und als Produktobjekte zurückgegeben werden sollen.

Auch Designs, die sich in weiten Teilen auf die Patterns *Composite (Kompositum*, siehe [Abschnitt 4.3](#)) und *Decorator (Dekorierer*, siehe [Abschnitt 4.4](#)) stützen, können vom Design Pattern *Prototype (Prototyp)* profitieren.

3.5 Singleton (Singleton)

Objektbasiertes Erzeugungsmuster

Zweck

Sicherstellung der Existenz nur einer einzigen Klasseninstanz sowie Bereitstellung eines globalen Zugriffspunkts für diese Instanz.

Motivation

In manchen Fällen ist es unabdingbar, dass eine Klasse lediglich genau eine einzige Instanz besitzt. So sollte es beispielsweise trotz mehrerer in einem System eingebundenen Druckern immer nur einen Druckerspooler geben. Ebenso sollte auch nur ein Dateisystem und ein Fenstermanager (engl. *Window Manager*) vorhanden sein. Ein Digitalfilter greift ebenfalls lediglich auf einen einzigen A/D-Wandler zu, und gleichermaßen ist auch ein Buchhaltungssystem in aller Regel auf die Anforderungen genau eines Unternehmens ausgelegt.

Wie aber lässt sich sicherstellen, dass eine Klasse nur eine einzige Instanz besitzt und problemlos auf diese Instanz zugegriffen werden kann? Eine globale Variable ermöglicht zwar den Zugriff auf ein Objekt, sie verhindert jedoch nicht, dass mehrere Instanzen erzeugt werden.

Die bessere Lösung besteht darin, der Klasse selbst die Verantwortung dafür zu übertragen, dass nur eine einzige Instanz von ihr existiert. Das heißt, sie selbst stellt sicher (durch das Abfangen von Requests zur Erzeugung neuer Objekte), dass keine weitere Instanz angelegt wird und ermöglicht zudem den Zugriff auf die einzige existierende Instanz. All dies wird mit dem Design Pattern *Singleton* (*Singleton*) realisiert.

Anwendbarkeit

Der Einsatz des Patterns *Singleton* (*Singleton*) ist dann sinnvoll, wenn

- es nur eine einzige Instanz einer Klasse geben darf und diese von einem bestimmten Zugriffspunkt für die Clients zugänglich sein muss.
- die einzige Instanz per Unterklassenbildung erweiterbar sein soll und die Clients in der Lage sein sollen, eine erweiterte Instanz zu nutzen, ohne ihren Code ändern zu müssen.

Struktur

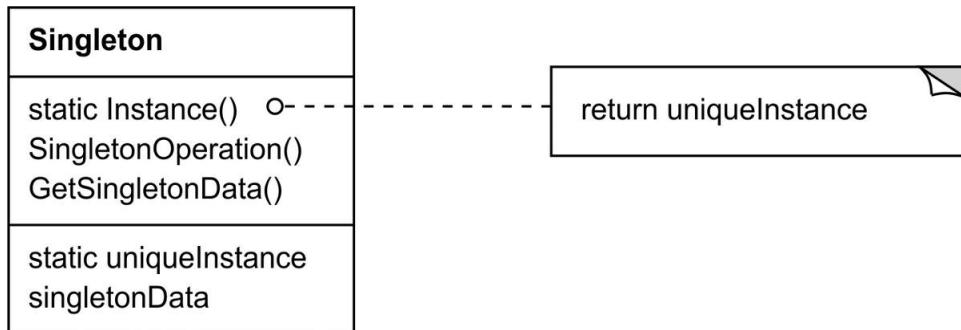


Abb. 3.12: Die Struktur des Design Patterns *Singleton* (*Singleton*)

Teilnehmer

- **Singleton**

- Definiert eine Instance-Operation, die den Clients den Zugriff auf die einzige existierende Instanz gestattet. Instance ist eine Klassenoperation (d. h. in Smalltalk eine Klassenmethode und in C++ eine statische Memberfunktion).
- Kann gegebenenfalls für die Erzeugung seiner einzigen Instanz verantwortlich sein.

Interaktionen

- Der clientseitige Zugriff auf eine Singleton-Instanz erfolgt ausschließlich über die Instance-Operation der Singleton-Klasse.

Konsequenzen

Das Design Pattern *Singleton* (*Singleton*) bringt mehrere Vorteile mit sich:

1. *Kontrollierter Zugriff auf die einzige Instanz.* Da die Singleton-Klasse ihre einzige vorhandene Instanz kapselt, hat sie die alleinige Kontrolle darüber, wie und wann die Clients darauf zugreifen.
2. *Eingeschränkter Namensraum.* Das Pattern *Singleton* (*Singleton*) stellt gegenüber globalen Variablen insofern eine Verbesserung dar, als dass es eine Überfrachtung des Namensraums mit globalen Variablen, die alleinige Instanzen speichern, verhindert.
3. *Verbesserte Operationen und Darstellung.* Die Singleton-Klasse kann durch Unterklassenbildung erweitert und spezialisiert werden. Darüber hinaus lässt sich die Anwendung mit einer Instanz dieser erweiterten Klasse leichter konfigurieren. Es ist sogar möglich, die Konfiguration mit einer Instanz der benötigten Klasse zur Laufzeit vorzunehmen.
4. *Variable Anzahl von Instanzen.* Selbst wenn zu einem späteren Zeitpunkt möglicherweise doch mehr als nur eine einzige Instanz der Singleton-Klasse benötigt werden sollten, ist dies mit dem Pattern *Singleton* (*Singleton*) kein Problem. Das Verfahren zur Überwachung der von der Anwendung zu nutzenden Instanzenanzahl bleibt dasselbe, es muss lediglich eine

entsprechende Anpassung der Operation erfolgen, die den Zugriff auf die Singleton-Instanz gewährt.

5. *Mehr Flexibilität als bei Klassenoperationen.* Eine weitere Möglichkeit, die Funktionalität eines Singletons zusammenzufassen, besteht in der Nutzung von Klassenoperationen (bzw. statischen Memberfunktionen in C++ und Klassenmethoden in Smalltalk). Allerdings erschweren diese Sprachtechniken die Modifikation des Designs, wenn doch mehr als eine Instanz einer Klasse benötigt wird. Zudem sind statische Memberfunktionen in C++ niemals virtuell und können daher nicht polymorph von Unterklassen überschrieben werden.

Implementierung

Beim Einsatz des Design Patterns *Singleton (Singleton)* sollten folgende Aspekte bedacht werden:

1. *Sicherstellung einer einzigen Instanz.* Das Pattern *Singleton (Singleton)* macht die einzige existierende Instanz zu einer normalen Klasseninstanz, deren Basisklasse bereits vorgibt, dass nur eine einzige Instanz angelegt werden kann. Üblicherweise ist die Operation zur Erzeugung dieser Instanz hinter einer Klassenoperation verborgen (also entweder einer statischen Memberfunktion oder einer Klassenmethode). Diese Operation hat Zugriff auf die Variable, die die einzige Instanz enthält, und stellt sicher, dass die Variable vor der Rückgabe des Wertes mit der einzigen Instanz initialisiert wird. Dadurch ist gewährleistet, dass ein Singleton bereits vor seiner ersten Verwendung generiert und initialisiert wird.

In C++ kann die Klassenoperation mit einer statischen Memberfunktion `Instance` der Singleton-Klasse definiert werden. Außerdem definiert Singleton eine statische Membervariable `_instance`, die einen Pointer auf ihre einzige Instanz enthält.

Die Singleton-Klasse ist wie folgt deklariert:

```
class Singleton {  
public:  
    static Singleton* Instance();  
protected:  
    Singleton();  
  
private:  
    static Singleton* _instance;
```

```
};
```

Ihre Implementierung sieht folgendermaßen aus:

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Der clientseitige Zugriff auf das Singleton erfolgt ausschließlich über die Memberfunktion `Instance`. Die Variable `_instance` wird mit 0 initialisiert, und die statische Memberfunktion `Instance` gibt den Wert des Singletons zurück. Sofern dieser 0 ist, wird das Singleton mit der einzigen existierenden Instanz initialisiert. `Instance` folgt dabei dem Prinzip der verzögerten Initialisierung (engl. *Lazy Initialization*): Der Rückgabewert wird erst zum Zeitpunkt des ersten Zugriffs erzeugt und gespeichert.

Wichtig ist hierbei, dass der Konstruktor geschützt ist: Sobald ein Client den Versuch unternimmt, `Singleton` direkt zu instanziieren, wird eine Fehlermeldung zur Laufzeit ausgegeben. Auf diese Weise ist gewährleistet, dass wirklich nur eine einzige Instanz erzeugt wird.

Da `_instance` einen Pointer auf ein `Singleton`-Objekt darstellt, kann die Memberfunktion `Instance` dieser Variablen außerdem einen Pointer auf eine Unterklasse von `Singleton` zuweisen (siehe auch Abschnitt »Beispielcode«).

Im Fall der C++-Implementierung reicht es jedoch nicht aus, das Singleton als globales oder statisches Objekt zu definieren und sich dann auf die automatische Initialisierung zu verlassen – und zwar aus drei Gründen:

- a. Es kann nicht garantiert werden, dass tatsächlich nur eine einzige Instanz eines statischen Objekts deklariert wird.
- b. Es stehen möglicherweise nicht genug Informationen zur Verfügung, um jedes Singleton zur statischen Initialisierungszeit erzeugen zu können. Mitunter erfordert ein Singleton Werte, die erst im Rahmen der späteren Programmausführung berechnet werden.
- c. C++ legt keine Reihenfolge fest, in der die Konstruktoren globaler Objekte bei der Kompilierung mehrerer Programmteile aufgerufen

werden [ES90]. Das bedeutet, dass es unter den verschiedenen Singletons keine Abhängigkeiten geben darf. Gibt es diese dennoch, kommt es unweigerlich zu Fehlern.

2. Ein zusätzliches (wenn auch weniger bedeutsames) Erfordernis des globalen/statischen Objektansatzes ist, dass grundsätzlich alle Singletons erzeugt werden müssen, unabhängig davon, ob sie nun benutzt werden oder nicht. All diese Probleme werden durch die Verwendung einer statischen Memberfunktion vermieden.

In Smalltalk wird die Funktion, die die einzige Instanz zurückgibt, als Klassenmethode der Singleton-Klasse implementiert. Um sicherzugehen, dass auch wirklich nur eine Instanz erstellt wird, wird die new-Operation überschrieben. Die daraus resultierende Singleton-Klasse kann dann beispielsweise die folgenden beiden Klassenmethoden enthalten, wobei die Klassenvariable SoleInstance nirgendwo anders verwendet wird:

```
new
    self error: 'cannot create new object'

default
    SoleInstance isNil ifTrue: [SoleInstance := super new].
    ^ SoleInstance
```

3. *Unterklassenbildung der Singleton-Klasse.* Das schwerwiegenderste Problem ist allerdings nicht so sehr die Definition der Unterklasse, sondern die Installation ihrer einzigen Instanz, so dass sie vonseiten der Clients genutzt werden kann. Im Wesentlichen muss die Variable, die auf die Singleton-Instanz verweist, mit einer Instanz der Unterklasse initialisiert werden. Am einfachsten ist dies durch die Bestimmung des Singletons zu erreichen, das in der Instance-Operation der Singleton-Klasse verwendet werden soll. Die entsprechende Implementierung mithilfe von Umgebungsvariablen ist im Abschnitt »Beispielcode« dokumentiert.

Eine andere Möglichkeit, die Unterklasse von Singleton auszuwählen, besteht darin, die Operation Instance aus der Basisklasse (z. B. MazeFactory) zu entfernen und in die Unterklasse einzufügen. Auf diese Weise braucht der C++-Programmierer die Klasse des Singletons erst zum Zeitpunkt der Verknüpfung zu bestimmen (z. B. durch das Verlinken einer Datei, die eine andere Implementierung enthält) und kann sie weiterhin vor den Clients des Singletons verbergen.

Dieser Ansatz behebt zwar das Problem der Festlegung der Singleton-Klasse

beim Verlinken, erschwert aber die Auswahl der Singleton-Klasse zur Laufzeit. Die Bestimmung der Unterklasse mithilfe bedingter Anweisungen bietet in dieser Hinsicht zwar mehr Flexibilität, andererseits werden die infrage kommenden Singleton-Klassen dabei aber fest im Code verankert – mit anderen Worten: Keine dieser Vorgehensweisen ist letztendlich flexibel genug, um alle Fälle abdecken zu können.

Eine weitreichendere Flexibilität lässt sich allerdings mit einer sogenannten **Singleton-Registratur** erzielen. Dabei wird die Auswahl der möglichen Singleton-Klassen nicht durch die `Instance`-Operation definiert, sondern die Singleton-Klassen registrieren ihre Singleton-Instanzen namentlich in einer entsprechenden Registratur.

Diese Registratur bildet ihrerseits wiederum die aus Strings bestehenden Namen auf die Singletons ab. Wenn also die `Instance`-Operation ein Singleton benötigt, übermittelt sie dessen Namen an die Registratur, die daraufhin das zugehörige Singleton heraussucht und (sofern vorhanden) zurückgibt. Bei diesem Verfahren muss `Instance` nun nicht mehr alle verfügbaren Singleton-Klassen oder -Instanzen selbst kennen – es ist einzige und allein eine gemeinsame Schnittstelle für alle Singleton-Klassen erforderlich, die unter anderem auch Operationen für die Registratur enthält:

```
class Singleton {
public:
    static void Register(const char* name, Singleton* );
    static Singleton* Instance();

protected:
    static Singleton* Lookup(const char* name);

private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

`Register` registriert die Singleton-Instanz unter dem angegebenen Namen. Damit die Registratur überschaubar bleibt, werden die Instanzen in diesem Fall in einer Liste von `NameSingletonPair`-Objekten gespeichert. Jedes `NameSingletonPair`-Objekt bildet einen Namen auf ein Singleton ab – und die `Lookup`-Operation sucht dann das mit dem übergebenden Namen übereinstimmende Singleton heraus. In dem hier angeführten Beispiel wird vorausgesetzt, dass der Name des gewünschten Singletons in einer Umgebungsvariablen spezifiziert ist:

```

Singleton* Singleton::Instance () {
    if (_instance ==0) {
        const char* singletonName = getenv("SINGLETON");
        // Wird beim Startup vom User oder der Umgebung angegeben

        _instance = Lookup(singletonName);
        // Gibt 0 zurück, wenn kein Singleton dieses Namens
vorhanden ist
    }
    return _instance;
}

```

Aber an welchem Punkt registrieren sich die Singleton-Klassen genau? Eine Möglichkeit dafür ist ihr Konstruktor. Dazu könnte eine Unterklasse `MySingleton` beispielsweise wie folgt vorgehen:

```

MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}

```

Aber natürlich wird der Konstruktor erst aufgerufen, wenn die Klasse instanziert wird – was genau das Problem widerspiegelt, das das Design Pattern *Singleton (Singleton)* zu lösen versucht! In C++ lässt sich diese Situation durch die Definition einer statischen Instanz von `MySingleton` umgehen. Beispielsweise könnte in der Datei, in der `MySingleton` implementiert wird, Folgendes definiert werden:

```
static MySingleton theSingleton;
```

Damit ist die Singleton-Klasse nicht mehr für die Erzeugung des Singletons zuständig – stattdessen ist sie in erster Linie dafür verantwortlich, das gewünschte Singleton-Objekt im System zugänglich zu machen. Aber auch dieser Ansatz mit den statischen Objekten hat einen potenziellen Nachteil: Es müssen Instanzen von allen möglichen Singleton-Unterklassen erzeugt werden, denn andernfalls werden sie nicht registriert.

Beispielcode

In diesem Beispiel soll, wie [hier](#) beschrieben, eine Klasse `MazeFactory` zur Errichtung von Labyrinthen definiert werden. Diese Klasse stellt eine Schnittstelle zur Erzeugung verschiedener Bestandteile des Labyrinths bereit. Anschließend können die Unterklassen die Definitionen der Operationen so anpassen, dass sie

Instanzen spezialisierter Produktklassen zurückgeben, wie z. B. `BombedWall`-Objekte statt reine `wall`-Objekte.

Wichtig ist in diesem Zusammenhang, dass die Labyrinth-Anwendung nur eine einzige Instanz von `MazeFactory` benötigt – und diese Instanz sollte jeglichem Code zur Verfügung stehen, der irgendeine Komponente des Labyrinths erzeugt. Hier kommt nun das Design Pattern *Singleton* (*Singleton*) ins Spiel: Durch die Verwendung der Klasse `MazeFactory` als Singleton wird das betreffende Labyrinth-Objekt allgemein zugänglich, ohne dass zu diesem Zweck globale Variablen eingesetzt werden müssen.

Der Einfachheit halber soll an dieser Stelle angenommen werden, dass eine Unterklassenbildung von `MazeFactory` auch zukünftig zu keinem Zeitpunkt notwendig sein wird. (Die gegenteilige Situation wird gleich noch erörtert.) In C++ wird die Klasse `MazeFactory` durch Hinzufügen einer statischen `Instance`-Operation und einer statischen `_instance`-Membervariablen zu einem Singleton, das die einzige und alleinige Instanz enthält. Außerdem muss der Konstruktor zur Vermeidung unbeabsichtigter Instanziierungen, die in mehr als einer Instanz resultieren könnten, geschützt sein:

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // Hier wird die vorhandene Schnittstelle angegeben
protected:
    MazeFactory();

private:
    static MazeFactory* _instance;
};
```

Die zugehörige Implementierung lautet:

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0 ) {
        _instance = new MazeFactory();
    }
    return _instance;
}
```

Was aber würde passieren, wenn es doch Unterklassen von `MazeFactory` gäbe und die Anwendung daher entscheiden müsste, welche von ihnen sie benutzen soll? In

diesem Fall wird die Labyrinth-Variante mittels einer Umgebungsvariablen ausgewählt und Code hinzugefügt, der die passende `MazeFactory`-Unterklasse auf der Grundlage des Wertes dieser Umgebungsvariablen instanziert. Am sinnvollsten lässt sich besagter Code in der `Instance`-Operation unterbringen, weil sie `MazeFactory` ohnehin instanziert:

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0 ) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

            // ... weitere mögliche Unterklassen

        } else {
            // Standard
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

Dabei ist jedoch zu beachten, dass `Instance` mit jeder Definition einer neuen Unterklasse von `MazeFactory` entsprechend modifiziert werden muss. In dieser Beispielanwendung mag das kein Problem sein, bei in einem Framework definierten abstrakten Factories kann dies allerdings schon ganz anders aussehen.

Eine mögliche Lösung wäre der im Abschnitt »Implementierung« beschriebene Registraturansatz. Auch das dynamische Binden könnte in dieser Situation hilfreich sein, denn damit würde die Anwendung davor bewahrt, sämtliche Unterklassen laden zu müssen – also auch diejenigen, die gar nicht benutzt werden.

Praxisbeispiele

Ein Beispiel für das Design Pattern *Singleton* (*Singleton*) in **Smalltalk-80** [Par90] ist der Änderungssatz des Codes, sprich `changeSet current`. Ein subtileres Beispiel ist die Beziehung zwischen Klassen und deren **Metaklassen**. Eine Metaklasse ist die Klasse einer Klasse – und jede Metaklasse hat genau eine Instanz. Metaklassen sind nicht namentlich bezeichnet (außer indirekt über ihre einzigen Instanzen), verwalten jedoch ihre einzigen Instanzen selbst und erzeugen normalerweise auch keine

weiteren.

Das **InterViews User Interface Toolkit** [LCI+92] setzt das Design Pattern *Singleton* (*Singleton*) unter anderem für den Zugriff auf die jeweils einzigen Instanzen ihrer Klassen `Session` und `WidgetKit` ein. `Session` definiert die Hauptschleife der Anwendung für das Event Dispatching (Ereignisverteilung), speichert die Datenbank der stilistischen Usereinstellungen und verwaltet die Anbindungen an ein oder mehrere physische Anzeigegeräte. `WidgetKit` ist eine abstrakte Factory (siehe [Abschnitt 3.1](#)) zur Definition des Look-and-Feels der auf der Benutzeroberfläche verwendeten Widgets. Die Operation `WidgetKit::instance()` bestimmt die spezifische `WidgetKit`-Unterklasse, die anhand einer von `Session` definierten Umgebungsvariablen instanziert wird. Eine ähnliche Operation von `Session` legt fest, ob Schwarzweiß- oder Farbdisplays unterstützt werden und konfiguriert die einzige `Session`-Instanz entsprechend.

Verwandte Patterns

Das Design Pattern *Singleton* (*Singleton*) gestattet die Implementierung zahlreicher anderer Patterns, wie z. B. *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)), *Builder* (*Erbauer*, siehe [Abschnitt 3.2](#)) und *Prototype* (*Prototyp*, siehe [Abschnitt 3.4](#)).

3.6 Weitere Erläuterungen zu den Erzeugungsmustern

Im Allgemeinen gibt es zwei Möglichkeiten, ein System mit den von ihm generierten Objektklassen zu parametrisieren. Zum einen käme die Unterklassenbildung der Klasse infrage, die die Objekte erzeugt. Dies entspricht der Anwendung des Design Patterns *Factory Method* (*Fabrikmethode*, siehe [Abschnitt 3.3](#)). Ein großer Nachteil dieses Ansatzes ist allerdings, dass schon allein in dem Fall, dass nur die Klasse des Produkts modifiziert wird, eine neue Unterklasse angelegt werden muss – und solche Änderungen können sich nicht selten kaskadenförmig fortpflanzen: Wenn beispielsweise der `Creator` eines Produkts von einer *Factory Method* erstellt wird, muss auch dessen `Creator` überschrieben werden.

Die zweite Möglichkeit zur Parametrisierung eines Systems stützt sich eher auf die Objektkomposition: Es wird ein Objekt definiert, dessen Aufgabe darin besteht, die Klasse der Produktobjekte zu kennen, und das anschließend als Systemparameter verwendet wird. Diese Methodik zählt zu den Kernaspekten der Design Patterns

Abstract Factory (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)), *Builder* (*Erbauer*, siehe [Abschnitt 3.2](#)) und *Prototype* (*Prototyp*, siehe [Abschnitt 3.4](#)), die allesamt die Erzeugung eines neuen »Factory-Objekts« vorsehen, das wiederum für die Generierung der Produktobjekte zuständig ist: Im Design Pattern *Abstract Factory* (*Abstrakte Fabrik*) wird das Factory-Objekt zum Anlegen mehrerer Klassen genutzt. Das Pattern *Builder* (*Erbauer*) setzt dieses Objekt für die inkrementelle Erzeugung eines komplexen Produkts unter Anwendung eines ebenso komplexen Protokolls ein. Und im Pattern *Prototype* (*Prototyp*) generiert das Factory-Objekt Produkte, indem es Prototype-Objekte kopiert. In diesem Fall sind das Factory-Objekt und der Prototyp ein und dasselbe Objekt, weil der Prototyp für die Rückgabe des Produkts zuständig ist.

Erinnern Sie sich noch an das Musikeditor-Beispiel aus der Beschreibung des Design Patterns *Prototype* (*Prototyp*)? Hier stehen mehrere Möglichkeiten zur Verfügung, `GraphicTool` mit der Klasse des Produkts zu parametrisieren:

- Die Anwendung des Design Patterns *Factory Method* (*Fabrikmethode*) bewirkt, dass für jede Unterklasse der abstrakten Klasse `Graphic` in der Palette je eine Unterklasse von `GraphicTool` erzeugt wird. Außerdem verfügt `GraphicTool` über eine Operation `NewGraphic`, die von jeder `GraphicTool`-Unterklasse neu definiert wird.
- Die Anwendung des Design Patterns *Abstract Factory* (*Abstrakte Fabrik*) bewirkt die Errichtung einer Klassenhierarchie von `GraphicsFactory`-Objekten mit jeweils einem Objekt für jede `Graphic`-Unterklasse. Jede Factory bringt genau eine Produktart hervor: `CircleFactory` generiert Kreise, `LineFactory` erzeugt Linien usw. In diesem Fall wird `GraphicTool` zur Erzeugung der zugehörigen Grafikobjekte mit einer entsprechenden Factory parametrisiert.
- Die Anwendung des Design Patterns *Prototype* (*Prototyp*) bewirkt, dass jede Unterklasse von `Graphics` die `Clone`-Operation implementiert. Hier wird `GraphicTool` mit einem Prototyp der Grafikobjekte parametrisiert, die es erzeugt.

Welches Pattern individuell am besten geeignet ist, hängt von mehreren Faktoren ab. In dem Musikeditor-Beispiel ist zunächst das Pattern *Factory Method* (*Fabrikmethode*) am leichtesten anzuwenden: Die Definition einer neuen Unterklasse von `GraphicTool` ist einfach und die Instanzen von `GraphicTool` werden erst dann erzeugt, wenn die Palette bereits definiert wurde. Der Nachteil beim Einsatz dieses Patterns ist, dass sich die `GraphicTool`-Unterklassen immer weiter fortpflanzen, aber keine von ihnen wirklich viel ausrichtet.

Im Vergleich dazu bewirkt auch das Design Pattern *Abstract Factory* (*Abstrakte Fabrik*) keine allzu große Verbesserung, weil es eine umfangreiche *GraphicsFactory*-Klassenhierarchie erfordert. Deshalb wäre es dem Pattern *Factory Method* (*Fabrikmethode*) nur dann vorzuziehen, wenn ohnehin bereits eine *GraphicsFactory*-Klassenhierarchie vorhanden ist – entweder weil der Compiler sie automatisch zur Verfügung stellt (wie in Smalltalk oder Objective-C) oder weil sie in einem anderen Bereich des Systems benötigt wird.

Alles in allem ist das Design Pattern *Prototype* (*Prototyp*) somit vermutlich am besten für das Musikeditor-Framework geeignet, weil es lediglich die Implementierung einer `Clone`-Operation für jede *Graphics*-Klasse voraussetzt. Dadurch reduziert sich die Anzahl der Klassen, und außerdem kann die Operation `Clone` auch für andere Zwecke verwendet werden als nur zur reinen Instanziierung (z. B. zur Implementierung eines Menübefehls **DUPLIZIEREN**).

Das Design Pattern *Factory Method* (*Fabrikmethode*) gestaltet das Design insgesamt anpassbarer und nur unwesentlich komplexer. Während andere Design Patterns neue Klassen erfordern, wird hier lediglich eine neue Operation benötigt. Im Allgemeinen wird dieses Pattern häufig standardmäßig zur Erzeugung von Objekten verwendet, das ist allerdings nicht unbedingt nötig, sofern sich die zu instanzierende Klasse nie ändert oder die Instanziierung im Rahmen einer Operation stattfindet, die die Unterklassen leicht überschreiben können, wie z. B. als Initialisierungsoperation.

Als Fazit lässt sich abschließend feststellen, dass Designs, in denen die Design Patterns *Abstract Factory* (*Abstrakte Fabrik*), *Builder* (*Erbauer*) oder *Prototype* (*Prototyp*) eingesetzt werden, zwar flexibler als diejenigen sind, die das Pattern *Factory Method* (*Fabrikmethode*) nutzen, allerdings sind sie dann auch komplexer. Nicht selten verwenden Designs zunächst das Pattern *Factory Method* (*Fabrikmethode*) und schwenken dann später, d. h., sobald der Entwickler feststellt, dass mehr Flexibilität benötigt wird, auf die anderen Erzeugungsmuster um. Grundsätzlich gilt also: Je mehr Design Patterns der Entwickler kennt, desto mehr Auswahlmöglichkeiten stehen ihm zur Verfügung, um die individuellen Designkriterien nutzbringend gegeneinander abwägen zu können.

Kapitel 4: Strukturmuster (Structural Patterns)

Die **Strukturmuster (Structural Patterns)** dienen in erster Linie der Komposition von Klassen und Objekten zur Errichtung umfassender Strukturen. *Klassenbasierte* Strukturmuster machen sich das Prinzip der Vererbung zunutze, um verschiedenartige Schnittstellen oder Implementierungen zusammenzuführen. So lassen sich beispielsweise mithilfe der Mehrfachvererbung zwei oder mehr Klassen zu einer einzigen Klasse verschmelzen, die im Ergebnis die Eigenschaften der ursprünglich zugrundeliegenden Basisklassen vereint. Patterns dieser Art ermöglichen insbesondere das Zusammenwirken von unabhängig voneinander entwickelten Klassenbibliotheken.

Dieses Konzept wird auch in der klassenbasierten Variante des Design Patterns *Adapter* (*Adapter*, siehe [Abschnitt 4.1](#)) umgesetzt. Grundsätzlich passt ein Adapter eine Schnittstelle (die des zu adaptierenden Objekts) an eine andere an und stellt damit eine einheitliche Abstraktion verschiedener Schnittstellen zur Verfügung. Ein klassenbasierter Adapter bewerkstelligt diese Aufgabe durch das private Erben von einer zu adaptierenden Klasse, d. h., der Adapter passt seine Schnittstelle an die des adaptierten Objekts an.

Objektbasierte Strukturmuster beschäftigen sich hingegen weniger mit der Zusammenführung von Schnittstellen oder Implementierungen, sondern verfolgen vielmehr das Ziel, Objekte so miteinander zu kombinieren, dass eine neue Funktionalität entsteht. Diese Form der Objektkomposition gestattet ein hohes Maß an Flexibilität, weil das Kompositionsgefüge zur Laufzeit geändert werden kann, was bei einer statischen Klassenkomposition nicht möglich ist.

Das Design Pattern *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) beschreibt als objektbasiertes Muster beispielsweise die Errichtung einer Klassenhierarchie, die sowohl aus primitiven als auch zusammengesetzten Objekten besteht. Composite-Objekte gestatten die Komposition von primitiven sowie weiteren zusammengefügten Objekten zu beliebig komplexen Strukturen.

Im Fall des Design Patterns *Proxy* (*Proxy*, siehe [Abschnitt 4.7](#)) agiert ein Proxy als zweckmäßiger Ersatz oder Platzhalter für ein anderes Objekt. Er kann auf vielfältige Art und Weise verwendet werden: als lokaler Stellvertreter für ein Objekt in einem

Remote-Adressbereich, als Repräsentant eines großen Objekts, das nur auf Anforderung geladen wird, als Instrument zur Zugriffskontrolle für kritische Objekte u.v.m. Proxys stellen eine Dereferenzierungsebene für spezifische Objekteigenschaften zur Verfügung – und können diese Eigenschaften somit einschränken, erweitern oder modifizieren.

Das Design Pattern *Flyweight* (*Fliegengewicht*, siehe [Abschnitt 4.6](#)) definiert eine Struktur für die gemeinsame Nutzung von Objekten (engl. *Object Sharing*). Zumeist geschieht dies aus wenigstens zwei Gründen: Effizienz und Konsistenz. Dieses Pattern kommt vorrangig der Speicherplatzeffizienz zugute. In Anwendungen, in denen viele Objekte benutzt werden, muss sorgfältig auf die »Kostenintensität« jedes einzelnen Objekts geachtet werden. Alternativ zur Replikation lassen sich durch die gemeinsame Objektnutzung gegebenenfalls deutliche Einsparungen erzielen. Allerdings können Objekte nur dann gemeinsam genutzt werden, wenn sie keine kontextabhängigen Zustände definieren, und Flyweight-Objekte besitzen solche Zustände nicht: Sie erhalten sämtliche zur Erledigung ihrer jeweiligen Aufgaben erforderlichen Zusatzinformationen immer erst genau dann, wenn sie sie auch tatsächlich benötigen – und können daher beliebig gemeinsam genutzt werden.

Anders als das Design Pattern *Flyweight* (*Fliegengewicht*), das die Erstellung und Verwendung vieler kleiner Objekte unterstützt, zeigt das Pattern *Facade* (*Fassade*, siehe [Abschnitt 4.5](#)) auf, wie sich ein ganzes Subsystem durch ein einzelnes Objekt repräsentieren lässt. Ein Facade-Objekt steht stellvertretend für einen ganzen Objektsatz und hat die Aufgabe, erhaltene Meldungen und Requests an die diversen Objekte weiterzuleiten, die es repräsentiert.

Das Design Patterns *Bridge* (*Brücke*, siehe [Abschnitt 4.2](#)) separiert die Abstraktion eines Objekts von seiner Implementierung, so dass sie unabhängig voneinander variiert werden können.

Und das Design Pattern *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#)) beschreibt die dynamische Erweiterung der Objekte. Dieses Strukturmuster setzt die Objekte rekursiv zusammen und ermöglicht so eine unbegrenzte Anzahl neuer Zuständigkeiten bzw. neue Funktionalität. Beispielsweise kann ein Decorator-Objekt die in ihm enthaltene Schnittstellenkomponente mit Schmuckelementen wie Rahmen oder Schatten ausstatten oder ihr zusätzliche Funktionalität verleihen, etwa eine Scroll- oder Zoomfunktion. Durch die Verschachtelung der Decorator-Objekte können auch zwei oder beliebig mehr Schmuckelemente auf einmal dekoriert werden. Voraussetzung dafür ist allerdings, dass jedes der verwendeten Decorator-Objekte der Schnittstelle seiner Komponente entspricht und erhaltene Aufrufe an sie weiterleitet. Im Übrigen kann der Decorator seine jeweilige Aufgabe, beispielsweise

das Zeichnen eines Rahmens um die Komponente, entweder vor oder nach der Weiterleitung eines Aufrufs ausführen.

Viele Strukturmuster stehen in einer verwandtschaftlichen Beziehung zueinander. Die genauen diesbezüglichen Zusammenhänge werden am Ende dieses Kapitels ausgeführt.

4.1 Adapter (Adapter)

Klassen- oder objektbasiertes Strukturmuster

Zweck

Anpassung der Schnittstelle einer Klasse an ein anderes von den Clients erwartetes Interface. Das Design Pattern *Adapter (Adapter)* ermöglicht die Zusammenarbeit von Klassen, die ansonsten aufgrund der Inkompatibilität ihrer Schnittstellen nicht dazu in der Lage wären.

Auch bekannt als

Wrapper (Hüllenklasse)

Motivation

Mitunter ist die Nutzung einer grundsätzlich als wiederverwendbar entwickelten Toolkit-Klasse nur deshalb nicht möglich, weil sie nicht der domainspezifischen Schnittstelle entspricht, die eine Anwendung voraussetzt.

Im Folgenden wird das Design Pattern *Adapter (Adapter)* am Beispiel eines Zeicheneditors erläutert, der es den Usern gestattet, grafische Elemente wie Linien, Polygone, Text etc. zu zeichnen und zu Bildern und Diagrammen zusammenzusetzen. Das grafische Objekt, dessen Form editierbar ist und das sich selbst zeichnen kann, repräsentiert die zentrale Abstraktion des Editors. Die

Schnittstelle für grafische Objekte ist durch eine abstrakte Klasse namens Shape definiert. Und darüber hinaus legt der Editor für jede Objektart je eine Unterkategorie von Shape an, also beispielsweise LineShape für Linien, PolygonShape für Polygone usw.

Da ihre Zeichen- und Editierfähigkeiten naturgemäß beschränkt sind, ist die Implementierung von Klassen für elementare geometrische Formen, wie z. B. LineShape und PolygonShape, verhältnismäßig einfach. Im Fall einer Unterkategorie TextShape, die die Anzeige und Bearbeitung von Textelementen gestattet, ist dies allerdings schwieriger, weil schon einfache Textbearbeitungsaktivitäten komplexe Bildschirmaktualisierungen und ein Datenpuffermanagement erfordern.

Möglicherweise kann hier ein im Handel erhältliches gebrauchsfertiges Toolkit für Benutzeroberflächen weiterhelfen, das bereits von Haus aus eine ausgereifte TextView-Klasse für die Textanzeige und -bearbeitung bereitstellt, wobei die Klasse TextView im Idealfall für die Implementierung der TextShape-Klasse wiederverwendet werden kann. Leider werden Shape-Klassen in solchen Toolkits in der Regel jedoch nicht berücksichtigt – und dann sind die TextView- und Shape-Objekte auch nicht austauschbar.

Wie also können zwar bereits vorhandene, aber gänzlich unabhängige Klassen wie TextView in einer Anwendung benutzt werden, die Klassen mit einer anderen und noch dazu inkompatiblen Schnittstelle erwartet? Vielleicht wäre es möglich, die TextView-Klasse zu modifizieren und an die Shape-Schnittstelle anzupassen, dazu müsste allerdings zwingend der Quellcode des Toolkits zur Verfügung stehen. Und selbst dann wäre diese Lösung nicht wirklich empfehlenswert, denn: Ein Toolkit sollte sich keinesfalls an domainspezifischen Schnittstellen orientieren müssen, nur damit eine einzelne Anwendung funktioniert.

Stattdessen könnte die Klasse TextShape aber so definiert werden, dass sie die Schnittstelle von TextView an die Schnittstelle von Shape *adaptiert*. Machbar wäre das entweder durch Vererbung der Shape-Schnittstelle sowie der TextView-Implementierung oder durch die Komposition einer TextView-Instanz innerhalb eines TextShape-Objekts und dessen anschließende an die TextView-Schnittstelle angepasste Implementierung. Diese beiden Ansätze entsprechen den klassen- und objektbasierten Versionen des Design Patterns *Adapter (Adapter)* – hier fungiert TextShape als **Adapter**.

Abbildung 4.1 veranschaulicht am Beispiel eines Objektadapters, wie die in der Klasse Shape deklarierten BoundingBox-Requests in GetExtent-Requests konvertiert werden, die ihrerseits wiederum in TextView definiert sind. Da TextShape die Schnittstelle TextView an die Shape-Schnittstelle adaptiert, kann der Zeicheneditor

die ansonsten inkompatible TextView-Klasse nun wiederverwenden.

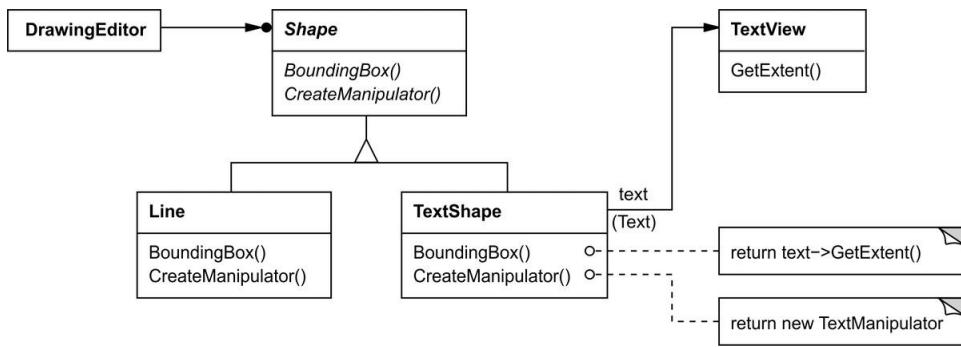


Abb. 4.1: Aufbau eines Objektadapters

Meist ist der Adapter dafür zuständig, die Funktionalität bereitzustellen, die die adaptierte Klasse nicht bietet. Wie er dies bewerkstelligen kann, ist ebenfalls aus der schematischen Darstellung in [Abbildung 4.1](#) ersichtlich: Der User sollte jedes Shape-Objekt interaktiv an einen neuen Standort bewegen können – allerdings ist die TextView-Klasse nicht dafür geeignet. Deshalb kann zur Ergänzung der fehlenden Funktionalität stattdessen die CreateManipulator-Operation von Shape implementiert werden, die eine Instanz der zugehörigen Manipulator-Unterklasse zurückgibt.

Manipulator ist eine abstrakte Klasse für Objekte, die die Reaktion eines Shape-Objekts auf eine Usereingabe, wie z. B. das Ziehen des Objekts an eine neue Position, passend animieren kann. Darüber hinaus existieren auch zugehörige Manipulator-Unterklassen für die verschiedenen Objektformen, beispielsweise ist TextManipulator die entsprechende Unterklasse für TextShape. Durch die Rückgabe einer TextManipulator-Instanz ergänzt TextShape die Funktionalität, die in TextView fehlt, aber von Shape benötigt wird.

Anwendbarkeit

Die Verwendung des Design Patterns *Adapter (Adapter)* ist in folgenden Fällen hilfreich:

- Wenn eine existierende Klasse genutzt werden soll, deren Schnittstelle den aktuellen Anforderungen nicht genügt.
- Wenn eine wiederverwendbare Klasse erzeugt werden soll, die mit voneinander unabhängigen und nicht vorhersehbaren Klassen zusammenarbeitet, also

Klassen, deren Schnittstellen gegebenenfalls nicht kompatibel sind.

- (*Nur Objektadapter*) Wenn mehrere bereits vorhandene Unterklassen verwendet werden sollen, es aber unpraktisch wäre, deren individuelle Schnittstellen durch Unterklassenbildung anzupassen. Hier schafft ein Objektadapter Abhilfe, indem er die Schnittstelle seiner Basisklasse adaptiert.

Struktur

Ein Klassenadapter greift zur Anpassung zweier verschiedener Schnittstellen auf das Prinzip der Mehrfachvererbung zurück:

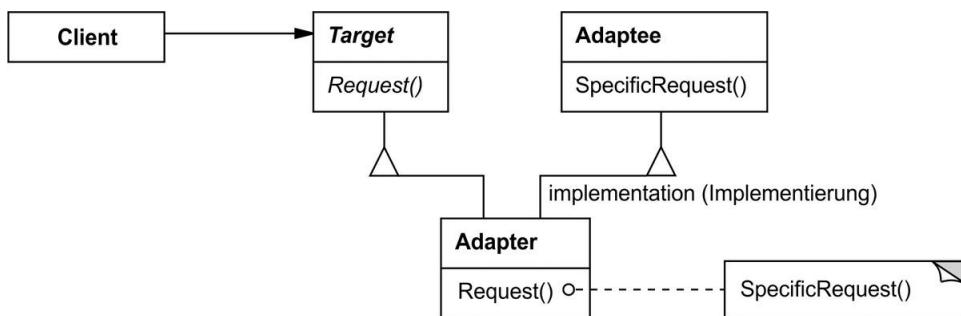


Abb. 4.2: Mehrfachvererbung beim Klassenadapter

Ein Objektadapter bedient sich hingegen der Objektkomposition:

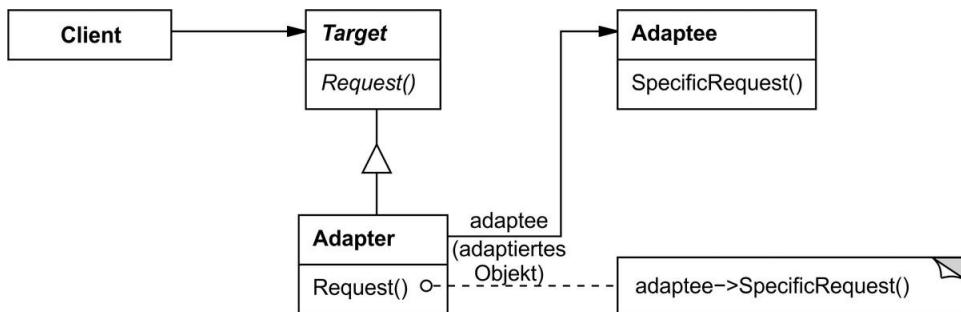


Abb. 4.3: Objektkomposition beim Objektadapter

Teilnehmer

- **Target (Shape)**
 - Definiert die vom Client verwendete domainspezifische Schnittstelle.

- **Client** (`DrawingEditor`)
 - Arbeitet mit Objekten zusammen, die der `Target`-Schnittstelle entsprechen.
- **Adaptee** (`TextView`)
 - Definiert eine existierende Schnittstelle, die adaptiert werden muss.
- **Adapter** (`TextShape`)
 - Adaptiert die Adaptee-Schnittstelle an die `Target`-Schnittstelle.

Interaktionen

- Die Clients rufen die Operationen auf einer Adapter-Instanz auf, und der Adapter ruft die Adaptee-Operationen auf, die den Request ausführen.

Konsequenzen

Klassen- und Objektadapter haben individuelle Vor- und Nachteile.

Ein Klassenadapter

- passt das zu adaptierende Objekt durch die Zuweisung einer konkreten `Adaptee`-Klasse an die `Target`-Schnittstelle an. Im Umkehrschluss bedeutet dies, dass ein Klassenadapter nicht funktioniert, wenn eine Klasse mitsamt all ihren Unterklassen adaptiert werden soll.
- gestattet dem Adapter, das Verhalten des adaptierten Objekts zu überschreiben, weil Adapter eine Unterklasse von `Adaptee` ist.
- benutzt lediglich ein einziges Objekt. Es ist keine zusätzliche Pointer-Dereferenzierung erforderlich, um zu dem adaptierten Objekt zu gelangen.

Ein Objektadapter

- ermöglicht einem einzelnen Adapter die Zusammenarbeit mit mehreren adaptierten Objekten, d. h. sowohl mit dem `Adaptee`-Objekt als auch, sofern vorhanden, mit dessen Unterklassen. Darüber hinaus kann der Adapter auch

alle adaptierten Objekte gleichzeitig um neue Funktionalität erweitern.

- erschwert das Überschreiben des Verhaltens des Adaptee-Objekts. In diesem Fall muss eine Unterklasse von dem Adaptee-Objekt gebildet werden und der Adapter muss dann statt des adaptierten Objekts diese Unterklasse referenzieren.

Darüber hinaus bietet das Design Pattern *Adapter* (*Adapter*) außerdem folgende Möglichkeiten:

1. *Anpassungsaufwand beim Einsatz eines Adapters.* Adapter können bei der Anpassung einer Adaptee- an eine Target-Schnittstelle in unterschiedlichem Umfang tätig werden. Der zu diesem Zweck zu betreibende Aufwand reicht von einer einfachen Schnittstellenkonvertierung – beispielsweise durch die Änderung der Operationsnamen – bis hin zur Unterstützung vollkommen unterschiedlicher Operationssätze. In welchem Ausmaß die Anpassungsmaßnahmen des Adapters stattfinden, ist davon abhängig, inwieweit sich die beiden Schnittstellen unterscheiden bzw. wie viele Gemeinsamkeiten sie aufweisen.
2. *Steckbare Adapter.* Die Wiederverwendung einer Klasse ist deutlich einfacher, wenn die darauf zurückgreifenden Klassen möglichst wenige Annahmen machen müssen. Um der Annahme, dass alle zugreifenden Klassen derselben Schnittstelle entsprechen müssen, von vornherein zu begegnen, ist es sinnvoll, die Schnittstellenadaptierung direkt in die wiederzuverwendende Klasse zu integrieren. Oder anders ausgedrückt: Die vorausschauende Berücksichtigung der Schnittstellenadaptierung gewährleistet die reibungslose Integration einer Klasse in existierende Systeme, die möglicherweise andere Schnittstellen erwarten. In ObjectWorks\Smalltalk [Par90] werden Klassen mit integrierter Schnittstellenadaptierung als **steckbare Adapter (Pluggable Adapter)** bezeichnet.

Für ein spezialisiertes Widget `TreeDisplay`, das Baumstrukturen grafisch darstellen kann und nur in einer einzigen Anwendung nutzbar ist, würde dies beispielsweise bedeuten, dass alle von ihm dargestellten Objekte gegebenenfalls einer bestimmten Schnittstelle entsprechen und somit von einer abstrakten `Tree`-Klasse abgeleitet sein müssten. Eine solche Vorbedingung wäre für eine Wiederverwendung von `TreeDisplay` (z. B. im Rahmen eines Widget-Toolkits) allerdings unsinnig, denn: Anwendungen definieren ohnehin eigene Klassen für Baumstrukturen und sollten nicht gezwungen sein, eine bestimmte Klasse, in diesem Fall besagte abstrakte `Tree`-Klasse, zu verwenden.

Unterschiedliche Baumstrukturen enthalten in aller Regel immer auch verschiedene Schnittstellen.

So könnte beispielsweise in einer Verzeichnishierarchie mittels einer `GetSubdirectories`-Operation auf die Kindobjekte zugegriffen werden, während die entsprechende Operation in einer Vererbungshierarchie `GetSubclasses` heißt. Ein wiederverwendbares `TreeDisplay`-Widget muss also selbst dann in der Lage sein, beide Hierarchiearten darzustellen, wenn sie unterschiedliche Schnittstellen verwenden – und deshalb sollte das `TreeDisplay`-Widget über eine integrierte Schnittstellenadaptierung verfügen.

Wie die Integration einer Schnittstellenadaptierung in die Klassen realisiert werden kann, ist im Abschnitt »Implementierung« beschrieben.

3. *Transparenz mittels 2-Wege-Adapter*. Ein potenzielles Problem bei der Nutzung von Adapters ist, dass sie nicht allen Clients gleichermaßen hinreichende Transparenz bieten. Ein adaptiertes Objekt ist nicht mehr mit der `Adaptee`-Schnittstelle konform und kann aus diesem Grund auch nicht mehr so eingesetzt werden wie das ursprünglich adaptierte Objekt – und diesen Umstand stellen **2-Wege-Adapter** transparent dar. Sie sind insbesondere dann geeignet, wenn zwei verschiedene Clients unterschiedliche Sichtweisen auf ein und dasselbe Objekt haben.

Der nachfolgend beschriebene 2-Wege-Adapter integriert sowohl das **Unidraw Application Framework für grafische Editoren** [VL90] als auch den Constraint-Löser **QOCA** (ein Toolkit zum Lösen zwingender Bedingungen und Einschränkungen) [HHMV92].

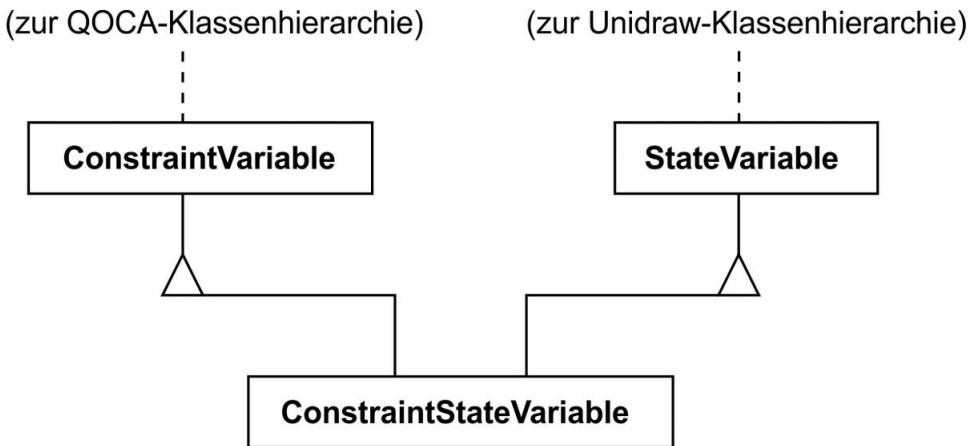


Abb. 4.4: QOCA- und Unidraw-Klassen für explizite Variablen

Beide Systeme enthalten Klassen, die explizit Variablen repräsentieren:

Unidraw verfügt über die Klasse `StateVariable` und QOCA besitzt die Klasse `ConstraintVariable`. Damit Unidraw mit QOCA zusammenarbeiten kann, muss die Klasse `ConstraintVariable` an die Klasse `StateVariable` angepasst werden – ebenso muss aber auch `StateVariable` an `ConstraintVariable` adaptiert werden, damit QOCA seinerseits Lösungen an Unidraw weiterreichen kann.

Hier wird dies durch den Einsatz eines 2-Wege-Klassenadapters namens `ConstraintStateVariable` erreicht, einer Unterklasse sowohl von `StateVariable` als auch von `ConstraintVariable`, die beide Schnittstellen aneinander anpasst. In diesem Fall stellt die Mehrfachvererbung eine tragfähige Lösung dar, weil die Schnittstellen der adaptierten Klassen gravierende Unterschiede aufweisen. Der 2-Wege-Adapter ist für beide adaptierte Klassen geeignet und kann somit auch in beiden Systemen verwendet werden.

Implementierung

Im Allgemeinen ist die Implementierung eines Adapters recht unkompliziert, dennoch sollten die folgenden Aspekte nicht außer Acht gelassen werden:

1. *Implementierung von Klassenadapters in C++*. Bei einer C++-Implementierung eines Klassenadapters würde die Adapter-Klasse von der Target-Schnittstelle öffentlich erben (engl. *Public Inheritance*), von der Adaptee-Schnittstelle hingegen privat (engl. *Private Inheritance*). Somit wäre die Adapter-Klasse also ein Subtyp der Target-Klasse, aber kein Subtyp der Adaptee-Klasse.
2. *Steckbare Adapter*. Die Implementierung von steckbaren Adapters lässt sich auf drei verschiedene Arten realisieren, die an dieser Stelle anhand des zuvor bereits erwähnten `TreeDisplay`-Beispiels demonstriert werden. Dieses Widget besitzt die Fähigkeit, eine hierarchische Struktur automatisch zu gestalten und darzustellen.

Der erste Schritt, den alle drei hier vorgestellten Implementierungsmöglichkeiten gemeinsam haben, besteht darin, eine »schmale« Schnittstelle für die Adaptee-Klasse zu finden – also die kleinste gemeinsame Menge der Operationen, die für die Adaptierung benötigt werden. Eine schmale Schnittstelle, die lediglich aus einigen wenigen Operationen besteht, ist leichter zu adaptieren als eine Schnittstelle mit Dutzenden von Operationen. Im Fall von `TreeDisplay` soll eine beliebige hierarchische

Struktur adaptiert werden. Eine minimalistische Schnittstelle könnte aus zwei Operationen bestehen: einer zur Definition der grafischen Darstellung eines Knotens in der hierarchischen Struktur und einer weiteren, die die Kindobjekte des Knotens zurückliefert.

Die schmale Schnittstelle gestattet drei Implementierungsansätze:

- Verwendung abstrakter Operationen.* Definiert in der Klasse `TreeDisplay` der schmalen Adaptee-Schnittstelle entsprechende abstrakte Operationen. Unterklassen müssen diese Operationen dann implementieren und das hierarchisch strukturierte Objekt adaptieren. Eine Unterklasse `DirectoryTreeDisplay` würde sie beispielsweise per Zugriff auf die Verzeichnisstruktur implementieren (siehe [Abbildung 4.5](#)).

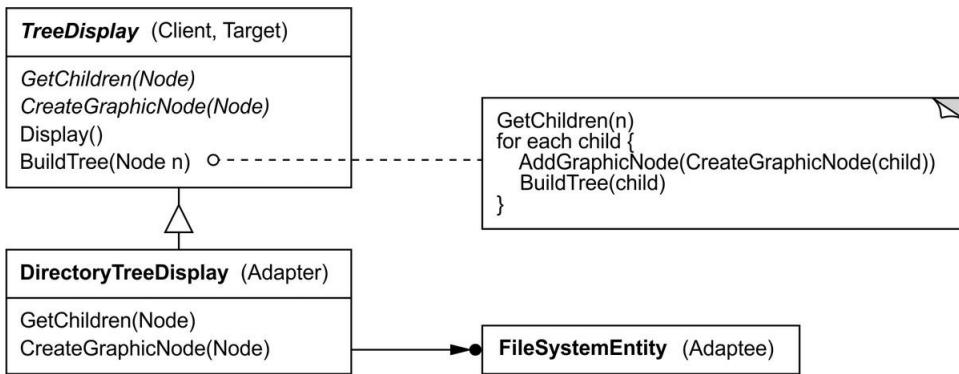


Abb. 4.5: Implementierung mithilfe von abstrakten Operationen

Die Klasse `DirectoryTreeDisplay` spezialisiert die schmale Schnittstelle so, dass sie aus `FileSystemEntity`-Objekten erstellte Verzeichnisstrukturen darstellen kann.

- Verwendung von Delegationsobjekten.* Bei diesem Ansatz leitet die Klasse `TreeDisplay` Requests zum Zugriff auf die hierarchische Struktur an ein **Delegationsobjekt** (auch **Delegat**, engl. *Delegate*) weiter. Zur Anwendung einer anderen Adoptionsstrategie kann sie das Delegationsobjekt einfach durch ein anderes austauschen.

Angenommen, es gäbe eine Klasse `DirectoryBrowser`, die von `TreeDisplay` genutzt wird. Diese Klasse könnte ein sinnvolles Delegationsobjekt für die Adaptierung von `TreeDisplay` an die hierarchische Verzeichnisstruktur darstellen. In dynamisch typisierten Programmiersprachen wie Smalltalk oder Objective-C würde dieser Ansatz lediglich eine Schnittstelle zur Registrierung des Delegaten am

Adapter erfordern, und anschließend würde `TreeDisplay` die Requests einfach an ihn weiterleiten. Das Betriebssystem **NeXT Step** [Add94] setzt dieses Verfahren zur Reduzierung der Unterklassenbildung ein.

Statisch typisierte Sprachen wie C++ erfordern dagegen eine explizite Schnittstellendefinition für das Delegationsobjekt. Zur Spezifizierung einer solchen Schnittstelle kann die von `TreeDisplay` benötigte schmale Schnittstelle in einer abstrakten Klasse `TreeAccessorDelegate` angelegt werden. Danach kann diese Schnittstelle mittels Vererbung mit einem gewünschten Delegationsobjekt zusammengefügt werden – in diesem Fall `DirectoryBrowser`. Sofern für die Klasse `DirectoryBrowser` keine Basisklasse existiert, kann dabei die einfache Vererbung genutzt werden, ansonsten die Mehrfachvererbung. Eine derartige Zusammenführung von Klassen ist einfacher zu bewerkstelligen, als eine neue Unterklasse `TreeDisplay` anzulegen und deren Operationen einzeln zu implementieren.

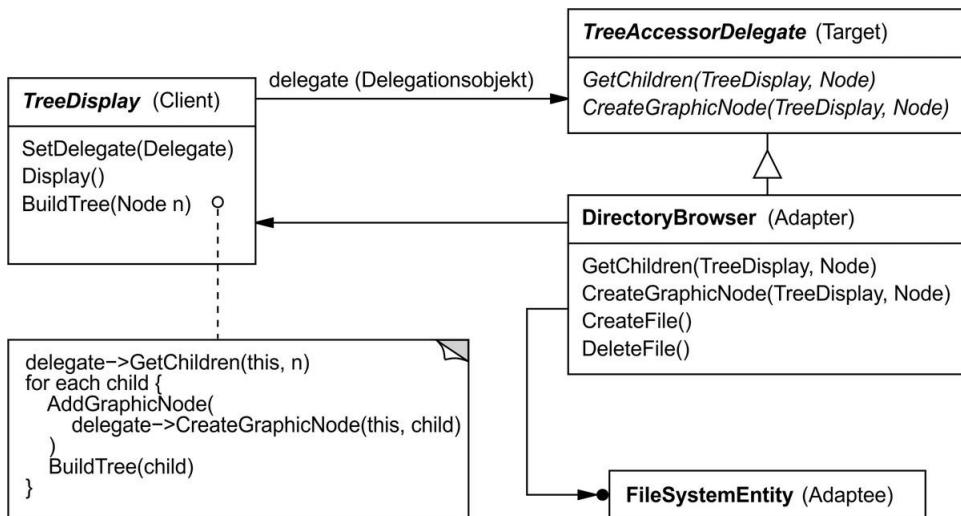


Abb. 4.6: Implementierung eines steckbaren Adapters mithilfe von Delegationsobjekten

- c. *Parametrisierte Adapter.* Üblicherweise werden steckbare Adapter in Smalltalk durch die Parametrisierung eines Adapters mit einem oder mehreren Blöcken unterstützt. Das Blockkonzept begünstigt die Adaptierung ohne Unterklassenbildung. Ein Block kann einen Request annehmen und der Adapter kann wiederum einen Block für jeden einzelnen Request speichern. Für das hier vorliegende Beispiel bedeutet das, dass `TreeDisplay` einen Block für die Konvertierung eines Knotens in ein Objekt `GraphicNode` speichert und einen weiteren Block für den

Zugriff auf die Kindobjekte des Knotens.

Zur Erzeugung einer Klasse `TreeDisplay` für eine Verzeichnishierarchie könnte man z. B. folgenden Code verwenden:

```
directoryDisplay :=
    (TreeDisplay on: treeRoot)
        getChildrenBlock:
            [:node | node getSubdirectories]
        createGraphicNodeBlock:
            [:node | node createGraphicNode].
```

Dieser Ansatz bietet im Hinblick auf die Integration einer Schnittstellenadaptierung in eine Klasse eine komfortable Alternative zur Unterklassenbildung.

Beispielcode

Im Folgenden wird die Implementierung von Klassen- und Objektadapters in Anlehnung an das im Abschnitt »Motivation« verwendete Beispiel zunächst mit den Klassen `Shape` und `TextView` skizziert:

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

Die Klasse `Shape` nimmt hier einen Begrenzungsrahmen (`BoundingBox`) an, der durch seine gegenüberliegenden Ecken definiert ist. Im Gegensatz dazu ist `TextView` durch Ursprungskoordinaten sowie Höhen- und Breitenangabe definiert. Weiterhin definiert `Shape` auch eine Operation `CreateManipulator` zur Erzeugung eines `Manipulator`-Objekts (siehe auch Design Pattern *Factory Method (Fabrikmethode)*, [Abschnitt 3.3](#), »Konsequenzen«), das das `Shape`-Objekt im Fall einer userseitigen

Manipulation entsprechend animieren kann. `TextView` besitzt keine vergleichbare Operation. Die Klasse `TextShape` fungiert als Adapter zwischen diesen beiden Schnittstellen.

Ein Klassenadapter wendet zur Adaptierung von Schnittstellen das Prinzip der Mehrfachvererbung an. Der springende Punkt besteht bei Klassenadapters darin, einen Vererbungszweig zum Erben der Schnittstelle und einen weiteren zum Erben der Implementierung zu verwenden. In C++ wird diese Unterscheidung üblicherweise dadurch realisiert, dass die Schnittstelle öffentlich erbt und die Implementierung privat. Diese Konvention wird nun auch auf die Definition des `TextShape`-Adapters angewendet:

```
class TextShape : public Shape, private TextView {  
public:  
    TextShape();  
  
    virtual void BoundingBox(  
        Point& bottomLeft, Point& topRight  
    ) const;  
    virtual bool IsEmpty() const;  
    virtual Manipulator* CreateManipulator() const;  
};
```

Die Operation `BoundingBox` konvertiert die Schnittstelle von `TextView` so, dass sie der Schnittstelle von `Shape` entspricht:

```
void TextShape::BoundingBox (  
    Point& bottomLeft, Point& topRight  
) const {  
    Coord bottom, left, width, height;  
  
    GetOrigin(bottom, left);  
    GetExtent(width, height);  
  
    bottomLeft = Point(bottom, left);  
    topRight = Point(bottom + height, left + width);  
}
```

Die Operation `IsEmpty` demonstriert die in Adapterimplementierungen übliche direkte Weiterleitung von Requests:

```
bool TextShape::IsEmpty () const {  
    return TextView::IsEmpty();  
}
```

Und zum Schluss wird noch die Operation `CreateManipulator` (die nicht von `TextView` unterstützt wird) von Grund auf neu definiert. In diesem Fall wird eine

bereits implementierte Klasse `TextManipulator` vorausgesetzt, die die Manipulation eines `TextShape`-Objekts unterstützt:

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

Der Objektkomposition bedient sich dagegen der Objektkomposition, um Klassen mit unterschiedlichen Schnittstellen zu kombinieren. Bei diesem Ansatz verwaltet der Adapter `TextShape` einen Pointer auf `TextView`:

```
class TextShape : public Shape {
public:
    TextShape(TextView* );
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;

private:
    TextView* _text;
};
```

Der `TextShape`-Adapter muss den Pointer auf die `TextView`-Instanz initialisieren und erledigt dies im Konstruktor. Darüber hinaus muss er bei jedem Aufruf seiner eigenen Operationen außerdem die Operationen seines `TextView`-Objekts aufrufen. In diesem Beispiel wird davon ausgegangen, dass das `TextView`-Objekt vom Client erzeugt und an den `TextShape`-Konstruktor übergeben wird:

```
TextShape::TextShape (TextView* t) {
    _text = t;
}

void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent (width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
```

}

Weil sie von Grund auf neu implementiert wurde und keine TextView-Funktionalität wiederverwendet, unterscheidet sich die Implementierung von CreateManipulator nicht von der im Klassenadapter:

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

Vergleicht man diesen Code mit dem des Klassenadapters, stellt man fest, dass das Schreiben des Objektadapters zwar etwas mehr Aufwand erfordert, er dafür aber auch flexibler ist. So funktioniert die im Objektadapter verwendete TextShape-Version ebenso gut mit Unterklassen von TextView – der Client übergibt einfach eine Instanz einer TextView-Unterklasse an den TextShape-Konstruktor.

Praxisbeispiele

Das im Abschnitt »Motivation« angeführte Beispiel stammt aus **ET++Draw**, einer auf ET++ basierenden Zeichenanwendung [WGM88], die die ET++-Klassen für die Textbearbeitung mithilfe einer TextShape-Adapterklasse wiederverwendet.

InterViews 2.6 definiert eine abstrakte Klasse **Interactor** für die Steuerungselemente auf der Benutzeroberfläche wie Scrollleisten, Schaltflächen und Menüs [VL88]. Darüber hinaus definiert dieses Toolkit auch eine abstrakte Klasse **Graphic** für strukturierte grafische Objekte wie Linien, Kreise, Polygone und Spline-Kurven. Sowohl die **Interactor**- als auch die **Graphics**-Objekte werden in grafischer Form dargestellt, besitzen jedoch unterschiedliche Schnittstellen und Implementierungen (sie haben keine gemeinsame Basisklasse) und sind daher nicht kompatibel – d. h., ein strukturiertes Grafikobjekt kann beispielsweise nicht direkt in eine Dialogbox eingebettet werden.

Stattdessen definiert InterViews 2.6 einen Objektadapter namens **GraphicBlock**, eine Unterklasse von **Interactor**, die eine **Graphic**-Instanz enthält. Dieser Objektadapter adaptiert die Schnittstelle der **Graphic**-Klasse an die von **Interactor** und ermöglicht so die Darstellung, das Scrollen und das Zoomen der **Graphic**-Instanz innerhalb einer **Interactor**-Struktur.

Steckbare Adapter sind in **ObjectWorks\Smalltalk** recht häufig anzutreffen [Par90]. **Standard Smalltalk** definiert eine Klasse **ValueModel** für Ansichten (engl. *Views*), die einen einzelnen Wert anzeigen. Diese Klasse definiert wiederum eine aus **value**

und `value:` bestehende Schnittstelle für den Zugriff auf den Wert. Hierbei handelt es sich um abstrakte Methoden. Die Anwendungsentwickler greifen eher mit domainspezifischeren Bezeichnungen wie `width` und `width:` auf den Wert zu, sollten jedoch keine Unterklass von `ValueModel` bilden müssen, um solche anwendungsspezifischen Namen für die `ValueModel`-Schnittstelle zu adaptieren.

Vielmehr enthält ObjectWorks\Smalltalk bereits eine Unterkasse von `ValueModel`, die mit `PluggableAdaptor` bezeichnet ist. Ein `PluggableAdaptor`-Objekt adaptiert andere Objekte an die `ValueModel`-Schnittstelle (`value/value:`) und kann mit Blöcken zum Abrufen und Setzen des gewünschten Wertes parametrisiert werden. Diese Blöcke werden von der Klasse `PluggableAdaptor` intern zur Implementierung der `value/value:`-Schnittstelle verwendet. Darüber hinaus ermöglicht `PluggableAdaptor` auch die direkte und syntaktisch bequeme Hereingabe von Selektorbezeichnern (z. B. `width/width:`) und konvertiert diese Selektoren dann automatisch in die entsprechenden Blöcke.

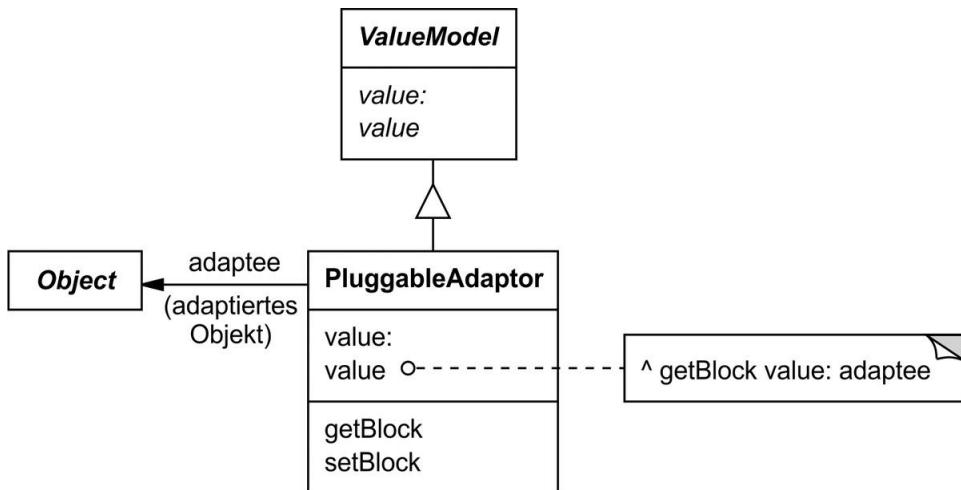


Abb. 4.7: Struktur der Unterkasse `ValueModel` in ObjectWorks\Smalltalk

Ein weiteres Beispiel aus **ObjectWorks\Smalltalks** ist auch die Klasse `TableAdaptor`. Ein `TableAdaptor`-Objekt kann eine Abfolge von Objekten in Form einer tabellarischen Darstellung adaptieren, wobei die Tabelle ein Objekt pro Zeile anzeigt. Der Client parametrisiert `TableAdaptor` mit einem Meldungssatz, den die Tabelle nutzen kann, um die Spaltenwerte von einem Objekt abzurufen.

Einige Klassen im **AppKit-Framework** des Unternehmens NeXT [Add94] realisieren die Schnittstellenadaptierung mithilfe von Delegationsobjekten. Ein Beispiel hierfür ist die Klasse `NXBrowser`, die in der Lage ist, hierarchisch strukturierte Datenlisten darzustellen. Diese Klasse verwendet sowohl für den Zugriff auf die Daten als auch für deren Adaptierung ein Delegationsobjekt.

Auch Meyers »**Marriage of Convenience**« [Mey88] stellt eine Form von Klassenadapter dar. Meyer beschreibt hier, wie die `FixedStack`-Klasse die Implementierung einer `Array`-Klasse an die Schnittstelle einer `Stack`-Klasse adaptiert. Das Resultat ist ein Stack mit einer festgelegten Anzahl von Einträgen.

Verwandte Patterns

Das Design Pattern *Bridge* (*Brücke*, siehe [Abschnitt 4.2](#)) besitzt eine ähnliche Struktur wie ein Objektadapter, dient jedoch einem anderen Zweck: Es trennt eine Schnittstelle von ihrer Implementierung, so dass beide reibungslos unabhängig voneinander variiert werden können. Ein Adapter soll hingegen die Schnittstelle eines *existierenden* Objekts ändern.

Auch das Design Pattern *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#)) ist artverwandt. Es erweitert ein anderes Objekt, ohne dessen Schnittstelle zu modifizieren. Aus Sicht der Anwendung ist dieses Pattern damit transparenter als ein Adapter und unterstützt darüber hinaus auch die rekursive Komposition, was bei reinen Adapters nicht möglich ist.

Das Design Pattern *Proxy* (*Proxy*, siehe [Abschnitt 4.7](#)) definiert einen Repräsentanten oder Stellvertreter für ein anderes Objekt, ohne jedoch dessen Schnittstelle zu verändern.

4.2 Bridge (Brücke)

Objektbasiertes Strukturmuster

Zweck

Entkopplung einer Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.

Auch bekannt als

Motivation

Wenn mehrere Implementierungen einer Abstraktion möglich sind, werden diese in aller Regel durch Vererbung realisiert. Eine abstrakte Klasse definiert die Schnittstelle zur Abstraktion und ihre konkreten Unterklassen implementieren sie dann auf verschiedene Arten. Dieser Ansatz ist jedoch nicht immer flexibel genug: Da die Vererbung eine Implementierung dauerhaft an die Abstraktion bindet, ist es schwierig, die Abstraktionen und ihre Implementierungen unabhängig voneinander zu modifizieren, zu erweitern und wiederzuverwenden.

In dem folgenden Beispiel wird die Implementierung einer portablen Fensterabstraktion in einem Toolkit für Benutzeroberflächen verwendet. Diese Abstraktion soll es dem User ermöglichen, Anwendungen zu entwickeln, die sowohl im X-Window-System als auch im Presentation Manager (PM) eingesetzt werden können. Nun könnten mittels Vererbung eine abstrakte Klasse `Window` sowie die Unterklassen `XWindow` und `PMWindow` erstellt werden, die die `window`-Schnittstelle für verschiedene Plattformen implementieren. Hierbei ergeben sich allerdings zwei Nachteile:

1. Die Erweiterung der Fensterabstraktion zur Berücksichtigung bzw. Einbeziehung verschiedener Fensterarten oder neuer Plattformen ist unkomfortabel und aufwendig – wie am Beispiel der Unterklasse `IconWindow` von `Window` deutlich wird, die die Fensterabstraktion auf die Verkleinerung auf Symbolgröße spezialisiert: Um die Unterstützung beider Plattformen zu gewährleisten, müssten zwei neue Klassen, `XIconWindow` und `PMIconWindow`, implementiert werden. Damit aber noch nicht genug: Es müssen für *jede* Fenstervariante jeweils zwei neue Klassen definiert werden. Und wenn dann noch eine dritte Plattform unterstützt werden soll, muss außerdem eine weitere `Window`-Unterklass für jede Fenstervariante angelegt werden.

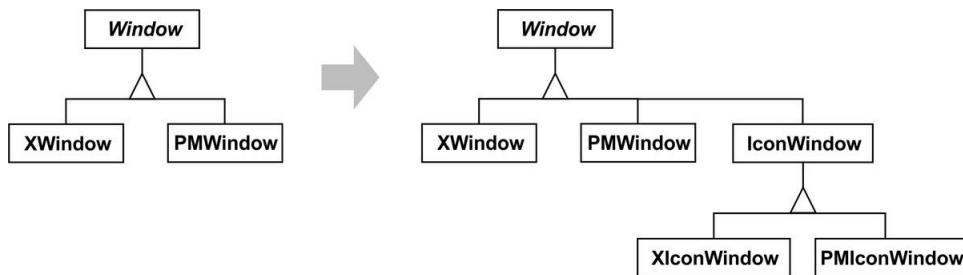


Abb. 4.8: Erweiterung der `window`-Abstraktion zur Unterstützung mehrerer

Plattformen

2. Darüber hinaus hat dieser Ansatz auch zur Folge, dass der Clientcode plattformabhängig wird. Wann immer ein Client ein Fenster erzeugt, instanziert er eine konkrete Klasse, die eine spezifische Implementierung besitzt. So wird die Fensterabstraktion beispielsweise durch die Erzeugung eines xwindow-Objekts an die X-Window-Implementierung gebunden – und damit wird auch der clientseitig auszuführende Code von dieser Implementierung abhängig. Das erschwert jedoch die Portierung des Clientcodes auf andere Plattformen.

Grundsätzlich sollte nur die Fensterimplementierung von der Plattform abhängig sein, auf der die Anwendung läuft. Die Clients sollten hingegen Fenster erzeugen können, ohne auf eine bestimmte Implementierung festgelegt zu sein – und dementsprechend sollte die Instanzierung von Fenstern durch den Clientcode immer ohne die Nennung spezifischer Plattformen erfolgen.

Das Design Pattern *Bridge (Brücke)* begegnet diesen Widrigkeiten durch die Unterbringung der Window-Abstraktion und deren Implementierung in getrennten Klassenhierarchien: Eine Klassenhierarchie enthält die Fensterschnittstellen (`Window`, `IconWindow`, `TransientWindow`) und eine weitere, separate Hierarchie beherbergt die plattformspezifischen Fensterimplementierungen mit `WindowImp` als Root-Klasse. In diesem Fall stellt die Unterklasse `XwindowImp` beispielsweise eine auf dem X-Window-System basierende Implementierung bereit:

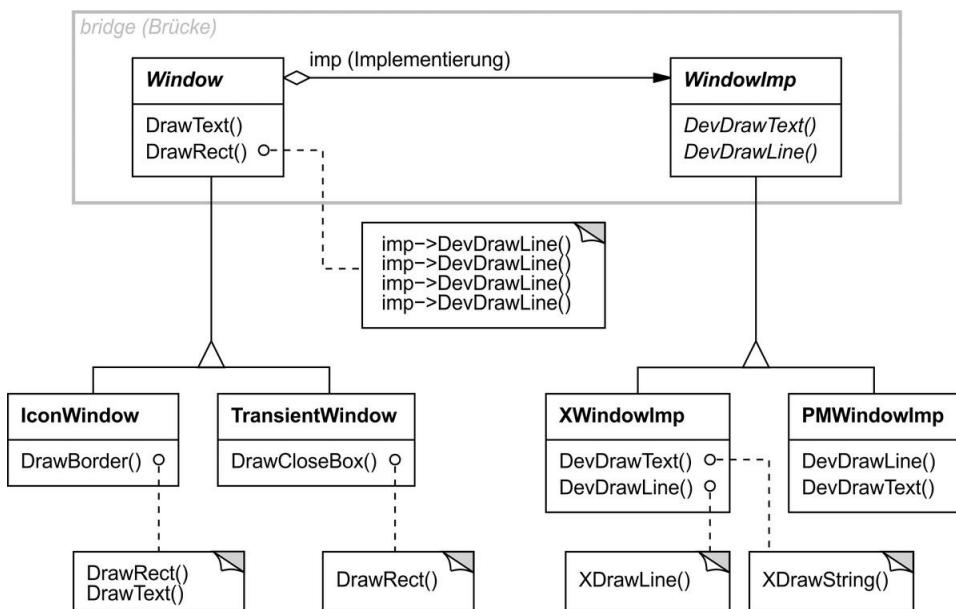


Abb. 4.9: Separate Klassenhierarchien für die window-Abstraktion und deren Implementierung

Alle auf die `window`-Unterklassen auszuführenden Operationen sind in Anlehnung an die abstrakten Operationen der `WindowImp`-Schnittstelle implementiert. Dadurch werden die Fensterabstraktionen von den verschiedenen plattformspezifischen Implementierungen abgekoppelt. Diese Beziehung zwischen `window` und `WindowImp` wird als **Bridge (Brücke)** bezeichnet, weil die Abstraktion und ihre Implementierung in einer Form miteinander verbunden werden, die es erlaubt, sie unabhängig voneinander zu variieren.

Anwendbarkeit

Die Verwendung des Design Patterns *Bridge (Brücke)* bietet sich unter folgenden Voraussetzungen an:

- Wenn eine permanente Bindung zwischen einer Abstraktion und ihrer Implementierung verhindert werden soll. Das kann z. B. dann gewünscht sein, wenn die Auswahl oder der Wechsel der Implementierung zur Laufzeit möglich sein soll.
- Wenn sowohl die Abstraktionen als auch ihre Implementierungen durch Unterklassenbildung erweiterbar sein sollen. Hier gewährleistet das Design Pattern *Bridge (Brücke)*, dass verschiedene Abstraktionen und Implementierungen kombiniert und unabhängig voneinander erweitert werden können.
- Wenn die an der Implementierung einer Abstraktion vorgenommenen Modifizierungen keine Auswirkungen auf die Clients haben sollen – d. h., der Clientcode soll nicht neu kompiliert werden müssen.
- (*Gilt für C++*) Wenn die Implementierung einer Abstraktion vollständig vor den Clients verborgen bleiben soll. In C++ ist eine Klasse in der Klassenschnittstelle sichtbar.
- Wenn ein starker zahlenmäßiger Anstieg der Klassen eintritt, wie in [Abbildung 4.8](#) dargestellt. Eine derart geprägte Klassenhierarchie deutet auf die Notwendigkeit hin, dass ein Objekt in zwei Teile aufgespalten werden sollte. Rumbaugh bezeichnet solche Klassenhierarchien als »verschachtelte Generalisierungen« (»*Nested Generalizations*«) [RBP+91].
- Wenn eine Implementierung von mehreren Objekten gemeinsam genutzt werden soll (beispielsweise mittels Referenzzählung (engl. *Reference Counting*

)) – und diese Tatsache dem Client verborgen bleiben soll. Ein simples Beispiel ist in diesem Zusammenhang Copliens String-Klasse [Cop92], in der ein und dieselbe Stringrepräsentation (die Klasse `StringRep`) von mehreren Objekten verwendet werden kann.

Struktur

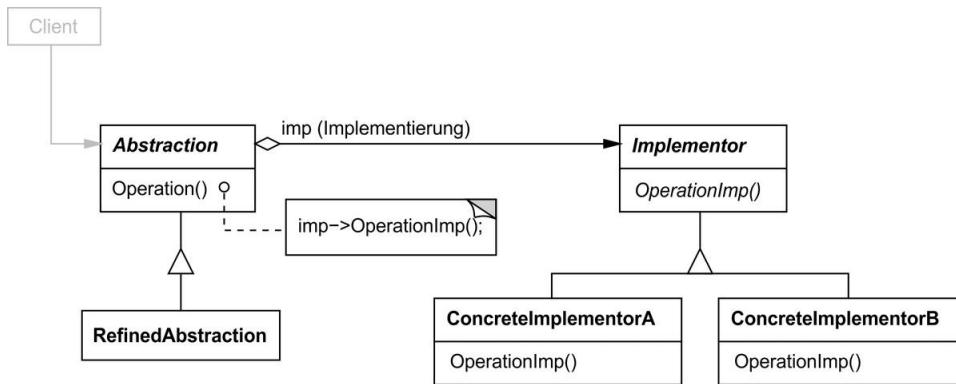


Abb. 4.10: Die Struktur des Design Patterns Bridge (Brücke)

Teilnehmer

- **Abstraction (Window)**
 - Definiert die Schnittstelle der Abstraktion.
 - Verwaltet eine Referenz auf ein Objekt des Typs `Implementor`.
- **RefinedAbstraction (IconWindow)**
 - Erweitert die durch `Abstraction` definierte Schnittstelle.
- **Implementor (WindowImp)**
 - Definiert die Schnittstelle für die Implementierungsklassen. Sie muss nicht notwendigerweise exakt mit der `Abstraction`-Schnittstelle übereinstimmen, sondern kann sich sogar deutlich von ihr unterscheiden. In der Regel liefert die `Implementor`-Schnittstelle lediglich die primitiven Operationen, während `Abstraction` darauf basierende höherrangigere Operationen bereitstellt.

- **ConcreteImplementor** (`XWindowImp`, `PMWindowImp`)
 - Implementiert die Schnittstelle `Implementor` und definiert ihre konkrete Implementierung.

Interaktionen

- Die Klasse `Abstraction` leitet clientseitige Requests an ihr Implementierungsobjekt weiter.

Konsequenzen

Das Design Pattern *Bridge* (*Brücke*) wirkt sich wie folgt aus:

1. *Entkopplung von Schnittstelle und Implementierung.* Eine Implementierung ist nicht dauerhaft an eine Schnittstelle gebunden. Die Implementierung einer Abstraktion kann zur Laufzeit konfiguriert werden, und sogar Objekte können ihre Implementierung zur Laufzeit wechseln.

Durch die Entkopplung der Schnittstellen `Abstraction` und `Implementor` werden auch die bei der Kompilierung auf eine Implementierung entfallenden Abhängigkeiten aufgehoben. Die Änderung einer Implementierungsklasse erfordert keine Neukompilierung der Klasse `Abstraction` und ihrer Clients. Für den Fall, dass eine Binärkompatibilität zwischen verschiedenen Versionen einer Klassenbibliothek sichergestellt sein muss, ist diese Eigenschaft unverzichtbar.

Darüber hinaus begünstigt die Entkopplung außerdem die Schichtenbildung (engl. *Layering*), die zu einer besseren Strukturierung des Systems beitragen kann. Die oben angesiedelten Systemschichten müssen dann lediglich Kenntnis von den Klassen `Abstraction` und `Implementor` haben.

2. *Bessere Erweiterbarkeit.* Die `Abstraction`- und `Implementor`-Hierarchien können unabhängig voneinander erweitert werden.
3. *Verbergen der Implementierungsdetails vor den Clients.* Die Clients können von detaillierteren Abläufen innerhalb der Implementierung unbekümmert bleiben, beispielsweise von der gemeinsamen Nutzung von `Implementor`-Objekten oder dem damit einhergehenden Referenzzählungsmechanismus (sofern vorhanden).

Implementierung

Beim Einsatz des Design Patterns *Bridge (Brücke)* sollten folgende Implementierungsaspekte beachtet werden:

1. *Nur ein Implementor-Objekt.* In Situationen, in denen nur eine einzige Implementierung vorliegt, ist die Erzeugung einer abstrakten `Implementor`-Klasse nicht erforderlich. Hierbei handelt es sich um eine degenerierte Variante des Design Patterns *Bridge (Brücke)*: Es besteht eine 1-zu-1-Beziehung zwischen `Abstraction` und `Implementor`. Trotzdem ist ihre Separierung immer noch sinnvoll, wenn eine Änderung an der Implementierung einer Klasse keine Auswirkungen auf ihre vorhandenen Clients haben soll – sprich diese nicht neu kompiliert, sondern lediglich neu verknüpft werden sollen.

Carolan [Car89] bezeichnet diese Entkopplung als »Cheshire Cat« (dt. »Grinsekatze«): In C++ kann die Klassenschnittstelle der `Implementor`-Klasse in einer privaten Headerdatei definiert werden, die den Clients nicht zur Verfügung gestellt wird – d. h., die Implementierung einer Klasse bleibt den Clients vollständig verborgen.

2. *Erzeugung des richtigen Implementor-Objekts.* Wie, wann und wo wird bei mehreren vorliegenden `Implementor`-Klassen entschieden, welche von ihnen instanziert werden muss?

Sofern die Klasse `Abstraction` von allen `ConcreteImplementor`-Klassen Kenntnis hat, kann sie eine davon in ihrem Konstruktor instanziiieren. Die Wahl zwischen den diversen Klassen kann dabei anhand der dem Konstruktor übergebenen Parameter getroffen werden. Wenn beispielsweise eine `Collection`-Klasse mehrere Implementierungen unterstützt, könnte die Größe der Collection als Entscheidungskriterium herangezogen werden. Bei kleineren Collections könnte auch eine verkettete Liste und bei größeren eine Hashtabelle zugrunde gelegt werden.

Eine weitere mögliche Herangehensweise besteht darin, zunächst eine Standardimplementierung auszuwählen und sie dann später entsprechend des Nutzungsverhaltens zu ändern. Sollte die Collection also beispielsweise über einen bestimmten Schwellenwert hinaus anwachsen, wechselt sie zu einer Implementierung, die für große Objektmengen besser geeignet ist.

Darüber hinaus kann die Entscheidung aber auch komplett an ein anderes Objekt delegiert werden. Im Fall des `Window/WindowImp`-Beispiels könnte dazu

ein Factory-Objekt (siehe *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#))) erstellt werden, dessen einzige Aufgabe darin besteht, die Plattformspezifika zu kapseln. Da die Factory weiß, welche Art von `WindowImp`-Objekt für die gerade verwendete Plattform zu erzeugen ist, braucht ein Fenster einfach nur ein `WindowImp`-Objekt von ihr abzufragen und bekommt dann das passende Objekt zurückgeliefert. Ein Vorteil dieses Ansatzes ist, dass die `Abstraction`-Klasse nicht unmittelbar an eine der Implementierungsklassen gekoppelt ist.

3. *Gemeinsame Nutzung von Implementor-Objekten.* Coplien beschreibt, wie das Handle/Body-Idiom in C++ für die gemeinsame Nutzung von Implementierungen zwischen mehreren Objekten [Cop92] verwendet werden kann. Der Body speichert eine Referenzzählung, die in der Handle-Klasse hoch- und runtergezählt wird. Der Code für die Zuweisung von Handle-Objekten an gemeinsam genutzte Body-Objekte hat folgende allgemeine Syntax:

```
Handle& Handle::operator= (const Handle& other) {  
    other._body->Ref();  
    _body->Unref();  
  
    if (_body->RefCount() == 0) {  
        delete _body;  
    }  
    _body = other._body;  
    return *this;  
}
```

4. *Anwendung der Mehrfachvererbung.* In C++ kann zur Zusammenführung einer Schnittstelle und ihrer Implementierung die Mehrfachvererbung genutzt werden [Mar91]. Zum Beispiel kann eine Klasse von `Abstraction` *öffentlich* erben und von `ConcreteImplementor` *privat*. Weil dieser Ansatz allerdings auf der statischen Vererbung basiert, wird die Implementierung dadurch dauerhaft an ihre Schnittstelle gebunden. Und deshalb kann in diesem Fall keine echte Bridge implementiert werden – zumindest nicht in C++.

Beispielcode

Der folgende C++-Code implementiert das im Abschnitt »Motivation« vorgestellte `Window/WindowImp`-Beispiel. Die `Window`-Klasse definiert dabei die Fensterabstraktion für Client-Anwendungen:

```

class Window {
public:
    Window(View* contents);

    // Von Window bearbeitete Requests
    virtual void DrawContents();

    virtual void Open();
    virtual void Close();
    virtual void Iconify();
    virtual void Deiconify();

    // An die Implementierung weitergeleitete Requests
    virtual void SetOrigin(const Point& at);
    virtual void SetExtent(const Point& extent);
    virtual void Raise();
    virtual void Lower();

    virtual void DrawLine(const Point&, const Point&);
    virtual void DrawRect(const Point&, const Point&);
    virtual void DrawPolygon(const Point[], int n);
    virtual void DrawText(const char*, const Point&);

protected:
    WindowImp* GetWindowImp();
    View* GetView();

private:
    WindowImp* _imp;
    View* _contents;
    // Die Fensterinhalte
};

```

Window enthält eine Referenz auf WindowImp – die abstrakte Klasse, die eine Schnittstelle für das zugrundeliegende Fenstersystem deklariert:

```

class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;

    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // Zahlreiche weitere Funktion zum Zeichnen auf den Fenstern ...

protected:
    WindowImp();
};

```

Die Unterklassen von `Window` definieren verschiedene Fenstervarianten, die die Anwendung nutzen könnte, z. B. Anwendungsfenster, auf Symbolgröße verkleinerbare Fenster, kurzzeitig einzublendende Dialogfenster, schwebende Werkzeugpaletten usw.

Beispielsweise implementiert `ApplicationWindow` die Operation `DrawContents`, damit sie die gespeicherte `View`-Instanz zeichnet:

```
class ApplicationWindow : public Window {  
public:  
    // ...  
    virtual void DrawContents();  
};  
  
void ApplicationWindow::DrawContents () {  
    GetView()->DrawOn(this);  
}
```

Die Klasse `IconWindow` speichert den Namen einer Bitmap für das Icon, das sie darstellt ...

```
class IconWindow : public Window {  
public:  
    // ...  
    virtual void DrawContents();  
  
private:  
    const char* _bitmapName;  
};
```

... und implementiert die Operation `DrawContents`, um die Bitmap auf dem Fenster zu zeichnen:

```
void IconWindow::DrawContents() {  
    WindowImp* imp = GetWindowImp();  
    if (imp != 0) {  
        imp->DeviceBitmap(_bitmapName, 0.0, 0.0);  
    }  
}
```

Darüber hinaus sind aber noch viele weitere Variationen von `Window` möglich. Ein `TransientWindow` muss möglicherweise mit dem Fenster, das es während des Dialogs erzeugt hat, kommunizieren – deshalb behält diese Unterklasse eine Referenz auf das betreffende Fenster bei. Ein `PalettesWindow` schwebt immer über anderen Fenstern. Und ein `IconDockWindow` verwaltet die `IconWindows` und ordnet sie in ansprechender Weise an.

Window-Operationen sind entsprechend der `WindowImp`-Schnittstelle definiert. Zum Beispiel extrahiert die `DrawRect`-Operation vier Koordinaten aus ihren beiden `Point`-Parametern, bevor sie die `WindowImp`-Operation aufruft, die das Rechteck in das Fenster einzeichnet:

```
void Window::DrawRect (const Point& p1, const Point& p2) {
    WindowImp* imp = GetWindowImp();
    imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

Konkrete Unterklassen von `WindowImp` gewährleisten die Unterstützung verschiedener Fenstersysteme. In diesem Beispiel wird eine Unterklasse `XWindowImp` für das X-Window-System erstellt:

```
class XWindowImp : public WindowImp {
public:
    XWindowImp ();

    virtual void DeviceRect(Coord, Coord, Coord, Coord);
    // Rest der öffentlichen Schnittstelle ...

private:
    // Weite Teile des X-Window-System-spezifischen Zustands,
    einschließlich:
    Display* _dpy;
    Drawable _winid;
    // Fenster-ID
    GC _gc;
    // Grafischer Kontext des Fensters
};
```

Für den Presentation Manager (PM) wird außerdem eine Klasse `PMWindowImp` definiert:

```
class PMWindowImp : public WindowImp {
public:
    PMWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord, Coord);

    // Rest der öffentlichen Schnittstelle...

private:
    // Weite Teile des für das PM-Fenstesystem spezifischen Zustands,
    // einschließlich:
    HPS _hps;
};
```

Diese Unterklassen implementieren den Primitiven des Fenstersystems

entsprechende `WindowImp`-Operationen. So ist z. B. `DeviceRect` für X Window wie folgt implementiert:

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dp, _winid, _gc, x, y, w, h);
}
```

Und die PM-Implementierung könnte dann so aussehen:

```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];

    point[0].x = left; point[0].y = top;
    point[1].x = right; point[1].y = top;
    point[2].x = right; point[2].y = bottom;
    point[3].x = left; point[3].y = bottom;

    if (
        (GpiBeginPath(_hps, 1L) == false) ||
        (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
        (GpiPolyLine(_hps, 4L, point) == GPI_ERROR) ||
        (GpiEndPath(_hps) == false)
    ) {
        // Fehlermeldung
    } else {
        GpiStrokePath(_hps, 1L, 0L);
    }
}
```

Wie aber erhält ein Fenster eine Instanz der richtigen `WindowImp`-Unterklasse? In diesem Beispiel wird angenommen, dass die Zuständigkeit bei der Klasse `Window` liegt. Deren `GetWindowImp`-Operation empfängt die richtige Instanz von einer abstrakten Factory (*Abstract Factory (Abstrakte Fabrik)*, siehe [Abschnitt 3.1](#)), die effektiv alle Spezifika des Fenstersystems kapselt:

```

WindowImp* Window::GetWindowImp () {
    if (_imp ==0) {
        _imp = WindowSystemFactory::Instance() -> MakewindowImp();
    }
    return _imp;
}

```

`WindowSystemFactory::Instance()` gibt eine abstrakte Factory zurück, die alle für das Fenstersystem spezifischen Objekte erzeugt. Der Einfachheit halber wird sie hier als *Singleton* (*Singleton*, siehe [Abschnitt 3.5](#)) verwendet und gewährt der `Window`-Klasse direkten Zugriff.

Praxisbeispiele

Das vorgenannte `window`-Beispiel entstammt der C++-Klassenbibliothek **ET++** [WGM88], in der die Klasse `WindowImp` als `WindowPort` bezeichnet wird und Unterklassen wie `XWindowPort` und `SunWindowPort` besitzt. Zur Erzeugung seines entsprechenden `Implementor`-Objekts fragt das `Window`-Objekt dieses von einer abstrakten Factory namens `WindowSystem` ab, die ihrerseits eine Schnittstelle zur Erzeugung plattformspezifischer Objekte wie Fonts, Cursor-Varianten, Bitmaps usw. bereitstellt.

Das **Window/WindowPort**-Design in ET++ erweitert das Design Pattern *Bridge* (*Brücke*) dahingehend, dass `WindowPort` ebenfalls eine Referenz auf `Window` enthält. Die `WindowPort`-Implementierungsklasse nutzt diese Referenz, um `Window` über `WindowPort`-spezifische Events wie z. B. Eingabeeignisse, Größenänderungen des Fensters etc. zu informieren.

Sowohl bei **Coplien** [Cop92] als auch bei **Stroustrup** [Str91] werden in diesem Zusammenhang auch `Handle`-Klassen erwähnt und einige Beispiele aufgezeigt, die sich jedoch insbesondere auf Problemstellungen hinsichtlich der Speicherverwaltung beziehen, etwa die gemeinsame Nutzung von Stringrepräsentationen und die Unterstützung von Objekten variabler Größe. Das in diesem Abschnitt angeführte Beispiel richtet sich allerdings vorrangig auf die Unterstützung der unabhängigen Erweiterbarkeit sowohl der Abstraktion als auch ihrer Implementierung.

Die GNU-Klassenbibliothek **libg++** [Lea88] definiert Klassen, die allgemeine Datenstrukturen implementieren, wie z. B. `Set`, `LinkedSet`, `HashSet`, `LinkedList` und `HashTable`. Die abstrakte Klasse `Set` definiert eine Mengenabstraktion, während `LinkedList` und `HashTable` konkrete Implementierungen einer verketteten Liste bzw.

einer Hashtabelle darstellen. `LinkedSet` und `HashSet` sind `Set`-Implementoren, die eine »Brücke« zwischen `Set` und ihren konkreten Entsprechungen `LinkedList` und `HashTable` bilden. Hierbei handelt es sich um eine degenerierte Variante des Design Patterns *Bridge (Brücke)*, da keine abstrakte `Implementor`-Klasse vorhanden ist.

Das **AppKit-Framework** von NeXT [Add94] macht sich das Design Pattern *Bridge (Brücke)* für die Implementierung und Darstellung von Grafiken zunutze. Die Anzeige eines Bildes kann auf unterschiedliche Art und Weise erfolgen. Seine optimale Darstellung ist dabei von den Eigenschaften des Anzeigegeräts, insbesondere dessen Farbfähigkeiten und Auflösung abhängig. Ohne das AppKit-Framework müssten die Entwickler selbst bestimmen, welche Implementierung unter den jeweiligen Umständen in jeder Anwendung zu benutzen ist.

Um ihnen diese Aufgabe abzunehmen, stellt ihnen das AppKit eine `NXImage/NXImageRep`-Bridge zur Verfügung. `NXImage` definiert die Schnittstelle für die Handhabung der Bilder, deren Implementierung in einer separaten `NXImageRep`-Klassenhierarchie mit Unterklassen wie `NXEPSImageRep`, `NXCachedImageRep` und `NXBitmapImageRep` hinterlegt ist. Die `NXImage`-Schnittstelle enthält eine Referenz auf ein oder mehrere `NXImageRep`-Objekte – und wählt, sofern mehrere Bildimplementierungen vorhanden sind, die für das aktuell verwendete Anzeigegerät am besten geeignete aus. Wenn nötig, kann `NXImage` sogar eine Implementierung in eine andere konvertieren. Das Interessante an dieser Variante des Design Patterns *Bridge (Brücke)* ist, dass `NXImage` mehr als eine `NXImageRep`-Implementierung auf einmal speichern kann.

Verwandte Patterns

Eine *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) kann ein bestimmtes *Bridge*-Objekt erzeugen und konfigurieren.

Das Design Pattern *Adapter* (*Adapter*, siehe [Abschnitt 4.1](#)) ist darauf ausgerichtet, die Zusammenarbeit nicht miteinander verwandter Klassen zu ermöglichen. In der Regel wird es erst nach der Fertigstellung des Designs auf die Systeme angewendet. Das Pattern *Bridge (Brücke)* fließt dagegen von vornherein in das Design mit ein, damit die Abstraktionen und Implementierungen unabhängig voneinander variiert werden können.

4.3 Composite (Kompositum)

Objektbasiertes Strukturmuster

Zweck

Komposition von Objekten in Baumstrukturen zur Abbildung von Teil-Ganzes-Hierarchien. Das Design Pattern *Composite (Kompositum)* gestattet den Clients einen einheitlichen Umgang sowohl mit individuellen Objekten als auch mit Objektkompositionen.

Motivation

Grafische Anwendungen wie Zeicheneditoren und Systeme zur Erstellung technischer Zeichnungen ermöglichen den Usern, einfache Komponenten zu komplexen Diagrammen zusammenzufügen. So können diese Komponenten zu größeren Komponenten kombiniert werden, aus denen sich wiederum noch größere Komponenten erstellen lassen. Eine simple Implementierung könnte beispielsweise Klassen für grafische Primitive wie Text und Linien sowie weitere Klassen definieren, die als Behälter für die Primitiven fungieren.

Allerdings birgt dieser Ansatz ein Problem: Programme, die solche Klassen verwenden, müssen die primitiven Objekte und die Behälter unterschiedlich behandeln, selbst wenn der User sie meistens identisch nutzt. Und aufgrund dieser Unterscheidung wird die Anwendung notgedrungen komplexer. Das Design Pattern *Composite (Kompositum)* beschreibt, wie sich die rekursive Komposition so anwenden lässt, dass die Clients die Objekte nicht zwingend unterscheiden müssen.

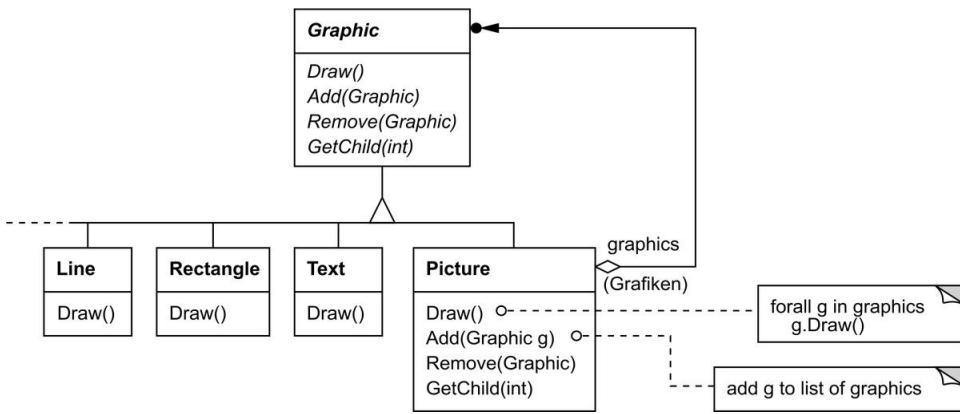


Abb. 4.11: Arbeitsweise des Design Patterns *Composite (Kompositum)* am Beispiel der abstrakten Klasse *Graphic*

Das Herzstück des Patterns *Composite (Kompositum)* ist eine abstrakte Klasse, die sowohl die Primitiven als auch deren Behälter repräsentiert. Im Fall des hier verwendeten Grafiksystems ist diese Klasse mit *Graphic* benannt. Sie deklariert speziell auf grafische Operationen ausgerichtet Objekte, hier z. B. *Draw*. Darüber hinaus stellt sie aber auch Operationen zur Verfügung, die allen zusammengefügten Objekten gemein sind, wie z. B. Operationen für den Zugriff auf deren Kindobjekte sowie deren Verwaltung.

Die in [Abbildung 4.11](#) gezeigten Unterklassen *Line*, *Rectangle* und *Text* definieren primitive grafische Objekte. Diese Klassen implementieren *Draw* jeweils zum Zeichnen von Linien, Rechtecken und Text. Da primitive Grafiken keine »Kindgrafiken« haben, sind auch keine kindbezogenen Operationen in diesen Unterklassen enthalten.

Die Klasse *Picture* definiert ein Aggregat, sprich eine Zusammenstellung von *Graphic*-Objekten. Sie implementiert die Operation *Draw* in der Form, dass sie eine weitere *Draw*-Operation auf ihre Kindobjekte aufruft. Ebenso werden auch alle anderen kindbezogenen Operationen implementiert. Weil die Schnittstelle *Picture* der Schnittstelle *Graphic* entspricht, können *Picture*-Objekte weitere *Picture*-Objekte rekursiv zusammenfügen.

[Abbildung 4.12](#) zeigt eine typische aus rekursiv zusammengefügten *Graphic*-Objekten bestehende Objektstruktur:

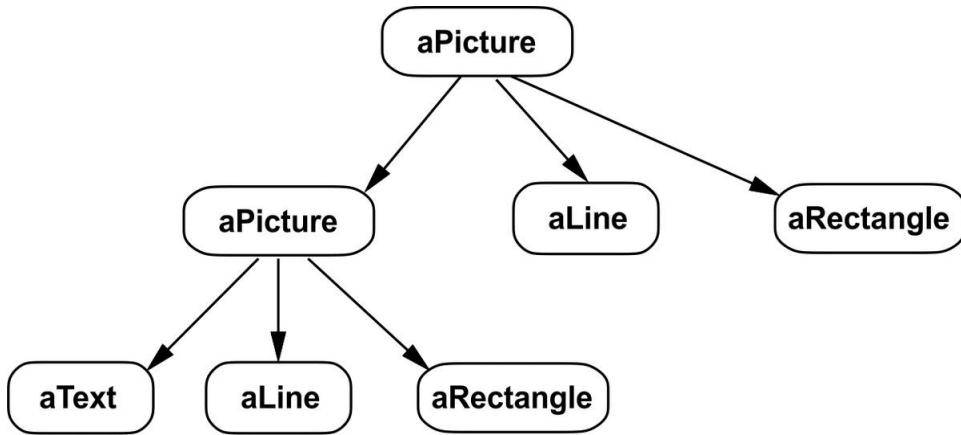


Abb. 4.12: Objektstruktur mit rekursiv zusammengefügten Objekten

Anwendbarkeit

Der Einsatz des Design Patterns *Composite (Kompositum)* ist dann sinnvoll, wenn

- Teil-Ganzes-Hierarchien von Objekten dargestellt werden sollen.
- die Clients in der Lage sein sollen, nicht zwischen individuellen Objekten und Objektkompositionen unterscheiden zu müssen, sondern stattdessen alle Objekte in der zusammengesetzten Struktur einheitlich zu behandeln.

Struktur

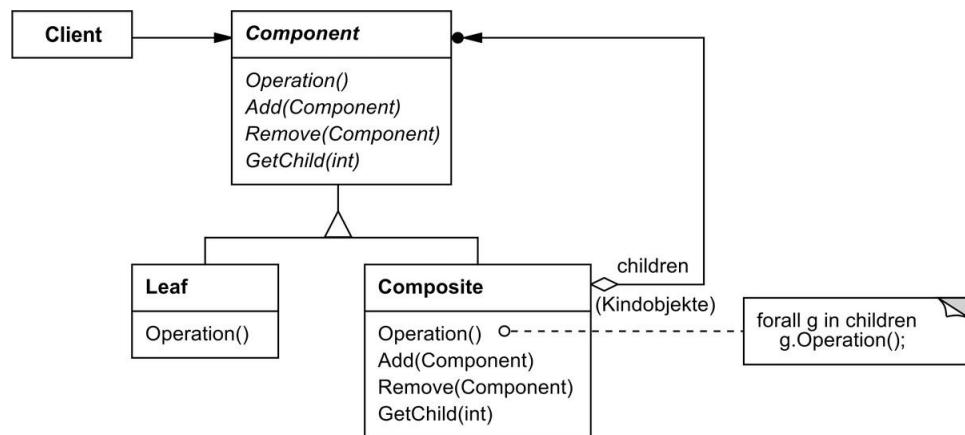


Abb. 4.13: Die Struktur des Design Patterns *Composite (Kompositum)*

Eine typische Objektstruktur im Design Pattern *Composite (Kompositum)* könnte

beispielsweise wie folgt aussehen:

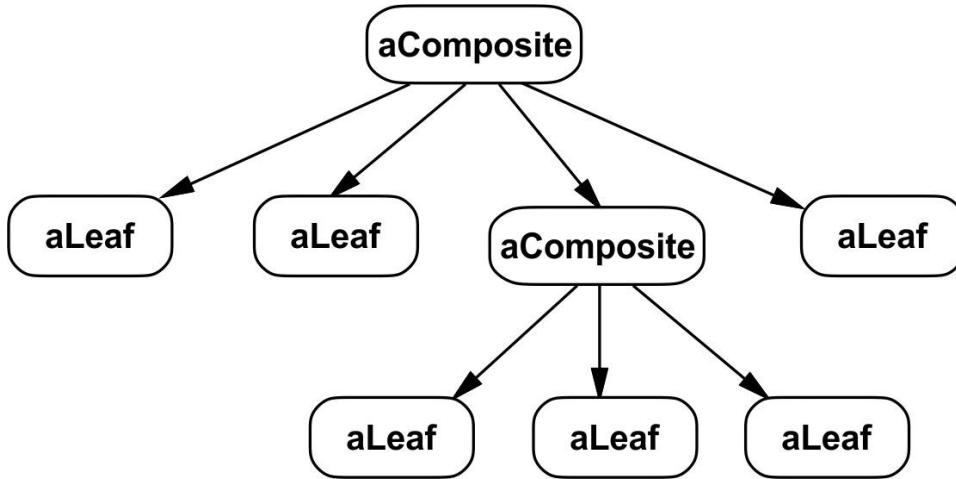


Abb. 4.14: Typische Objektstruktur im Design Pattern Composite (Kompositum)

Teilnehmer

- **Component** (Graphic)
 - Deklariert die Schnittstelle für Objekte in der Komposition.
 - Implementiert bei Bedarf ein Standardverhalten für die Schnittstelle, die alle Klassen gemeinsam haben.
 - Deklariert eine Schnittstelle für den Zugriff auf die Component-Kindobjekte und deren Verwaltung.
 - (*Optional*) Definiert eine Schnittstelle für den Zugriff auf das Elternobjekt einer Komponente in der rekursiven Struktur und implementiert diese gegebenenfalls.
- **Leaf** (Rectangle, Line, Text etc.)
 - Repräsentiert die Leaf-Objekte (Blattobjekte) in der Komposition. Dieser Objekttyp besitzt keine Kindobjekte.
 - Definiert das Verhalten der primitiven Objekte in der Komposition.
- **Composite** (Picture)

- Definiert das Verhalten der Komponenten, die über Kindobjekte verfügen.
 - Speichert die Kindkomponenten.
 - Implementiert kindbezogene Operationen der Component-Schnittstelle.
- **Client**
 - Manipuliert die Objekte in der Komposition über die Component-Schnittstelle.

Interaktionen

- Clients nutzen die Klassenschnittstelle von Component zur Interaktion mit Objekten in der Composite-Struktur. Handelt es sich bei dem Empfänger um ein Leaf-Objekt, werden die Requests unmittelbar bearbeitet. Ist der Empfänger hingegen ein Composite-Objekt, dann werden die Requests in der Regel an dessen Kindkomponenten weitergeleitet, wobei möglicherweise vor und/oder nach der Weiterleitung noch zusätzliche Operationen ausgeführt werden.

Konsequenzen

Das Design Pattern *Composite (Kompositum)*

- definiert aus primitiven und zusammengesetzten Objekten bestehende Klassenhierarchien. Primitive Objekte können zu größeren Objekten zusammengefügt werden, die wiederum rekursiv zu noch komplexeren Objekten kombiniert werden können usw. Wann immer der Clientcode ein primitives Objekt erwartet, kann er ebenso gut auch ein zusammengesetztes Objekt entgegennehmen.
- gestattet den Clients eine einheitliche Handhabung sowohl der zusammengesetzten Strukturen als auch der individuellen Objekte. Im Normalfall haben die Clients keine Kenntnis davon (und es sollte für sie auch nicht von Belang sein), ob sie es mit einem Leaf-Objekt oder einer zusammengesetzten Komponente zu tun haben. Dadurch muss der Clientcode keine Funktionen zur Unterscheidung der kompositionsdefinierenden Klassen von den übrigen Klassen enthalten.

- erleichtert das Hinzufügen neuer Komponententypen. Neu definierte Composite- oder Leaf-Klassen arbeiten automatisch mit den vorhandenen Strukturen und dem existierenden Clientcode zusammen. Zur Ergänzung neuer Component-Klassen ist somit keine Modifizierung der Clients erforderlich.
- kann das Design zu allgemeingültig werden lassen. Die vereinfachte Ergänzung neuer Komponenten birgt den Nachteil, dass es auf der anderen Seite schwieriger wird, die Komponenten einer Composite-Struktur zu begrenzen. Manchmal soll eine Komposition aber nur aus bestimmten Komponenten bestehen. Beim Einsatz des Design Patterns *Composite (Kompositum)* werden solche Einschränkungen nicht mehr vom Typsystem vorgegeben, sondern es müssen entsprechende Laufzeitüberprüfungen durchgeführt werden.

Implementierung

Bei der Anwendung des Design Patterns *Composite (Kompositum)* sind zahlreiche Kriterien zu beachten:

1. *Explizite Referenzen auf Elternobjekte.* Die Aufrechterhaltung der Referenzen von Kindkomponenten auf ihre Elternobjekte kann in Bezug auf die Traversierung und Verwaltung einer Composite-Struktur sehr hilfreich sein, weil sie die Navigation innerhalb der Hierarchie, z. B. zum Entfernen einer Komponente, deutlich einfacher gestaltet. Darüber hinaus erleichtern solche Referenzen auch die Implementierung des Design Patterns *Chain of Responsibility (Zuständigkeitskette*, siehe [Abschnitt 5.1](#)).

Üblicherweise wird die Referenz eines Kindobjekts auf das zugehörige Elternobjekt in der Klasse `Component` definiert. Dadurch können die Klassen `Leaf` und `Composite` sowohl die Referenz selbst als auch die von ihr verwalteten Operationen erben.

Ein wichtiger Aspekt ist in diesem Zusammenhang die Beibehaltung der Invariante, dass alle Kindobjekte eines Kompositums exakt dieses Composite-Objekt als Elternobjekt besitzen, das seinerseits exakt diese Objekte als Kindobjekte besitzt. Am einfachsten ist das zu gewährleisten, indem eine Änderung des Elternobjekts einer Komponente *ausschließlich* dann erfolgt, wenn es zu einer Composite-Struktur ergänzt oder daraus entfernt wird. Wird diese Vorgabe genau einmal in die Add- (Hinzufügen) und Remove (Entfernen)-Operationen der Composite-Klasse implementiert, kann sie von allen

Unterklassen geerbt werden – und damit bleibt die Invariante automatisch erhalten.

2. *Gemeinsame Nutzung von Komponenten.* Oftmals ist es sinnvoll, Komponenten gemeinsam zu nutzen, z. B. um die Speicherbeanspruchung zu reduzieren. Schwierig wird es allerdings dann, wenn eine Komponente mehr als ein Elternobjekt haben kann.

Eine Lösungsmöglichkeit ist in diesem Fall das Speichern mehrerer Elternobjekte in einem Kindobjekt. Das kann jedoch bei der Weiterleitung eines Requests in die höheren Ebenen der Struktur zu Mehrdeutigkeiten führen. Das Design Pattern *Flyweight* (*Fliegengewicht*, siehe [Abschnitt 4.6](#)) zeigt auf, wie sich ein Design so umgestalten lässt, dass ein Speichern der Elternobjekte vollständig vermieden wird. Es funktioniert in allen Fällen, in denen Kindobjekte die Weiterleitung von an die Elternobjekte gerichteten Requests durch die teilweise oder vollständige Auslagerung ihres Zustands umgehen.

3. *Maximierung der Component-Schnittstelle.* Eine Zielsetzung des Design Patterns *Composite* (*Kompositum*) besteht darin, die spezifischen Leaf- oder Composite-Klassen, die von den Clients genutzt werden, vor diesen zu verbergen. Dazu sollte die Component-Klasse so viele gemeinsame Operationen für Composite und Leaf definieren wie möglich. In der Regel stellt die Component-Klasse entsprechende Standardimplementierungen bereit, die dann von den Unterklassen Leaf und Composite überschrieben werden.

Mitunter gerät diese Zielsetzung allerdings mit dem Prinzip des Klassenhierarchiedesigns in Konflikt, das besagt, dass eine Klasse nur solche Operationen definieren sollte, die für ihre Unterklassen sinnvoll sind. Denn viele der von Component unterstützten Operationen scheinen für die Leaf-Klassen nicht unbedingt sinnvoll zu sein. Wie also kann die Klasse Component eine Standardimplementierung für sie zur Verfügung stellen?

Mit ein wenig Kreativität kann eine Operation, die im Prinzip nur für Composite-Klassen sinnvoll erscheint, durch Verschieben in die betreffende Klasse aber doch noch für alle Component-Unterklassen implementiert werden. Zum Beispiel stellt die Schnittstelle für den Zugriff auf die Kindobjekte einen fundamentalen Bestandteil einer Composite-Klasse dar – nicht notwendigerweise aber auch der Leaf-Klassen. Betrachtet man eine Leaf-Klasse jedoch als eine Component-Klasse, die *niemals* Kindobjekte besitzen wird, dann kann in dieser Component-Klasse eine Standardoperation für den Zugriff auf Kindobjekte definiert werden, die *niemals* irgendwelche Kindobjekte

zurückgibt. Das heißt, Leaf-Klassen können die Standardimplementierung nutzen, aber die Composite-Klassen implementieren sie in der Form neu, dass sie ihre Kindobjekte zurückgeben.

Die im nächsten Abschnitt beschriebenen Operationen zur Verwaltung von Kindobjekten sind im Vergleich dazu etwas problematischer.

4. *Deklaration der Operationen zur Verwaltung von Kindobjekten.* Wenngleich die Add- (Hinzufügen) und Remove (Entfernen)-Operationen zur Verwaltung der Kindobjekte von der Composite-Klasse implementiert werden, lautet eine der Kernfragen beim Design Pattern *Composite (Kompositum)*, welche Klassen diese Operationen in der Composite-Klassenhierarchie *deklarieren*: Sollten sie in der Klasse Component deklariert sein, wo sie für die Leaf-Klassen von Bedeutung sind, oder sollten sie nur in der Klasse Composite und deren Unterklassen deklariert und definiert sein?

Die Entscheidungsfindung erfordert in diesem Fall ein Abwägen der Faktoren Sicherheit und Transparenz:

- Die Definition der Schnittstelle zur Verwaltung der Kindobjekte in der Root-Klasse der Klassenhierarchie gewährleistet Transparenz, weil alle Komponenten einheitlich behandelt werden können. Andererseits wird dadurch jedoch die Sicherheit beeinträchtigt, weil die Clients in diesem Fall sinnlose Aktionen wie das Hinzufügen und Entfernen von Objekten aus den Leaf-Klassen versuchen könnten.
- Die Definition der Schnittstelle zur Verwaltung der Kindobjekte in der Composite-Klasse gewährleistet dagegen Sicherheit, weil jeder Versuch, Objekte zu den Leaf-Klassen zu ergänzen oder daraus zu entfernen, in einer statisch typisierten Sprache wie C++ schon beim Kompilieren abgefangen wird. Andererseits wird dadurch jedoch die Transparenz beeinträchtigt, weil die Leaf- und die Composite-Klassen unterschiedliche Schnittstellen haben.

In dem hier angeführten Beispiel wurde dem Faktor Transparenz gegenüber der Sicherheit der Vorzug gegeben. Im umgekehrten Szenario kann es gelegentlich vorkommen, dass Typinformationen verloren gehen und eine Komponente daher in ein Kompositum konvertiert werden muss. Doch wie lässt sich dies bewerkstelligen, ohne einen unsicheren Downcast ausführen zu müssen?

Ein Ansatz ist die Deklaration einer Operation `Composite* GetComposite()` in der `Component`-Klasse. Diese Klasse stellt eine Standardoperation zur Verfügung, die einen Nullzeiger zurückgibt – und die `Composite`-Klasse definiert diese Operation dann so um, dass sie sich mithilfe des `this`-Zeigers selbst zurückgibt:

```
class Composite;
class Component {
public:
    //...
    virtual Composite* GetComposite() { return 0; }
};

class Composite : public Component {
public:
    void Add(Component* );
    // ...
    virtual Composite* GetComposite() { return this; }
};

class Leaf : public Component {
    // ...
};
```

Mit `GetComposite` kann abgefragt werden, ob es sich bei der Komponente um ein Kompositum handelt. Ist das der Fall, können `Add` und `Remove` sicher auf dem zurückgegebenen `Composite`-Objekt ausgeführt werden:

```
Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;

Component * aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->GetComposite()) {
    test->Add(new Leaf);
}

aComponent = aLeaf;

if (test = aComponent->GetComposite()) {
    test->Add(new Leaf);
    // Es wird kein Leaf-Objekt hinzugefügt
}
```

In C++ können ähnliche Überprüfungen für eine `Composite`-Klasse mit dem `dynamic_cast`-Operator ausgeführt werden.

Dabei stellt sich aber das Problem, dass nicht alle Komponenten einheitlich behandelt werden – deshalb muss vor der Ausführung der passenden Operationen zunächst auf verschiedene Typprüfungen zurückgegriffen werden.

Der einzige gangbare Weg zur Gewährleistung von Transparenz besteht in der Definition standardmäßiger Add- und Remove-Operationen in der Component-Klasse. Das ruft jedoch wiederum ein neues Problem auf den Plan: Bei der Implementierung von Component::Add muss neben der Möglichkeit der erfolgreichen Durchführung der Operation auch die Möglichkeit ihres Fehlschlags berücksichtigt werden. Man könnte sie vielleicht leer implementieren – damit würde allerdings eine wichtige Tatsache ignoriert, nämlich: Der Versuch, ein Objekt zu einer Leaf-Klasse hinzuzufügen, weist in aller Regel auf einen Bug hin. Und in diesem Fall produziert die Operation Add »Müll«. Sie könnte zwar so implementiert werden, dass sie ihr Argument löscht – das würde jedoch möglicherweise nicht dem entsprechen, was die Clients erwarten.

In der Regel ist es sinnvoller, Add und Remove standardmäßig fehlschlagen zu lassen (ggf. durch das Auslösen einer Ausnahmebehandlung (engl. *Exception*)), wenn die Komponente keine Kindobjekte haben darf oder das Argument von Remove kein Kindobjekt der Komponente ist.

Eine weitere Alternative wäre eventuell auch eine leicht veränderte Auslegung des Begriffs »Remove« (Entfernen): Wenn eine Komponente eine Referenz auf ein Elternobjekt enthält, dann könnte die Operation Component::Remove so definiert werden, dass sie sich selbst aus dem Elternobjekt entfernt – für die Add-Operation hat sich allerdings bislang noch keine ähnlich sinnvolle Interpretation gefunden.

5. *Component-Klasse als Behälter für Komponenten.* Möglicherweise steht die Überlegung im Raum, den Behälter der Kindobjekte als Instanzvariable in der Component-Klasse zu definieren, in der die Operationen für den Zugriff auf die Kindobjekte und deren Verwaltung deklariert sind. Das Hinzufügen des Pointers, der auf die Kindobjekte zeigt, in die Basisklasse bedeutet jedoch für jedes Leaf-Objekt Kapazitätseinbußen – obwohl ein Leaf-Objekt zu keinem Zeitpunkt Kindobjekte besitzen wird. Insofern ist diese Verfahrensweise nur dann nutzbringend, wenn relativ wenige Kindobjekte in der Struktur vorhanden sind.
6. *Ordnung der Kindobjekte.* Viele Designs spezifizieren eine bestimmte Ordnung der Kindobjekte der Composite-Klasse. So könnte in dem eingangs erwähnten

Graphics-Beispiel eine Front-to-back-Ordnung gefordert sein, d. h., die Überlagerung der grafischen Objekte muss von vorne nach hinten geordnet erfolgen. Wenn Composite-Objekte Parse-Bäume repräsentieren, können Verbundanweisungen (engl. *Compound Statement*) Instanzen einer Composite-Klasse sein, deren Kindobjekte so geordnet sein müssen, dass sie das Programm widerspiegeln.

Soll die Einhaltung einer solchen Ordnung gewährleistet sein, dann müssen die Schnittstellen für den Zugriff auf die Kindobjekte und deren Verwaltung in der Lage sein, die Reihenfolge der Objekte entsprechend zu organisieren. Hierbei kann sich das Design Pattern *Iterator* (*Iterator*, siehe [Abschnitt 5.4](#)) als nützlich erweisen.

7. *Performancesteigerung durch Caching (Zwischenspeichern)*. Wenn Kompositionen häufig traversiert oder durchsucht werden müssen, können die für ihre Kindobjekte relevanten Traversierungs- und Suchinformationen in der Composite-Klasse zwischengespeichert werden – und zwar entweder in Form der tatsächlichen Resultate oder lediglich in Form von Informationen, die die Traversierung bzw. Suche abkürzen. So könnte z. B. die Picture-Klasse aus dem im Abschnitt »Motivation« vorgestellten Beispiel den Begrenzungsrahmen ihrer Kindobjekte zwischenspeichern und somit diesbezügliche Zeichen- oder Suchoperationen von vornherein verhindern, wenn ihre Kindobjekte im aktuellen Fenster nicht sichtbar sind.

Änderungen an einer Komponente bedingen eine Invalidierung des Zwischenspeichers ihres Elternobjekts – und das funktioniert am besten, wenn die Komponenten ihre Elternobjekte kennen. Das bedeutet: Zur Verwendung des Caching-Verfahrens muss eine Schnittstelle definiert werden, die den zusammengesetzten Objekten mitteilt, dass ihre Zwischenspeicher ungültig geworden sind.

8. *Zuständigkeit für die Entfernung von Komponenten*. In Sprachen, die keine automatische Speicherbereinigung (engl. *Garbage Collection*) anbieten, ist es in der Regel am besten, dem Composite-Objekt selbst die Zuständigkeit für das Löschen seiner Kindobjekte zu übertragen, wenn es selbst entfernt wird. Eine Ausnahme von dieser Regel bilden allerdings unveränderliche Leaf-Objekte, die dementsprechend gemeinsam genutzt werden können.
9. *Optimale Datenstruktur zum Speichern von Komponenten*. Composite-Objekte können eine Vielzahl von Datenstrukturen verwenden, um ihre Kindobjekte einschließlich verkettete Listen, Bäumen, Arrays und Hashtabellen zu

speichern. Die Wahl der Datenstruktur hängt (wie immer) von der gewünschten zu erzielenden Effizienz ab. Tatsächlich ist die Verwendung einer allgemeinen Datenstruktur allerdings gar nicht zwingend erforderlich. Manchmal haben Composite-Objekte für jedes Kindesobjekt eine eigene Variable, wenngleich dies bedeutet, dass jede UnterkLASSE der Composite-KLASSE eine eigene Verwaltungsschnittstelle implementieren muss. Ein entsprechendes Beispiel ist in den Ausführungen zum Design Pattern *Interpreter* (*Interpreter*, siehe [Abschnitt 5.3](#)) beschrieben.

Beispielcode

Hardwaregeräte wie Computer- und Stereosysteme sind häufig in Form einer Teil-Ganzes- (engl. *Part-Whole Hierarchy*) oder Enthaltsseinshierarchie (engl. *Containment Hierarchy*) organisiert. So kann ein Gehäuse beispielsweise Laufwerke und Platinen enthalten, ein Bus kann Steckkarten enthalten und ein Schrank kann wiederum Gehäuse, Busse usw. enthalten. Solche Strukturen können mithilfe des Design Patterns *Composite* (*Kompositum*) auf natürliche Art und Weise modelliert werden.

Im nachfolgenden Beispiel definiert die Klasse `Equipment` eine Schnittstelle für sämtliche Geräte in einer Teil-Ganzes-Hierarchie:

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment* );
    virtual void Remove(Equipment* );
    virtual Iterator<Equipment*>* CreateIterator();

protected:
    Equipment(const char* );

private:
    const char* _name;
};
```

`Equipment` deklariert Operationen, die die Attribute eines Geräts zurückgeben, also

z. B. seine Leistungsaufnahme und das Kostenaufkommen, und die Unterklassen implementieren diese Operationen für spezifische Gerätearten. Außerdem deklariert die Klasse Equipment eine Operation CreateIterator, die einen Iterator (siehe [Anhang C](#)) für den Zugriff auf ihre individuellen Bestandteile bereitstellt. Die Standardimplementierung dieser Operation liefert einen NullIterator zurück, der über die leere Menge iteriert.

Zu den Unterklassen von Equipment können auch Leaf-Klassen gehören, die die Laufwerke, integrierte Schaltkreise sowie Switches repräsentieren:

```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char* );
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

Die Klasse CompositeEquipment ist die Basisklasse für Geräte, die weitere Geräte enthalten. Zudem ist sie auch eine Unterklasse von Equipment:

```
class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment* );
    virtual void Remove(Equipment* );
    virtual Iterator<Equipment*>* CreateIterator();

protected:
    CompositeEquipment(const char* );

private:
    List<Equipment*> _equipment;
};
```

In dieser Klasse sind die Operationen für den Zugriff auf die enthaltenen Geräte und deren Verwaltung definiert. Mit der Operation Add werden der in der Membervariablen _equipment gespeicherten Geräteliste weitere Geräte hinzugefügt, während Remove sie daraus entfernt. Die Operation CreateIterator gibt einen Iterator (genauer gesagt eine Instanz von ListIterator) zurück, der diese Liste dann

traversiert.

Eine Standardimplementierung von NetPrice könnte CreateIterator dazu nutzen, die Nettopreise der enthaltenen Geräte aufzuaddieren:

```
Currency CompositeEquipment::NetPrice () {
    Iterator<Equipment*>* i = CreateIterator();
    Currency total = 0;

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
```

Hinweis

Leider wird allzu häufig vergessen, den Iterator nach Gebrauch wieder zu löschen – wie sich dies vermeiden lässt, zeigt das Design Pattern *Iterator (Iterator*, siehe [Abschnitt 5.4](#)).

Nun kann ein Computergehäuse als eine mit Chassis benannte Unterklasse von CompositeEquipment repräsentiert werden. Diese Unterklasse erbt die kindbezogenen Operationen von CompositeEquipment:

```
class Chassis : public CompositeEquipment {
public:
    Chassis(const char* );
    virtual ~Chassis();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

Anschließend können auf ähnliche Weise noch weitere Gerätbehälter wie Cabinet und Bus definiert werden. Damit sind nun alle für den Bau eines (recht einfachen) PCs benötigten Gerätekomponenten beisammen:

```
Cabinet* cabinet = new Cabinet("PC Cabinet");
Chassis* chassis = new Chassis("PC Chassis");
```

```

cabinet->Add(chassis);

Bus* bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

count << "The net price is " << chassis->NetPrice() << endl;

```

Praxisbeispiele

Beispiele für den Einsatz des Design Patterns *Composite (Kompositum)* finden sich in fast allen objektorientierten Systemen. Die ursprüngliche *View*-Klasse des **Smalltalk-MVC-Architekturschemas** [KP88] war ein Composite-Objekt – und fast jedes Toolkit oder Framework für Benutzeroberflächen hat diese Konvention beibehalten, so auch **ET++** (mit VObjects [WGM88]) und **InterViews** (Styles [LCI+92], Graphics [VL88] und Glyphs [CL90]). Interessanterweise besaß die originale *View*-Klasse des MVC-Schemas einen Subview-Satz, d. h.: *View* war sowohl die Component- als auch die Composite-Klasse. Im Release 4.0 von Smalltalk-80 wurde das MVC-Schema mit einer *visualComponent*-Klasse überarbeitet, die die Unterklassen *View* und *CompositeView* enthielt.

Das **RTL-Smalltalk Compiler Framework** [JML92] macht ausgiebigen Gebrauch von dem Design Pattern *Composite (Kompositum)*: *RTLExpression* ist eine Component-Klasse für Parse-Bäume. Sie besitzt Unterklassen, wie z. B. *BinaryExpression*, die *RTLExpression*-Kindobjekte enthalten und eine Kompositionsstruktur für Parse-Bäume definieren. Bei *RegisterTransfer* handelt es sich dagegen um die Component-Klasse für die Single-Static-Assigment (SSA)-Übergangsform eines Programms. Ihre *Leaf*-Unterklassen definieren verschiedene statische Zuweisungen, wie z. B.:

- primitive Zuweisungen, die eine Operation auf zwei Register anwenden und das Resultat dann einem dritten Register zuweisen,
- eine Zuweisung mit einem Quellregister, aber ohne Zielregister – wodurch angezeigt wird, dass das Register *nach der Rückgabe* einer Routine verwendet wird,
- eine Zuweisung mit einem Zielregister, aber ohne Quellregister – wodurch angezeigt wird, dass das Register *vor der Ausführung* der Operation zugewiesen wird.

Eine weitere Unterklasse namens `RegisterTransferSet` ist eine Composite-Klasse zur Darstellung von Zuweisungen, die mehrere Register gleichzeitig ändern.

Auch in der Finanzwelt findet sich mit einem Portfolio, das mehrere individuelle Vermögenswerte zusammenfasst, ein Beispiel für das Design Pattern *Composite (Kompositum)*: Zur Erstellung komplexer Aggregationen von Vermögenswerten kann das Portfolio als composite-Objekt implementiert werden, das der Schnittstelle eines individuellen Vermögenswertes entspricht [BE93].

Das Design Pattern *Command (Befehl*, siehe [Abschnitt 5.2](#)) beschreibt, wie sich Command-Objekte mithilfe einer Composite-Klasse namens `MacroCommand` zusammenführen und sequenzieren lassen.

Verwandte Patterns

Die Verknüpfung von Komponente und Elternobjekt wird häufig zur Implementierung einer *Chain of Responsibility (Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) genutzt.

Das Design Pattern *Decorator (Dekorierer*, siehe [Abschnitt 4.4](#)) wird vielfach gemeinsam mit dem Pattern *Composite (Kompositum)* verwendet. Wenn Decorator- und Composite-Klassen zusammen eingesetzt werden, haben sie in aller Regel eine gemeinsame Basisklasse. Dementsprechend müssen die Decorator-Objekte die Component-Schnittstelle mit Operationen wie `Add`, `Remove` und `GetChild` unterstützen.

Das Design Pattern *Flyweight (Fliegengewicht*, siehe [Abschnitt 4.6](#)) ermöglicht die gemeinsame Nutzung von Komponenten, allerdings können diese ihre Elternobjekte dann nicht mehr referenzieren.

Das Design Pattern *Iterator (Iterator*, siehe [Abschnitt 5.4](#)) kann zur Traversierung von Composite-Objekten verwendet werden.

Und das Design Pattern *Visitor (Besucher*, siehe [Abschnitt 5.11](#)) lokalisiert Operationen und Verhaltensweisen, die andernfalls über die composite- und Leaf-Klassen verteilt wären.

4.4 Decorator (Dekorierer)

Objektbasiertes Strukturmuster

Zweck

Dynamische Erweiterung der Funktionalität eines Objekts. Decorator-Objekte stellen hinsichtlich der Ergänzung einer Klasse um weitere Zuständigkeiten eine flexible Alternative zur Unterklassenbildung dar.

Auch bekannt als

Wrapper (Gebundener Umwickler)

Motivation

Es gibt Situationen, in denen lediglich einzelne Objekte mit einer erweiterten Funktionalität ausgestattet werden sollen, nicht aber eine ganze Klasse. Beispielsweise sollte ein Toolkit für grafische Benutzeroberflächen die Option zur Verfügung stellen, jede beliebige Schnittstellenkomponente mit zusätzlichen Eigenschaften wie Rahmen oder Funktionen, etwa einer Scrollfunktion, versehen zu können.

Eine Möglichkeit für die Zuweisung weiterer Zuständigkeiten bietet das Vererbungsprinzip: Durch das Erben eines Rahmens von einer anderen Klasse wird auch jede Unterklasseninstanz mit einem Rahmen umgeben. Dieses Verfahren ist allerdings insofern unflexibel, als die Auswahl des Rahmens statisch erfolgt, d. h., ein Client hat keinen Einfluss darauf, wie und wann die Komponente mit einem Rahmen dekoriert wird.

Ein flexiblerer Ansatz ist dagegen, die Komponente in ein anderes Objekt einzubinden, das den Rahmen ergänzt. Das umschließende Objekt wird als **Decorator (Dekorierer)** bezeichnet. Da der Decorator der Schnittstelle der zu dekorierten Komponente entspricht, ist seine Präsenz für die Clients der Komponente transparent. Das Decorator-Objekt leitet Requests an die Komponente

weiter und kann zudem vor oder nach der Weiterleitung zusätzliche Operationen (wie z. B. das Zeichnen eines Rahmens) ausführen. Diese Transparenz ermöglicht die rekursive Schachtelung von Decorator-Objekten und gestattet somit die uneingeschränkte Ergänzung weiterer Funktionalität.

Als Beispiel soll an dieser Stelle ein TextView-Objekt dienen, das Text in einem Fenster ausgibt. Dieses Objekt verfügt standardmäßig nicht über Scrollleisten, weil sie möglicherweise nicht ständig benötigt werden. Werden sie aber doch gebraucht, dann lassen sie sich über ein ScrollDecorator-Objekt ergänzen. Außerdem soll die Textausgabe mit einem dicken schwarzen Rahmen versehen werden. Dieser kann wiederum mithilfe eines BorderDecorator-Objekts hinzugefügt werden. Im Grunde werden also einfach passende Decorator-Objekte mit dem Objekt TextView zusammengeführt, um das gewünschte Ergebnis zu erzeugen.

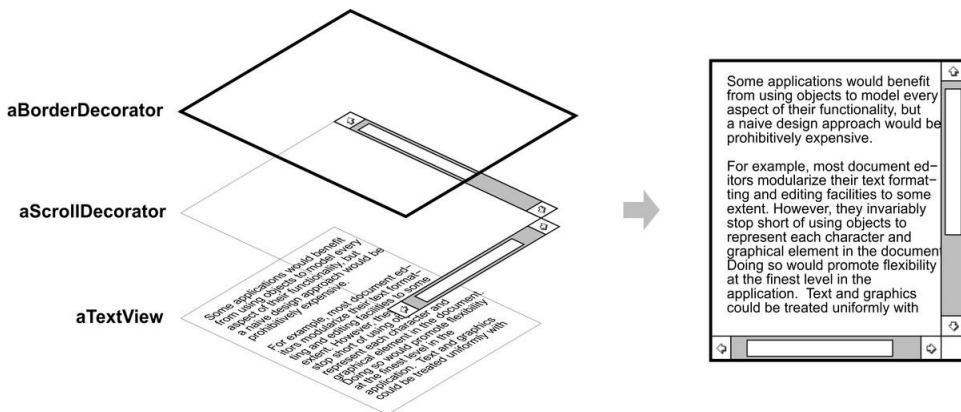


Abb. 4.15: Zusammenführung von *TextView*- und *Decorator*-Objekten

Das in [Abbildung 4.16](#) dargestellte Objektdiagramm zeigt, wie ein TextView-Objekt mit einem BorderDecorator- und einem ScrollDecorator-Objekt zusammengeführt werden kann, so dass im Ergebnis eine umrahmte, scrollbare Textanzeige generiert wird:



Abb. 4.16: Zusammenführung der Objekte *TextView* und *BorderDecorator* sowie *ScrollDecorator*

Die Klassen ScrollDecorator und BorderDecorator sind Unterklassen von Decorator, einer abstrakten Klasse für visuelle Komponenten, die andere visuelle Komponenten dekorieren.

`VisualComponent` ist die abstrakte Klasse für visuelle Objekte und definiert deren Schnittstelle zum Zeichnen und zur Ereignisbehandlung (engl. *Event Handling*). Beachtenswert ist hier vor allem, dass die `Decorator`-Klasse Requests zum Zeichnen einfach an ihre Komponenten weiterleitet und die Unterklassen von `Decorator` diese Operation erweitern können.

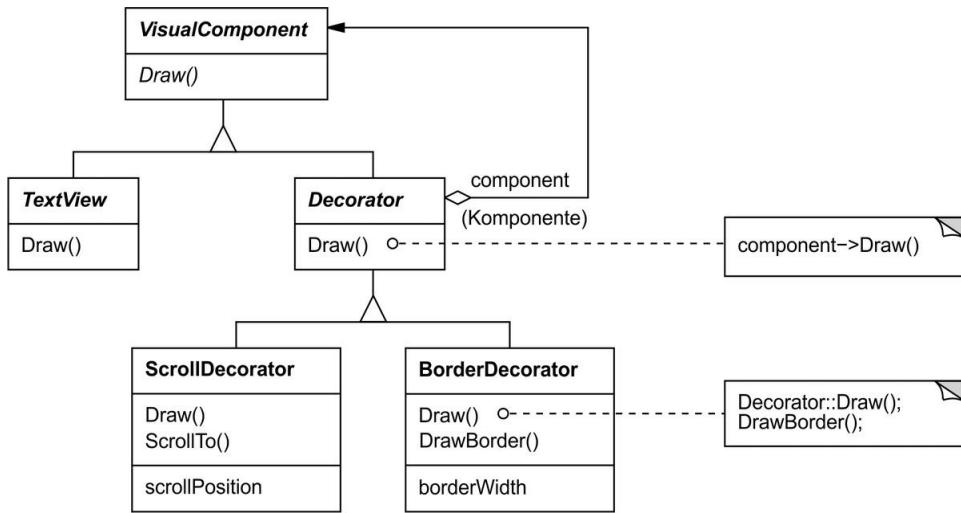


Abb. 4.17: Schematische Darstellung der abstrakten Klasse `visualComponent`

Die `Decorator`-Unterklassen sind in der Lage, zur Einbindung neuer Funktionalität beliebige weitere Operationen zu ergänzen. Beispielsweise ermöglicht die `ScrollDecorator`-Operation `ScrollTo` anderen Objekten, die Benutzeroberfläche zu scrollen, sofern ihnen bekannt ist, dass die Schnittstelle ein `ScrollDecorator`-Objekt beinhaltet. Der maßgebliche Aspekt am Design Pattern *Decorator (Dekorierer)* ist, dass die `Decorator`-Objekte überall dort in Erscheinung treten können, wo auch ein `VisualComponent`-Objekt verwendet werden kann. Aus diesem Grund können Clients im Allgemeinen nicht zwischen einer dekorierten und einer nicht dekorierten Komponente unterscheiden – und sind damit auch in keiner Weise von der Dekorierung abhängig.

Anwendbarkeit

Die Nutzung des Design Patterns *Decorator (Dekorierer)* bietet sich an, wenn

- einzelne Objekte dynamisch und transparent mit weiterer Funktionalität ausgestattet werden sollen, ohne dass andere Objekte davon betroffen sind.
- Funktionalität ergänzt werden soll, die aber auch wieder entfernt werden kann.

- eine Erweiterung mittels Unterklassenbildung praktisch nicht durchführbar ist. Manchmal sind zwar zahlreiche voneinander unabhängige Erweiterungen möglich, deren Unterstützung in jeder möglichen Kombination allerdings zu einem explosiven zahlenmäßigen Anstieg der Unterklassen führen würde. Oder eine Klassendefinition ist versteckt oder steht aus anderen Gründen nicht für die Unterklassenbildung zur Verfügung.

Struktur

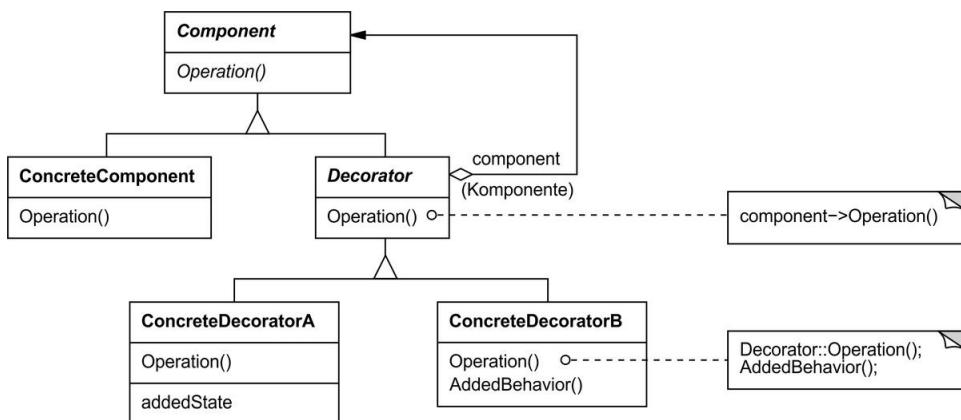


Abb. 4.18: Die Struktur des Design Patterns *Decorator* (*Dekorierer*)

Teilnehmer

- **Component** (`VisualComponent`)
 - Definiert die Schnittstelle für Objekte, die dynamisch um zusätzliche Funktionalität erweitert werden können.
- **ConcreteComponent** (`TextView`)
 - Definiert ein Objekt, das um zusätzliche Funktionalität erweitert werden kann.
- **Decorator**
 - Enthält eine Referenz auf ein **Component**-Objekt und definiert eine Schnittstelle, die der **Component**-Schnittstelle entspricht.

- **ConcreteDecorator** (`BorderDecorator`, `ScrollDecorator`)
 - Stattet die Komponente mit zusätzlicher Funktionalität aus.

Interaktionen

- Die Klasse `Decorator` leitet Requests an ihr `Component`-Objekt weiter und führt ggf. vor und/oder nach der Weiterleitung noch zusätzliche Operationen aus.

Konsequenzen

Das Design Pattern *Decorator* (*Dekorierer*) bringt wenigstens zwei bedeutende Vorteile und zwei wichtige Verbindlichkeiten mit sich:

1. *Mehr Flexibilität als bei der statischen Vererbung.* Das Design Pattern *Decorator* (*Dekorierer*) bietet flexiblere Anpassungsmöglichkeiten in Bezug auf die Objektfunktionalität, als dies im Rahmen der statischen (Mehrfach-)Vererbung der Fall ist: Neue oder nicht länger erwünschte Funktionen können ganz einfach durch Hinzufügen bzw. Löschen entsprechender `Decorator`-Objekte zur Laufzeit ergänzt bzw. entfernt werden. Im Gegensatz dazu muss bei der Vererbung für jede weitere Funktion eine neue Klasse erzeugt werden (z. B. `BorderedScrollView`, `BorderedTextView` etc.) – was in der Konsequenz zu einem rapiden Anstieg der Klassenanzahl und damit auch der Komplexität des Systems führt. Ein ebenfalls nicht zu unterschätzender Vorteil der Bereitstellung unterschiedlicher `Decorator`-Klassen für eine spezifische `Component`-Klasse ist außerdem, dass sich die Funktionen auf diese Art gut miteinander kombinieren und aufeinander abstimmen lassen.

Darüber hinaus erleichtern die `Decorator`-Objekte auch die zweifache Verwendung von Eigenschaften: Um beispielsweise die Klasse `TextView` mit einem doppelten Rahmen zu versehen, muss sie einfach nur um zwei `BorderDecorator`-Objekte erweitert werden – das zweifache Erben von einer `Border`-Klasse ist dagegen bestenfalls fehleranfällig.

2. *Vermeidung funktionsüberladener Klassen in den oberen Hierarchieebenen.* Dank der `Decorator`-Objekte ist es möglich, weitere Funktionen erst zu dem Zeitpunkt zu ergänzen, an dem sie auch tatsächlich benötigt werden. Statt also versuchen zu müssen, alle irgendwann in der Zukunft eventuell benötigten

Funktionen durch eine komplexe anpassbare Klasse zu unterstützen, reicht es hier aus, eine einfache Klasse zu definieren und die benötigte Funktionalität mittels Decorator-Objekten inkrementell zu ergänzen. Die Gesamtfunktionalität lässt sich aus einfachen einzelnen Bestandteilen zusammenstellen – so dass die Anwendung nicht unnötigerweise die Last nicht genutzter Funktionen tragen muss. Ebenso einfach ist es auch, neue Arten von Decorator-Objekten unabhängig von den Klassen der von ihnen erweiterten Objekte zu definieren – selbst für nicht vorhersehbare Erweiterungen. Ansonsten werden bei der Erweiterung einer komplexen Klasse in aller Regel Details offengelegt, die gar nicht mit der ergänzten Funktionalität in Zusammenhang stehen.

3. *Ein Decorator-Objekt ist nicht mit seiner Komponente identisch.* Ein Decorator-Objekt dient seiner Komponente als eine durchsichtige Umhüllung (engl. *Transparent Enclosure*). Was die Identität des Objekts angeht, ist eine dekorierte Komponente nicht mit der Komponente selbst identisch – insofern sollte die Verwendung von Decorator-Objekten auch nicht auf der Objektidentität basieren.
4. *Viele kleine Objekte.* Ein Design, in dem Decorator-Objekte verwendet werden, resultiert häufig in einem System mit vielen kleineren Objekten, die zudem alle sehr ähnlich sind. Sie unterscheiden sich nur durch die Art und Weise, wie sie miteinander verbunden sind – nicht jedoch durch ihre Klasse oder den Wert ihrer Variablen. Wer sich mit diesen Systemen auskennt, wird sie problemlos anpassen können – wer sie jedoch nicht beherrscht, wird Schwierigkeiten haben, sie zu durchschauen und zu debuggen.

Implementierung

Bei der Anwendung des Design Patterns *Decorator (Dekorierer)* empfiehlt es sich, die folgenden Aspekte zu berücksichtigen:

1. *Schnittstellenkonformität.* Die Schnittstelle eines Decorator-Objekts muss mit der Schnittstelle der Komponente, die es dekoriert, konform sein. ConcreteDecorator-Klassen müssen daher von einer gemeinsamen Klasse erben (zumindest in C++).
2. *Wegfall der abstrakten Decorator-Klasse.* Zur Ergänzung einer einzelnen Funktion muss keine abstrakte Decorator-Klasse definiert werden. Solche Situationen treten häufig dann auf, wenn keine von Grund auf neu erstellte, sondern eine bereits vorhandene Klassenhierarchie genutzt wird. In diesem Fall

kann die Funktionalität der allgemeinen `Decorator`-Klasse zum Weiterleiten von Requests an die Komponente mit der Funktionalität einer `ConcreteDecorator`-Klasse zusammengeführt werden.

3. *Leichtgewichtige Komponentenklassen.* Damit eine Schnittstellenkonformität gewährleistet ist, müssen Komponenten und `Decorator`-Objekte von einer gemeinsamen `Component`-Klasse abgeleitet sein – die wiederum möglichst leichtgewichtig bleiben muss, d. h., sie sollte nicht auf die Datenspeicherung, sondern auf die Definition einer Schnittstelle ausgerichtet sein. Zu diesem Zweck ist es hilfreich, die Definition der Datenrepräsentation an Unterklassen zu delegieren, ansonsten könnte die Komplexität der `Component`-Klasse die `Decorator`-Objekte zu schwergewichtig machen, um sie in größerer Zahl nutzen zu können. Wird die `Component`-Klasse mit sehr viel Funktionalität ausgestattet, erhöht sich außerdem die Wahrscheinlichkeit, dass konkrete Unterklassen die Last der Funktionen tragen müssen, die sie gar nicht brauchen.
4. *Modifizierung des Objektäußeren kontra Modifizierung des Objektinneren.* Man kann sich ein `Decorator`-Objekt in etwa wie die äußere Hülle eines Objekts vorstellen, das sein Verhalten verändert. Eine Alternative wäre, das Innere des Objekts zu ändern, was zum Beispiel beim Einsatz des Design Patterns *Strategy (Strategie)*, siehe [Abschnitt 5.9](#)) geschieht.

`Strategy`-Objekte sind in Situationen, in denen die `Component`-Klasse schwergewichtig und die Anwendung des Patterns *Decorator (Dekorierer)* daher zu kostspielig ist, die bessere Wahl: Das Design Pattern *Strategy (Strategie)*, siehe [Abschnitt 5.9](#)) gewährleistet, dass die Komponente einen Teil ihres Verhaltens einem separaten `Strategy`-Objekt überlässt, das dann bedarfsweise ersetzt werden kann, um die Funktionalität der Komponente zu ändern oder zu erweitern.

Zum Beispiel könnten verschiedene Rahmenstile unterstützt werden, indem die Komponente das Zeichnen des Rahmens an ein separates `Border`-Objekt delegiert. Das `Border`-Objekt ist ein `Strategy`-Objekt, das eine Strategie zum Zeichnen eines Rahmens kapselt. Durch die Erweiterung der Anzahl der `Strategy`-Objekte von eins auf eine unbegrenzte Anzahl wird derselbe Effekt erzielt wie beim rekursiven Verschachteln von `Decorator`-Objekten.

So verwalten die grafischen Komponenten (»Views« genannt) in MacApp 3.0 [App89] und Bedrock [Sym93a] beispielsweise eine Liste von »Verzierungsobjekten« (engl. *Adorners*), die eine `view`-Komponente mit zusätzlichen Ausgestaltungselementen wie Rahmen versehen können. Verfügen

Views über solche Verzierungsobjekte, dann gestattet sie ihnen, zusätzliche Ausgestaltungselemente zu zeichnen. MacApp und Bedrock müssen diesen Ansatz verfolgen, weil ihre view-Klassen schwergewichtig sind – und es daher viel zu kostenaufwendig wäre, vollwertige Views zu verwenden, um lediglich einen Rahmen zu ergänzen.

Da das Design Pattern *Decorator (Dekorierer)* nur das Äußere einer Komponente ändert, muss die Komponente selbst keinerlei Kenntnis von ihren Decorator-Objekten haben, d. h., die Decorator-Objekte sind aus Sicht der Komponente transparent.

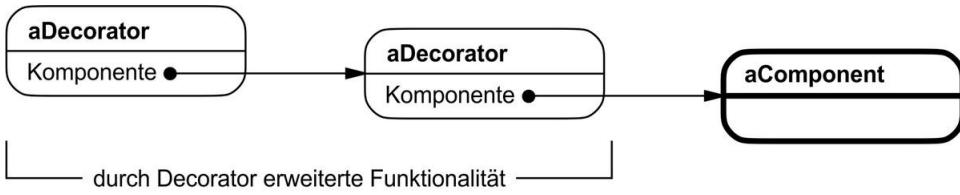


Abb. 4.19: Funktionalitätserweiterung mit *Decorator*-Objekten

Im Gegensatz dazu hat die Komponente bei der *Strategy*-Variante Kenntnis von möglichen Erweiterungen und muss die entsprechenden *Strategy*-Objekte daher auch referenzieren und verwalten.



Abb. 4.20: Funktionalitätserweiterung mit *Strategy*-Objekten

Bei dem auf dem Design Pattern *Strategy (Strategie*, siehe [Abschnitt 5.9](#)) basierenden Ansatz muss die Komponente gegebenenfalls für neue Erweiterungen modifiziert werden. Andererseits kann ein *Strategy*-Objekt aber auch eine eigene spezialisierte Schnittstelle besitzen, während die Schnittstelle eines *Decorator*-Objekts mit der Schnittstelle der Komponente konform sein muss. Ein *Strategy*-Objekt zum Zeichnen eines Rahmens braucht beispielsweise nur die passende Schnittstelle zu definieren (*DrawBorder*, *GetWidth* etc.), d. h., die Schnittstelle der *Strategy*-Klasse kann auch dann noch schwergewichtet sein, wenn die *component*-Klasse schwergewichtet ist.

MacApp und Bedrock nutzen diesen Ansatz über die reine Verzierung von Views hinaus auch zur Verbesserung des Event-Handling-Verhaltens von

Objekten. In beiden Systemen verwaltet ein View eine Liste von »Verhaltensobjekten«, die Ereignisse verändern und abfangen können. Der View gestattet allen registrierten Verhaltensobjekten, das Ereignis vor den nicht registrierten Verhaltensobjekten zu bearbeiten – und behandelt sie damit effektiv vorrangig. So kann ein View beispielsweise mit einer speziellen Unterstützung für den Umgang mit Tastatureingaben dekoriert werden, indem ein Verhaltensobjekt registriert wird, das Tastenereignisse abfängt und bearbeitet.

Beispielcode

Der folgende Code veranschaulicht die Implementierung von Decorator-Objekten für Benutzeroberflächen in C++. Den Ausgangspunkt bildet in diesem Fall eine Component-Klasse namens `VisualComponent`:

```
class VisualComponent {  
public:  
    VisualComponent();  
  
    virtual void Draw();  
    virtual void Resize();  
    // ...  
};
```

Nun wird eine Unterklasse `Decorator` dieser Klasse `VisualComponent` definiert, von der zwecks Generierung verschiedener Dekorierungen wiederum weitere Unterklassen abgeleitet werden:

```
class Decorator : public VisualComponent {  
public:  
    Decorator(VisualComponent*);  
  
    virtual void Draw();  
    virtual void Resize();  
    // ...  
  
private:  
    VisualComponent* _component;  
};
```

`Decorator` dekoriert die von der im Konstruktor initialisierten Instanzvariablen `_component` referenzierte Klasse `VisualComponent`. Für jede Operation in der Schnittstelle von `VisualComponent` definiert `Decorator` eine Standardimplementierung, die den Request an `_component` weiterleitet:

```

void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}

```

Die Unterklassen von Decorator definieren spezifische Dekorationen.

Beispielsweise ergänzt die Klasse BorderDecorator einen Rahmen um die von ihr umschlossene Komponente. BorderDecorator ist eine Unterklasse von Decorator, die zum Zeichnen des Rahmens die `Draw`-Operation überschreibt. Darüber hinaus definiert BorderDecorator außerdem eine private Hilfsoperation `DrawBorder`, die den eigentlichen Vorgang des Zeichnens übernimmt. Alle anderen Operationsimplementierungen erbt die Unterklasse von Decorator:

```

class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderwidth);

    virtual void Draw();

private:
    void DrawBorder(int);

private:
    int _width;
};

void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}

```

Die Implementierung von `ScrollDecorator` und `DropShadowDecorator`, die die visuelle Komponente um eine Scrollleiste bzw. einen Schlagschatten ergänzen, würde dann auf ähnliche Art und Weise erfolgen.

Nun können zur Umsetzung verschiedener Dekorierungen Instanzen dieser Klassen zusammengefügt werden. Der folgende Code zeigt, wie sich mithilfe von Decorator-Objekten ein umrahmter und scrollfähiger `TextView` erzeugen lässt.

Dazu muss zunächst einmal eine Möglichkeit gefunden werden, eine visuelle Komponente in ein `Window`-Objekt einzubetten. Hier wird angenommen, dass die `Window`-Klasse zu diesem Zweck eine `SetContents`-Operation bereitstellt:

```
void Window::SetContents (VisualComponent* contents) {
```

```
    // ...
}
```

Anschließend kann zum einen das TextView-Objekt und zum anderen ein Fenster erzeugt werden, in das es eingebettet wird:

```
Window* window = new Window;
TextView* textView = new TextView;
```

TextView ist ein visualComponent-Objekt, das es ermöglicht, den View in das Fenster einzubetten:

```
window->SetContents(textView);
```

Da das TextView-Objekt allerdings umrahmt und scrollfähig sein soll, wird es zunächst entsprechend dekoriert und erst dann in das Fenster eingefügt:

```
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

Weil window über die visualComponent-Schnittstelle auf seine Inhalte zugreift, hat es keine Kenntnis von der Existenz des Decorator-Objekts. Der Client kann das TextView-Objekt jedoch trotzdem überwachen, wenn er direkt mit ihm interagieren muss, z. B. um Operationen aufzurufen, die nicht Teil der visualComponent-Schnittstelle sind. Auch Clients, die sich auf die Identität der Komponente verlassen, sollten es direkt referenzieren.

Praxisbeispiele

Viele objektorientierte Toolkits für Benutzeroberflächen verwenden Decorator-Objekte zur grafischen Ausgestaltung von Widgets. Beispiele dafür sind unter anderem **InterViews** [LVC89, LCI+92], **ET++** [WGM88] und die Klassenbibliothek **ObjectWorks\Smalltalk** [Par90]. Exotischere Anwendungen des Design Patterns *Decorator (Dekorierer)* sind **DebuggingGlyph** von InterViews und **PassivityWrapper** von **ParcPlace Smalltalk**. Ein DebuggingGlyph gibt vor und nach der Weiterleitung eines Layout-Requests an seine Komponente Debugging-Daten aus. Diese Trace-Daten können zur Analyse und zum Debugging des Layout-Verhaltens von Objekten in einer komplexen Komposition genutzt werden. Und der PassivityWrapper kann userseitige Interaktionen mit der Komponente ein- oder

ausschalten.

Das Design Pattern *Decorator* (*Dekorierer*) ist allerdings keinesfalls nur auf grafische Benutzeroberflächen beschränkt, wie das folgende auf den **ET++ Streamklassen** [WGM88] basierende Beispiel zeigt.

Streams stellen in den meisten I/O-Bibliotheken eine fundamentale Abstraktion dar. Ein Stream kann eine Schnittstelle zur Konvertierung von Objekten in eine Abfolge von Bytes oder Zeichen bereitstellen. Sie ermöglicht es, ein Objekt für den späteren Abruf in eine Datei oder einen String in den Speicher zu schreiben. Eine direkte Möglichkeit, dies zu tun, ist die Definition einer abstrakten Stream-Klasse mit den Unterklassen `MemoryStream` und `FileStream`. An dieser Stelle soll jedoch Folgendes möglich sein:

- Komprimierung der Streamdaten mithilfe verschiedener Komprimierungsalgorithmen (Lauflängenkodierung (engl. *Run-Length Encoding*), Lempel-Ziv etc.).
- Reduzierung der Streamdaten in 7-Bit-ASCII-Zeichen, so dass sie über einen ASCII-Kommunikationskanal übertragen werden können.

Das Design Pattern *Decorator* (*Dekorierer*) bietet eine elegante Möglichkeit, die Streams um diese Funktionalität zu erweitern. [Abbildung 4.21](#) zeigt eine Lösung für dieses Problem:

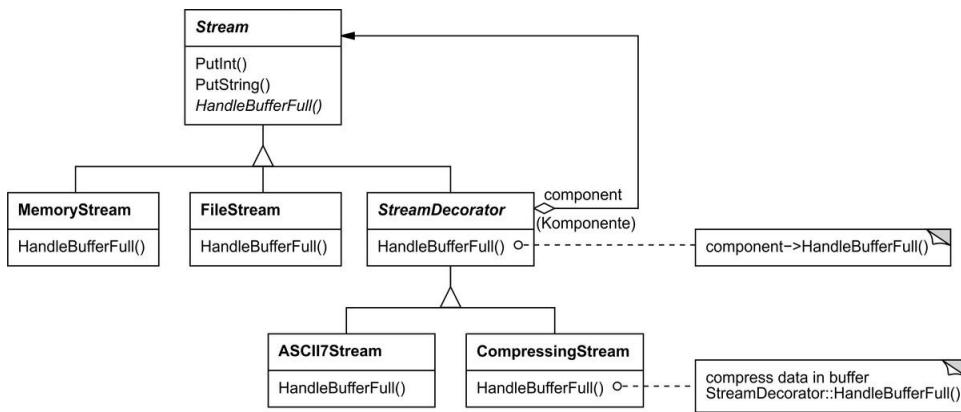


Abb. 4.21: Stream-Klasse mit den Unterklassen `MemoryStream` und `FileStream`

Die abstrakte Stream-Klasse unterhält einen internen Puffer und stellt Operationen zum Speichern von Daten im Stream zur Verfügung (`PutInt`, `PutString`). Sobald der Puffer voll ist, ruft der Stream die abstrakte Operation `HandleBufferFull` auf, die dann den aktuellen Datentransfer durchführt. Zur Übertragung des Pufferinhalts

in eine Datei wird diese Operation von der `FileStream`-Version überschrieben.

In diesem Fall ist `StreamDecorator` die maßgebliche Klasse. Sie hält eine Referenz auf einen Komponentenstream vor und leitet Requests an ihn weiter. Die `StreamDecorator`-Unterklassen überschreiben `HandleBufferFull` und führen weitere Operationen aus, bevor sie schließlich die `StreamDecorator`-eigene Operation `HandleBufferFull` aufrufen.

Die Unterklasse `CompressingStream` komprimiert zum Beispiel die Daten, und die Unterklasse `ASCII7Stream` konvertiert sie in 7-Bit-ASCII. Um nun einen `FileStream` zu erzeugen, der seine Daten komprimiert und die komprimierten Binärdaten in 7-Bit-ASCII konvertiert, wird `FileStream` mit einem `CompressingStream`- und einem `ASCII7Stream`-Objekt dekoriert:

```
Stream* aStream = new CompressingStream(
    new ASCII7Stream(
        new FileStream("aFileName")
    )
);
aStream->PutInt(12);
aStream->PutString("aString");
```

Verwandte Patterns

Adapter (*Adapter*, siehe [Abschnitt 4.1](#)): Anders als ein Adapter-Objekt modifiziert ein Decorator-Objekt lediglich die Funktionalität eines Objekts, lässt dessen Schnittstelle aber unangetastet. Ein Adapter-Objekt stattet ein Objekt hingegen mit einer vollkommen neuen Schnittstelle aus.

Composite (*Kompositum*, siehe [Abschnitt 4.3](#)): Ein Decorator-Objekt ist im Prinzip ein degeneriertes Composite-Objekt mit nur einer Komponente. Allerdings ergänzt ein Decorator zusätzliche Funktionalität und ist nicht für die Objektaggregation gedacht.

Strategy (*Strategie*, siehe [Abschnitt 5.9](#)): Während ein Decorator-Objekt die Modifizierung des Äußeren eines Objekts ermöglicht, gestattet ein Strategy-Objekt die Modifizierung des Inneren eines Objekts – sie bieten also zwei alternative Möglichkeiten zur Objektänderung.

4.5 Facade (Fassade)

Objektbasiertes Strukturmuster

Zweck

Bereitstellung einer einheitlichen Schnittstelle zu einem Schnittstellensatz in einem Subsystem. Das Design Pattern *Facade (Fassade)* definiert eine Schnittstelle höherer Ebene, die die Nutzung des Subsystems vereinfacht.

Motivation

Durch die Unterteilung eines Systems in mehrere Subsysteme lässt sich dessen Komplexität deutlich reduzieren. Ein grundsätzliches Designziel besteht darin, die Kommunikation und Abhängigkeiten zwischen den Subsystemen so minimal wie möglich zu gestalten. Dies lässt sich zum Beispiel mithilfe eines Facade-Objekts erreichen, das eine einzelne vereinfachte Schnittstelle zu der allgemeineren Funktionalität eines Subsystems zur Verfügung stellt.

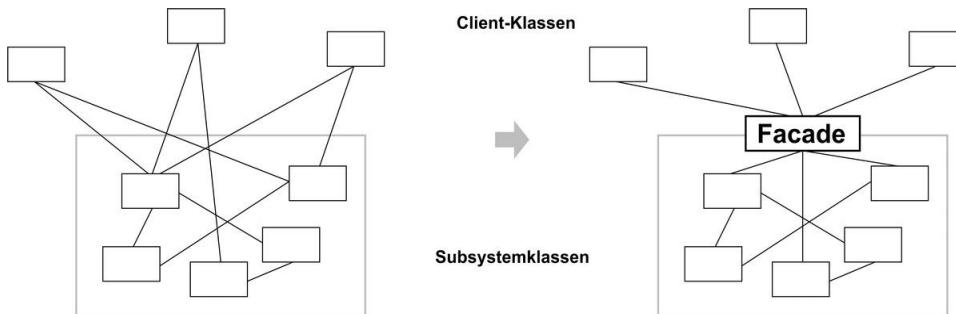


Abb. 4.22: Bereitstellung einer vereinfachten Schnittstelle durch ein Facade-Objekt

Als Beispiel soll hier eine Programmierumgebung dienen, die Anwendungen den Zugriff auf das Subsystem ihres Compilers ermöglicht. Dieses Subsystem enthält Klassen wie Scanner, Parser, ProgramNode, BytecodeStream und ProgramNodeBuilder, die den Compiler implementieren. Möglicherweise müssen einige spezialisierte Anwendungen direkt auf diese Klassen zugreifen können, die meisten Clients eines Compilers befassen sich allerdings im Allgemeinen nicht mit Aufgaben wie dem Parsen von Quellcode und der Codegenerierung – sondern lediglich mit der Kompilierung von Code. Aus ihrer Sicht verkomplizieren die

leistungsfähigen, aber dennoch untergeordneten Schnittstellen im Subsystem des Compilers ihre Arbeit nur.

Zur Bereitstellung einer höherrangigeren Schnittstelle, die die Clients gegen diese Klassen abschirmen kann, enthält das Subsystem des Compilers außerdem eine **Compiler**-Klasse. Sie definiert eine einheitliche Schnittstelle zu der Funktionalität des Compilers und dient als eine Art Fassade: Sie stellt den Clients eine einzige einfache Schnittstelle zum Subsystem des Compilers zur Verfügung und verbindet die Klassen, die die Compiler-Funktionalität implementieren, ohne sie vollständig zu verbergen. Diese Compiler-Fassade bedeutet für die meisten Entwickler eine echte Arbeitserleichterung, ohne jedoch den wenigen Programmierern, die sie wirklich benötigen, den Zugang zur Low-Level-Funktionalität vorzuenthalten.

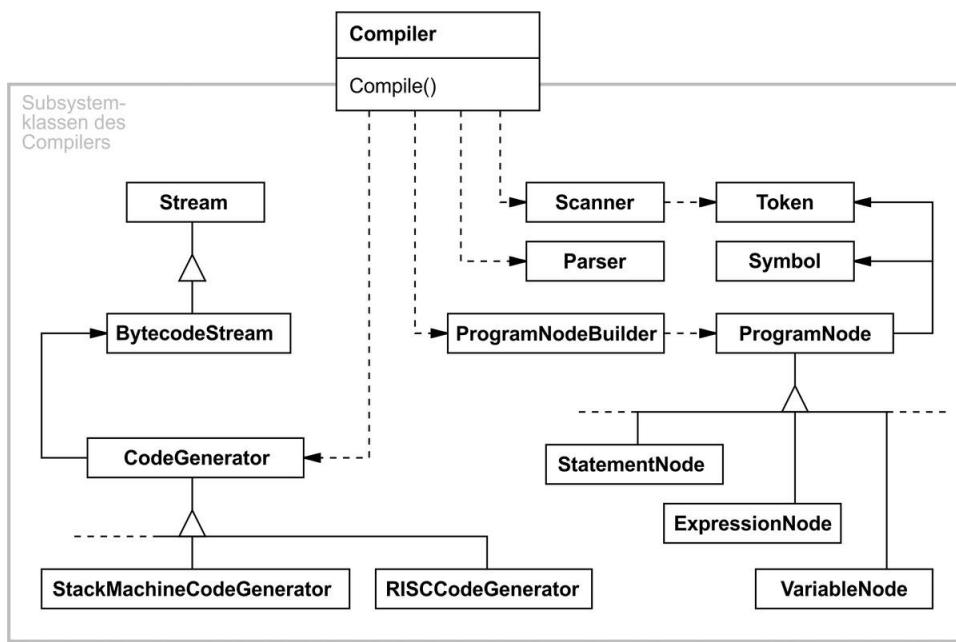


Abb. 4.23: Subsystemklassen des Compilers

Anwendbarkeit

Die Nutzung des Design Patterns *Facade (Fassade)* ist in folgenden Situationen geeignet:

- Wenn eine einfache Schnittstelle zu einem komplexen Subsystem bereitgestellt werden soll. Subsysteme werden im Laufe des Entwicklungsprozesses häufig zunehmend komplexer – und die meisten Patterns resultieren dann in mehreren kleineren Klassen. Dadurch ist das Subsystem zwar einfacher

wiederzuverwenden und anzupassen, andererseits für Clients, die es nicht anpassen müssen, aber auch schwieriger zu benutzen. Das Design Pattern *Facade (Fassade)* kann dagegen eine einfache Standardsicht des Subsystems bereitstellen, die den meisten Clients vollkommen ausreicht. Lediglich Clients, die eine bessere Anpassbarkeit benötigen, müssen hinter die Fassade blicken.

- Wenn viele Abhängigkeiten zwischen den Clients und den Implementierungsklassen einer Abstraktion bestehen. Mithilfe eines Facade-Objekts lässt sich das Subsystem von den Clients und anderen Subsystemen entkoppeln, was im Endeffekt die Unabhängigkeit und Portabilität des Subsystems begünstigt.
- Wenn die Subsysteme in Schichten unterteilt werden sollen. Durch die Anwendung des Patterns *Facade (Fassade)* wird ein Eintrittspunkt zu jeder Subsystemschicht definiert. Zur Vereinfachung von gegebenenfalls zwischen den Subsystemen bestehenden Abhängigkeiten kann ihre Kommunikation in der Art eingeschränkt werden, dass sie ausschließlich über ihre Facade-Objekte erfolgt.

Struktur

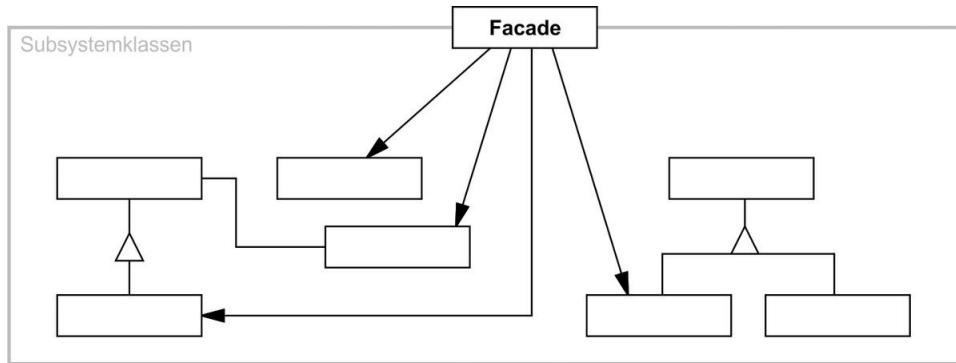


Abb. 4.24: Die Struktur des Design Patterns *Facade (Fassade)*

Teilnehmer

- **Facade (Compiler)**
 - Weiß, welche Subsystemklassen für einen Request zuständig sind.

- Delegiert Client-Requests an die passenden Subsystemobjekte.
- **Subsystemklassen** (Scanner, Parser, ProgramNode etc.)
 - Implementieren die Subsystemfunktionalität.
 - Führen die vom Facade-Objekt zugewiesenen Aufgaben aus.
 - Haben keine Kenntnis von der Fassade und besitzen somit auch keine Referenzen darauf.

Interaktionen

- Die Kommunikation der Clients mit dem Subsystem erfolgt durch die Übersendung der Requests an das Facade-Objekt, das sie anschließend an die passenden Subsystemobjekte weiterleitet. Obwohl das Subsystemobjekt die eigentliche Arbeit erledigt, muss auch das Facade-Objekt gegebenenfalls zusätzlich tätig werden, um seine Schnittstelle auf die Schnittstellen des Subsystems abzubilden.
- Clients, die das Facade-Objekt benutzen, müssen nicht direkt auf dessen Subsystemobjekte zugreifen.

Konsequenzen

Das Design Pattern *Facade (Fassade)* bietet folgende Vorteile:

1. Es schirmt die Clients von den Komponenten des Subsystems ab und reduziert auf diese Weise die Anzahl der Objekte, die von den Clients verwaltet werden müssen, so dass das Subsystem einfacher zu benutzen ist.
2. Es fördert die lose Kopplung zwischen dem Subsystem und seinen Clients. Oftmals sind die Komponenten eines Subsystems stark miteinander gekoppelt. Allerdings ermöglichen lose Kopplungen den Austausch der Komponenten, ohne dass die Clients davon betroffen sind. Facade-Objekte begünstigen die Unterteilung eines Systems sowie der Abhängigkeiten zwischen den Objekten in Schichten und können komplexe oder zirkuläre Abhängigkeiten vollständig aufheben. Dies kann dann wichtig sein, wenn der Client und das Subsystem voneinander unabhängig implementiert werden.

Die Reduzierung der Kompilierungsabhängigkeiten spielt insbesondere in umfangreichen Softwaresystemen eine bedeutende Rolle, um den Zeitaufwand für die erneute Kompilierung im Anschluss an Modifikationen der Subsystemklassen zu minimieren. Durch die Verwendung von Facade-Objekten lässt sich dieser Kompilierungsaufwand nach kleineren Änderungen in einem wichtigen Subsystem erheblich einschränken. Zudem können Facade-Objekte auch die Portierung von Systemen auf andere Plattformen erleichtern, weil sie die Wahrscheinlichkeit verringern, dass die Neuerrichtung eines Subsystems zugleich die Neuerrichtung aller anderen Subsysteme nach sich zieht.

3. Die bedarfsweise Verwendung der Subsystemklassen durch die Anwendungen wird nicht behindert, so dass jederzeit zwischen einfacher Bedienbarkeit und umfassender Allgemeingültigkeit gewählt werden kann.

Implementierung

Bei der Implementierung des Design Patterns *Facade* (*Fassade*) sind folgende Aspekte zu beachten:

1. *Reduzierte Kopplung zwischen Client und Subsystem*. Für eine noch weitreichendere Reduzierung der Kopplung zwischen Clients und Subsystem kann das Facade-Objekt als abstrakte Klasse mit konkreten Unterklassen für verschiedene Implementierungen eines Subsystems deklariert werden. So können die Clients über die Schnittstelle der abstrakten Facade-Klasse mit dem Subsystem kommunizieren. Diese abstrakte Kopplung verhindert, dass die Clients Kenntnis davon haben, welche Implementierung eines Subsystems verwendet wird.

Eine Alternative zur Unterklassenbildung ist die Konfigurierung eines Facade-Objekts mit verschiedenen Subsystemobjekten. Für eine maßgeschneiderte Anpassung der Fassade müssen dann lediglich ein oder mehrere ihrer Subsystemobjekte ersetzt werden.

2. *Öffentliche kontra private Subsystemklassen*. Zwischen Subsystem und Klasse besteht insofern eine Analogie, als dass sie beide Schnittstellen besitzen – und auch beide etwas kapseln: Eine Klasse kapselt Zustand und Operationen, ein Subsystem kapselt Klassen. Und ebenso wie es hilfreich ist, Überlegungen in Bezug auf die öffentliche und private Schnittstelle einer Klasse anzustellen, ist auch eine genauere Betrachtung der öffentlichen und privaten Schnittstelle eines Subsystems lohnenswert.

Die öffentliche Schnittstelle eines Subsystems besteht aus Klassen, die für alle Clients zugänglich sind. Die private Schnittstelle ist dagegen ausschließlich den Klassen vorbehalten, die das Subsystem erweitern. Die Facade-Klasse ist selbstverständlich ein Bestandteil der öffentlichen Schnittstelle, aber nicht der einzige. Normalerweise sind auch andere Subsystemklassen öffentlich – die Klassen Parser und Scanner im Compiler-Subsystem sind zum Beispiel ebenfalls Teil der öffentlichen Schnittstelle.

Der private Zugriff auf Subsystemklassen wäre zwar hilfreich, er wird jedoch nur von wenigen objektorientierten Sprachen unterstützt. Sowohl in C++ als auch in Smalltalk war in der Vergangenheit traditionell ein globaler Namensraum für Klassen vorgesehen – allerdings hat das C++-Standardisierungskomitee die Sprache in jüngerer Vergangenheit um Namensräume erweitert [Str94], die es ermöglichen, nur die öffentlichen Subsystemklassen offenzulegen.

Beispielcode

Im Folgenden wird aufgezeigt, wie sich eine Fassade auf ein Compiler-Subsystem aufsetzen lässt.

Das Compiler-Subsystem definiert eine Klasse `BytecodeStream`, die einen Stream von Bytecode-Objekten implementiert. Ein Bytecode-Objekt kapselt einen Bytecode zur Spezifizierung von Maschineninstruktionen. Außerdem definiert das Subsystem eine Klasse `Token` für Objekte, die Tokens einer Programmiersprache kapseln.

Die Klasse `Scanner` nimmt einen String-Stream entgegen und erzeugt einen Token-Stream, der ein Token nach dem anderen liefert:

```
class Scanner {
public:
    Scanner(istream&);
    virtual ~Scanner();

    virtual Token& Scan();

private:
    istream& _inputStream;
};
```

Die Klasse `Parser` nutzt zur Erstellung eines aus den Tokens einer Klasse `Scanner` bestehenden Parse-Baums ihrerseits die Klasse `ProgramNodeBuilder`:

```

class Parser {
public:
    Parser();
    virtual ~Parser();

    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};

```

Der Aufruf der Klasse `ProgramNodeBuilder` durch die Klasse `Parser` resultiert in einem inkrementellen Aufbau des Parse-Baums. Die Interaktion dieser Klassen entspricht dem Design Pattern *Builder* (*Erbauer*, siehe [Abschnitt 3.2](#)):

```

class ProgramNodeBuilder {
public:
    ProgramNodeBuilder();

    virtual ProgramNode* NewVariable(
        const char* variableName
    ) const;

    virtual ProgramNode* NewAssignment(
        ProgramNode* variable, ProgramNode* expression
    ) const;

    virtual ProgramNode* NewReturnStatement(
        ProgramNode* value
    ) const;

    virtual ProgramNode* NewCondition(
        ProgramNode* condition,
        ProgramNode* truePart, ProgramNode* falsePart
    ) const;
    // ...

    ProgramNode* Get rootNode();
private:
    ProgramNode* _node;
};

```

Der Parse-Baum besteht aus Instanzen der `ProgramNode`-Unterklassen wie `StatementNode`, `ExpressionNode` usw. Die `ProgramNode`-Hierarchie ist ein Beispiel für das Design Pattern *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)). `ProgramNode` definiert eine Schnittstelle zur Manipulation der Programmnoten und, sofern vorhanden, ihrer Kindobjekte:

```

class ProgramNode {
public:
    // Programmnotenmanipulation
    virtual void GetSourcePosition(int& line, int& index);

```

```

// ...

// Kindobjektmanipulation
virtual void Add(ProgramNode* );
virtual void Remove(ProgramNode* );
// ...

virtual void Traverse(CodeGenerator&);

protected:
    ProgramNode();
};

```

Die `Traverse`-Operation nimmt ein `CodeGenerator`-Objekt entgegen, das von den `ProgramNode`-Unterklassen verwendet wird, um Maschinencode in Form von `Bytecode`-Objekten eines `BytecodeStream` zu generieren. Bei der Klasse `CodeGenerator` handelt es sich um ein `Visitor`-Objekt (siehe *Visitor (Besucher)*, siehe [Abschnitt 5.11](#)).

```

class CodeGenerator {
public:
    virtual void Visit(StatementNode* );
    virtual void Visit(ExpressionNode* );
    // ...

protected:
    CodeGenerator(BytecodeStream& );
protected:
    BytecodeStream& _output;
};

```

Die Unterklassen von `CodeGenerator`, wie z. B. `StackMachineCodeGenerator` und `RISCCodeGenerator`, erzeugen Maschinencode für verschiedene Hardwarearchitekturen.

Jede Unterklasse von `ProgramNode` implementiert die Operation `Traverse` so, dass sie ihre `ProgramNode`-Kindobjekte durchläuft. Jedes Kindobjekt geht bei seinen eigenen Kindobjekten wiederum genauso vor und das wird dann rekursiv fortgesetzt. Zum Beispiel definiert `ExpressionNode` die Operation `Traverse` wie folgt:

```

void ExpressionNode::Traverse (CodeGenerator& cg) {
    cg.Visit(this);

    ListIterator<ProgramNode*> i (_children);

    for (i. First (); !i.IsDone (); i.Next()) {
        i.CurrentItem()->Traverse(cg);
    }
}

```

```
    }
```

Die bisher erwähnten Klassen bilden das Subsystem des Compilers. Im Folgenden kommt nun eine `Compiler`-Klasse als Fassade hinzu, die all diese Einzelteile zusammensetzt. Sie stellt eine einfache Schnittstelle zur Kompilierung von Quellcode und zur Codegenerierung für eine bestimmte Maschine zur Verfügung:

```
class Compiler {
public:
    Compiler();

    virtual void Compile(istream&, BytecodeStream&);
};

void Compiler::Compile (
    istream& input, BytecodeStream& output
) {
    Scanner scanner(input);
    ProgramNodeBuilder builder;
    Parser parser;

    parser.Parse(scanner, builder);

    RISCCodeGenerator generator(output);
    ProgramNode* parseTree = builder.GetRootNode();
    parseTree->Traverse(generator);
}
```

Da diese Implementierung den Typ des zu verwendenden Codegenerators hartkodiert, muss der Programmierer die Zielarchitektur nicht gesondert spezifizieren. Und das mag auch sinnvoll sein, wenn grundsätzlich nur eine Zielarchitektur erwünscht ist – andernfalls sollte der `Compiler`-Konstruktor allerdings gegebenenfalls dahingehend modifiziert werden, dass er einen `CodeGenerator`-Parameter entgegennimmt. Anschließend kann der Entwickler bei der Instanziierung von `Compiler` den zu verwendenden Generator spezifizieren. Die `Compiler`-Fassade kann zwecks erweiterter Flexibilität auch andere Teilnehmer wie `Scanner` und `ProgramNodeBuilder` parametrisieren, damit weicht sie dann jedoch von der eigentlichen Aufgabe des Design Patterns *Facade (Fassade)* ab, die ja letztendlich darin besteht, die Schnittstelle für den Normalfall zu vereinfachen.

Praxisbeispiele

Das im Abschnitt »Beispielcode« angeführte Compiler-Beispiel ist an das

ObjectWorks\Smalltalk-Compiler-System angelehnt [Par90].

Im **ET++ Application Framework** [WGM88] kann eine Anwendung auf integrierte Werkzeuge zurückgreifen, um ihre Objekte zur Laufzeit zu durchsuchen und zu überprüfen. Diese Browsing-Tools sind in einem separaten Subsystem implementiert, das eine Facade-Klasse namens `ProgrammingEnvironment` enthält und Operationen wie `InspectObject` und `InspectClass` für den Zugriff auf die Browser definiert.

Allerdings kann eine ET++-Anwendung gegebenenfalls auch auf eine solche Funktionalität verzichten. In diesem Fall implementiert `ProgrammingEvent` die entsprechenden Requests als Null-Operationen, d. h., sie zeigen keinerlei Wirkung. Stattdessen werden diese Requests mithilfe von Operationen, die die zugehörigen Browser anzeigen, ausschließlich von der `ETProgrammingEnvironment`-Unterkategorie implementiert. Durch die abstrakte Kopplung zwischen der Anwendung und dem Browsing-Subsystem hat die Anwendung selbst keine Kenntnis davon, ob eine Browserumgebung zur Verfügung steht oder nicht.

Das Betriebssystem **Choices** [CIRM93] nutzt Facade-Objekte, um mehrere Frameworks zu einem einzigen zusammenzufügen. Die maßgeblichen Abstraktionen sind hier Prozesse, Speicher und Adressräume. Für jede dieser Abstraktionen existiert ein entsprechendes, als Framework implementiertes Subsystem, das die Portierung von Choices auf eine Reihe unterschiedlicher Hardwareplattformen unterstützt. Diese Repräsentanten sind `FileSystemInterface` (Speicher) und `Domain` (Adressräume).

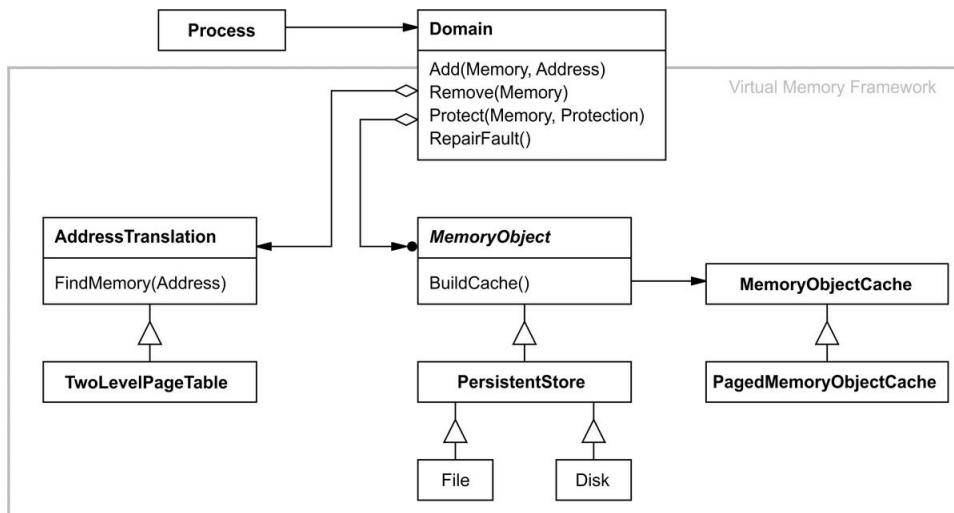


Abb. 4.25: Virtual Memory Framework im Betriebssystem Choices

Das Virtual Memory Framework verfügt beispielsweise über die Klasse `Domain` als

Fassade. Ein Domain-Objekt repräsentiert einen Adressbereich. Es stellt eine Abbildung zwischen virtuellen Adressen und Offsets auf Speicherobjekte, Dateien oder Zusatzspeicher dar. Die wichtigen Operationen von Domain unterstützen die Ergänzung eines Speicherobjekts an einer bestimmten Adresse, die Entfernung eines Speicherobjekts sowie die Handhabung eines Seitenfehlers.

Wie in [Abbildung 4.25](#) zu sehen, verwendet das Virtual-Memory-Subsystem intern die folgenden Komponenten:

- MemoryObject repräsentiert einen Datenspeicher.
- MemoryObjectCache speichert die Daten von MemoryObjects im physikalischen Speicher.
- AddressTranslation kapselt die Adressübersetzungshardware.

Die RepairFault-Operation wird immer dann aufgerufen, wenn aufgrund eines Seitenfehlers eine Unterbrechung auftritt. Domain sucht das Speicherobjekt an der Adresse, die den Fehler verursacht, und delegiert die RepairFault-Operation an den mit diesem Speicherobjekt verknüpften Zwischenspeicher. Domains können durch den Austausch ihrer Komponenten wunschgemäß angepasst werden.

Verwandte Patterns

Zur Bereitstellung einer Schnittstelle für die subsystemunabhängige Erzeugung von Subsystemobjekten kann das Design Pattern *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)) gemeinsam mit dem Design Pattern *Facade* (*Fassade*) genutzt werden. Außerdem kann das Pattern *Abstract Factory* (*Abstrakte Fabrik*) als Alternative zum Pattern *Facade* (*Fassade*) eingesetzt werden, um plattformspezifische Klassen zu verbergen.

Das Design Pattern *Mediator* (*Vermittler*, siehe [Abschnitt 5.5](#)) ähnelt dem Pattern *Facade* (*Fassade*) insoweit, als es die Funktionalität existierender Klassen abstrahiert. Der eigentliche Zweck des Patterns *Mediator* (*Vermittler*) besteht allerdings darin, die Kommunikation zwischen miteinander in Beziehung stehenden Objekten zu abstrahieren, wobei häufig die Funktionalität im Mittelpunkt steht, die keinem der Objekte zugewiesen werden kann. Die mit dem Mediator-Objekt in Beziehung stehenden Objekte haben Kenntnis von seiner Existenz und kommunizieren nicht direkt miteinander, sondern mit ihm. Im Gegensatz dazu abstrahiert ein Facade-Objekt die Schnittstelle zu Subsystemobjekten lediglich zum

Zweck der einfacheren Verwendbarkeit – sie definiert jedoch keine neue Funktionalität und die Subsystemklassen haben auch keine Kenntnis von ihr.

Da normalerweise nur ein Facade-Objekt benötigt wird, handelt es sich dabei häufig um ein *Singleton* (siehe Design Pattern *Singleton (Singleton)*, [Abschnitt 3.5](#)).

4.6 Flyweight (Fliegengewicht)

Objektbasiertes Strukturmuster

Zweck

Gemeinsame Nutzung feingranularer Objekte, um sie auch in großer Anzahl effizient einsetzen zu können.

Motivation

Manche Anwendungsdesigns könnten sicherlich von einer durchgängigen Objektnutzung profitieren – allerdings ist eine naive Implementierung der Objekte mit untragbaren Laufzeitkosten verbunden.

Beispielsweise sind die Layout- und Editierfunktionen in den meisten Texteditorimplementierungen in gewissem Umfang modular aufgebaut. Objektorientierte Texteditoren machen sich zur Darstellung von eingebetteten Elementen wie etwa Tabellen oder Abbildungen zwar Objekte zunutze – in der Regel gehen sie aber selten so weit, dass jedes einzelne Zeichen im Dokument durch ein Objekt repräsentiert wird, obwohl die Anwendung dadurch auf vielen Ebenen deutlich mehr Flexibilität erhalten würde: Es wäre ein einheitlicher Umgang in Bezug auf die Erstellung und Formatierung von Zeichen- und anderen eingebetteten Elementen möglich. Die Anwendung könnte weitere, neue Zeichensätze unterstützen, ohne mit der übrigen Funktionalität in Konflikt zu geraten. Und die Objektstruktur der Anwendung könnte die physische Struktur des Dokuments widerspiegeln. [Abbildung 4.26](#) zeigt eine Variante der Objektnutzung zur Darstellung von Zeichen in einem Texteditor:

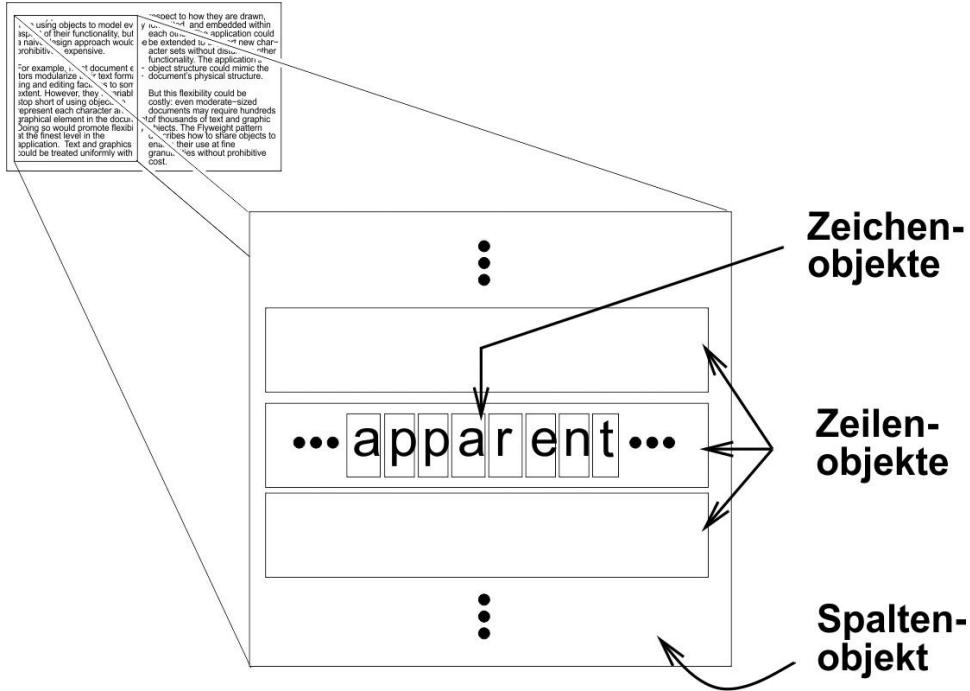


Abb. 4.26: Objektnutzung zur Darstellung von Zeichen in einem Texteditor

Der Nachteil eines solchen Designs sind die damit verbundenen Laufzeitkosten: Selbst Dokumente moderater Größe könnten Hunderttausende Zeichenobjekte benötigen, die sehr viel Speicherplatz belegen und letztendlich zu einem inakzeptabel hohen Mehraufwand zur Laufzeit führen würden. Das Design Pattern *Flyweight (Fliegengewicht)* beschreibt, wie Objekte in einer Form gemeinsam genutzt werden können, dass sie auf feingranularer Ebene einsetzbar sind, ohne unzumutbare Laufzeitkosten zu verursachen.

Ein Flyweight-Objekt ist ein gemeinsam genutztes Objekt, das gleichzeitig, aber dennoch jeweils unabhängig in verschiedenen Kontexten verwendet werden kann – und somit von einer nicht gemeinsam genutzten Instanz derselben Klasse nicht zu unterscheiden ist. Flyweight-Objekte können keine Annahmen über den Kontext machen, in dem sie agieren. Der maßgebliche Grundsatz ist hierbei die Unterscheidung zwischen **intrinsischem** und **extrinsischem** Zustand: Der *intrinsische* (von innen her kommende) Zustand wird unmittelbar im Flyweight-Objekt selbst gespeichert und besteht ausschließlich aus kontextunabhängigen Informationen, wodurch eine gemeinsame Nutzung möglich ist. Im Gegensatz dazu ist der *extrinsische* (von außen her kommende) Zustand kontextabhängig und variiert dementsprechend, weshalb eine gemeinsame Nutzung ausgeschlossen ist. In diesem Fall sind die Client-Objekte dafür zuständig, den extrinsischen Zustand bei Bedarf an das Flyweight-Objekt weiterzuleiten.

Flyweight-Objekte modellieren Konzepte oder Entitäten, die für eine Darstellung mittels Objekten normalerweise zu umfassend sind. So kann ein Texteditor zum Beispiel je ein Flyweight-Objekt für jeden Buchstaben des Alphabets erzeugen. Jedes Flyweight-Objekt speichert genau einen Zeichencode – die Standortkoordinaten für die genaue Position des Buchstabens im Dokument und der darauf anzuwendende Schriftstil können jedoch von den Algorithmen für das Textlayout und den aktiven Formatierungsbefehlen bestimmt werden. In diesem Beispiel repräsentiert der Zeichencode den intrinsischen Zustand, während die übrigen Informationen den extrinsischen Zustand wiedergeben.

Aus logischer Sicht existiert für jedes einzelne im Dokument verwendete Textzeichen jeweils ein eigenes Objekt:

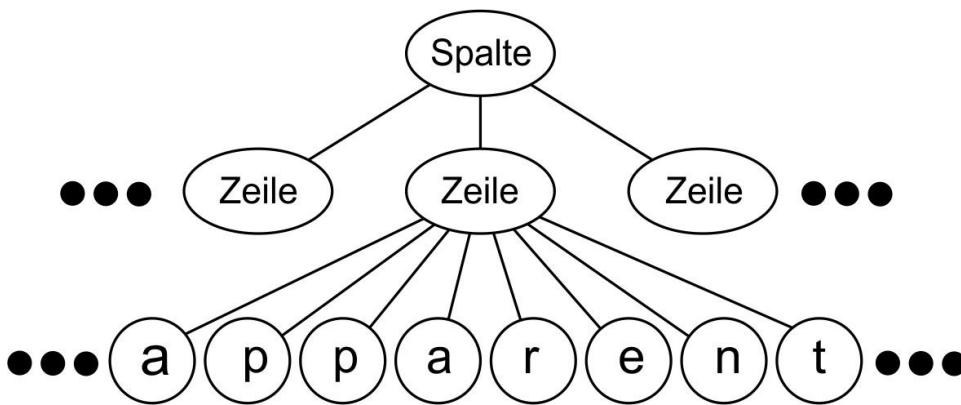


Abb. 4.27: Flyweight-Objekte zur Darstellung von Buchstaben in einem Textdokument

Physisch existiert jedoch nur ein gemeinsam genutztes Flyweight-Objekt pro Zeichen, das in verschiedenen Kontexten in der Dokumentstruktur in Erscheinung tritt. Jedes Vorkommen eines bestimmten Zeichenobjekts bezieht sich auf dieselbe Instanz in dem gemeinsam genutzten Pool von Flyweight-Objekten:

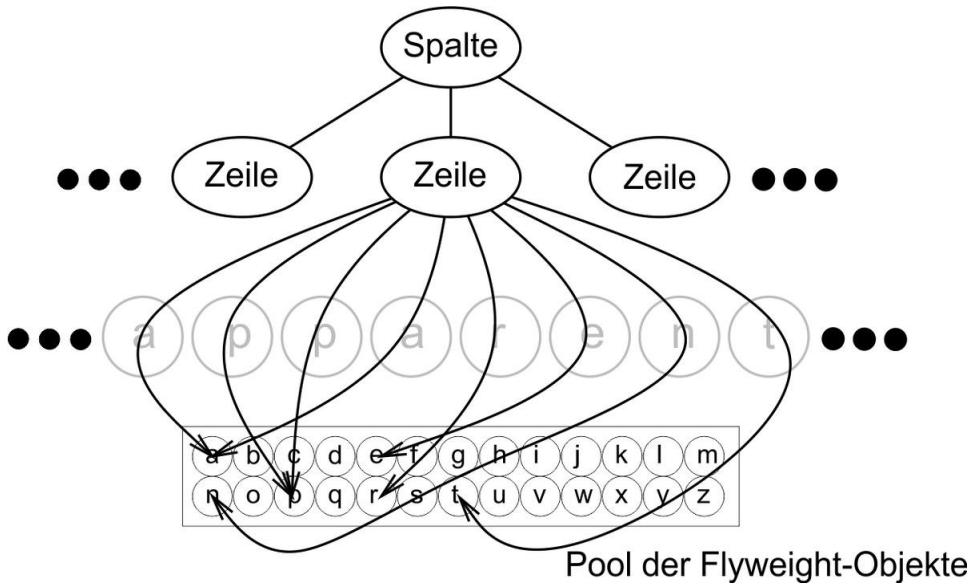


Abb. 4.28: Gemeinsam genutzter Pool von Flyweight-Objekten

Die Klassenstruktur dieser Objekte ist in [Abbildung 4.29](#) dargestellt. **Glyph** ist die abstrakte Klasse für grafische Objekte, von denen einige auch Flyweight-Objekte sein können. Operationen, die vom extrinsischen Zustand abhängig sind, bekommen ihn mittels Parametern übergeben. Zum Beispiel müssen **Draw** und **Intersects** den Kontext der Glyphe kennen, um ihre Arbeit erledigen zu können:

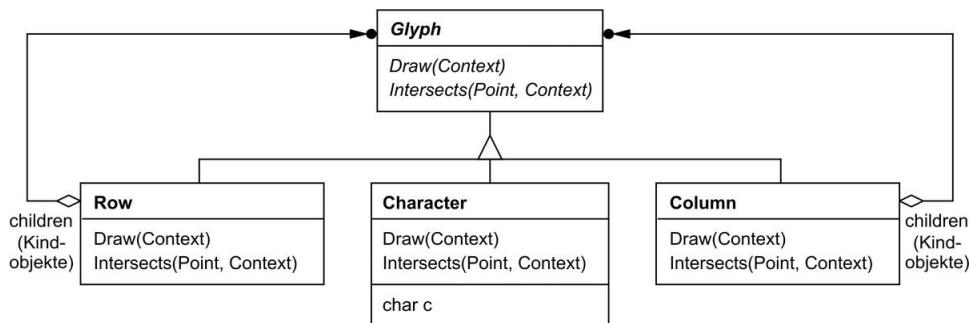


Abb. 4.29: Flyweight-Klassenstruktur

Ein Flyweight-Objekt, das den Buchstaben »a« darstellt, speichert nur den zugehörigen Zeichencode – es braucht weder seine Position, noch seinen Zeichensatz zu speichern. Die zum Zeichnen des Flyweight-Objekts benötigten kontextabhängigen Informationen werden vom Client geliefert. Zum Beispiel weiß eine Row-Glyphe, wo ihre Kindobjekte gezeichnet werden sollten, damit sie horizontal angeordnet sind, und kann somit im Fall eines Requests zum Zeichnen jedem Kindobjekt seinen jeweiligen Standort übermitteln.

Da die Anzahl der verschiedenen Zeichenobjekte deutlich unter der Anzahl der in

dem Dokument verwendeten Zeichen liegt, ist die Gesamtzahl der Objekte erheblich geringer, als es in einer naiven Implementierung der Fall wäre. Ein Dokument, in dem alle Zeichen in derselben Schriftart und Farbe dargestellt werden, weist unabhängig von der Dokumentlänge in etwa 100 Zeichenobjekte auf (was ungefähr dem Umfang des ASCII-Zeichensatzes entspricht). Und da die meisten Dokumente nicht mehr als 10 verschiedene Schriftart-/Farbe-Kombinationen nutzen, wird diese Anzahl in der Praxis kaum anwachsen. Insofern wird eine Objektabstraktion für einzelne Zeichen zu einer praktikablen Lösung.

Anwendbarkeit

Wie effizient sich das Design Pattern *Flyweight (Fliegengewicht)* auswirkt, ist in hohem Maße davon abhängig, wie und wo es angewendet wird. Deshalb sollte es nur eingesetzt werden, wenn *alle* nachfolgend aufgeführten Kriterien erfüllt sind:

- Eine Anwendung nutzt eine große Zahl von Objekten.
- Die Speicheranforderungen sind aufgrund der zahlreichen verwendeten Objekte recht hoch.
- Der Objektzustand kann in weiten Teilen extrinsisch gemacht werden.
- Viele Objektgruppen können nach der Beseitigung des extrinsischen Zustands durch relativ wenige gemeinsam genutzte Objekte ersetzt werden.
- Die Anwendung ist nicht von der Objektidentität abhängig. Da Flyweight-Objekte gemeinsam genutzt werden können, geben die Identitätsprüfungen für konzeptuell verschiedene Objekte den Wert `true` (`wahr`) zurück.

Struktur

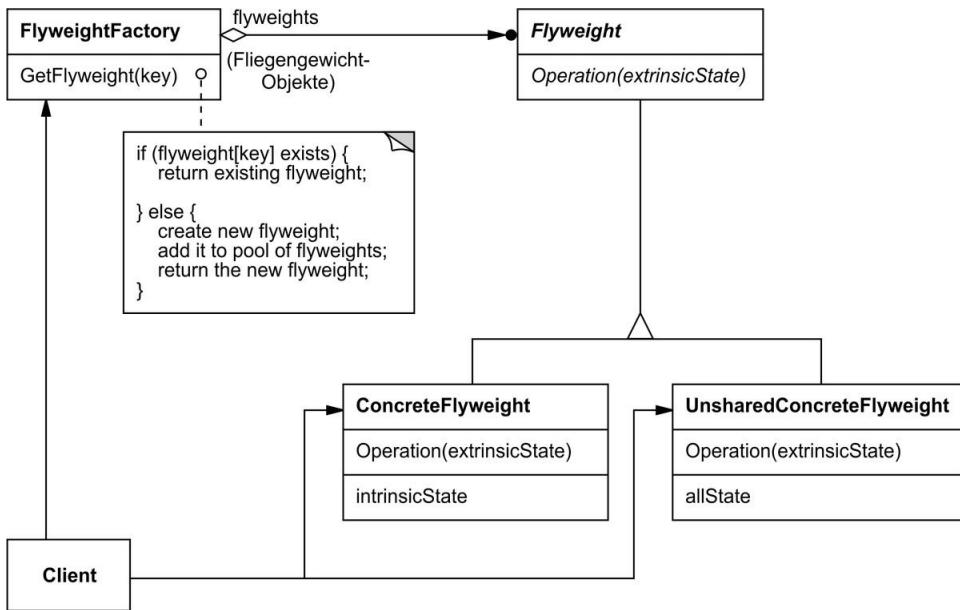


Abb. 4.30: Die Struktur des Design Patterns Flyweight (Fliegengewicht)

Abbildung 4.31 zeigt, wie Flyweight-Objekte gemeinsam genutzt werden:

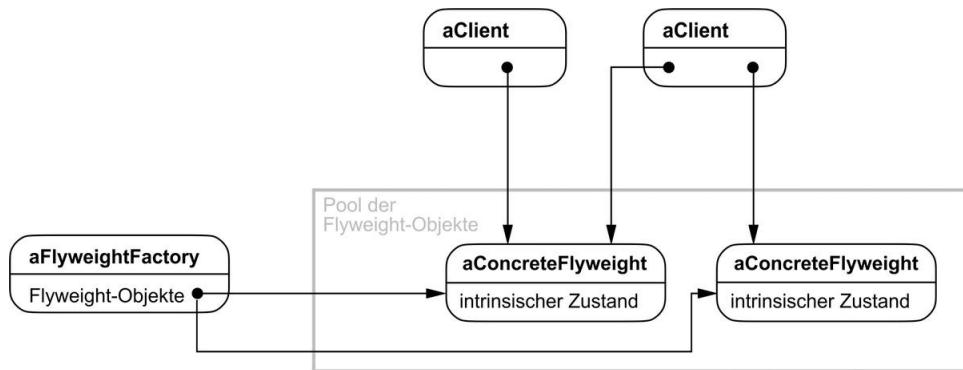


Abb. 4.31: Gemeinsame Nutzung von Flyweight-Objekten

Teilnehmer

- **Flyweight** (Glyph)
 - Deklariert eine Schnittstelle, durch die Flyweight-Objekte einen extrinsischen Zustand erhalten und entsprechend reagieren können.
- **ConcreteFlyweight** (Character)
 - Implementiert die Flyweight-Schnittstelle und ergänzt Speicherkapazität

für einen intrinsischen Zustand, sofern vorhanden. Ein `ConcreteFlyweight`-Objekt muss gemeinsam nutzbar sein. Jeder darin gespeicherte Zustand muss intrinsisch sein, d. h., er darf nicht vom Kontext des `ConcreteFlyweight`-Objekts abhängig sein.

- **UnsharedConcreteFlyweight** (Zeile, Spalte)

- Nicht alle Flyweight-Unterklassen müssen gemeinsam genutzt werden. Die Flyweight-Schnittstelle *ermöglicht* zwar eine gemeinsame Nutzung, erzwingt sie aber nicht. Im Allgemeinen haben `UnsharedConcreteFlyweight`-Objekte ab irgendeiner Ebene in der Flyweight-Objektstruktur `ConcreteFlyweight`-Objekte als Kindobjekte (wie z. B. die `Row`- und `Column`-Klassen).

- **FlyweightFactory**

- Erzeugt und verwaltet Flyweight-Objekte.
- Gewährleistet die korrekte gemeinsame Nutzung von Flyweight-Objekten. Sobald ein Client ein Flyweight-Objekt anfordert, stellt das `FlyweightFactory`-Objekt eine schon vorhandene Instanz bereit oder erzeugt eine neue, sofern noch keine existiert.

- **Client**

- Verwaltet eine Referenz auf Flyweight-Objekte.
- Berechnet oder speichert den extrinsischen Status von Flyweight-Objekten.

Interaktionen

- Der für das Funktionieren eines Flyweight-Objekts erforderliche Zustand muss ausdrücklich als entweder intrinsisch oder extrinsisch ausgewiesen werden. Ein intrinsischer Zustand wird im Objekt `ConcreteFlyweight` gespeichert, ein extrinsischer Zustand wird von `client`-Objekten gespeichert oder berechnet. Die Übergabe dieses Zustands durch die Clients erfolgt beim Aufruf der Operationen des Flyweight-Objekts.
- Die Clients sollten `ConcreteFlyweight`-Objekte nie direkt instanziieren. Damit

eine korrekte gemeinsame Nutzung gewährleistet ist, müssen sie ausschließlich vom FlyweightFactory-Objekt erhalten.

Konsequenzen

Flyweight-Objekte sind unter Umständen mit Laufzeiteinbußen in Bezug auf die Übertragung, das Auffinden und/oder die Berechnung des extrinsischen Zustands verbunden, insbesondere wenn Letzterer vorher als intrinsischer Zustand gespeichert war. Diese Einbußen werden jedoch durch Speicherplatz einsparungen wieder wettgemacht, die parallel zum Ausmaß der gemeinsamen Nutzung von Flyweight-Objekten ansteigen.

Speicherplatz einsparungen hängen von mehreren Faktoren ab:

- der Reduzierung der Gesamtzahl der Instanzen, die sich aus der gemeinsamen Nutzung ergeben,
- dem Ausmaß des intrinsischen Zustands pro Objekt,
- der Frage, ob der extrinsische Zustand berechnet oder gespeichert wird.

Je mehr Flyweight-Objekte gemeinsam genutzt werden, desto höher ist die Speicherplatzersparnis. Dabei steigt die eingesparte Speicherkapazität parallel zum Ausmaß des gemeinsam genutzten Zustands an. Die größte Ersparnis wird erzielt, wenn die Objekte sowohl den intrinsischen als auch den extrinsischen Zustand in möglichst umfangreichem Maße nutzen und der extrinsische Zustand nicht gespeichert, sondern berechnet werden kann. Damit wird der Speicherbedarf gleich auf zwei Arten geschont: Zum einen reduziert die gemeinsame Nutzung die Speicherkosten für den intrinsischen Zustand, und zum anderen kann der extrinsische Zustand gegen Rechenzeit eingetauscht werden.

Das Design Pattern *Flyweight (Fliegengewicht)* wird häufig in Kombination mit dem Pattern *Composite (Kompositum*, siehe [Abschnitt 4.3](#)) verwendet, um eine hierarchische Struktur als Graph mit gemeinsam genutzten Blattknoten abzubilden. Eine Folge der gemeinsamen Nutzung ist, dass die Flyweight-Blattknoten keine Pointer auf ihre Elternobjekte speichern können. Stattdessen wird die Elternreferenz als Teil seines extrinsischen Zustands an das Flyweight-Objekt übergeben – und das hat erheblichen Einfluss auf die Art und Weise, wie die Objekte in der Hierarchie miteinander kommunizieren.

Implementierung

Bei der Anwendung des Design Patterns *Flyweight (Fliegengewicht)* sollten folgende Kriterien in Betracht gezogen werden:

1. *Entfernung des extrinsischen Zustands.* Die Anwendbarkeit dieses Patterns wird in weiten Teilen davon bestimmt, wie leicht sich der extrinsische Zustand ermitteln und von den gemeinsam genutzten Objekten entfernen lässt. Die Entfernung des extrinsischen Zustands trägt nicht zur Reduzierung der Speicheranforderungen bei, wenn es genauso viele verschiedene extrinsische Zustände wie Objekte gibt wie vor der gemeinsamen Nutzung. Idealerweise kann der extrinsische Zustand aus einer separaten Objektstruktur berechnet werden, die deutlich geringere Anforderungen an die Speicherkapazität stellt.

In dem hier verwendeten Texteditor könnten beispielsweise die typografischen Daten in einer separaten Struktur abgebildet werden, statt die Schriftart und den Schriftstil in jedem einzelnen Zeichenobjekt zu speichern. Diese Abbildung verwaltet Abfolgen von Zeichen mit denselben typografischen Attributen. Wenn sich ein Zeichen selbst zeichnet, werden ihm als Nebeneffekt der für den Zeichenvorgang ausgeführten Traversierung auch seine typografischen Attribute übermittelt. Weil in Dokumenten normalerweise nur wenige verschiedene Zeichensätze und Schriftstile verwendet werden, ist die externe Speicherung dieser Daten für jedes einzelne Zeichenobjekt deutlich effizienter als die interne Speicherung.

2. *Verwaltung gemeinsam genutzter Objekte.* Bei der gemeinsamen Nutzung von Objekten sollten die Clients sie nicht direkt instanziieren. Das FlyweightFactory-Objekt ermöglicht den Clients einerseits ein bestimmtes Flyweight-Objekt zu lokalisieren und bietet ihnen andererseits häufig auch einen assoziativen Speicher, den sie nutzen können, um die gewünschten Flyweight-Objekte herauszusuchen. So kann die FlyweightFactory in dem hier verwendeten Texteditor beispielsweise eine mit Zeichencodes indizierte Flyweight-Tabelle verwalten und gibt dann anhand des übergebenen Codes das richtige Flyweight-Objekt zurück oder erzeugt es gegebenenfalls.

Die gemeinsame Nutzbarkeit bedingt auch irgendeine Form der Referenzzählung (engl. *Reference Counting*) oder Speicherbereinigung (engl. *Garbage Collection*), damit die von dem Flyweight-Objekt belegte Speicherkapazität wieder freigegeben werden kann, wenn sie nicht mehr benötigt wird. Das gilt allerdings nicht, wenn es sich um eine festgelegte und noch dazu überschaubare Anzahl von Flyweight-Objekten handelt, also z. B.

nur für den ASCII-Zeichensatz – in diesem Fall lohnt es sich, die Objekte permanent im Speicher zu halten.

Beispielcode

An dieser Stelle soll noch einmal auf das Beispiel der Dokumentformatierung zurückgegriffen werden. Zunächst wird eine Basisklasse `Glyph` für die grafischen Flyweight-Objekte definiert. Aus logischer Sicht sind Glyphen Composite-Objekte (siehe *Composite (Kompositum)*, [Abschnitt 4.3](#)), die grafische Attribute besitzen und sich selbst zeichnen können. In diesem Fall geht es nur um das `Font`-Attribut, also das Attribut für den Zeichensatz. Dieselbe Vorgehensweise lässt sich jedoch auch für alle anderen grafischen Attribute anwenden, die eine Glyphe haben kann.

```
class Glyph {  
public:  
    virtual ~Glyph();  
  
    virtual void Draw(Window*, GlyphContext&);  
  
    virtual void SetFont(Font*, GlyphContext&);  
    virtual Font* GetFont(GlyphContext&);  
  
    virtual void First(GlyphContext&);  
    virtual void Next(GlyphContext&);  
    virtual bool IsDone(GlyphContext&);  
    virtual Glyph* Current(GlyphContext&);  
  
    virtual void Insert(Glyph*, GlyphContext&);  
    virtual void Remove(GlyphContext&);  
  
protected:  
    Glyph();  
};
```

Die Unterklasse `Character` speichert lediglich einen Zeichencode:

```
class Character : public Glyph {  
public:  
    Character(char);  
  
    virtual void Draw(Window*, GlyphContext&);  
  
private:  
    char _charcode;  
};
```

Damit nicht in jeder Glyph Speicherplatz für ein Zeichensatzattribut vorgehalten werden muss, werden die Attribute extrinsisch in einem `GlyphContext`-Objekt gespeichert. Dieses Objekt fungiert als Repository für den extrinsischen Zustand. Es bildet die Beziehung zwischen einem `Glyph`-Objekt und seinem Zeichensatz (sowie jedem anderen grafischen Attribut) in unterschiedlichen Kontexten in kompakter Form ab. Jeder Operation, die von dem Zeichensatz einer Glyph in einem gegebenen Kontext Kenntnis haben muss, wird eine `GlyphContext`-Instanz als Parameter übergeben. Anschließend kann die Operation dann den richtigen Zeichensatz für den betreffenden Kontext von dem `GlyphContext`-Objekt abfragen. Und da der Kontext vom Standort des `Glyph`-Objekts in der Glyphenstruktur abhängig ist, müssen die Iterations- und Manipulationsoperationen der Kindobjekte von `Glyph` das `GlyphContext`-Objekt bei jeder Verwendung aktualisieren:

```
class GlyphContext {
public:
    GlyphContext();
    virtual ~GlyphContext();

    virtual void Next(int step = 1);
    virtual void Insert(int quantity = 1);

    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1);

private:
    int _index;
    BTTree* _fonts;
};
```

`GlyphContext` muss also während des Traversierungsvorgangs stets über die aktuelle Position in der Glyphenstruktur informiert sein. Zugleich wird `_index` im weiteren Verlauf der Traversierung von `GlyphContext::Next` inkrementiert. `Glyph`-Unterklassen, die Kindobjekte besitzen (z. B. `Row` und `Column`), müssen die Operation `Next` so implementieren, dass sie an jedem Punkt der Traversierung `GlyphContext::Next` aufruft.

`GlyphContext::GetFont` benutzt den Index als Schlüssel zu einer `BTTree`-Struktur, die die Abbildung der »Glyphen-zu-Zeichensatz«-Beziehungen in Form einer Map speichert. Jeder Knoten in der Baumstruktur ist mit der Länge des Strings gekennzeichnet, dessen Zeichensatzdaten darin bereitgestellt sind. Dabei verweisen die Blattobjekte im Baum auf einen Zeichensatz, während die inneren Knoten den String in jeweils einen Substring für jedes Kindobjekt aufbrechen.

[Abbildung 4.32](#) zeigt einen Auszug aus einer Glyphenkomposition:

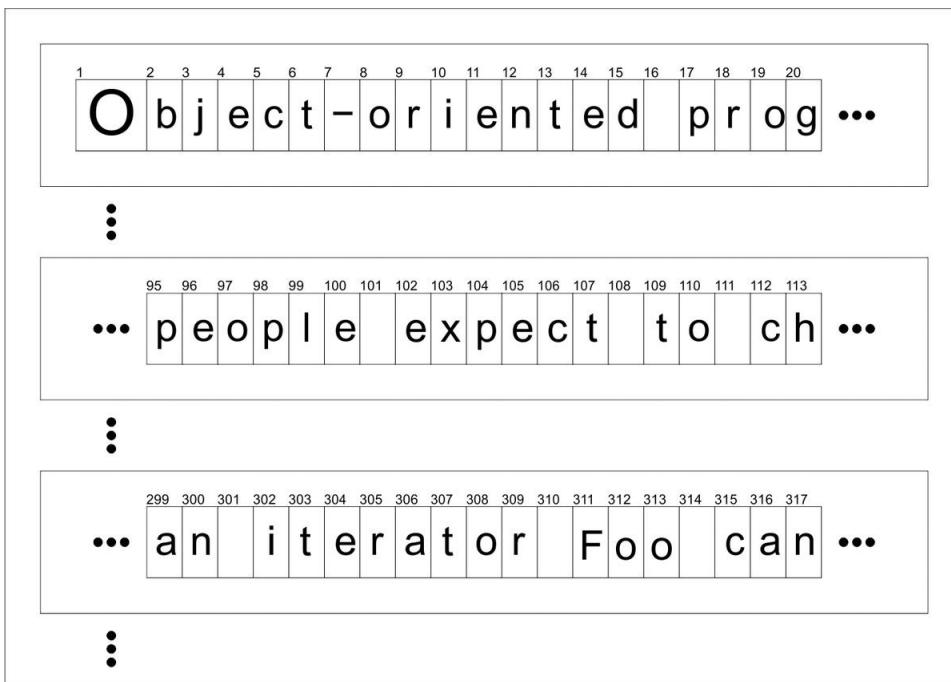


Abb. 4.32: Auszug aus einer Glyphenkomposition

Die **BTree**-Struktur für die Zeichensatzdaten könnte beispielsweise wie folgt aussehen:

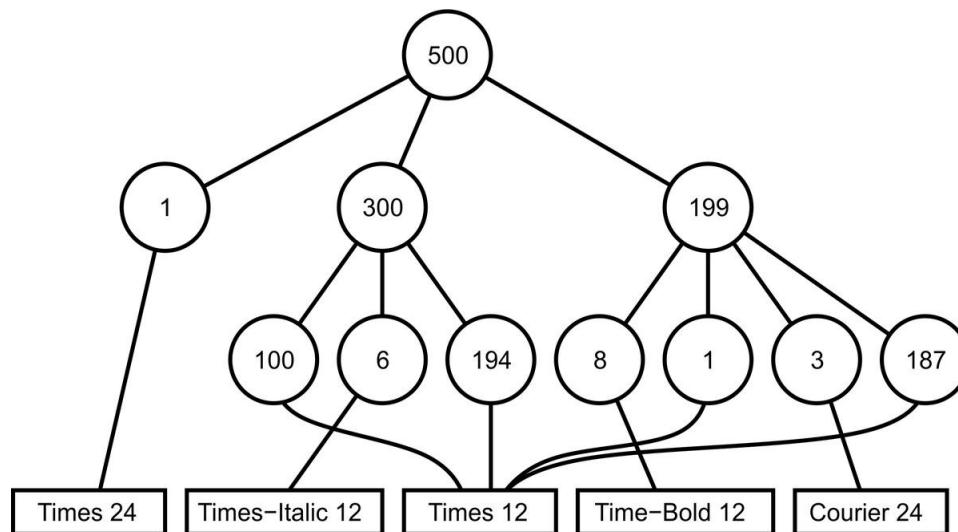


Abb. 4.33: BTree-Struktur

Die inneren Knoten definieren Bereiche der **Glyph**-Indizes. Jede Zeichensatzänderung, Ergänzung eines neuen oder Entfernung eines vorhandenen **Glyph**-Objekts hat eine Aktualisierung der **BTree**-Struktur zur Folge. Wenn sich die Traversierung also beispielsweise am Index 102 befindet, würde der folgende Code in diesem Beispielfall den Zeichensatz jedes Zeichens in dem Wort »expect« auf den

des umgebenden Textes setzen – in diesem Fall also `Times 12`, eine Instanz von Font für die Schriftart Times Roman in der Größe 12 Punkt:

```
GlyphContext gc;
Font* times12 = new Font("Times-Roman-12");
Font* timesItalic12 = new Font("Times-Italic-12");
// ...

gcSetFont(times12, 6);
```

Die neue BTree-Struktur (Änderungen sind in Schwarz dargestellt) sieht damit folgendermaßen aus:

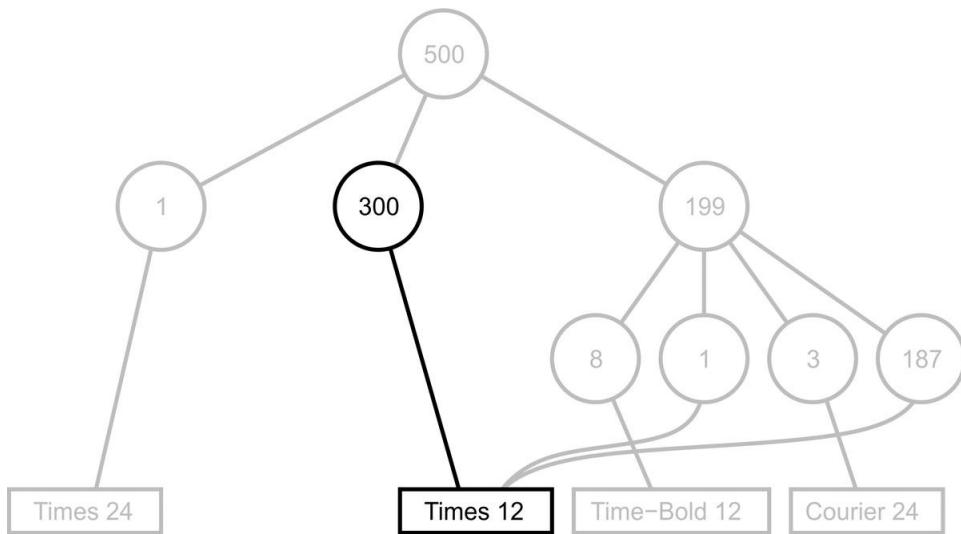


Abb. 4.34: BTree-Struktur mit geändertem Zeichensatz

Nun soll noch das Wort »don't« (mit einem darauffolgenden Leerzeichen) in Times Italic 12 Punkt vor »expect« eingefügt werden. Der folgende Code informiert den `gc` (`GlyphContext`) über dieses Ereignis, wobei davon ausgegangen wird, dass sich die Traversierung immer noch am Index 102 befindet:

```
gc.Insert(6);
gcSetFont(timesItalic12, 6);
```

Damit ändert sich die BTree-Struktur wie folgt:

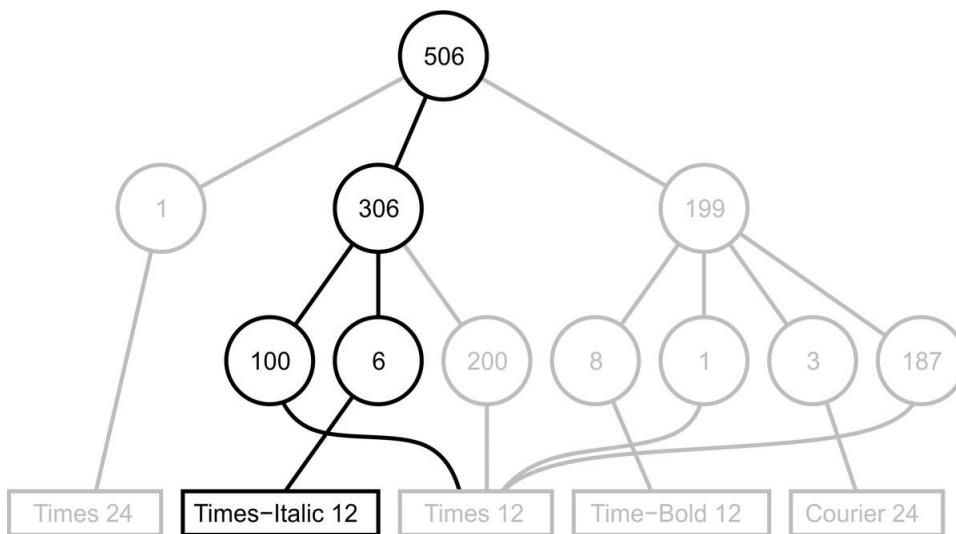


Abb. 4.35: BTree-Struktur nach dem Einfügen-Ereignis

Wird nun der Zeichensatz der aktuellen Glyphe vom `GlyphContext`-Objekt abgefragt, durchläuft es die BTree-Struktur von oben nach unten und addiert dabei die Indizes, bis es den Zeichensatz für den aktuellen Index gefunden hat. Da Zeichensatzänderungen normalerweise nur relativ selten vorkommen, bleibt der Baum in Relation zu der Glyphenstruktur überschaubar – und dementsprechend halten sich auch die Speicheranforderungen in Grenzen, ohne dass die Lookup-Zeit unverhältnismäßig ansteigt.

Hinweis

In diesem Beispiel entwickelt sich die Lookup-Zeit proportional zu der Häufigkeit der Zeichensatzwechsel. Das Worst-Case-Szenario in Bezug auf die Performance wäre die Situation, wenn der Zeichensatz mit jedem Zeichen gewechselt würde – das passiert in der Praxis allerdings so gut wie nie.

Als Letztes wird nun noch ein `FlyweightFactory`-Objekt benötigt, das Glyphen erzeugt und deren korrekte gemeinsame Nutzung sicherstellt. Die Klasse `GlyphFactory` instanziert neben der `character`-Glyphe auch andere Arten von `Glyph`-Objekten. In diesem Fall werden nur `character`-Objekte gemeinsam genutzt. Zusammengesetzte `Glyph`-Objekte treten relativ selten auf und ihre relevanten Zustände, sprich ihre Kindobjekte, sind ohnehin intrinsisch:

```
const int NCHARCODES = 128;
```

```

class GlyphFactory {
public:
    GlyphFactory();
    virtual ~GlyphFactory ();
    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
    // ...

private:
    Character* _character[NCHARCODES];
};

```

Das `_character`-Array enthält Pointer auf die mittels Zeichencodes indizierten `Character`-Glyphen und wird im Konstruktor mit 0 initialisiert:

```

GlyphFactory::GlyphFactory () {
    for (int i = 0; i < NCHARCODES; ++i) {
        _character[i] = 0;
    }
}

```

`CreateCharacter` sucht ein Zeichen aus dem `Glyph`-Objekt `Character` im Array heraus und gibt dann die entsprechende Glyphe zurück, sofern vorhanden. Andernfalls wird diese Glyphe erzeugt, in das Array gelegt und anschließend zurückgegeben:

```

Character* GlyphFactory::CreateCharacter (char c) {
    if (! _character[c]) {
        _character[c] = new Character(c);
    }

    return _character[c];
}

```

Da `Glyph`-Objekte, die keine Zeichen repräsentieren, nicht gemeinsam genutzt werden, instanziieren die anderen Operationen bei jedem Aufruf einfach ein neues Objekt:

```

Row* GlyphFactory::CreateRow () {
    return new Row;
}

Column* GlyphFactory::CreateColumn () {
    return new Column;
}

```

Grundsätzlich könnten diese Operationen auch weggelassen und stattdessen den

Clients die Aufgabe übertragen werden, nicht gemeinsam genutzte Glyph-Objekte selbst zu instanziieren. Wenn diese Glyphen dann allerdings zu einem späteren Zeitpunkt doch gemeinsam genutzt werden sollen, müsste der gesamte Clientcode, der für ihre Erzeugung zuständig ist, geändert werden.

Praxisbeispiele

Das Konzept der Flyweight-Objekte als Designtechnik wurde erstmals in **InterViews 3.0** [CL90] beschrieben und untersucht. Zum Nachweis dieses Konzepts entwickelten die Macher dieses Toolkits einen leistungsstarken Texteditor namens **Doc** [CL92], der zur Darstellung der Zeichen in einem Dokument Glyph-Objekte verwendet. Der Editor erzeugt für jedes einzelne Zeichen eines bestimmten Stils je eine Glyph-Instanz, die die zugehörigen grafischen Attribute definiert – somit besteht der intrinsische Zustand eines Zeichens jeweils aus dem Zeichencode und seiner Stildaten, sozusagen einem Index in einer Stiltabelle. (In dem zuvor beschriebenen Beispielcode sind die Stildaten extrinsisch, wodurch der Zeichencode als einziger intrinsischer Zustand verbleibt.) Das bedeutet, dass nur die Standortkoordinaten des Zeichens im Dokument extrinsisch sind – und das kommt der Geschwindigkeit von Doc spürbar zugute. Dokumente werden durch die Klasse `Document` repräsentiert, die gleichzeitig als FlyweightFactory fungiert. In Doc vorgenommene Messungen belegen die beachtliche Effizienz, die durch die gemeinsame Nutzung von Flyweight-Zeichen erzielt wird: In einem typischen Anwendungsfall mussten für ein Dokument, das insgesamt 180.000 Zeichen umfasste, lediglich 480 character-Objekte zugewiesen werden.

In **ET++** [WGM88] werden Flyweight-Objekte zur Gewährleistung der Unabhängigkeit von Look-and-Feel-Standards verwendet (siehe auch *Abstract Factory (Abstrakte Fabrik, Abschnitt 3.1*, für einen weiteren Ansatz zur Unterstützung mehrerer verschiedener Look-and-Feels). Der Look-and-Feel-Standard beeinflusst das Layout von Elementen auf der Benutzeroberfläche (z. B. Scrollleisten, Schaltflächen, Menüs etc., allgemein als »Widgets« bezeichnet) und deren Dekorierungen (z. B. Schlagschatten, Schrägstellung etc.). Ein Widget delegiert sein Layout- und Zeichenverhalten vollständig an ein separates Layout-Objekt, d. h., wenn das Objekt modifiziert wird, ändert sich auch das Look-and-Feel – sogar zur Laufzeit.

Darüber hinaus existiert für jede Widget-Klasse eine passende Layout-Klasse (z. B. `ScrollbarLayout`, `MenubarLayout` etc.). Offenkundig problematisch ist bei diesem Ansatz, dass sich die Zahl der Objekte für die Benutzeroberfläche durch die

Verwendung separater Layout-Objekte verdoppelt: Auf jedes Objekt der Benutzeroberfläche kommt ein zusätzliches Layout-Objekt. Um diesen Überhang zu vermeiden, werden die Layout-Objekte als Flyweight-Objekte implementiert – wofür sie gut geeignet sind, weil sie in erster Linie mit der Definition von Verhalten befasst sind und sich problemlos mit dem für das Layouten oder Zeichnen eines Objekts erforderlichen extrinsischen Zustand ausstatten lassen.

Die Layout-Objekte werden wiederum von Look-Objekten erzeugt und verwaltet. Die Look-Klasse ist eine *Abstract Factory* (siehe auch Design Pattern *Abstract Factory (Abstrakte Fabrik)*, [Abschnitt 3.1](#)), die ein spezifisches Layout-Objekt mit Operationen wie `GetButtonLayout`, `GetMenuBarLayout` usw. abruft. Für jeden Look-and-Feel-Standard existiert eine passende Look-Unterklasse (z. B. `MotifLook`, `OpenLook`), die die zugehörigen Layout-Objekte liefert.

Im Übrigen handelt es sich bei den Layout-Objekten im Wesentlichen um *Strategy*-Objekte (siehe auch Design Pattern *Strategy (Strategie)*, [Abschnitt 5.9](#)). Sie sind ein Beispiel für ein *Strategy*-Objekt, das als Flyweight-Objekt implementiert ist.

Verwandte Patterns

Das Design Pattern *Flyweight (Fliegengewicht)* wird häufig in Kombination mit dem Pattern *Composite (Kompositum*, siehe [Abschnitt 4.3](#)) genutzt, um eine an einen gerichteten azyklischen Graphen angelehnte logische hierarchische Struktur mit gemeinsam genutzten Blattknoten zu implementieren.

State- und *Strategy*-Objekte (siehe auch die Design Patterns *State (Zustand)*, [Abschnitt 5.8](#), und *Strategy (Strategie)*, [Abschnitt 5.9](#))) lassen sich meist am besten als Flyweight-Objekte implementieren.

4.7 Proxy (Proxy)

Objektbasiertes Strukturmuster

Zweck

Bereitstellung eines vorgelagerten Stellvertreterobjekts bzw. eines Platzhalters zwecks Zugriffssteuerung eines Objekts.

Auch bekannt als

Surrogate (Stellvertreter)

Motivation

Ein maßgeblicher Beweggrund für die Einrichtung einer Zugriffskontrolle für ein Objekt ist, dass sie den Aufschub der für dessen Instanziierung und Initialisierung anfallenden Ressourcenlast bis zum Zeitpunkt seiner tatsächlichen Nutzung ermöglicht. Im Folgenden wird die Arbeitsweise des Design Patterns *Proxy* (*Proxy*) am Beispiel eines Texteditors demonstriert, der die Einbettung grafischer Objekte in einem Dokument gestattet. Die Erzeugung von z. B. großen gerasterten Bildern kann die Performance einer Anwendung mitunter stark beeinträchtigen. Allerdings sollte das Öffnen eines Dokuments jederzeit möglichst zügig vonstattengehen – und deshalb ist die zeitgleiche Instanziierung ressourcenlastiger Objekte generell zu vermeiden. Im Allgemeinen ist dies aber ohnehin nicht nötig, weil normalerweise nicht alle Objekte zur gleichen Zeit im Dokument sichtbar sein werden.

Angesichts dieser Rahmenbedingungen erscheint es naheliegend, die ressourcenlastigen Objekte nur *auf Anforderung* zu erzeugen – was im vorliegenden Beispiel genau dann passiert, wenn ein Bild sichtbar wird. Doch was soll anstelle des Bildes im Dokument zu sehen sein? Und wie lässt sich die Tatsache, dass das Bild nur auf Anforderung erzeugt wird, in einer Weise verbergen, die die Implementierung des Editors nicht unnötig verkompliziert? Diese Optimierungsmaßnahme sollte sich zum Beispiel weder auf den Render- noch den Formatierungscode auswirken.

Die Lösung besteht darin, stellvertretend ein anderes Objekt, ein sogenanntes **Bild-Proxy**, zu verwenden, das als Platzhalter für das reale Bild verwendet wird. Das Proxy verhält sich genauso wie das Bild und sorgt außerdem im Bedarfsfall für dessen Instanziierung.

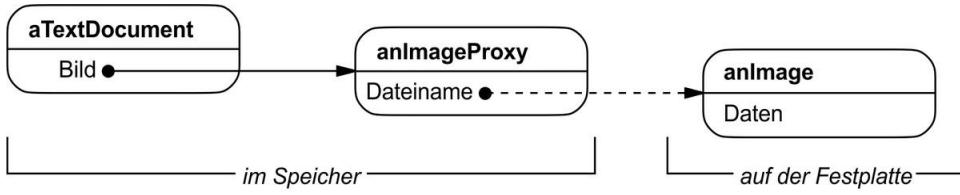


Abb. 4.36: Verwendung eines Bild-Proxys in einem Dokument

Das reale Bild wird nur dann erzeugt, wenn der Texteditor das Bild-Proxy durch den Aufruf der `Draw`-Operation anweist, es darzustellen. Da das Proxy alle darauffolgenden Requests direkt an das Bild weiterleitet, muss es auch nach der Erzeugung des Bildes eine Referenz darauf beibehalten.

Für den Fall, dass die Bilder in separaten Dateien gespeichert sind, können die Dateinamen als Referenz auf das jeweils richtige Objekt verwendet werden. Das Proxy speichert auch die Maße (engl. *Extent*) des Bildes, also dessen Breite und Höhe. Auf diese Weise kann es Requests bezüglich der Dimensionen des Bildes beantworten, ohne das Bild selbst tatsächlich instanziieren zu müssen.

Das in [Abbildung 4.37](#) dargestellte Klassendiagramm veranschaulicht dieses Beispiel im Detail:

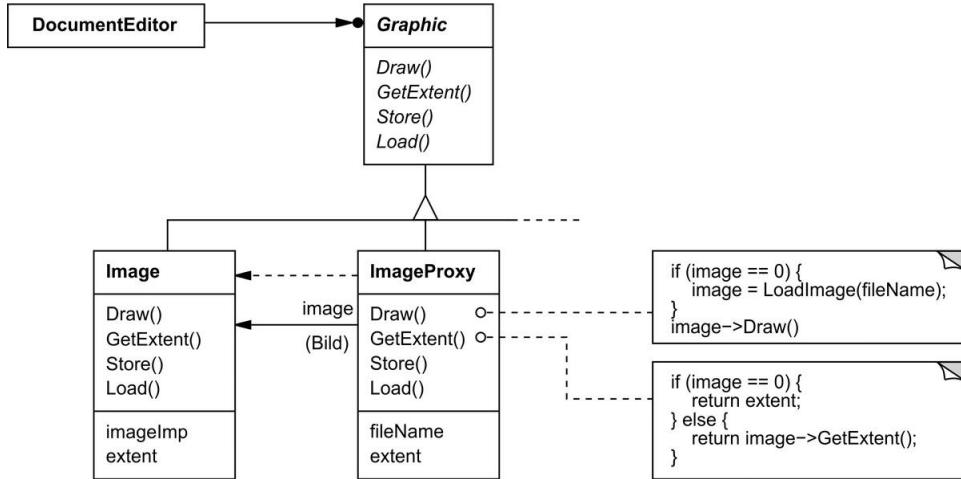


Abb. 4.37: Klassendiagramm zur Verwendung eines Bild-Proxys

Der Texteditor nutzt für den Zugriff auf eingebettete Bilder die von der abstrakten `Graphic`-Klasse definierte Schnittstelle. `ImageProxy` ist eine Klasse für Bilder, die auf Anforderung erzeugt werden. Sie enthält den Dateinamen als Referenz auf das Bild auf der Festplatte. Der Dateiname wird als Argument an den `ImageProxy`-Konstruktor übergeben.

Darüber hinaus sind auch der Begrenzungsrahmen sowie eine Referenz auf die

Instanz des realen Bildes in der `ImageProxy`-Klasse gespeichert. Diese Referenz wird erst dann gültig, wenn das Proxy das Bild instanziert. Die `Draw`-Operation überprüft vor der Weiterleitung des Requests an das Bild zunächst, ob die Instanziierung überhaupt stattgefunden hat. `GetExtent` übermittelt den Request nur dann an das Bild, wenn es instanziert wurde – andernfalls gibt die Klasse `ImageProxy` die von ihr gespeicherten Maße zurück.

Anwendbarkeit

Die Nutzung des Design Patterns *Proxy (Proxy)* ist immer dann sinnvoll, wenn über einen simplen Pointer hinausgehender Bedarf nach einer anpassungsfähigeren und intelligenteren Referenzierung eines Objekts besteht. Nachfolgend sind einige allgemeine Situationen beschrieben, für die das Pattern *Proxy (Proxy)* gut geeignet ist:

1. Ein **Remote Proxy** stellt einen lokalen Stellvertreter für ein Objekt in einem anderen Adressbereich zur Verfügung. NeXTStep [Add94] verwendet zu diesem Zweck die Klasse `NXProxy`. Coplien [Cop92] bezeichnet diese Art von Proxy als »Ambassador« (dt. »Botschafter«).
2. Ein **virtuelles Proxy (Virtual Proxy)** erzeugt ressourcenlastige Objekte auf Anforderung. Ein Beispiel dafür ist das im Abschnitt »Motivation« beschriebene Objekt `ImageProxy`.
3. Ein **Schutz-Proxy (Protection Proxy)** kontrolliert den Zugriff auf das Originalobjekt. Die Verwendung von Schutz-Proxys ist dann sinnvoll, wenn Objekte unterschiedliche Zugriffsrechte haben sollen. So ermöglichen beispielsweise im Betriebssystem Choices [CIRM93] die `KernelProxies` den geschützten Zugriff auf die zum Betriebssystem gehörigen Objekte.
4. Eine **Smart Reference** dient als Ersatz für einen einfachen Pointer und führt beim Zugriff auf ein Objekt zusätzliche Operationen aus. Typische Verwendungen sind zum Beispiel:
 - das Zählen der Referenzen auf das reale Objekt, so dass es automatisch wieder freigegeben werden kann, wenn keine weiteren Referenzen mehr existieren (auch **Smart Pointers** genannt [Ede92]).
 - das Laden eines persistenten Objekts in den Speicher, sobald es erstmals referenziert wird.

- die Überprüfung, ob das reale Objekt vor dem Zugriffsversuch gesperrt war, damit Manipulationen durch ein anderes Objekt ausgeschlossen sind.

Struktur

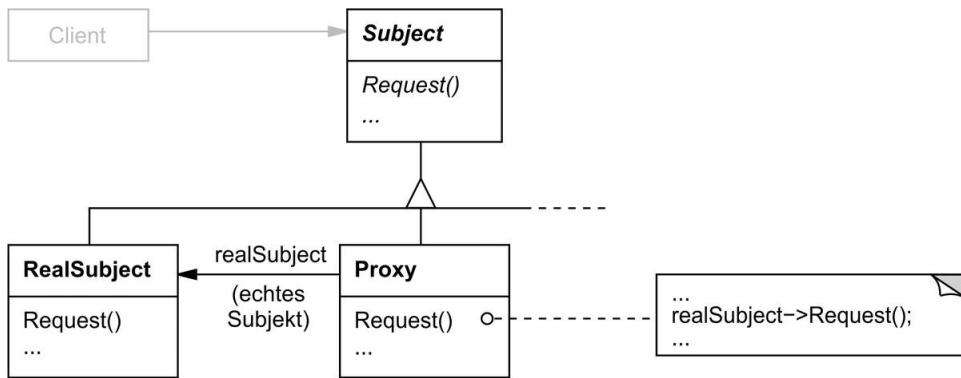


Abb. 4.38: Die Struktur des Design Patterns Proxy (Proxy)

Das Objektdiagramm einer Proxy-Struktur zur Laufzeit könnte wie folgt aussehen:

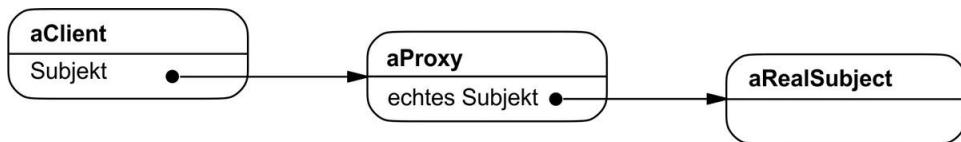


Abb. 4.39: Objektdiagramm einer Proxy-Struktur zur Laufzeit

Teilnehmer

- **Proxy (ImageProxy)**

- Verwaltet eine Referenz, die dem Proxy den Zugriff auf das reale Subjekt ermöglicht. Wenn die `RealSubject`- und `Subject`-Schnittstellen identisch sind, kann `Proxy` ein `Subject` gegebenenfalls referenzieren.
- Stellt eine mit der von `Subject` identische Schnittstelle zur Verfügung, so dass das reale Subjekt durch ein `Proxy` ersetzt werden kann.
- Kontrolliert den Zugriff auf das echte Subjekt und kann auch für dessen Erzeugung und Löschung zuständig sein.

- Weitere Zuständigkeiten sind von der Art des Proxys abhängig:
 - *Remote Proxys* sind für die Kodierung eines Requests und seiner Argumente sowie für die Übersendung des kodierten Requests an das echte Subjekt in einem anderen Adressbereich zuständig.
 - *Virtuelle Proxys* können zusätzliche Informationen über das echte Subjekt zwischenspeichern und damit den Zugriff darauf verzögern. So legt das im Abschnitt »Motivation« beschriebene *ImageProxy* beispielsweise die Maße des realen Bildes in den Zwischenspeicher.
 - *Schutz-Proxys* prüfen, ob der Aufrufer die für die Ausführung eines Requests erforderlichen Zugriffsrechte besitzt.
- **Subject (Graphic)**
 - Definiert die gemeinsame Schnittstelle für *RealSubject* und *Proxy*, so dass das *Proxy* überall dort eingesetzt werden kann, wo ein reales Subjekt erwartet wird.
- **RealSubject (Image)**
 - Definiert das von dem *Proxy* repräsentierte reale Objekt.

Interaktionen

- Je nach Art des Proxys leitet *Proxy* Requests im Bedarfsfall an *RealSubject* weiter.

Konsequenzen

Das Design Pattern *Proxy (Proxy)* ergänzt eine Dereferenzierungsebene für den Zugriff auf ein Objekt, die in Abhängigkeit von der Art des Proxys mehrere Zwecke erfüllt:

1. Ein Remote Proxy kann die Tatsache verbergen, dass sich ein Objekt in einem anderen Adressbereich befindet.
2. Ein virtuelles Proxy kann Optimierungen wie die Erzeugung eines Objekts auf

Anforderung durchführen.

3. Sowohl Schutz-Proxys als auch Smart References ermöglichen zusätzliche Verwaltungsaufgaben, sobald auf das Objekt zugegriffen wird.

Darüber hinaus kann das Pattern *Proxy* (*Proxy*) auch noch eine andere Optimierung vor dem Client verbergen: Das sogenannte **Copy-On-Write**-Verfahren ist mit der Objekterzeugung auf Anforderung verwandt. Das Kopieren eines umfangreichen und komplexen Objekts kann sich als eine recht ressourcenlastige Operation erweisen. Wenn die Kopie allerdings nie modifiziert wird, dann besteht auch kein Anlass, diesen Vorgang überhaupt durchzuführen. Mithilfe eines Proxy-Objekts zur Verzögerung des Kopiervorgangs lässt sich sicherstellen, dass dies nur dann passiert, wenn das Objekt definitiv verändert wird.

Damit das Copy-On-Write-Verfahren funktioniert, muss eine Referenzzählung auf das Subjekt ausgeführt werden. Das Kopieren des Proxy-Objekts bewirkt dabei nichts anderes als das Hochzählen dieser Referenzzählung. Erst wenn der Client eine Operation anfordert, die das Subjekt modifiziert, führt das Proxy den Kopiervorgang auch tatsächlich aus. In diesem Fall muss das Proxy die Referenzzählung für das Subjekt außerdem herunterzählen – und ist sie bei null angelangt, wird das Subjekt gelöscht.

Die Anwendung des Copy-On-Write-Verfahrens kann den Aufwand für das Kopieren von schwergewichtigen Subjekten erheblich reduzieren.

Implementierung

Mit dem Design Pattern *Proxy* (*Proxy*) lassen sich die folgenden sprachspezifischen Funktionen sinnvoll nutzen:

1. *Überladen (engl. Overloading) des Member-Zugriffsoperators in C++*. Die Programmiersprache C++ unterstützt das Überladen des Member-Zugriffsoperators `operator ->`. Dadurch können bei jeder Dereferenzierung eines Objekts zusätzliche Operationen ausgeführt werden, was für die Implementierung einiger Proxy-Varianten hilfreich sein kann, denn das Proxy verhält sich genau wie ein Pointer.

Das folgende Beispiel veranschaulicht die Anwendung dieser Technik anhand der Implementierung eines virtuellen Proxys namens `ImagePtr`:

```

class Image;
extern Image* LoadAnImageFile(const char* );
    // Externe Funktion

class ImagePtr {
public:
    ImagePtr(const char* imageFile);
    virtual ~ImagePtr();

    virtual Image* operator->();
    virtual Image& operator*();

private:
    Image* LoadImage();

private:
    Image * _image;
    const char* _imageFile;
};

ImagePtr::ImagePtr (const char* theImageFile) {
    _imageFile = theImageFile;
    _image = 0;
}

Image* ImagePtr::LoadImage () {
    if (_image ==0) {
        _image = LoadAnImageFile(_imageFile);
    }
    return _image;
}

```

Die Operatoren `overloaded->` und `operator*` verwenden `LoadImage`, um `_image` an die Aufrufer zurückzugeben (und, wenn nötig, zu laden):

```

Image* ImagePtr::operator-> () {
    return LoadImage();
}

Image& ImagePtr::operator* () {
    return *LoadImage();
}

```

Dieser Ansatz ermöglicht den Aufruf der `Image`-Operationen mithilfe von `ImagePtr`-Objekten, ohne dass sie erst zu einem Bestandteil der `ImagePtr`-Schnittstelle gemacht werden müssen:

```

ImagePtr image = ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
    // (image.operator->())->Draw(Point(50, 100))

```

Beachtenswert ist hier, dass das Proxy `image` zwar wie ein Pointer funktioniert, aber nicht als Pointer auf `Image` deklariert ist, d. h., es kann nicht wie ein echter Pointer auf `Image` verwendet werden. Die Clients müssen die Objekte `Image` und `ImagePtr` bei dieser Vorgehensweise daher entsprechend unterschiedlich behandeln.

Das Überladen des Member-Zugriffsoperators stellt jedoch nicht für jede Art von Proxy eine wirklich gute Lösung dar. Es gibt einige Proxys, die genauestens darüber informiert sein müssen, welche Operation aufgerufen wird – in diesen Fällen funktioniert das Überladen des Member-Zugriffsoperators nicht.

Dies wurde auch in dem im Abschnitt »Motivation« angeführten Beispiel des virtuellen Proxys deutlich: Hier sollte das Bild zu einem bestimmten Zeitpunkt geladen werden – nämlich beim Aufruf der `Draw`-Operation und nicht etwa bei einer Referenzierung des Bildes. Beim Überladen des Zugriffsoperators ist eine solche Unterscheidung jedoch nicht möglich: Hier muss jede Proxy-Operation, die den Request an das Subjekt weiterleitet, manuell implementiert werden.

Wie im Abschnitt »Beispielcode« beschrieben, ähneln sich diese Operationen im Allgemeinen sehr. Eine ihrer Gemeinsamkeiten besteht zum Beispiel darin, dass sie üblicherweise allesamt vor der Weiterleitung des Requests zunächst einmal prüfen, ob der Request überhaupt zulässig ist, ob das Originalobjekt existiert etc. Da es jedoch ziemlich mühselig wäre, den zugehörigen Code immer wieder neu schreiben zu müssen, wird meist ein Präprozessor verwendet, der ihn automatisch generiert.

2. *Verwendung von doesNotUnderstand in Smalltalk.* Zur Unterstützung der automatischen Weiterleitung von Requests stellt Smalltalk einen Eingriffspunkt zur Verfügung: Sobald ein Client eine Meldung an einen Empfänger sendet, der über keine entsprechende Methode verfügt, ruft Smalltalk die Operation `doesNotUnderstand: aMessage` auf. Die Klasse `Proxy` kann `doesNotUnderstand:` dann so umdefinieren, dass die Meldung an ihr Subjekt weitergeleitet wird.

Um sicherzustellen, dass die Weiterleitung des Requests auch tatsächlich erfolgt und er nicht bloß heimlich, still und leise im Proxy verschwindet, kann eine `Proxy`-Klasse definiert werden, die *keine* Meldungen versteht. In Smalltalk wird dies durch die Definition der Klasse `Proxy` ohne zugehörige Basisklasse realisiert.

Hinweis

Diese Technik wird im Betriebssystem NeXTStep [Add94] zur Implementierung von verteilten Objekten (genauer gesagt für die Klasse NXProxy) angewendet und definiert in diesem Fall forward als äquivalenten Eingriffspunkt.

Der größte Nachteil von doesNotUnderstand: besteht darin, dass es in den meisten Smalltalk-Systemen einige spezielle Meldungen gibt, die unmittelbar von der virtuellen Maschine bearbeitet und nicht über den üblichen Methoden-Look-up abgewickelt werden. Die einzige Methode, die üblicherweise in Object implementiert ist (und daher Proxys betreffen kann), ist die Identitätsoperation ==.

Wenn doesNotUnderstand: zur Implementierung von Proxy verwendet werden soll, muss diese Problematik im Design berücksichtigt werden – weil nicht davon ausgegangen werden kann, dass die Objektidentität der Proxys mit der ihrer realen Subjekte identisch ist. Ein weiterer Nachteil ist auch, dass doesNotUnderstand: für die Fehlerbehandlung und nicht zur Erstellung von Proxys entwickelt wurde, so dass es in der Regel nicht allzu zügig arbeitet.

3. *Mögliche Unkenntnis vom Typ des realen Subjekts aufseiten des Proxys.* Wenn der Umgang einer Proxy-Klasse mit ihrem Subjekt ausschließlich über eine abstrakte Schnittstelle erfolgt, ist es nicht notwendig, für jede Klasse RealSubject eine eigene Proxy-Klasse anzulegen – denn das Proxy kann dann alle RealSubject-Klassen einheitlich behandeln. Sollten die Proxys die RealSubjects allerdings instanziiieren (was z. B. auf ein virtuelles Proxy zutrifft), müssen sie auch deren konkrete Klassen kennen.

Eine weitere Implementierungsproblematik betrifft die Referenzierung des Subjekts vor seiner Instanziierung. Einige Proxys müssen ihre Subjekte unabhängig davon, ob sie sich auf der Festplatte oder im Speicher befinden, referenzieren. Und das bedeutet wiederum, dass sie auf vom Adressbereich unabhängige Objektidentifizierer zurückgreifen müssen. In dem im Abschnitt »Motivation« beschriebenen Beispiel wurde zu diesem Zweck ein Dateiname verwendet.

Beispielcode

Der folgende Code implementiert zwei Arten von Proxys: das im Abschnitt »Motivation« beschriebene virtuelle Proxy sowie ein mit `doesNotUnderstand` implementiertes Proxy:

Hinweis

In den Ausführungen zum Design Pattern *Iterator* (*Iterator*, siehe [Abschnitt 5.4](#)) wird noch eine weitere Proxy-Variante vorgestellt.

1. *Ein virtuelles Proxy*. Die `Graphic`-Klasse definiert die Schnittstelle für grafische Objekte:

```
class Graphic {
public:
    virtual ~Graphic();

    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;

    virtual const Point& GetExtent() = 0;

    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;

protected:
    Graphic();
};
```

Die `Image`-Klasse implementiert die `Graphic`-Schnittstelle, die die Bilddateien ausgibt. `Image` überschreibt `HandleMouse`, um dem User die interaktive Größenänderung des Bildes zu ermöglichen:

```
class Image : public Graphic {
public:
    Image(const char* file);
    // Lädt das Bild aus einer Datei
    virtual ~Image();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();
    virtual void Load(istream& from);
    virtual void Save(ostream& to);
```

```
private:  
    // ...  
};
```

ImageProxy besitzt dieselbe Schnittstelle wie Image:

```
class ImageProxy : public Graphic {  
public:  
    ImageProxy(const char* imageFile);  
    virtual ~ImageProxy();  
  
    virtual void Draw(const Point& at);  
    virtual void HandleMouse(Event& event);  
  
    virtual const Point& GetExtent();  
  
    virtual void Load(istream& from);  
    virtual void Save(ostream& to);  
  
protected:  
    Image* GetImage();  
  
private:  
    Image* _image;  
    Point _extent;  
    char* _fileName;  
};
```

Der Konstruktor speichert eine lokale Kopie des Namens der Datei, in der das Bild gespeichert ist, und initialisiert _extent und _image:

```
ImageProxy::ImageProxy (const char* fileName) {  
    _fileName = strdup(fileName);  
    _extent = Point::Zero;  
    // Bildmaße noch unbekannt  
    _image = 0;  
}  
  
Image* ImageProxy::GetImage() {  
    if (_image == 0) {  
        _image = new Image(_fileName);  
    }  
    return _image;  
}
```

Die Implementierung von GetExtent gibt, sofern verfügbar, die zwischengespeicherten Bildmaße zurück. Andernfalls wird das Bild aus der Datei geladen. Draw lädt das Bild, und HandleMouse leitet das Ereignis an das

reale Bild weiter:

```
const Point& ImageProxy::GetExtent () {
    if (_extent == Point::Zero) {
        _extent = GetImage()->GetExtent();
    }
    return _extent;
}

void ImageProxy::Draw (const Point& at) {
    GetImage()->Draw(at);
}

void ImageProxy::HandleMouse (Event& event) {
    GetImage()->HandleMouse(event);
}
```

Die Save-Operation speichert die Maße des zwischengespeicherten Bildes sowie den Bilddateinamen in einen Stream. Und Load ruft diese Daten wiederum ab und initialisiert die zugehörigen Membervariablen:

```
void ImageProxy::Save (ostream& to) {
    to << _extent << _fileName;
}

void ImageProxy::Load (istream& from) {
    from >> _extent >> _fileName;
}
```

Schließlich wird in diesem Beispiel außerdem das Vorhandensein einer Klasse TextDocument angenommen, die Graphic-Objekte enthalten kann:

```
class TextDocument {
public:
    TextDocument();

    void Insert(Graphic* );
    // ...
};
```

Das Einfügen eines ImageProxy in das Textdokument lässt sich damit wie folgt bewerkstelligen:

```
TextDocument* text = new TextDocument;
// ...
text->Insert(new ImageProxy("anImageFileName"));
```

2. Proxys, die doesNotUnderstand verwenden. Zur Erzeugung generischer Proxys

können in Smalltalk Klassen definiert werden, deren Basisklasse `nil` ist und die eine Methode `doesNotUnderstand:` nutzen, die so definiert ist, dass sie Meldungen bearbeitet.

Hinweis

Da fast alle Klassen letztendlich `Object` als Basisklasse haben, ist die Zuweisung des Wertes `nil` – wie in diesem Beispiel – praktisch gleichbedeutend mit der Definition einer Klasse, deren Basisklasse *nicht Object* ist.

Die folgende Methode setzt voraus, dass das Proxy über eine Methode `realSubject` verfügt, die das reale Subjekt zurückgibt. Im Fall von `ImageProxy` würde sie prüfen, ob das Bild erzeugt wurde, es nötigenfalls selbst erzeugen und es schließlich zurückgeben. Darüber hinaus verwendet diese Methode `perform:withArguments:`, um die abgefangene Meldung auf das reale Subjekt anzuwenden:

```
doesNotUnderstand: aMessage
    ^ self realSubject
        perform: aMessage selector
        withArguments: aMessage arguments
```

Das Argument für `doesNotUnderstand:` ist eine Instanz von `Message`, die die vom Proxy nicht verstandene Meldung enthält. Somit antwortet das Proxy auf alle Meldungen, indem es zunächst sicherstellt, dass das reale Subjekt existiert, bevor es die Meldung dorthin weiterleitet.

Einer der Vorteile von `doesNotUnderstand:` besteht darin, dass es beliebige Verarbeitungsprozesse ausführen kann. Beispielsweise könnte durch die Spezifikation eines Satzes `legalMessages` mit zu akzeptierenden Meldungen ein Schutz-Proxy erstellt werden, dem dann folgende Methode übergeben wird:

```
doesNotUnderstand: aMessage
    ^ (legalMessages includes: aMessage selector)
        ifTrue: [self realSubject
            perform: aMessage selector
            withArguments: aMessage arguments]
        ifFalse: [self error: 'Illegal operator']
```

Diese Methode überprüft vor der Weitergabe der Meldung an das reale Subjekt,

ob sie zulässig ist. Sollte das nicht der Fall sein, sendet sie `error`: an das Proxy – und löst damit eine Endlosschleife von Fehlern aus, sofern `error`: nicht im Proxy definiert ist. Aus diesem Grund sollte die Definition von `error`: mitsamt aller dazugehörigen Methoden von der Klasse `Object` kopiert bzw. übernommen werden.

Praxisbeispiele

Das im Abschnitt »Motivation« angeführte Beispiel zur Verwendung eines virtuellen Proxys entstammt den **ET++-Textbausteinklassen**.

NeXTStep [Add94] setzt Proxys (Instanzen der Klasse `NXProxy`) als lokale Repräsentanten für potenziell verteilte Objekte ein. Ein Server erstellt Proxys für entfernte Objekte, wenn die Clients sie anfordern. Beim Empfang einer Meldung kodiert das Proxy diese mitsamt ihrer Argumente und leitet die kodierte Fassung dann an das entfernte Subjekt weiter. In ähnlicher Weise kodiert auch das Subjekt jegliche Rückgabewerte und sendet diese wiederum an das `NXProxy`-Objekt zurück.

Genauere Erläuterungen zum Einsatz von Proxys für den Zugriff auf entfernte Objekte in Smalltalk finden sich bei **McCullough** [McC87]. **Pascoe** [Pas86] beschreibt die Anwendung von Nebeneffekten auf Methodenaufrufe sowie die Zugriffssteuerung mithilfe von »Encapsulators« (dt. »Kapselungsobjekte«).

Verwandte Patterns

Adapter (*Adapter*, siehe [Abschnitt 4.1](#)): Ein Adapter-Objekt liefert eine andere Schnittstelle als die des adaptierten Objekts. Im Gegensatz dazu stellt ein Proxy-Objekt dieselbe Schnittstelle wie sein Subjekt bereit. Ein für die Zugriffskontrolle verwendetes Proxy kann allerdings die Ausführung einer Operation, die das Subjekt durchführen würde, verweigern, so dass die Schnittstelle effektiv lediglich eine Untermenge der Schnittstelle des Subjekts darstellen kann.

Decorator (*Dekorierer*, siehe [Abschnitt 4.4](#)): Obwohl Decorator-Objekte ähnliche Implementierungen aufweisen können wie Proxys, erfüllen sie einen anderen Zweck: Während ein Decorator-Objekt ein Objekt um eine oder mehrere Zuständigkeiten erweitert, kontrolliert ein Proxy den Zugriff auf ein Objekt.

Der Ähnlichkeitsgrad der Implementierungen von Proxy- und Decorator-Objekten

variiert. Ein Schutz-Proxy kann gegebenenfalls exakt wie ein Decorator-Objekt implementiert sein. Ein Remote-Proxy enthält dagegen in aller Regel keine direkte, sondern lediglich eine indirekte Referenz auf sein reales Subjekt, beispielsweise »Host-ID und lokale Adresse auf dem Host«. Und ein virtuelles Proxy verwendet zwar zunächst eine indirekte Referenz, wie z. B. einen Dateinamen, erhält und nutzt letztendlich jedoch eine direkte Referenz.

4.8 Weitere Erläuterungen zu den Strukturmustern

Die Strukturmuster weisen insbesondere im Hinblick auf ihre Teilnehmer und Interaktionen offenkundige Ähnlichkeiten auf. Das ist vor allem darauf zurückzuführen, dass sie alle auf demselben überschaubaren Fundus von Sprachmechanismen zur Strukturierung von Code und Objekten basieren: Einfach- und Mehrfachvererbung für klassenbasierte Patterns und Objektkomposition für objektbasierte Patterns. Diese Ähnlichkeiten sollten jedoch keinesfalls über die unterschiedlichen Zielsetzungen der Patterns hinwegtäuschen – deshalb werden die Strukturmuster nachfolgend zum besseren Verständnis noch einmal in Gruppen gegenübergestellt und ihre individuellen Vorteile hervorgehoben.

4.8.1 *Adapter* (*Adapter*, siehe [Abschnitt 4.1](#)) kontra *Bridge* (*Brücke*, siehe [Abschnitt 4.2](#))

Die Design Patterns *Adapter* (*Adapter*) und *Bridge* (*Brücke*) weisen einige gemeinsame Eigenschaften auf: Beide fördern die Flexibilität, indem sie eine zusätzliche Dereferenzierung für den Zugriff auf ein anderes Objekt verwenden. Und ebenso beinhalten sie beide die von einer objektfremden Schnittstelle ausgehende Weiterleitung von Requests und Befehlen an dieses Objekt.

Das wichtigste Unterscheidungsmerkmal dieser zwei Patterns ist dagegen ihre individuelle Zielsetzung: Die Hauptaufgabe des Patterns *Adapter* (*Adapter*) besteht in der Auflösung von Inkompatibilitäten zwischen existierenden Schnittstellen. Dabei ist sowohl die Art und Weise, wie diese Schnittstellen implementiert sind als auch ihre voneinander unabhängige Entwicklung unerheblich. Dieses Design Pattern bietet eine Möglichkeit, zwei autark entworfene Klassen zur Zusammenarbeit zu bewegen, ohne sie erneut implementieren zu müssen. Das Pattern *Bridge* (*Brücke*) bildet dagegen eine Brücke zwischen einer Abstraktion und ihren (unter Umständen recht zahlreichen) Implementierungen. Es stellt den Clients eine stabile Schnittstelle

zur Verfügung, ohne jedoch die Variationsmöglichkeiten der Klassen, die das Pattern implementieren, einzuschränken. Zudem gewährleistet es die Anpassung neuer Implementierungen, die sich aus der Weiterentwicklung des Systems ergeben.

Angesichts dieser Unterschiede werden die Design Patterns *Adapter (Adapter)* und *Bridge (Brücke)* oftmals zu verschiedenen Zeitpunkten innerhalb der Evolution eines Systems eingesetzt. Adapter-Objekte werden häufig dann benötigt, wenn sich herausstellt, dass zwei inkompatible Klassen zusammenarbeiten sollen – üblicherweise um eine Doppelung des Codes zu vermeiden – und diese Paarung nicht vorhersehbar war. Im Gegensatz dazu ist bei der Verwendung eines Bridge-Objekts von vornherein klar, dass eine Abstraktion über mehrere Implementierungen verfügen muss und dass sie sich unabhängig voneinander entwickeln werden. Das Design Pattern *Adapter (Adapter)* ermöglicht die gemeinsame Nutzung von Klassen *nach* ihrem Entwurf. Das Pattern *Bridge (Brücke)* stellt hingegen schon *vor* ihrem Entwurf sicher, dass sie zusammenarbeiten können. Dies bedeutet jedoch nicht, dass das Pattern *Adapter (Adapter)* dem Pattern *Bridge (Brücke)* unterlegen ist – faktisch nehmen sie sich lediglich jeweils einer anderen Problemstellung an.

Nun könnte man auch ein Facade-Objekt (siehe Design Pattern *Facade (Fassade)*, [Abschnitt 4.5](#)) als eine Art Adapter-Objekt für einen Satz anderer Objekte betrachten. Diese Interpretation würde allerdings die Tatsache außer Acht lassen, dass das Facade-Objekt eine *neue* Schnittstelle definiert, während ein Adapter-Objekt *bereits vorhandene* Schnittstellen wiederverwendet und zur Zusammenarbeit bewegt.

4.8.2 Composite (Kompositum, siehe [Abschnitt 4.3](#)) kontra Decorator (Dekorierer, siehe [Abschnitt 4.4](#)) kontra Proxy (Proxy, siehe [Abschnitt 4.7](#))

Die Design Patterns *Composite (Kompositum)* und *Decorator (Dekorierer)* besitzen ähnliche Strukturen, die erkennen lassen, dass sie auf dem Prinzip der rekursiven Komposition zur Strukturierung einer uneingeschränkten Menge von Objekten basieren. Diese Gemeinsamkeit kann dazu verleiten, ein Decorator-Objekt als ein degeneriertes Composite-Objekt zu betrachten, womit allerdings der eigentliche Sinn des Design Patterns *Decorator (Dekorierer)* ignoriert würde. Vielmehr beginnen die Gemeinsamkeiten der Patterns *Composite (Kompositum)* und *Decorator (Dekorierer)* mit der rekursiven Komposition – und enden dort auch gleich wieder, weil sie verschiedene Zielsetzungen verfolgen.

Das Design Pattern *Decorator* (*Dekorierer*) ist so strukturiert, dass die Objekte ohne Unterklassenbildung um weitere Funktionalität erweitert werden können. Dadurch wird ein explosionsartiger zahlenmäßiger Anstieg von Unterklassen vermieden, der im Fall einer statischen Definition jeglicher Kombination von Zuständigkeiten entstehen könnte. Das Design Pattern *Composite* (*Kompositum*) erfüllt hingegen einen anderen Zweck: Es konzentriert sich auf die Strukturierung von Klassen, damit möglichst viele verwandte Objekte einheitlich und mehrere Objekte wie ein einziges behandelt werden können. Der Schwerpunkt dieses Patterns liegt nicht auf der Ausgestaltung der Objekte, sondern auf deren Darstellung.

Trotz ihrer unterschiedlichen Zielsetzungen sind die Patterns *Decorator* (*Dekorierer*) und *Composite* (*Kompositum*) dennoch komplementär einsetzbar – und dementsprechend werden sie auch häufig gemeinsam verwendet. Beide ermöglichen jene Art von Design, bei dem Anwendungen einfach durch das Zusammenfügen von Objekten erstellt werden können, ohne dass die Definition neuer Klassen erforderlich ist: eine abstrakte Klasse mit ein paar Unterklassen, von denen einige Composite-Objekte und andere Decorator-Objekte darstellen, sowie ein paar Unterklassen, die die grundlegenden Bausteine des Systems implementieren. In diesem Fall verfügen die Decorator- und die Composite-Objekte über eine gemeinsame Schnittstelle. Aus Sicht des Patterns *Decorator* (*Dekorierer*) ist ein Composite-Objekt ein *ConcreteComponent*, also eine konkrete Komponente. Und aus Sicht des Patterns *Composite* (*Kompositum*) ist ein Decorator-Objekt ein *Leaf*-Objekt, sprich ein Blattobjekt. Natürlich müssen sie nicht gemeinsam genutzt werden – immerhin dienen sie doch recht unterschiedlichen Zwecken.

Das Design Pattern *Proxy* (*Proxy*, siehe [Abschnitt 4.7](#)) weist ebenfalls eine ähnliche Struktur auf wie das Pattern *Decorator* (*Dekorierer*). Beide Patterns beschreiben die Bereitstellung einer Ebene der Dereferenzierung zu einem Objekt. Sowohl die Implementierung eines Proxy- als auch die eines Decorator-Objekts enthält jeweils eine Referenz auf ein anderes Objekt, an das sie Requests und Befehle weiterleitet. Aber auch hier gilt, dass verschiedene Zielsetzungen verfolgt werden.

Wie das Pattern *Decorator* (*Dekorierer*) fügt auch das Design Pattern *Proxy* (*Proxy*) ein Objekt zusammen und stellt den Clients eine identische Schnittstelle zu Verfügung – allerdings befasst es sich nicht mit dem dynamischen Hinzufügen und Entfernen von Eigenschaften und ist prinzipiell auch nicht für die rekursive Komposition ausgelegt. Stattdessen besteht seine Zielsetzung vielmehr darin, einen Stellvertreter für ein Subjekt bereitzustellen, wenn der direkte Zugriff auf das Subjekt unkomfortabel oder unerwünscht ist, weil es beispielsweise auf einer anderen entfernten Maschine liegt, lediglich ein eingeschränkter Zugriff möglich ist oder es persistent ist.

Beim Design Pattern *Proxy* (*Proxy*) wird die maßgebliche Funktionalität durch das Subjekt definiert, und das *Proxy*-Objekt ermöglicht oder verhindert den Zugriff darauf. Beim Design Pattern *Decorator* (*Dekorierer*) stellt die Komponente lediglich einen Teil der Funktionalität zur Verfügung, während ein oder mehrere *Decorator*-Objekte alles Übrige liefern. Es ist vor allem in solchen Situationen nützlich, in denen die Funktionalität eines Objekts beim Kompilieren gar nicht oder nur auf Umwegen bestimmt werden kann. Durch diese Unbestimmtheit wird die rekursive Komposition zu einem zentralen Bestandteil des Patterns *Decorator* (*Dekorierer*). Das ist beim Pattern *Proxy* (*Proxy*) hingegen nicht der Fall, weil es sich auf eine einzelne Beziehung – zwischen dem *Proxy* und dem Subjekt – konzentriert und diese Beziehung statisch ausgedrückt werden kann.

Die hier beschriebenen Unterschiede sind insofern von Bedeutung, als sie die Lösungen zu bestimmten immer wiederkehrenden Problemen in objektorientierten Designs erfassen. Das bedeutet allerdings nicht, dass diese Patterns nicht miteinander kombiniert werden könnten. So wären zum Beispiel eine *Proxy-Decorator*-Kombination zur Erweiterung eines *Proxy*-Objekts um zusätzliche Funktionalität oder eine *Decorator-Proxy*-Kombination zur Dekoration eines entfernten Objekts durchaus vorstellbar. Aber auch wenn solche Hybridkombinationen möglicherweise *hilfreich sein könnten* (gute Beispiele dafür liegen uns allerdings nicht vor), lassen sie sich jedenfalls immer auch auf einzelne Patterns verteilen, die *definitiv hilfreich sind*.

Kapitel 5: Verhaltensmuster (Behavioral Patterns)

Verhaltensmuster (Behavioral Patterns) beschäftigen sich schwerpunktmäßig mit Algorithmen und der Zuweisung von Zuständigkeiten an Objekte. Sie beschreiben nicht nur Patterns von Objekten und Klassen, sondern auch deren wechselseitige Kommunikationsmuster. Zudem erfassen sie komplexe Programmabläufe, die zur Laufzeit nur schwer nachzuvollziehen sind – und verlagern somit den Fokus vom Programmablauf auf die Art und Weise, wie die Objekte miteinander verbunden sind.

Klassenbasierte Verhaltensmuster wenden für die Verhaltenszuordnung zu den Klassen das Vererbungsprinzip an. In diesem Kapitel werden zwei klassenbasierte Patterns vorgestellt. Das einfachere und gebräuchlichere Design Pattern *Template Method (Schablonenmethode*, siehe [Abschnitt 5.10](#)) repräsentiert eine schrittweise abstrakte Definition eines Algorithmus, wobei jeder Schritt entweder eine abstrakte oder eine primitive Operation auslöst und die abstrakten Operationen zur Ausgestaltung des Algorithmus von einer UnterkLASSE definiert werden. Bei dem zweiten klassenbasierten Verhaltensmuster handelt es sich um das Design Pattern *Interpreter (Interpreter*, siehe [Abschnitt 5.3](#)), das eine Grammatik als Klassenhierarchie darstellt und einen Interpreter als Anwendung einer Operation auf die Instanzen dieser Klassen implementiert.

Objektbasierte Verhaltensmuster machen sich statt des Vererbungsprinzips die Objektkomposition zunutze. Einige dieser Patterns beschreiben die Zusammenarbeit einer Gruppe von Peer-Objekten zur Ausführung eines Tasks, der von einem Objekt allein nicht bewerkstelligt werden kann. Ein wichtiger Faktor ist hierbei, in welcher Form die Peer-Objekte Kenntnis voneinander haben. Zwar könnten sie sich gegenseitig explizit referenzieren, dadurch würden sie jedoch auch stärker miteinander gekoppelt – im Extremfall hätte sogar jedes einzelne Objekt Kenntnis von allen anderen Objekten. Damit dies nicht passiert, stellt das Design Pattern *Mediator (Vermittler*, siehe [Abschnitt 5.5](#)) ein Mediator-Objekt zur Verfügung, das zwischen den Peer-Objekten vermittelt und die für die Aufrechterhaltung loser Kopplungen erforderlichen Dereferenzierungen gewährleistet.

Das Design Pattern *Chain of Responsibility (Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) ermöglicht sogar eine noch losere Kopplungsstruktur, indem es die Verkettung

geeigneter Objektkandidaten zur impliziten Request-Übermittlung an ein Objekt zulässt. Ausschlaggebend ist hierbei, dass grundsätzlich jedes einzelne Glied in der Kette in Abhängigkeit von den vorliegenden Laufzeitbedingungen in der Lage ist, den Request anzunehmen und auszuführen. Darüber hinaus kann die Kette aus einer unbegrenzten Anzahl von Objektkandidaten bestehen, deren Auswahl und Integration zudem zur Laufzeit erfolgen kann.

Das Design Pattern *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)) definiert und verwaltet die Abhängigkeiten zwischen den Objekten. Ein klassisches Beispiel für den Einsatz dieses Patterns findet sich im Smalltalk Model/View/Controller-Architektschema (kurz: Smalltalk MVC), in dem alle View-Objekte des Model-Objekts über Zustandsänderungen des Model-Objekts benachrichtigt werden.

Andere objektbasierte Verhaltensmuster befassen sich wiederum vor allem mit der Kapselung des Verhaltens in einem Objekt sowie der Delegierung von Requests an dieses Objekt: Das Design Pattern *Strategy* (*Strategie*, siehe [Abschnitt 5.9](#)) kapselt einen Algorithmus in einem Objekt und erleichtert die Spezifikation sowie Modifikation des von einem Objekt verwendeten Algorithmus. Das Design Pattern *Command* (*Befehl*, siehe [Abschnitt 5.2](#)) kapselt einen Request in einem Objekt, so dass er als Parameter weitergegeben, in einem Befehlsverlauf gespeichert oder in anderer Weise manipuliert werden kann. Das Design Pattern *State* (*Zustand*, siehe [Abschnitt 5.8](#)) kapselt die Zustände eines Objekts und versetzt es so in die Lage, sein Verhalten im Fall einer Zustandsänderung entsprechend anzupassen. Das Design Pattern *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)) kapselt Verhalten, das andernfalls über die Klassen verteilt würde. Und das Design Pattern *Iterator* (*Iterator*, siehe [Abschnitt 5.4](#)) abstrahiert die Art des Zugriffs auf und der Traversierung von Objekten in einem Aggregat.

5.1 Chain of Responsibility (Zuständigkeitskette)

Objektbasiertes Verhaltensmuster

Zweck

Vermeidung der Kopplung eines Request-Absenders mit seinem Empfänger, indem mehr als ein Objekt in die Lage versetzt wird, den Request zu bearbeiten. Die

empfangenden Objekte werden miteinander verkettet und der Request wird dann so lange entlang dieser Kette weitergeleitet, bis er von einem Objekt angenommen und bearbeitet wird.

Motivation

Zur Veranschaulichung der Arbeitsweise des Design Patterns *Chain of Responsibility (Zuständigkeitskette)* soll an dieser Stelle ein kontextsensitives Hilfesystem für eine grafische Benutzeroberfläche als Beispiel dienen, das es dem User ermöglicht, durch Anklicken eines beliebigen Elements auf der Benutzeroberfläche zugehörige Hilfetexte abzurufen. Die Art der bereitgestellten Informationen ist dabei von dem gewählten Element und dessen aktuellem Kontext abhängig. So könnte beispielsweise für eine Schaltfläche in einem Dialogfenster ein anderer Hilfetext hinterlegt sein als für eine vergleichbare Schaltfläche im Hauptfenster. Für den Fall, dass keine spezifischen Informationen zu dem betreffenden Element verfügbar sind, sollte das Hilfesystem einen allgemeineren Informationstext zu dem unmittelbaren Kontext ausgeben, wie z. B. zu dem Dialogfenster, in dem es sich befindet.

Demnach ist es nur folgerichtig, die Hilfetexte nach dem Allgemeinheitsgrad ihres Informationsgehalts zu organisieren, angefangen mit dem detailliertesten bis hin zum allgemeinsten Informationsgehalt. Weiterhin ist klar, dass ein eingehender Hilfe-Request von einem der vielen Objekte der Benutzeroberfläche bearbeitet wird – welches das allerdings sein wird, hängt vom jeweiligen Kontext und der Detailgenauigkeit der verfügbaren Hilfeinformationen ab.

Das Problem dabei ist, dass das Objekt, das den Hilfetext *bereitstellt*, dem Objekt (z. B. der Schaltfläche), das den Hilfe-Request *auslöst*, nicht explizit bekannt ist. Deshalb muss (in diesem Fall) die Schaltfläche, die den Hilfe-Request absetzt, von den Objekten, die eine passende Hilfestellung anbieten, entkoppelt werden. Und das Design Pattern *Chain of Responsibility (Zuständigkeitskette)* definiert, in welcher Art und Weise dies geschieht.

Ziel dieses Patterns ist die Entkopplung von Absender und Empfänger, indem eine Möglichkeit geschaffen wird, dass ein Hilfe-Request von mehreren Objekten bearbeitet werden kann. Der Request wird dann entlang einer Kette von Objekten weitergeleitet, bis eins von ihnen sich seiner annimmt:

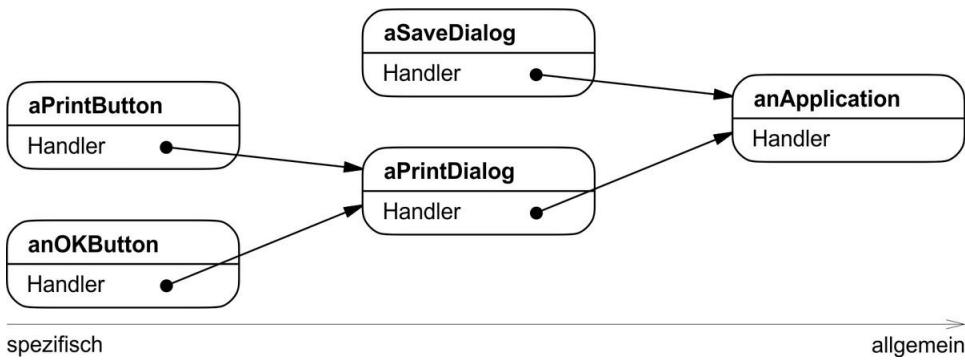


Abb. 5.1: Bearbeitungsoptionen für einen Hilfe-Request

Das erste Objekt in der Kette erhält den Request und nimmt ihn entweder zur Bearbeitung entgegen oder reicht ihn an sein Nachfolgeobjekt weiter, das wiederum die gleichen Handlungsoptionen hat. Dem auslösenden Objekt ist nicht explizit bekannt, welches Objekt den Request am Ende tatsächlich bearbeiten wird – der Request hat also einen **impliziten Empfänger**.

Angenommen, der User setzt einen Hilfe-Request für ein mit DRUCKEN beschriftetes Schaltflächen-Widget ab. Diese Schaltfläche befindet sich in einer Instanz der Klasse `PrintDialog`, der ihrerseits bekannt ist, welchem Application-Objekt sie zugeordnet ist (siehe Abbildung 5.1). Das in Abbildung 5.2 dargestellte Interaktionsdiagramm verdeutlicht die Weiterleitung des Hilfe-Requests entlang der Zuständigkeitskette:

aPrintButton aPrintDialog anApplication

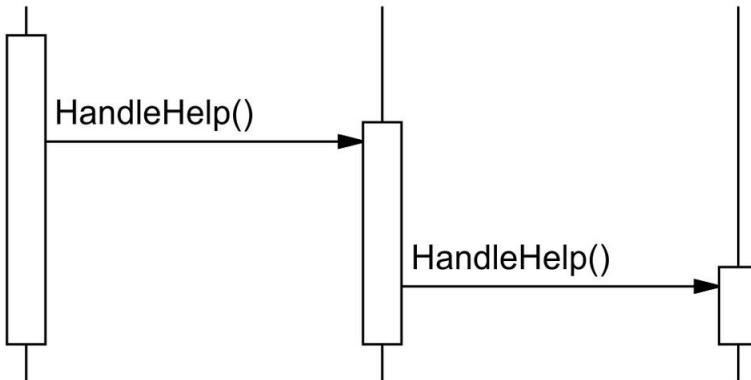


Abb. 5.2: Weiterleitung eines Hilfe-Requests in der Zuständigkeitskette

Hier wird der Hilfe-Request weder von `aPrintButton` noch von `aPrintDialog` angenommen, sondern zum Objekt `anApplication` durchgereicht, das ihn entweder bearbeiten oder ignorieren kann. Der Client, der den Request ausgelöst hat, besitzt keine direkte Referenz auf das Objekt, das ihn schlussendlich bearbeitet.

Zur Weiterleitung des Requests entlang der Zuständigkeitskette und um sicherzustellen, dass die Empfänger implizit bleiben, verwenden alle Objekte in der Kette für das Handling der Requests und den Zugriff auf ihre jeweiligen **Nachfolgeobjekte** (engl. *Successors*) eine gemeinsame Schnittstelle. Beispielsweise könnte das Hilfesystem eine Klasse `HelpHandler` mit einer zugehörigen `HandleHelp`-Operation definieren. Diese Klasse kann von vornherein als Basisklasse für die Klassen der Objektkandidaten oder auch als Mixin-Klasse definiert sein – dann können Klassen, die Hilfe-Requests bearbeiten wollen, `HelpHandler` zu ihrer Basisklasse machen:

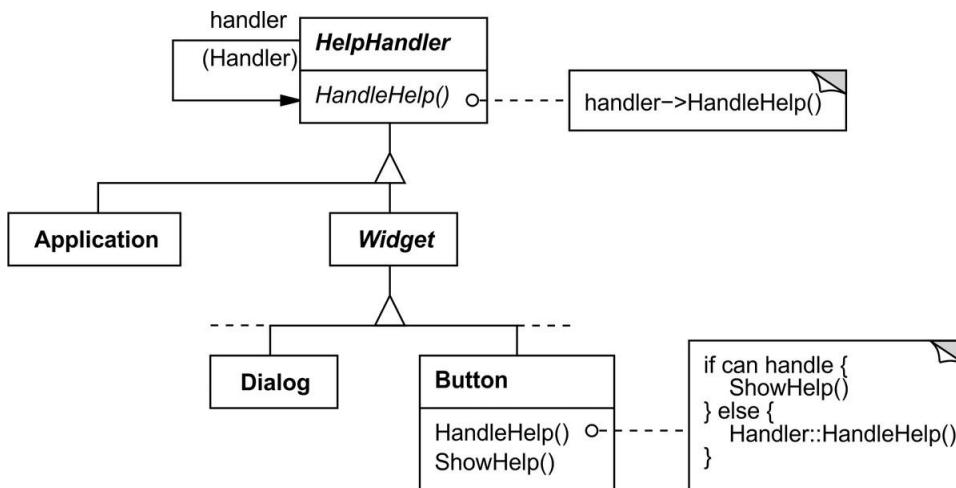


Abb. 5.3: Die Klasse HelpHandler

Die Klassen `Button`, `Dialog` und `Application` nutzen die `HelpHandler`-Operationen zur Bearbeitung von Hilfe-Requests. Die Operation `HandleHelp` der Klasse `HelpHandler` leitet den Request standardmäßig an das Nachfolgeobjekt weiter. Unterklassen können diese Operation überschreiben, um unter den richtigen Umständen Hilfeinformationen bereitzustellen – oder ansonsten die Standardimplementierung verwenden, um den Request weiterzuleiten.

Anwendbarkeit

Der Einsatz des Design Patterns *Chain of Responsibility (Zuständigkeitskette)* bietet sich an, wenn:

- Requests von mehr als einem Objekt bearbeitet können werden sollen und der Handler (dt. Bearbeiter) nicht *von vornherein* bekannt ist, sondern automatisch zur Laufzeit bestimmt werden soll.

- ein Request ohne explizite Angabe des Empfängers an eins von mehreren Objekten gerichtet werden soll.
- der Objektsatz, der einen Request bearbeiten kann, dynamisch bestimmt werden soll.

Struktur

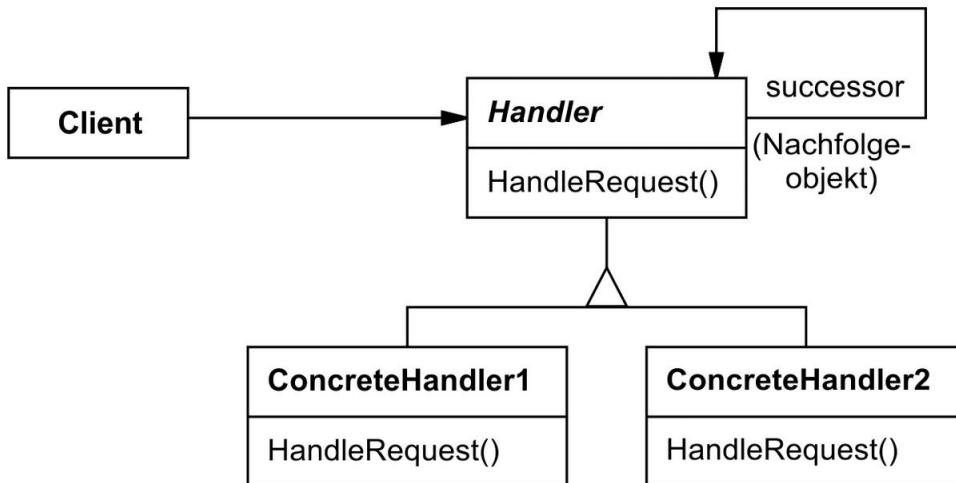


Abb. 5.4: Die Struktur des Design Patterns *Chain of Responsibility* (Zuständigkeitskette)

Eine typische Objektstruktur könnte wie folgt aussehen:



Abb. 5.5: Mögliche Objektstruktur einer Zuständigkeitskette

Teilnehmer

- **Handler (HelpHandler)**
 - Definiert eine Schnittstelle für die Bearbeitung von Requests.
 - (*Optional*) Implementiert die Verknüpfung zum Nachfolgeobjekt.

- **ConcreteHandler** (`PrintButton`, `PrintDialog`)
 - Bearbeitet den Request, der seiner Zuständigkeit unterliegt.
 - Kann auf sein Nachfolgeobjekt zugreifen.
 - Sofern `ConcreteHandler` den Request bearbeiten kann, führt er ihn auch aus – andernfalls leitet er ihn an sein Nachfolgeobjekt weiter.
- **Client**
 - Initiiert den Request an ein `ConcreteHandler`-Objekt in der Kette.

Interaktionen

- Wenn ein Client einen Request ausgibt, wird dieser entlang der Kette weitergeleitet, bis ein `ConcreteHandler`-Objekt die Aufgabe übernimmt, ihn zu bearbeiten.

Konsequenzen

Das Design Pattern *Chain of Responsibility (Zuständigkeitskette)* begünstigt bzw. bewirkt Folgendes:

1. *Reduzierte Kopplung.* Das Pattern entbindet ein Objekt von der Notwendigkeit, Kenntnis davon haben zu müssen, *welches* andere Objekt einen Request bearbeitet. Es muss lediglich wissen, *dass* ein Request »adäquat« bearbeitet werden wird. Sowohl der Empfänger als auch der Absender haben keine explizite Kenntnis voneinander, und die Objekte in der Kette benötigen keinerlei Informationen über die Kettenstruktur.

Damit vereinfacht das Design Pattern *Chain of Responsibility (Zuständigkeitskette)* auch die Beziehungen zwischen den Objekten: Statt Referenzen auf alle anderen Objektkandidaten enthalten zu müssen, reicht eine einfache Referenz auf ihr jeweiliges Nachfolgeobjekt aus.

2. *Zusätzliche Flexibilität bei der Zuweisung von Zuständigkeiten an Objekte.* Das Design Pattern *Chain of Responsibility (Zuständigkeitskette)* bietet eine hohe Flexibilität hinsichtlich der Verteilung der Zuständigkeiten unter den Objekten:

Die Zuständigkeit für die Bearbeitung eines Requests kann durch die Ergänzung weiterer Objekte oder sonstige Modifikationen innerhalb der Kette zur Laufzeit zugewiesen oder geändert werden. In Kombination mit der Bildung von Unterklassen ermöglicht diese Vorgehensweise zudem die statische Spezialisierung der Handler.

3. *Keine Bearbeitungsgarantie.* Da kein expliziter Empfänger für einen Request existiert, ist nicht *garantiert*, dass er tatsächlich ausgeführt wird, d. h.: Der Request kann selbst dann, wenn er am Ende der Kette angelangt ist, unbearbeitet geblieben sein. Das gilt auch, wenn die Kette nicht korrekt konfiguriert ist.

Implementierung

Bei der Implementierung des Design Patterns *Chain of Responsibility* (*Zuständigkeitskette*) sollten folgende Aspekte beachtet werden:

1. *Implementierung der Kette der Nachfolgeobjekte.* Die Kette der Nachfolgeobjekte kann auf zwei Arten implementiert werden:
 - a. durch die Definition neuer Verknüpfungen (normalerweise im Handler, ist aber auch in den ConcreteHandlers möglich),
 - b. durch die Verwendung existierender Verknüpfungen.

Während in den bisher angeführten Beispielen neue Verknüpfungen erstellt wurden, lassen sich häufig auch bereits vorhandene Objektreferenzen zur Bildung der Nachfolgerkette nutzen. So können zum Beispiel die Elternreferenzen in einer Teil-Ganzes-Hierarchie das Nachfolgeobjekt eines Kettenabschnitts definieren. Möglicherweise verfügt eine Widget-Struktur bereits über solche Verknüpfungen. Eine genauere Beschreibung der Elternreferenzen findet sich in den Ausführungen zum Design Pattern *Composite (Kompositum)*, siehe [Abschnitt 4.3](#).

2. *Verbinden der Nachfolgeobjekte.* Sollten keine bereits existierenden Referenzen für die Definition einer Kette vorhanden sein, müssen sie von Hand erstellt werden. In diesem Fall definiert der Handler nicht nur die Schnittstelle für die Requests, sondern verwaltet üblicherweise auch das Nachfolgeobjekt. Dadurch kann er eine Standardimplementierung der Klasse `HandleRequest` bereitstellen, die den Request an das Nachfolgeobjekt weiterleitet (sofern vorhanden). Sollte eine `ConcreteHandler`-Unterklasse nicht an dem Request interessiert sein, muss

sie die Weiterleitungsoperation nicht überschreiben, weil ihre Standardimplementierung ihn bedingungslos weiterleitet.

Eine Basisklasse, die eine Verknüpfung auf ein Nachfolgeobjekt verwaltet, könnte folgendermaßen aussehen:

```
class HelpHandler {  
public:  
    HelpHandler(HelpHandler* s) : _successor(s) { }  
    virtual void HandleHelp();  
private:  
    HelpHandler* _successor;  
};  
  
void HelpHandler::HandleHelp () {  
    if (_successor) {  
        _successor->HandleHelp();  
    }  
}
```

3. *Repräsentation von Requests.* Zur Darstellung von Requests stehen verschiedene Optionen zur Verfügung. In der einfachsten Form stellt der Request wie im Fall von `HandleHelp` einen hartkodierten Operationsaufruf dar. Diese Option ist bequem und sicher, allerdings kann hierbei lediglich der in der `Handler`-Klasse fest definierte Satz von Requests weitergeleitet werden.

Alternativ dazu kann auch eine einzelne `Handler`-Funktion verwendet werden, die einen Request-Code (z. B. eine Ganzzahl oder einen String) als Parameter entgegennimmt. Die Anzahl der möglichen Requests ist dabei nicht beschränkt. Einige Voraussetzung ist, dass der Absender und der Empfänger sich über die Verschlüsselung des Requests einig sind.

Dieser Ansatz bietet mehr Flexibilität, erfordert aber bedingte Anweisungen für die codebasierte Weiterleitung des Requests. Darüber hinaus ist keine typsichere Übergabe von Parametern möglich, so dass sie manuell ge- und entpackt werden müssen – und das ist naturgemäß unsicherer als der direkte Aufruf einer Operation.

Um der Problematik der Parameterübergabe zu begegnen, können separate *Request-Objekte* verwendet werden, die die zugehörigen Parameter bündeln. Eine Request-Klasse kann die Requests explizit darstellen, und die Definition neuer Request-Arten kann durch Unterklassen erfolgen, die ihrerseits verschiedene Parameter definieren können. Die Handler müssen zum Zugriff auf diese Parameter wissen, mit welcher Art von Request sie es zu tun haben

bzw. welche Request-Unterklasse sie benutzen.

Zur Identifizierung des Requests kann die Basisklasse Request eine Zugriffsfunktion definieren, die einen Identifizierer für die Klasse zurückgibt. Wenn die Implementierungssprache dies zulässt, kann der Empfänger alternativ auch die Laufzeittypinformation nutzen.

Der folgende Code skizziert eine Verteilungsfunktion, die die Request-Identifizierung mittels Request-Objekten vornimmt. In diesem Beispiel wird die Art des Requests über eine in der Basisklasse Request definierte GetKind-Operation ermittelt:

```
void Handler::HandleRequest (Request* theRequest) {  
    switch (theRequest->GetKind()) {  
        case Help:  
            // Argumentumwandlung in den passenden Typ  
            HandleHelp((HelpRequest*) theRequest);  
            break;  
  
        case Print:  
            HandlePrint((PrintRequest*) theRequest);  
            // ...  
            break;  
  
        default:  
            // ...  
            break;  
    }  
}
```

Unterklassen können die Verteilungsoperation durch Überschreiben von HandleRequest erweitern. Eine Unterklasse bearbeitet immer nur diejenigen Requests, die für sie von Interesse sind – alle anderen Requests werden an die Basisklasse weitergeleitet. Damit erweitert sie effektiv die HandleRequest-Operation (statt sie zu überschreiben). Der nachfolgende Code zeigt als Beispiel die Erweiterung der Handler-Version von HandleRequest durch eine Unterklasse ExtendedHandler:

```
class ExtendedHandler : public Handler {  
public:  
    virtual void HandleRequest(Request* theRequest);  
    // ...  
};  
  
void ExtendedHandler::HandleRequest (Request* theRequest) {  
    switch (theRequest->GetKind()) {  
        case Preview:  
            // ...  
    }  
}
```

```

    // Bearbeitet den Preview-Request
    break;

    default:
        // Lässt den Handler andere Requests bearbeiten
        Handler::HandleRequest(theRequest);
    }
}

```

4. *Automatische Weiterleitung in Smalltalk.* In Smalltalk können Requests mithilfe des doesNotUnderstand-Mechanismus weitergeleitet werden. Meldungen, die keine entsprechenden Methoden aufweisen, werden in der doesNotUnderstand-Implementierung abgefangen, die so überschrieben werden kann, dass sie die Meldung an ein Nachfolgeobjekt weiterleitet. Auf diese Weise erübrigts sich die manuelle Implementierung der Weiterleitung: Die Klasse bearbeitet nur solche Requests, die für sie interessant sind, und überlässt doesNotUnderstand die Weiterleitung aller anderen Requests.

Beispielcode

Das folgende Beispiel demonstriert die mögliche Request-Bearbeitung durch eine Zuständigkeitskette in einem Online-Hilfesystem wie dem eingangs beschriebenen. Der Hilfe-Request repräsentiert eine explizite Operation. In diesem Fall werden bereits existierende Elternreferenzen in der Widget-Hierarchie verwendet, um die Requests entlang der Widgets in der Kette weiterzuleiten. Außerdem wird zur Weiterleitung der Requests zwischen denjenigen Objekten in der Kette, bei denen es sich nicht um Widgets handelt, eine Referenz in der `Handler`-Klasse definiert.

Die Klasse `HelpHandler` definiert die Schnittstelle zur Bearbeitung von Hilfe-Requests. Sie verwaltet ein Hilfethema (das standardmäßig leer ist) und enthält eine Referenz auf ihr Nachfolgeobjekt in der Kette der Hilfe-Handler. `HandleHelp` ist hier die zentrale Operation, die von Unterklassen überschrieben wird. `HasHelp` ist dagegen eine Komfortfunktion zur Überprüfung auf das Vorhandensein zugehöriger Hilfetexte:

```

typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
    HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
}

```

```

    virtual void HandleHelp();

private:
    HelpHandler* _successor;
    Topic _topic;
};

HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}

void HelpHandler::HandleHelp () {
    if (_successor != 0) {
        _successor->HandleHelp();
    }
}

```

Alle Widgets sind Unterklassen der abstrakten `Widget`-Klasse. `Widget` ist ihrerseits wiederum eine Unterklasse von `HelpHandler`, da alle Elemente der Benutzeroberfläche mit einer zugehörigen Hilfeinformation verknüpft sein können. (Hier ließe sich aber ebenso gut eine Mixin-basierte Implementierung verwenden.)

```

class Widget : public HelpHandler {
protected:
    Widget(Widget* parent, Topic t = NO_HELP_TOPIC);

private:
    Widget* _parent;
};

Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) {
    _parent = w;
}

```

In diesem Beispiel ist der erste Handler in der Kette eine Schaltfläche. Die `Button`-Klasse ist eine Unterklasse von `Widget`. Der `Button`-Konstruktor nimmt zwei Parameter entgegen: eine Referenz auf das ihn umschließende `Widget` und eine auf das Hilfethema:

```

class Button : public Widget {
public:
    Button(Widget* d, Topic t = NO_HELP_TOPIC);

    virtual void HandleHelp();
    // Widget-Operationen, die von der Button-Klasse überschrieben
}

```

```
werden...
};
```

Die `HandleHelp`-Version von `Button` prüft zunächst, ob Hilfeinformationen für Schaltflächen vorhanden sind. Sollte dies nicht der Fall sein, dann wird der Request mithilfe der `HandleHelp`-Operation im `HelpHandler` an das Nachfolgeobjekt weitergeleitet. Ist dagegen ein zugehöriger Hilfetext *vorhanden*, zeigt die Schaltfläche ihn an und die Suche ist beendet:

```
Button::Button (Widget* h, Topic t) : Widget(h, t) { }

void Button::HandleHelp () {
    if (HasHelp()) {
        // Bietet Hilfeinformationen für die Schaltfläche an
    } else {
        HelpHandler::HandleHelp();
    }
}
```

Die Klasse `Dialog` implementiert ein ähnliches Schema, allerdings ist das Nachfolgeobjekt hier kein `Widget`, sondern ein *beliebiger* `HelpHandler` – in dieser Beispielanwendung eine Instanz der Klasse `Application`:

```
class Dialog : public Widget {
public:
    Dialog(HelpHandler* h, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();

    // Widget-Operationen, die von Dialog überschrieben werden...
    // ...
};

Dialog::Dialog (HelpHandler* h, Topic t) : Widget(0) {
    SetHandler(h, t);
}

void Dialog::HandleHelp () {
    if (HasHelp()) {
        // Bietet Hilfeinformationen zum Dialog an
    } else {
        HelpHandler::HandleHelp();
    }
}
```

Am Ende der Kette steht schließlich eine Instanz von `Application`. Da es sich bei der Anwendung nicht um ein `Widget` handelt, wird die Unterklasse `Application` direkt von `HelpHandler` abgeleitet. In dem Fall, dass ein Hilfe-Request bis zu dieser Ebene weitergeleitet wurde, kann die Anwendung entweder allgemeine

Informationen über sich selbst zur Verfügung stellen oder eine Auflistung verschiedener Hilfethemen anbieten:

```
class Application : public HelpHandler {  
public:  
    Application(Topic t) : HelpHandler(0, t) {}  
  
    virtual void HandleHelp();  
    // Anwendungsspezifische Operationen...  
};  
  
void Application::HandleHelp () {  
    // Zeigt eine Liste mit Hilfethemen an  
}
```

Der folgende Code erstellt und verbindet diese Objekte. Weil es sich in diesem Fall um einen DRUCKEN-Dialog handelt, sind den Objekten dementsprechend druckbezogene Themen zugewiesen:

```
const Topic PRINT_TOPIC = 1;  
const Topic PAPER_ORIENTATION_TOPIC = 2;  
const Topic APPLICATION_TOPIC = 3;  
  
Application* application = new Application(APPLICATION_TOPIC);  
Dialog* dialog = new Dialog(application, PRINT_TOPIC);  
Button* button = new Button(dialog, PAPER_ORIENTATION_TOPIC);
```

Der Hilfe-Request kann nun mittels `HandleHelp` bei jedem Objekt in der Kette ausgelöst werden. Um die Suche bei dem `Button`-Objekt zu starten, wird dort also lediglich `HandleHelp` aufgerufen:

```
button->HandleHelp();
```

In diesem Fall bearbeitet die Schaltfläche den Request sofort. Beachtenswert ist hierbei, dass jede beliebige `HelpHandler`-Klasse zum Nachfolgeobjekt der `Dialog`-Klasse gemacht werden könnte. Zudem ist auch eine dynamische Änderung des Nachfolgeobjekts möglich – und das heißt: Unabhängig davon, wo ein Dialog verwendet wird, wird immer der passende kontextabhängige Hilfetext dazu geliefert.

Praxisbeispiele

Einige Klassenbibliotheken nutzen das Design Pattern *Chain of Responsibility* (*Zuständigkeitskette*) zur Bearbeitung von Eingabeereignissen. Sie verwenden zwar

unterschiedliche Bezeichnungen für die Handler-Klasse, basieren aber grundsätzlich auf demselben Prinzip: Wenn der Anwender einen Mausklick ausführt oder eine Taste betätigt, wird ein Ereignis generiert und entlang der Kette weitergereicht. Bei **MacApp** [App89] und ET++ [WGM88] heißt diese Klasse **EventHandler**, in der **TCL-Bibliothek** von Symantec [Sym93b] ist sie mit **Bureaucrat** bezeichnet, und im **AppKit-Framework** von NeXT [Add94] trägt sie den Namen **Responder**.

Das **Unidraw Application Framework** für grafische Editoren [VL90] definiert **Command**-Objekte, die Requests für **Component**- und **ComponentView**-Objekte kapseln. Sie sind in dem Sinne Requests, als sie von einem **Component**- oder **ComponentView**-Objekt als Befehl zur Ausführung einer Operation interpretiert werden können. Dies entspricht dem im Abschnitt »Implementierung« beschriebenen Ansatz der »Repräsentation von Requests« als Objekte. **Component**- und **ComponentView**-Objekte können hierarchisch strukturiert werden und die Interpretation eines **Command**-Objekts an ihr Elternobjekt weiterleiten, das es möglicherweise seinerseits wiederum an sein Elternobjekt weitergibt usw., so dass eine Zuständigkeitskette entsteht.

In ET++ dient das Design Pattern *Chain of Responsibility (Zuständigkeitskette)* zur Anzeigenaktualisierung: Sobald ein grafisches Objekt seine Bildschirmschirmdarstellung aktualisieren muss, wird die Operation **InvalidateRect** aufgerufen. Weil das Objekt jedoch keine hinreichende Kenntnis von seinem jeweiligen Kontext hat, kann es die Operation nicht selbst ausführen. Beispielsweise könnte es etwa von Scroll- oder Zoom-Objekten umhüllt sein, die sein Koordinatensystem transformieren – und das hieße wiederum, dass das Objekt aktuell gescrollt oder gezoomt ist und sich gegebenenfalls zum Teil außerhalb des sichtbaren Bereichs befindet. Damit solche Eventualitäten berücksichtigt werden, leitet die Standardimplementierung von **InvalidateRect** den Request an das umgebende Behälterobjekt weiter. Das letzte Objekt in der Weiterleitungskette ist dann eine **Window**-Instanz, das mit dem Empfang des Requests sicherstellt, dass das zu invalidierende Rechteck korrekt transformiert wurde. Das **Window**-Objekt bearbeitet die Operation **InvalidateRect** in Form der Benachrichtigung der Schnittstelle des Fenstersystems sowie der Anforderung einer Aktualisierung.

Verwandte Patterns

Das Design Pattern *Chain of Responsibility (Zuständigkeitskette)* wird häufig in Kombination mit dem Design Pattern *Composite (Kompositum*, siehe [Abschnitt 4.3](#))

eingesetzt, in dem das Elternobjekt einer Komponente als dessen Nachfolgeobjekt genutzt werden kann.

5.2 Command (Befehl)

Objektbasiertes Verhaltensmuster

Zweck

Kapselung eines Requests als Objekt, um so die Parametrisierung von Clients mit verschiedenen Requests, Warteschlangen- oder Logging-Operationen sowie das Rückgängigmachen von Operationen zu ermöglichen.

Auch bekannt als

Action, Transaction

Motivation

Es kann vorkommen, dass Requests an Objekte gerichtet werden müssen, ohne dass irgendetwas über die auszuführende Operation oder den Empfänger bekannt ist. Zum Beispiel enthalten Toolkits für Benutzeroberflächen Elemente wie Schaltflächen und Menüs, die als Reaktion auf eine Benutzereingabe einen Request ausführen. Allerdings kann das Toolkit den Request nicht explizit in die Schaltfläche oder das Menü implementieren, weil nur diejenigen Anwendungen, die das Toolkit einsetzen, überhaupt wissen, was mit welchem Objekt zu tun ist. Toolkit-Designer haben keine Möglichkeit, den Empfänger des Requests oder die daraufhin auszuführenden Operationen zu ermitteln.

Das Design Pattern *Command (Befehl)* gestattet es Toolkit-Objekten, Requests an nicht spezifizierte Anwendungsobjekte zu richten, indem es den Request selbst in ein Objekt verwandelt. Dieses Objekt kann dann wie jedes andere Objekt auch gespeichert und weitergeleitet werden. Das zentrale Element dieses Patterns ist eine

abstrakte Command-Klasse, die eine Schnittstelle für die Ausführung von Operationen deklariert und in seiner einfachsten Form eine abstrakte Execute-Operation enthält. Konkrete Command-Unterklassen spezifizieren eine Empfänger/Operation-Paarung, indem sie den Empfänger als Instanzvariable speichern und zum Auslösen des Requests die Execute-Operation implementieren. Der Empfänger besitzt alle nötigen Informationen, um den Request auszuführen.

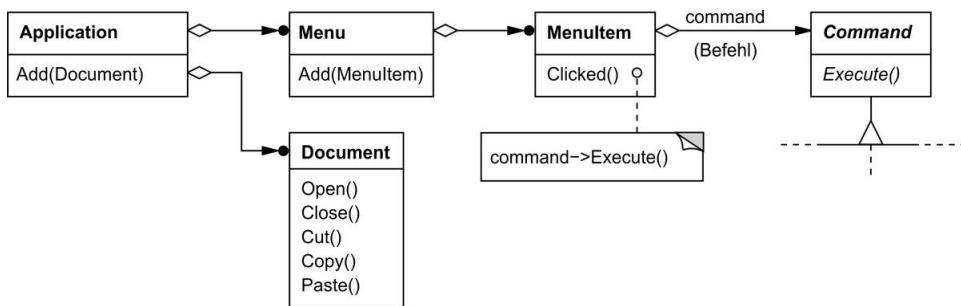


Abb. 5.6: Die Klasse *Command*

Menüs lassen sich einfach mit Command-Objekten implementieren. Dabei entspricht jeder Menüeintrag einer Instanz einer MenuItem-Klasse. Diese Menüs und ihre Menüinträge werden zusammen mit den übrigen Elementen der Benutzeroberfläche von einer Application-Klasse erzeugt, die darüber hinaus auch über die vom User geöffneten Document-Objekte Buch führt.

Die Anwendung konfiguriert jedes MenuItem-Objekt mit einer Instanz einer konkreten Command-Unterklasse. Wenn der User ein MenuItem-Objekt auswählt, ruft es die Execute-Operation seines Befehlsobjekts auf, die daraufhin ausgeführt wird. MenuItem-Objekte haben keine Kenntnis davon, welche Unterklasse von Command sie nutzen. Command-Unterklassen speichern den Empfänger des Requests und lösen eine oder mehrere Operationen bei ihm aus.

Beispielsweise unterstützt die Klasse PasteCommand das Einfügen von Text aus der Zwischenablage in ein Dokument. Der Empfänger von PasteCommand ist das Objekt Document, das während seiner Instanziierung an das Command-Objekt übergeben wurde. Die Execute-Operation ruft Paste beim empfangenden Document-Objekt auf:

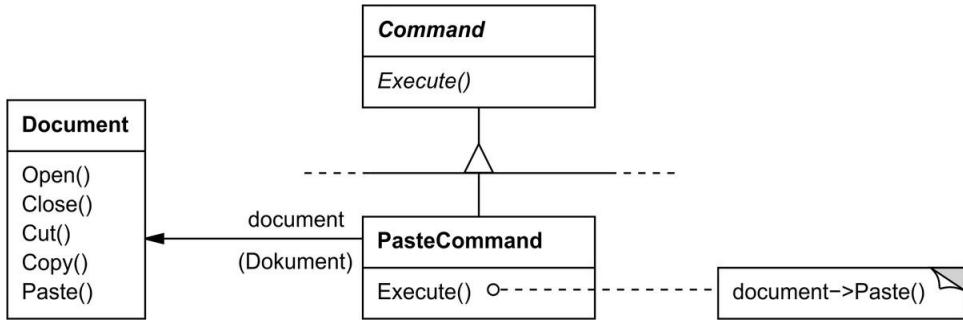


Abb. 5.7: Einfügen eines Textes aus der Zwischenablage

Bei der Execute-Operation der Klasse OpenCommand verhält sich dies anders: Sie fordert den User zur Angabe eines Dokumentnamens auf, erstellt ein entsprechendes Document-Objekt, fügt das Dokument in die empfangende Anwendung ein und öffnet es:

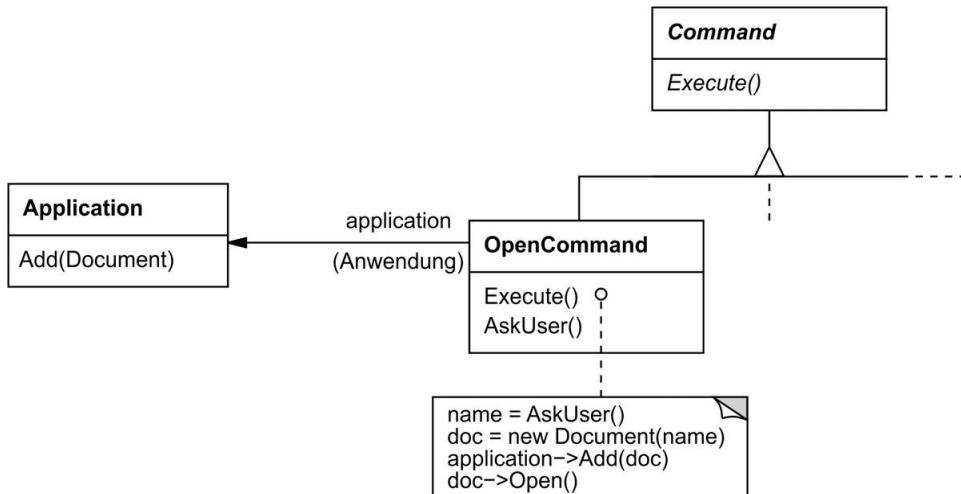


Abb. 5.8: Öffnen eines neuen Dokuments in einer Anwendung

Manchmal muss ein MenuItem-Objekt auch eine *Befehlsfolge* ausführen. Beispielsweise könnte es zum Zentrieren einer Seite aus einem CenterDocumentCommand- und einem NormalSizeCommand-Objekt konstruiert sein. Da eine solche Aneinanderreihung von Befehlen durchaus häufig vorkommt, kann eine MacroCommand-Klasse definiert werden, die es dem MenuItem-Objekt gestattet, eine unbegrenzte Zahl von Befehlen auszuführen. Die Klasse MacroCommand ist eine konkrete Command-Unterklasse, die einfach nur eine Abfolge von Befehlen ausführt. Sie selbst hat keinen expliziten Empfänger, weil die Befehle jeweils ihre eigenen Empfänger definieren.

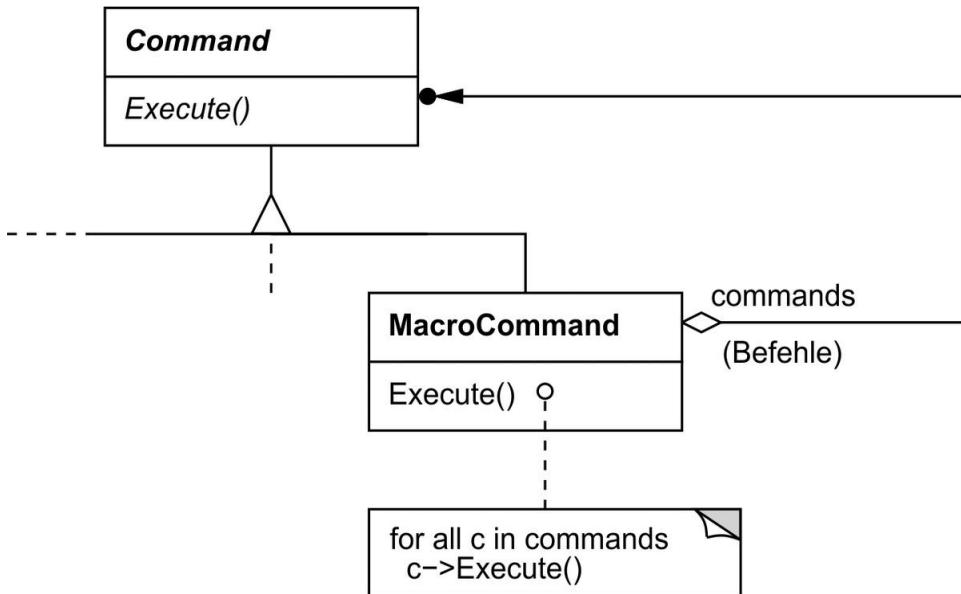


Abb. 5.9: Ausführung einer Befehlsabfolge

Jedes dieser Beispiele demonstriert, in welcher Art und Weise das Design Pattern *Command (Befehl)* die Entkopplung des Objekts, das die Operation auslöst, von dem Objekt, das letztlich in der Lage ist, die Operation tatsächlich auszuführen, bewerkstelligt. Diese Maßnahme bietet viel Flexibilität für das Design der Benutzeroberfläche. Eine Anwendung kann sowohl ein Menü als auch eine Schaltfläche als Schnittstelle für eine Funktion zur Verfügung stellen, indem diese Elemente einfach eine Instanz derselben konkreten Command-Unterklasse gemeinsam nutzen. Dadurch können Befehle dynamisch ersetzt werden – was der Implementierung von kontextsensitiven Menüs zuträglich ist. Und das Zusammenfügen einzelner Befehle zu größeren Command-Objekten gewährleistet die Unterstützung von Befehlsskripten. All dies ist möglich, weil das Objekt, das einen Request absetzt, nur wissen muss, wie er ausgelöst wird – nicht aber, wie er ausgeführt wird.

Anwendbarkeit

Das Design Pattern *Command (Befehl)* ist für folgende Zwecke geeignet:

- Wenn Objekte wie in dem `MenuItem`-Beispiel aus dem Abschnitt »Motivation« mit einer auszuführenden Operation parametrisiert werden sollen. In einer prozeduralen Sprache kann eine solche Parametrisierung über eine Callback-Funktion ausgedrückt werden, d. h. eine Funktion, die irgendwo registriert und zu einem späteren Zeitpunkt aufgerufen wird. Befehlsobjekte sind ein

objektorientierter Ersatz für Callbacks.

- Wenn Requests spezifiziert, in eine Warteschlange gestellt und zu verschiedenen Zeitpunkten ausgeführt werden sollen. Ein Command-Objekt kann eine von dem Original-Request unabhängige Lebensdauer haben. Wenn der Empfänger eines Requests in einer vom Adressbereich unabhängigen Art und Weise repräsentiert werden kann, lässt sich ein Command-Objekt für diesen Request an einen anderen Prozess übertragen und dort ausführen.
- Wenn die UNDO-Funktion, also das Rückgängigmachen von Operationen, unterstützt werden soll. Die Execute-Operation des Befehls kann den Zustand für dessen Umkehr in dem Befehlsobjekt selbst speichern. Die Command-Schnittstelle muss über eine zusätzliche Unexecute-Operation verfügen, die die Auswirkungen eines vorherigen Execute-Aufrufs rückgängig macht. Ausgeführte Befehle werden in einer **Befehlshistorie** gespeichert. Die unbegrenzte Verwendung der UNDO- und REDO-Funktionen wird durch die rückwärts und vorwärts gerichtete Traversierung der Liste und den entsprechenden Aufruf von Unexecute bzw. Execute erzielt.
- Wenn die Protokollierung vorgenommener Änderungen unterstützt werden soll, so dass diese im Fall eines Systemcrashes wiederhergestellt werden können. Durch die Erweiterung der Command-Schnittstelle um Operationen zum Laden und Speichern kann ein dauerhaftes Änderungsprotokoll angelegt werden. Die Wiederherstellung des Systems nach einem Absturz beinhaltet dann auch das erneute Laden protokollierter Befehlsobjekte von der Festplatte sowie ihre erneute Ausführung mittels der Execute-Operation.
- Wenn ein System mit komplexen Operationen strukturiert werden soll, die auf primitiven Operationen aufbauen. Derartige Strukturen finden sich insbesondere in Informationssystemen, die **Transaktionen** unterstützen. Das Design Pattern *Command (Befehl)* ermöglicht die Modellierung dieser Transaktionen. Dabei besitzen die Command-Objekte eine gemeinsame Schnittstelle, die es ihnen gestattet, alle Transaktionen auf die gleiche Weise aufzurufen. Zudem erleichtert es das Pattern auch, das System um neue Transaktionen zu ergänzen.

Struktur

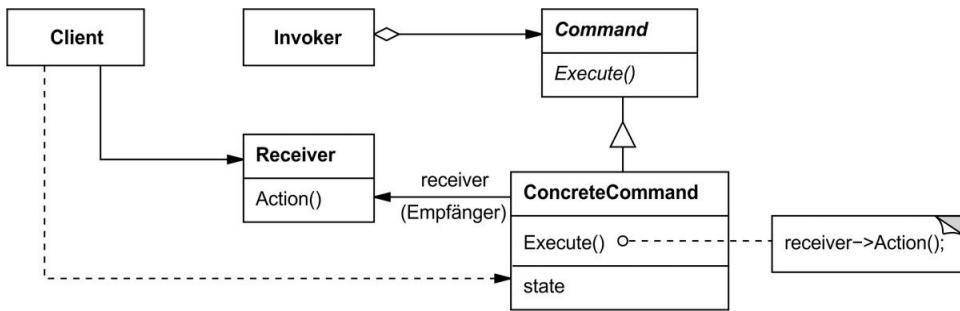


Abb. 5.10: Die Struktur des Design Patterns Command (Befehl)

Teilnehmer

- **Command**
 - Deklariert eine Schnittstelle zur Ausführung einer Operation.
- **ConcreteCommand** (PasteCommand, OpenCommand)
 - Definiert eine Bindung zwischen einem Empfängerobjekt und einer Operation.
 - Implementiert den Execute-Befehl durch den Aufruf der entsprechenden Operation/en beim Empfängerobjekt.
- **Client** (Application)
 - Erzeugt ein ConcreteCommand-Objekt und bestimmt seinen Empfänger.
- **Invoker** (MenuItem)
 - Weist das Command-Objekt an, den Request auszuführen.
- **Receiver** (Document, Application)
 - Kann die mit der Bearbeitung eines Requests verknüpften Operationen ausführen. Dabei kann jede beliebige Klasse als Receiver (dt. Empfänger) dienen.

Interaktionen

- Der Client erzeugt ein `ConcreteCommand`-Objekt und spezifiziert dessen Empfänger.
- Ein `Invoker` (dt. Aufrufer) speichert das `ConcreteCommand`-Objekt.
- Das `Invoker`-Objekt löst durch den Aufruf von `Execute` beim Befehl einen Request aus. Wenn Befehle rückgängig gemacht werden können, speichert `ConcreteCommand` vor der Ausführung des `Execute`-Befehls den aktuellen Zustand.
- Das `ConcreteCommand`-Objekt ruft zur Ausführung des Requests Operationen seines Empfängers auf.

Das in [Abbildung 5.11](#) dargestellte Interaktionsdiagramm veranschaulicht die Zusammenarbeit zwischen den Objekten und zeigt auf, wie das Design Pattern *Command (Befehl)* das `Invoker`-Objekt vom `Receiver`-Objekt (und dem von ihm auszuführenden Request) entkoppelt:

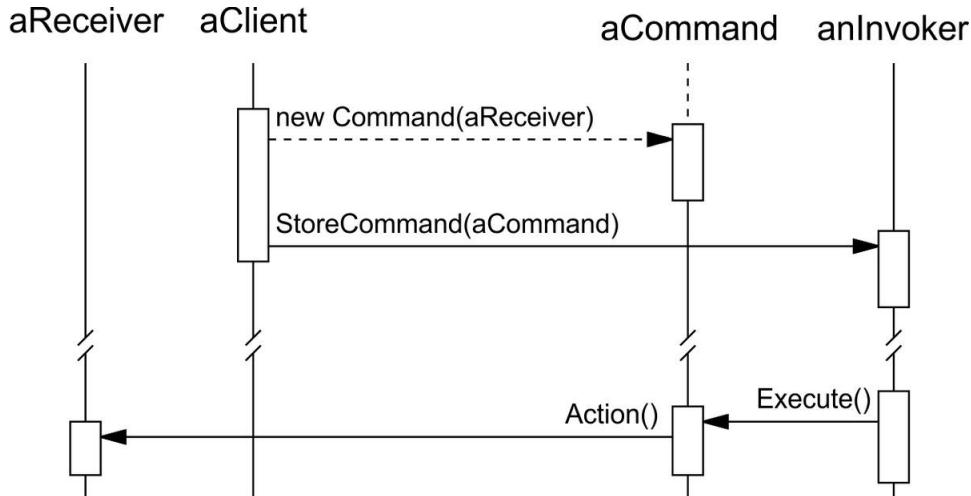


Abb. 5.11: Interaktion zwischen den Objekten

Konsequenzen

Das Design Pattern *Command (Befehl)* bewirkt Folgendes:

1. Es entkoppelt das Objekt, das die Operation auslöst, von dem Objekt, das in der Lage ist, sie auszuführen.
2. Command-Objekte sind Objekte erster Ordnung, die wie jedes andere Objekt

auch manipuliert und erweitert werden können.

3. Mehrere Befehle können zu einem einzigen Command-Objekt zusammengefügt werden, wie z. B. im Fall der zuvor beschriebenen Klasse MacroCommand. Im Allgemeinen sind zusammengesetzte Command-Objekte ein Ableger des Design Patterns *Composite (Kompositum*, siehe [Abschnitt 4.3](#)).
4. Das Hinzufügen neuer Command-Objekte ist aufgrund der Tatsache, dass eine Änderung bzw. Anpassung existierender Klassen nicht notwendig ist, problemlos und einfach möglich.

Implementierung

Bei der Implementierung des Design Patterns *Command (Befehl)* sollten folgende Aspekte Berücksichtigung finden:

1. *Wie intelligent soll ein Command-Objekt sein?* Ein Command-Objekt verfügt über eine beachtliche Bandbreite an Möglichkeiten: Im einen Extrem definiert es schlicht eine Bindung zwischen einem Empfänger und den Operationen, die den Request ausführen. Im anderen Extrem implementiert es ausnahmslos alles selbst, ohne irgendetwas an den Empfänger zu delegieren. Letzteres ist dann hilfreich, wenn Befehle definiert werden sollen, die nicht von den existierenden Klassen abhängig sein sollen, wenn kein geeigneter Empfänger vorhanden ist oder wenn der Empfänger dem Command-Objekt implizit bekannt ist. Ein Command-Objekt, das ein weiteres Anwendungsfenster erzeugt, kann möglicherweise ebenso gut ein beliebiges anderes Objekt erstellen. Und irgendwo zwischen diesen beiden Extremen gibt es auch Command-Objekte, die in der Lage sind, ihre Empfänger dynamisch zu finden.
2. *Unterstützung der Funktionen UNDO und REDO.* Sofern Command-Objekte eine Möglichkeit zur Umkehr ihrer Ausführung anbieten (d. h., sie stellen eine Unexecute- oder Undo-Operation zur Verfügung), können sie die Funktionen UNDO und REDO unterstützen. Dazu muss die ConcreteCommand-Klasse gegebenenfalls zusätzliche Daten zum jeweils aktuellen Zustand speichern, was Folgendes beinhalten kann:
 - das Receiver-Objekt, das die auf den Request anzuwendenden Operationen effektiv ausführt,
 - die Argumente für die vom Empfänger ausgeführten Operationen sowie

- sämtliche Originalwerte im Receiver-Objekt, die sich infolge der Request-Bearbeitung ändern können. Das Receiver-Objekt muss Operationen zur Verfügung stellen, die es dem Command-Objekt ermöglichen, den Empfänger in seinen vorherigen Zustand zurückzuversetzen.

Um lediglich die Rücknahme des jeweils letzten durchgeföhrten Arbeitsschrittes zu ermöglichen, braucht die Anwendung nur den zuletzt ausgeführten Befehl zu speichern. Sollen dagegen mehrere UNDO- und REDO-Schritte möglich sein, muss die Anwendung eine Historie der ausgeführten Befehle speichern, wobei die Länge der Liste zugleich auch die maximale Zahl der möglichen UNDO-/REDO-Schritte bestimmt. Diese Befehlshistorie bildet die ausgeführte Befehlsabfolge in ihrer Gesamtheit ab. Die rückwärtige Traversierung der Liste sowie die umgekehrte Ausführung der Befehle hebt deren vorherige Wirkung wieder auf (UNDO) – bei einer Vorwärtstraversierung werden die Befehle erneut ausgeführt (REDO).

Unter Umständen muss ein rückgängig zu machender Befehl vor dem Einfügen in die Befehlshistorie zunächst kopiert werden – weil das Command-Objekt, das den ursprünglichen Request (z. B. von einem MenuItem-Objekt) ausgeführt hat, anschließend noch weitere Requests bearbeiten wird. In diesem Fall ist das Kopieren des Befehls zwingend erforderlich, damit die diversen Aufrufe des betreffenden Command-Objekts unterschieden werden können, wenn sich dessen Zustand im Rahmen der Aufrufe ändern kann.

Zum Beispiel muss ein DeleteCommand-Objekt, das ausgewählte Objekte löscht, bei jeder Ausführung verschiedene Objektsätze speichern. Aus diesem Grund muss es nach jeder Ausführung kopiert und die Kopie dann in die Befehlshistorie eingefügt werden. Sollte sich der Zustand des Command-Objekts aufgrund seiner Ausführung allerdings zu keiner Zeit ändern, ist es auch nicht nötig, es zu kopieren – dann genügt die Hinterlegung einer passenden Objektreferenz in der Befehlshistorie. Command-Objekte, die vor dem Einfügen in die Befehlshistorie kopiert werden müssen, dienen als Prototypen (siehe Design Pattern *Prototype (Prototyp)*, [Abschnitt 3.4](#)).

3. *Vermeidung von Fehlerhäufungen im UNDO-Prozess.* Hysteresen können sich hinsichtlich der Sicherstellung eines zuverlässigen, semantikgetreuen UNDO-/REDO-Mechanismus als problematisch erweisen: Während der Ausführung, Rückgängigmachung und erneuten Ausführung von Befehlen können sich Fehler anhäufen, die dazu führen, dass sich der Zustand einer Anwendung schließlich von den ursprünglichen Werten entfernt. Deshalb kann es notwendig

sein, zusätzliche Informationen in dem Befehl zu speichern, damit die Wiederherstellung der Objekte in ihrem Originalzustand gewährleistet ist. Um sicherzustellen, dass der Befehl auf diese Daten zugreifen kann, ohne dass die Interna auch anderen Objekten zugänglich werden, kann das Design Pattern *Memento* (*Memento*, siehe [Abschnitt 5.6](#)) angewendet werden.

4. *Verwendung von C++-Templates*. Für Command-Objekte, die weder rückgängig zu machen sind, noch Argumente benötigen, können C++-Templates eingesetzt werden, wodurch sich die Erzeugung einer Command-Unterklasse für jede Art von Operation und Empfänger erübrigt. Wie das funktioniert, wird im Abschnitt »Beispielcode« demonstriert.

Beispielcode

Das nachfolgende C++-Beispiel skizziert die Implementierung der im Abschnitt »Motivation« beschriebenen Command-Klassen. Neben den Klassen OpenCommand, PasteCommand und MacroCommand, die im weiteren Verlauf noch folgen werden, wird hier zunächst die abstrakte Command-Klasse definiert:

```
class Command {  
public:  
    virtual ~Command();  
  
    virtual void Execute() = 0;  
  
protected:  
    Command();  
};
```

Die Klasse OpenCommand öffnet ein vom User spezifiziertes Dokument. Dem Konstruktor dieser Klasse muss ein Application-Objekt übergeben werden. Der Name des zu öffnenden Dokuments wird mithilfe der Implementierungsroutine AskUser vom User abfragt:

```
class OpenCommand : public Command {  
public:  
    OpenCommand(Application*);  
  
    virtual void Execute();  
  
protected:  
    virtual const char* AskUser();  
  
private:
```

```

        Application* _application;
        char* _response;
    };

OpenCommand::OpenCommand (Application* a) {
    _application = a;
}

void OpenCommand::Execute () {
    const char* name = AskUser();

    if (name != 0) {
        Document* document = new Document(name);
        _application->Add(document);
        document->Open();
    }
}

```

Einem PasteCommand-Objekt muss ein Document-Objekt als Empfänger zugewiesen werden, der dann wiederum dem Konstruktor von PasteCommand als Parameter übergeben wird:

```

class PasteCommand : public Command {
public:
    PasteCommand(Document *);

    virtual void Execute();

private:
    Document* _document;
};

PasteCommand::PasteCommand (Document* doc) {
    _document = doc;
}

void PasteCommand::Execute () {
    _document->Paste();
}

```

Einfache Befehle, die nicht rückgängig zu machen sind und keine Argumente benötigen, können zur Parametrisierung des Empfängers des Command-Objekts eine template-Klasse verwenden. Für solche Befehle wird eine template-Unterklasse namens SimpleCommand definiert. Anschließend wird SimpleCommand mit dem Empfängertyp parametrisiert und verwaltet die Bindung zwischen einem Receiver-Objekt und einer Operation, die als Zeiger auf eine Memberfunktion gespeichert wird:

```
template <class Receiver>
```

```

class SimpleCommand : public Command {
public:
    typedef void (Receiver::* Action)();

    SimpleCommand(Receiver* r, Action a) :
        _receiver(r), _action(a) { }

    virtual void Execute();

private:
    Action _action;
    Receiver* _receiver;
};

```

Der Konstruktor speichert den Empfänger und die Operation in den zugehörigen Instanzvariablen. Execute wendet lediglich die Operation auf den Empfänger an:

```

template <class Receiver>
void SimpleCommand<Receiver>::Execute () {
    (_receiver->*_action)();
}

```

Zur Erzeugung eines Befehls, der die Action-Operation einer Instanz der Klasse MyClass aufruft, schreibt ein Client einfach Folgendes:

```

MyClass* receiver = new MyClass;
// ...
Command* aCommand =
    new SimpleCommand<MyClass>(receiver, &MyClass::Action);
// ...
aCommand->Execute();

```

Hierbei ist allerdings zu beachten, dass diese Lösung nur für einfache Befehle geeignet ist. Komplexere Befehle, die nicht nur ihre Empfänger, sondern darüber hinaus auch über Argumente und/oder den Zustand für die Nutzung der UNDO-Funktion Buch führen, benötigen dagegen eine Command-Unterklasse.

Die Klasse MacroCommand verwaltet eine Abfolge von Unterbefehlen und stellt zudem Operationen für deren Ergänzung und Entfernung zur Verfügung. Ein expliziter Empfänger ist nicht erforderlich, weil die Unterbefehle bereits ihre jeweiligen Empfänger definieren:

```

class MacroCommand : public Command {
public:
    MacroCommand();
    virtual ~MacroCommand();

    virtual void Add(Command* );

```

```

    virtual void Remove(Command* );
    virtual void Execute();

private:
    List<Command*>* _cmds;
};

```

Das Schlüsselement der Klasse `MacroCommand` ist ihre `Execute`-Memberfunktion. Sie traversiert alle Unterbefehle und führt jeweils die `Execute`-Operation aus:

```

void MacroCommand::Execute () {
    ListIterator<Command*> i(_cmds);

    for (i.First(); !i.IsDone(); i.Next()) {
        Command* c = i.CurrentItem();
        c->Execute();
    }
}

```

Wichtig ist hierbei, dass die Unterklassen in dem Fall, wenn die Klasse `MacroCommand` eine `Unexecute`-Operation implementiert, anders als bei der `Execute`-Implementierung in *umgekehrter* Reihenfolge rückgängig gemacht werden müssen.

Außerdem muss die Klasse `MacroCommand` auch Operationen zur Verwaltung ihrer Unterbefehle bereitstellen und ist ebenso für das Löschen ihrer Unterbefehle zuständig:

```

void MacroCommand::Add (Command* c) {
    _cmds->Append(c);
}

void MacroCommand::Remove (Command* c) {
    _cmds->Remove(c);
}

```

Praxisbeispiele

Das möglicherweise erste Beispiel für das Design Pattern *Command (Befehl)* überhaupt findet sich in einer Abhandlung von **Lieberman** [Lie85]. Und das Macintosh Application Framework **MacApp** [App89] machte das Konzept von Befehlen für die Implementierung von rückgängig zu machenden Operationen breiteren Kreisen zugänglich. Auch **ET++** [WGM88], **InterViews** [LCI+92] sowie **Unidraw** [VL90] definieren Klassen, die dem Design Pattern *Command (Befehl)* folgen. InterViews definiert eine abstrakte Action-Klasse, die Befehlsfunktionalität

bereitstellt. Darüber hinaus wird hier ein mit der Action-Methode parametrisiertes ActionCallback-Template definiert, das für die automatische Instanzierung der Command-Unterklassen sorgt.

Die **THINK-Klassenbibliothek** [Sym93b] verwendet ebenfalls Befehle, um das Rückgängigmachen von Operationen zu ermöglichen. Diese Befehle werden in THINK als »Tasks« (»Aufgaben«) bezeichnet. Task-Objekte werden zur Bearbeitung entlang einer Zuständigkeitskette (*Chain of Responsibility* (*Zuständigkeitskette*, siehe [Abschnitt 5.1](#))) weitergereicht.

Die in Unidraw verwendeten Befehlsobjekte sind insofern einzigartig, als dass sie sich wie Meldungen verhalten können. Ein Unidraw-Befehl kann an ein anderes Objekt geschickt und von diesem interpretiert werden, wobei das Resultat der Interpretation in Abhängigkeit von dem Empfängerobjekt variiert. Darüber hinaus kann der Empfänger die Interpretation seinerseits ebenfalls an ein anderes Objekt delegieren – in einer umfangreicher Struktur wie einer Zuständigkeitskette normalerweise an sein Elternobjekt. Der Empfänger eines Unidraw-Befehls wird somit berechnet und nicht gespeichert. Der Interpretationsmechanismus von Unidraw ist von den Laufzeittypinformationen abhängig.

Coplien beschreibt die Implementierung von sogenannten **Functors** in C++ [Cop92] – also von Objekten, die Funktionen darstellen. Er führt in seinen Beispielen an, dass sich mithilfe dieser Objekte durch die Überladung des Operators für den Funktionsaufruf (operator()) eine gewisse Transparenz erzielen lässt. Das Design Pattern *Command (Befehl)* geht hier anders vor: Es konzentriert sich auf die *Bindung* zwischen einem Empfänger und einer Funktion (sprich Operation) und nicht nur auf die Verwaltung einer Funktion.

Verwandte Patterns

Das Design Pattern *Composite (Kompositum*, siehe [Abschnitt 4.3](#)) eignet sich zur Implementierung von MacroCommand-Objekten.

Das Design Pattern *Memento (Memento*, siehe [Abschnitt 5.6](#)) kann den Zustand erfassen, den das Command-Objekt braucht, um seine Wirkung rückgängig zu machen.

Ein Command-Objekt, das kopiert werden muss, bevor es in die Befehlshistorie eingefügt werden kann, dient als Prototyp (siehe Design Pattern *Prototype (Prototyp)*, siehe [Abschnitt 3.4](#))).

5.3 Interpreter (Interpreter)

Klassenbasierter Verhaltensmuster

Zweck

Definition einer Repräsentation der Grammatik einer gegebenen Sprache sowie Bereitstellung eines Interpreters, der diese Grammatik nutzt, um in der betreffenden Sprache verfasste Sätze zu interpretieren.

Motivation

Um einer bestimmten, immer wiederkehrenden Art von Problem- bzw. Aufgabenstellung zu begegnen, kann es hilfreich sein, sie in Form von »Sätzen« in einer einfachen Sprache auszudrücken und diese Sätze mithilfe eines passend dazu erzeugten Interpreters auszuwerten bzw. zu interpretieren.

Ein Beispiel für ein solches häufig auftretendes Problem ist die Suche nach Strings (Zeichenfolgen), die einem bestimmten Muster entsprechen. Dabei stellen reguläre Ausdrücke eine Standardsprache für die Spezifikation von Stringmustern dar: Statt auf eigens zum Abgleichen jedes einzelnen Stringmusters entwickelte Algorithmen zurückgreifen zu müssen, können die Suchalgorithmen ebenso gut auch einen regulären Ausdruck interpretieren, der die abzugleichenden Strings spezifiziert.

Das Design Pattern *Interpreter (Interpreter)* beschreibt neben der Definition einer Grammatik für einfache Sprachen auch, wie sich Sätze in der jeweiligen Sprache darstellen und anschließend interpretieren lassen. Zur Veranschaulichung dieses Konzepts wird im folgenden Beispiel eine Grammatik für reguläre Ausdrücke definiert, ein bestimmter regulärer Ausdruck dargestellt und dieser schließlich interpretiert.

Die regulären Ausdrücke werden in diesem Fall durch folgende Grammatik definiert:

```
expression ::= literal | alternation | sequence | repetition | '(' expression ')'
alternation ::= expression '|' expression
```

```

sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

```

expression repräsentiert hier das Startsymbol und literal das Terminalsymbol (abschließendes Symbol), die einfache Wörter definieren.

Wie in [Abbildung 5.12](#) zu sehen, wird im Design Pattern *Interpreter* (*Interpreter*) jede einzelne Grammatikregel durch eine eigene Klasse repräsentiert. Bei den Symbolen, die sich auf der rechten Seite der Regel befinden, handelt es sich um Instanzvariablen dieser Klassen. Die übergeordnete Grammatik wird durch fünf Klassen repräsentiert: eine abstrakte Klasse `RegularExpression` und deren vier Unterklassen `LiteralExpression`, `AlternationExpression`, `SequenceExpression` sowie `RepetitionExpression`, wobei die drei letztgenannten Klassen Variablen definieren, die ihrerseits Unterausdrücke enthalten:

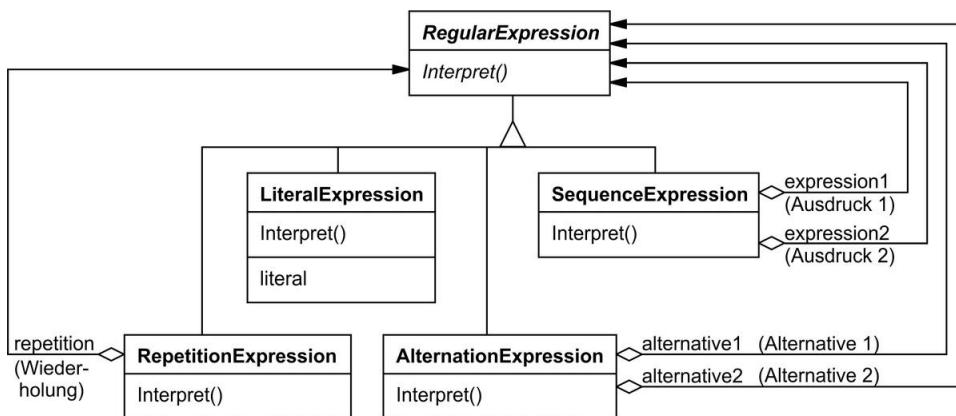


Abb. 5.12: Aufbau einer Grammatik

Jeder im Rahmen dieser Grammatik definierte reguläre Ausdruck wird durch einen abstrakten Syntaxbaum dargestellt, der aus Instanzen dieser Klassen besteht. So findet sich der reguläre Ausdruck

```
raining & (dogs | cats) *
```

beispielsweise in diesem abstrakten Syntaxbaum wieder:

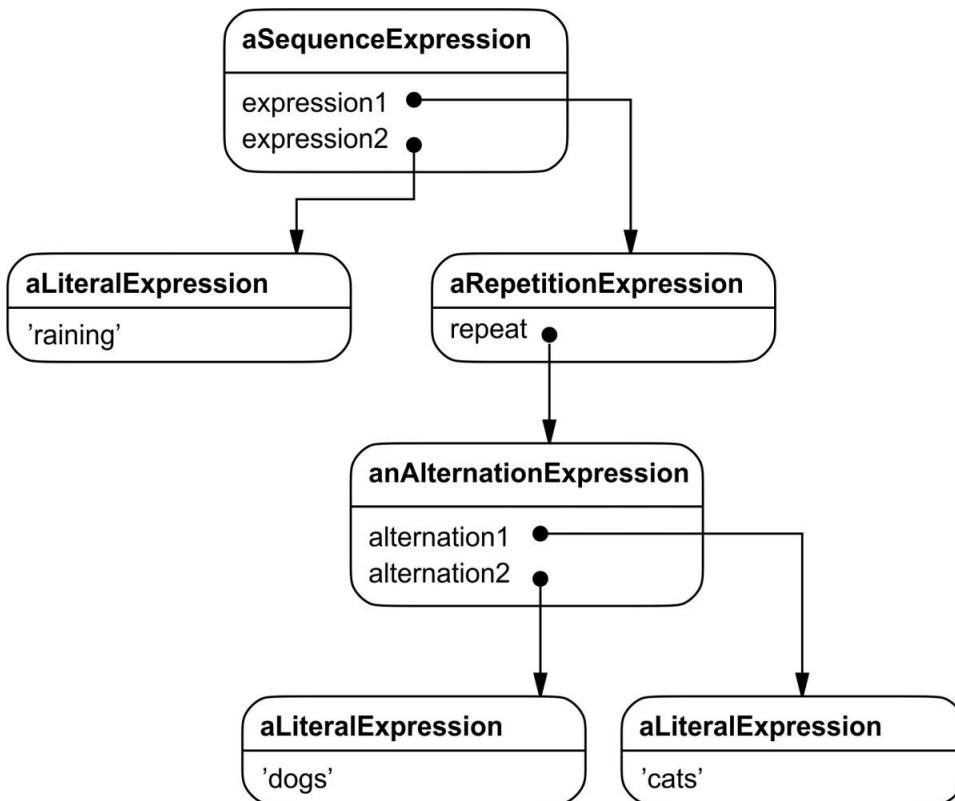


Abb. 5.13: Abstrakter Syntaxbaum zur Darstellung eines regulären Ausdrucks

Zur Erzeugung eines Interpreters für diese regulären Ausdrücke kann in jeder Unterklasse von `RegularExpression` die `Interpret`-Operation definiert werden. Sie nimmt den Kontext, in dem der Ausdruck zu interpretieren ist, als Argument entgegen. Der Kontext enthält den Eingabestring sowie Informationen über den bislang erzielten Fortschritt des Abgleichs. Die Operation `Interpret` wird derart in jede `RegularExpression`-Unterklasse implementiert, dass sie den nächsten Teil des Eingabestrings unter Einbeziehung des aktuellen Kontexts abgleicht und beispielsweise folgende Überprüfungen vornimmt:

- Die Unterklasse `LiteralExpression` prüft, ob die Eingabe mit dem von ihr definierten Literal übereinstimmt.
- Die Unterklasse `AlternationExpression` prüft, ob die Eingabe einer der von ihr angebotenen Alternativen entspricht.
- Die Unterklasse `RepetitionExpression` prüft, ob die Eingabe mehrere Vorkommen des von ihr zu wiederholenden Ausdrucks enthält.

Und so weiter.

Anwendbarkeit

Das Design Pattern *Interpreter* (*Interpreter*) ist immer dann geeignet, wenn eine Sprache interpretiert werden muss und sich die darin verwendeten Ausdrücke in Form abstrakter Syntaxbäume abbilden lassen. Die besten Ergebnisse werden dabei erzielt, wenn

- die Grammatik einfach gehalten ist. Klassenhierarchien für komplexe Grammatiken sind schlicht zu umfangreich und nicht mehr zu handhaben. In diesen Fällen sind Tools wie Parser-Generatoren die bessere Alternative, weil sie in der Lage sind, Ausdrücke auch ohne abstrakte Syntaxbäume zu interpretieren, wodurch nicht nur Speicherkapazität, sondern häufig auch Zeit eingespart werden kann.
- der Faktor Effizienz keine vorrangige Rolle spielt. Besonders leistungsstarke Interpreter werden normalerweise *nicht* durch die unmittelbare Interpretation von Parse-Bäumen realisiert, sondern vielmehr durch die vorausgehende Transformation der Bäume in eine andere Form. Reguläre Ausdrücke werden zum Beispiel oftmals in endliche Automaten (auch Zustandsmaschinen genannt) transformiert. Aber selbst dann kann der *Übersetzer* immer noch mithilfe des Design Patterns *Interpreter* (*Interpreter*) implementiert werden, so dass es nach wie vor anwendbar ist.

Struktur

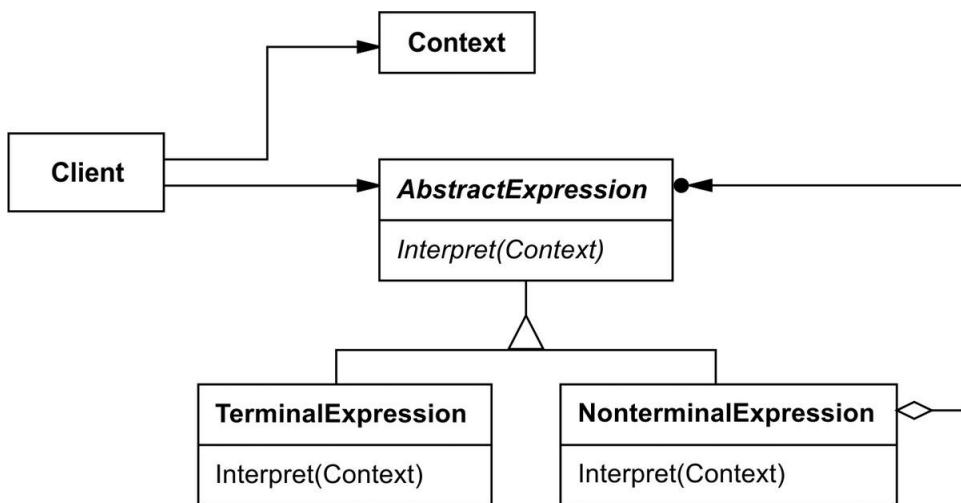


Abb. 5.14: Die Struktur des Design Patterns *Interpreter* (*Interpreter*)

Teilnehmer

- **AbstractExpression** (RegularExpression)
 - Deklariert eine abstrakte Interpret-Operation, die alle Knoten in dem abstrakten Syntaxbaum gemeinsam haben.
- **TerminalExpression** (LiteralExpression)
 - Implementiert eine mit den Terminalsymbolen in der Grammatik verknüpfte Interpret-Operation.
 - Jedes Terminalsymbol in einem Satz erfordert eine eigene Instanz.
- **NonterminalExpression** (AlternationExpression, RepetitionExpression, SequenceExpressions)
 - Für jede $R ::= R_1 R_2 \dots R_n$ -Regel in der Grammatik ist je eine solche Klasse erforderlich.
 - Enthält Instanzvariablen des Typs AbstractExpression für jedes einzelne Symbol von R_1 bis R_n .
 - Implementiert eine Interpret-Operation für Nichtterminalsymbole in der Grammatik. Sie ruft sich normalerweise selbst rekursiv bei den Variablen auf, die R_1 bis R_n repräsentieren.
- **Context**
 - Enthält die für den Interpreter allgemeingültigen Informationen.
- **Client**
 - Erzeugt (oder erhält) einen abstrakten Syntaxbaum, der einen bestimmten Satz in der von der Grammatik definierten Sprache repräsentiert. Er setzt sich aus Instanzen der Klassen NonterminalExpression und TerminalExpression zusammen.
 - Ruft die Interpret-Operation auf.

Interaktionen

- Der Client erzeugt (oder erhält) den Satz in Form eines abstrakten Syntaxbaums, der aus NonterminalExpression- und TerminalExpression-Instanzen besteht. Anschließend initialisiert der Client den Kontext und ruft die Operation Interpret auf.
- Jeder NonterminalExpression-Knoten definiert die Interpret-Operation jeweils im Sinne einer Interpret-Operation für jeden einzelnen Unterausdruck. Die Interpret-Operation jeder TerminalExpression-Klasse definiert den Basisfall in der Rekursion.
- Die Interpret-Operationen jedes Knotens nutzen den Kontext, um auf den Zustand des Interpreters zuzugreifen und ihn zu speichern.

Konsequenzen

Das Design Pattern *Interpreter* (*Interpreter*) begünstigt bzw. bewirkt Folgendes:

1. *Einfache Änderung und Erweiterung der Grammatik.* Da dieses Pattern zur Abbildung der Grammatikregeln auf Klassen zurückgreift, kann zum Ändern und Erweitern der Grammatik das Prinzip der Vererbung angewendet werden. Bereits vorhandene Ausdrücke sind inkrementell modifizierbar, und neue Ausdrücke können als Variationen alter Ausdrücke definiert werden.
2. *Einfache Implementierung der Grammatik.* Klassen, die Knoten im abstrakten Syntaxbaum definieren, verfügen über ähnliche Implementierungen. Solche Klassen sind einfach zu erstellen und können mithilfe eines Compilers oder Parser-Generators häufig sogar automatisch generiert werden.
3. *Schwierige Verwaltung komplexer Grammatiken.* Das Design Pattern *Interpreter* (*Interpreter*) definiert mindestens eine Klasse für jede in der Grammatik vorhandene Regel. (Für Grammatikregeln, die gemäß der Backus-Normalform (BNF) definiert sind, sind ggf. mehrere Klassen erforderlich.) Dementsprechend können Grammatiken, in denen viele Regeln zur Anwendung kommen, nur schwer zu verwalten sein. Hier kann es unter Umständen hilfreich sein, weitere geeignete Design Patterns hinzuzuziehen (siehe Abschnitt »Implementierung«), für sehr komplexe Grammatiken sind speziellere Techniken wie Parser- oder Compiler-Generatoren jedoch besser geeignet.
4. *Ergänzung neuer Interpretationsarten für Ausdrücke.* Das Design Pattern *Interpreter* (*Interpreter*) erleichtert die Anwendung neuer

Auswertungsmethoden für die Ausdrücke. So kann beispielsweise die Definition einer neuen Operation in den Expression-Klassen die Quelltextformatierung oder auch eine Typprüfung unterstützen. Für den Fall, dass mehrere neue Varianten der Ausdrucksinterpretation genutzt werden sollen, empfiehlt sich die Verwendung des Design Patterns *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)), um Änderungen an den Grammatikklassen selbst zu vermeiden.

Implementierung

Die Implementierung des Patterns *Interpreter* (*Interpreter*) weist viele Gemeinsamkeiten mit der Implementierung des Design Pattern *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) auf. Darüber hinaus sind allerdings auch einige spezifische Aspekte zu beachten:

1. *Erstellung des abstrakten Syntaxbaums.* Die Art und Weise, wie ein abstrakter Syntaxbaum zu erzeugen ist, wird im Design Pattern *Interpreter* (*Interpreter*) nicht näher spezifiziert, d. h., es gibt keine Vorgaben hinsichtlich des Parsings. Der abstrakte Syntaxbaum kann mithilfe eines tabellenbasierten Parsers, eines manuell erstellten (und üblicherweise rekursiv absteigenden) Parsers oder auch direkt vom Client erzeugt werden.
2. *Definition der Interpret-Operation.* Die Interpret-Operation muss nicht zwingend in den Expression-Klassen definiert werden. Wenn generell ein neuer Interpreter erzeugt wird, empfiehlt sich die Anwendung des Design Patterns *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)), damit die Interpret-Operation in einem separaten *Visitor*-Objekt untergebracht werden kann. Eine Grammatik für eine Programmiersprache wird beispielsweise viele Operationen für den abstrakten Syntaxbaum besitzen, etwa zur Typprüfung, zur Optimierung, zur Codegenerierung usw. Hier bietet sich der Einsatz von *visitor*-Objekten an, damit diese Operationen nicht in jeder einzelnen Grammatikklassie definiert werden müssen.
3. *Gemeinsame Nutzung von Terminalsymbolen mit dem Design Pattern Flyweight* (*Fliegengewicht*, siehe [Abschnitt 4.6](#)). Grammatiken, deren Sätze zahlreiche Terminalsymbole enthalten, können bereits von der gemeinsamen Nutzung einer einzigen Kopie des betreffenden Symbols profitieren. Ein gutes Beispiel hierfür sind Grammatiken für Computerprogramme: Jede Programmvariable wird an vielen verschiedenen Stellen innerhalb des Codes verwendet werden. So könnte das im Abschnitt »Motivation« erwähnte Terminalsymbol `dog` (das

in der Klasse `LiteralExpression` modelliert wird) durchaus mehrfach in einem Satz verwendet werden.

Terminalknoten speichern im Allgemeinen keine Informationen über ihre Position im abstrakten Syntaxbaum. Stattdessen übergeben ihnen die Elternknoten jeglichen Kontext, den sie im Laufe der Interpretation benötigen. Hier wird also zwischen dem gemeinsamen genutzten (*intrinsischen*) und dem übergebenen (*extrinsischen*) Zustand unterschieden – und damit ist das Design Pattern *Flyweight (Fliegengewicht*, siehe [Abschnitt 4.6](#)) anwendbar.

Beispielsweise erhält jede Instanz der Klasse `LiteralExpression` einen Kontext für `dog`, der den bis zu diesem Zeitpunkt bereits abgeglichenen Teilstring enthält. Und jede dieser `LiteralExpression`-Instanzen geht dann wiederum genauso mit ihrer zugehörigen `Interpret`-Operation vor: Sie prüft, ob der nächste Teil der Eingabe den String `dog` enthält – egal, an welcher Stelle des Baums sich die Instanz gerade befindet.

Beispielcode

Im Folgenden werden zwei Beispiele für die Anwendung des Design Patterns *Interpreter (Interpreter)* aufgezeigt. Als Erstes wird Smalltalk-Code erstellt, der überprüft, ob eine Zeichenfolge einem regulären Ausdruck entspricht. Bei dem zweiten Codebeispiel handelt es sich um ein C++-Programm zur Auswertung von booleschen Ausdrücken.

Der Code zum Abgleich von regulären Ausdrücken prüft, ob ein String in der von dem regulären Ausdruck definierten Sprache vorliegt. Der reguläre Ausdruck ist durch folgende Grammatik definiert:

```
expression ::= literal | alternation | sequence | repetition | '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&'amp; expression
repetition ::= expression 'repeat'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

Hierbei handelt es sich um eine leicht modifizierte Version des Beispiels aus dem Abschnitt »Motivation«: Weil das Symbol `*` in Smalltalk nicht als Postfix-Operation verwendet werden kann, wurde die konkrete Syntax der regulären Ausdrücke ein wenig verändert und `*` durch `repeat` ersetzt. So passt zum Beispiel der reguläre Ausdruck

```
(('dog' | 'cat') repeat & 'weather')
```

zum Eingabestring dog dog cat weather.

Zur Implementierung dieser Abgleichsroutine werden die fünf [hier](#) beschriebenen Expression-Klassen definiert:

Die Kindobjekte der Klasse SequenceExpression sind in den Instanzvariablen expression1 und expression2 im abstrakten Syntaxbaum untergebracht, während sich die Alternativausdrücke der Klasse AlternationExpression in den Instanzvariablen alternative1 und alternative2 wiederfinden. Der von der Klasse RepetitionExpression zu wiederholende Ausdruck ist in deren Instanzvariable repetition gespeichert. Und in der Instanzvariablen components der Klasse LiteralExpression ist eine (mutmaßlich aus Zeichen bestehende) Objektliste hinterlegt. Diese Klassen repräsentieren zusammengenommen den Literalstring, der mit dem Eingabestring übereinstimmen muss.

Die Operation `match`: implementiert einen Interpreter für den regulären Ausdruck und ist in jeder Klasse des abstrakten Syntaxbaums definiert. Sie nimmt `inputState` als ein Argument entgegen, das den aktuellen Zustand des Abgleichsprozesses nach dem partiellen Einlesen des Eingabetrings repräsentiert.

Dieser aktuelle Zustand wird durch einen Satz von Eingabestreams charakterisiert, der die Gesamtheit der bislang von dem regulären Ausdruck ggf. akzeptierten Eingaben repräsentiert. (Das entspricht in etwa der Aufzeichnung aller Zustände, die ein entsprechender endlicher Automat im Rahmen der Erkennung des Eingabestreams bis zu diesem Zeitpunkt angenommen hätte.)

Für die `repeat`-Operation ist der aktuelle Zustand von besonderer Bedeutung. Würde der reguläre Ausdruck beispielsweise

```
'a' repeat
```

lauten, dann könnte der Interpreter Strings wie a, aa, aaa etc. als Übereinstimmungen erkennen. Der Ausdruck

```
'a' repeat & 'bc'
```

würde Übereinstimmungen mit den Strings abc, aabc, aaabc etc. feststellen. Wenn es sich jedoch um einen regulären Ausdruck wie

```
'a' repeat & 'abc'
```

handeln würde, dann würde der Abgleich der Eingabe aabc mit dem Unterausdruck 'a' repeat zwei Eingabestreams ergeben, wobei im einen Fall eine Übereinstimmung mit genau einem Zeichen (a) und im anderen Fall mit genau zwei Zeichen der Eingabe (aa) gefunden würde – aber nur der Stream, der das eine Zeichen (a) erkannt hat, würde zusätzlich auch das verbleibende abc als Übereinstimmung akzeptieren.

Ein genauerer Blick auf die Definition von `match`: in den einzelnen für die regulären Ausdrücke maßgeblichen Klassen zeigt: Die Definition von `SequenceExpression` stimmt in der Reihenfolge mit jedem ihrer Unterausdrücke überein. Normalerweise werden die Eingabestreams dann aus dem `inputState` dieser Klasse entfernt:

```
match: inputState
  ^ expression2 match: (expression1 match: inputState).
```

Eine `AlternationExpression` gibt einen Zustand zurück, der die Zustände der vorliegenden Alternativen zusammenfasst. Die Definition von `match`: für die Klasse `AlternationExpression` lautet:

```
match: inputState
  | finalState |
  finalState := alternative1 match: inputState.
  finalState addAll: (alternative2 match: inputState).
  ^ finalState
```

Die `match`:-Operation einer Klasse `RepetitionExpression` versucht so viele übereinstimmende Zustände wie möglich zu finden:

```
match: inputState
  | aState finalState |
  aState := inputState.
  finalState := inputState copy.
  [aState isEmpty]
    whileFalse:
      [aState := repetition match: aState.
       finalState addAll: aState].
  ^ finalState
```

Ihr Ausgabezustand umfasst in aller Regel mehr Zustände als ihr Eingabezustand, weil eine `RepetitionExpression` im Eingabezustand mit ein, zwei oder mehreren Vorkommen des zu wiederholenden Ausdrucks `repetition` übereinstimmen kann. All diese Möglichkeiten werden in den Ausgabezuständen abgebildet und ermöglichen damit den nachfolgenden Elementen des regulären Ausdrucks die Auswahl des korrekten Zustands.

Die Definition der Operation `match`: für die Klasse `LiteralExpression` versucht schließlich, ihre Komponenten mit jedem infrage kommenden Eingabestream abzulegen und bewahrt nur diejenigen Streams, die eine Übereinstimmung aufweisen:

```
match: inputState
  | finalState tStream |
  finalState := Set new.
  inputState
    do:
      [:stream | tStream := stream copy.
       (tStream nextAvailable:
        components size
       ) = components
        ifTrue: [finalState add: tStream]
      ].
  ^ finalState
```

Die `nextAvailable`-Meldung ist die einzige `match`:-Operation, die im Eingabestream weiter vorrückt. Beachtenswert ist hierbei, dass der zurückgegebene Zustand eine Kopie des Eingabestreams enthält. Dadurch ist sichergestellt, dass der Stream zu keinem Zeitpunkt infolge der Übereinstimmung mit einem Literal modifiziert wird – und das ist insofern von Bedeutung, als jeder Alternative einer `AlternationExpression` identische Versionen des Eingabestreams zur Verfügung stehen sollten.

Nachdem nun alle Klassen, aus denen der abstrakte Syntaxbaum besteht, bereitstehen, kann als Nächstes festgelegt werden, wie er zusammengesetzt wird. Statt einen gesonderten Parser für reguläre Ausdrücke zu schreiben, werden hier einige Operationen für die `RegularExpression`-Klassen definiert, so dass durch die Auswertung eines Smalltalk-Ausdrucks ein entsprechender abstrakter Syntaxbaum gebildet wird. Auf diese Weise kann der integrierte Smalltalk-Compiler so genutzt werden, als wäre er ein Parser für reguläre Ausdrücke.

Zur Erzeugung des abstrakten Syntaxbaums müssen `|`, `repeat` und `&` wie folgt als Operationen in der Klasse `RegularExpression` definiert werden:

```
& aNode
  ^ SequenceExpression new
    expression1: self expression2: aNode asRExp

repeat
  ^ RepetitionExpression new repetition: self

| aNode
  ^ AlternationExpression new
```

```

    Alternative1: self alternative2: aNode asRExp
asRExp
^ self

```

Die Operation `asRExp` konvertiert Literale in `RegularExpression`-Objekte. Operationen dieser Art sind in der Klasse `String` definiert:

```

& aNode
^ SequenceExpression new
    Expression1: self asRExp expression2: aNode asRExp

repeat
^ RepetitionExpression new repetition: self

| aNode
^ AlternationExpression new
    Alternative1: self asRExp alternative2: aNode asRExp

asRExp
^ LiteralExpression new components: self

```

Würden diese Operationen weiter oben in der Klassenhierarchie angesiedelt (`SequenceableCollection` in Smalltalk-80, `IndexedCollection` in Smalltalk/V), dann würden sie auch für Klassen wie `Array` und `OrderedCollection` definiert – und damit wären reguläre Ausdrücke mit Objektsequenzen aller Art abgleichbar.

Das zweite Codebeispiel zum Design Pattern *Interpreter* (*Interpreter*) beschreibt ein in C++ implementiertes System zur Manipulation und Auswertung von booleschen Ausdrücken. Bei den Terminalsymbolen in C++ handelt es sich um boolesche Variablen, sprich die Konstanten `true` und `false`. Die Nichtterminalsymbole repräsentieren Ausdrücke, die die Operatoren `and`, `or` und `not` enthalten. Die Grammatik ist in diesem Fall wie folgt definiert:

Hinweis

Der Einfachheit halber wird der Operator-Rangfolge in diesem Beispiel keine Beachtung geschenkt, sondern stattdessen vorausgesetzt, dass sie der Zuständigkeit des Objekts unterliegt, das den Syntaxbaum erzeugt.

```

BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp | '('
BooleanExp ')'
AndExp ::= BooleanExp 'and' BooleanExp

```

```

OrExp ::= BooleanExp 'or' BooleanExp
NotExp ::= 'not' BooleanExp
Constant ::= 'true' | 'false'
VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'

```

Es werden zwei Operationen für boolesche Ausdrücke definiert: Evaluate prüft einen booleschen Ausdruck in einem Kontext, der jeder Variablen entweder den Wert `true` oder `false` zuweist. Und Replace erzeugt mittels Ersetzen einer Variablen durch einen Ausdruck einen neuen booleschen Ausdruck. In dem vorliegenden Beispiel modifiziert die Operation also den Ausdruck selbst.

An dieser Stelle werden nur die Klassen `BooleanExp`, `VariableExp` und `AndExp` eingehender betrachtet. (Abgesehen von dem verwendeten Operator stimmen die Klassen `OrExp` und `NotExp` weitestgehend mit der Klasse `AndExp` überein.) Die Klasse `Constant` repräsentiert die booleschen Konstanten.

Die Klasse `BooleanExp` definiert die Schnittstelle für alle Klassen, die einen booleschen Ausdruck repräsentieren:

```

class BooleanExp {
public:
    BooleanExp();
    virtual ~BooleanExp();

    virtual bool Evaluate(Context&) = 0;
    virtual BooleanExp* Replace(const char*, BooleanExp&) = 0;
    virtual BooleanExp* Copy() const = 0;
};

```

Die Klasse `Context` definiert die Abbildung von Variablen auf boolesche Werte, die durch die C++-Konstanten `true` und `false` dargestellt werden. Sie besitzt folgende Schnittstelle:

```

class Context {
public:
    bool Lookup(const char*) const;
    void Assign(VariableExp*, bool);
};

```

`VariableExp` beschreibt eine benannte Variable:

```

class VariableExp : public BooleanExp {
public:
    VariableExp(const char*);
    virtual ~VariableExp();

    virtual bool Evaluate(Contexts);
};

```

```

        virtual BooleanExp* Replace(const char*, BooleanExp&);
        virtual BooleanExp* Copy() const;
private:
    char * _name;
};

```

Der Konstruktor nimmt den Variablenamen als Argument entgegen:

```

VariableExp::VariableExp (const char* name) {
    _name = strdup(name);
}

```

Die Auswertung einer Variablen ermittelt deren Wert im aktuellen Kontext:

```

bool VariableExp::Evaluate (Context& aContext) {
    return aContext.Lookup(_name);
}

```

Durch das Kopieren einer Variablen wird eine neue VariableExp-Instanz zurückgeliefert:

```

BooleanExp* VariableExp::Copy () const {
    return new VariableExp(_name);
}

```

Um eine Variable durch einen Ausdruck zu ersetzen, wird zunächst überprüft, ob sie denselben Namen hat wie die als Argument übergebene Variable:

```

BooleanExp* VariableExp::Replace (
    const char* name, BooleanExp& exp
) {
    if (strcmp(name, _name) ==0) {
        return exp.Copy();
    } else {
        return new VariableExp(_name);
    }
}

```

Die Klasse AndExp repräsentiert einen durch eine UND-Verknüpfung zweier boolescher Ausdrücke erzeugten Ausdruck:

```

class AndExp : public BooleanExp {
public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~AndExp();

    virtual bool Evaluate(Context&);

    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
}

```

```

private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
};

AndExp::AndExp ( BooleanExp* op1, BooleanExp* op2 ) {
    _operand1 = op1;
    _operand2 = op2;
}

```

Das AndExp-Objekt wertet zunächst seine zwei Operanden aus und gibt anschließend das logische »UND« der beiden Ergebnisse zurück:

```

bool AndExp::Evaluate ( Context& aContext ) {
    return
        _operand1->Evaluate(aContext) &&
        _operand2->Evaluate(aContext);
}

```

Außerdem implementiert AndExp die Operationen Copy und Replace durch rekursive Aufrufe seiner Operanden:

```

BooleanExp* AndExp::Copy () const {
    return
        new AndExp(_operand1->Copy(), _operand2->Copy());
}

BooleanExp* AndExp::Replace (const char* name, BooleanExp& exp) {
    return
        new AndExp(
            _operand1->Replace(name, exp),
            _operand2->Replace(name, exp)
        );
}

```

Nun kann der boolesche Ausdruck

```
(true and x) or (y and (not x ))
```

definiert und für die Belegung der Variablen x und y mit true oder false ausgewertet werden:

```

BooleanExp* expression;
Context context;

VariableExp* x = new VariableExp("X");
VariableExp* y = new VariableExp("Y");

```

```

expression = new OrExp(
    new AndExp(new Constant(true), x),
    new AndExp(y, new NotExp(x))
);

context.Assign(x, false);
context.Assign(y, true);

bool result = expression->Evaluate(context);

```

Bei dieser Belegung von x und y ergibt die Auswertung des Ausdrucks den Wert `true`. Zur Auswertung des Ausdrucks mit einer anderen Variablenbelegung muss lediglich der Kontext verändert werden.

Und schließlich kann die Variable y durch einen neuen Ausdruck ersetzt und erneut ausgewertet werden:

```

VariableExp* z = new VariableExp("Z");
NotExp not_z(z);

BooleanExp* replacement = expression->Replace("Y", not_z);

context.Assign(z, true);

result = replacement->Evaluate(context);

```

Dieses Beispiel veranschaulicht einen wichtigen Aspekt des Design Patterns *Interpreter* (*Interpreter*): Ein Satz kann mithilfe vieler verschiedener Operationsarten »interpretiert« werden. Von den drei für die Klasse `BooleanExp` definierten Operationen kommt `Evaluate` der allgemeinen Vorstellung von der Arbeitsweise eines Interpreters am nächsten – die im Wesentlichen in der Interpretation eines Codes oder Ausdrucks sowie der Rückgabe eines einfachen Ergebnisses besteht.

Aber auch `Replace` kann als Interpreter verstanden werden, dessen Kontext sich durch den Namen der zu ersetzenen Variablen sowie den Ausdruck, durch den sie ersetzt wird, definiert und dessen Ergebnis ein neuer Ausdruck ist. Selbst `Copy` ist im Grunde genommen ein Interpreter, wenn auch mit leerem Kontext. Da es sich bei `Replace` und `Copy` ja eigentlich bloß um einfache Baumoperationen handelt, mag es zwar auf den ersten Blick etwas merkwürdig erscheinen, sie als Interpreter aufzufassen – dass sie aber tatsächlich weitreichende Ähnlichkeiten aufweisen, wird spätestens in den zum Design Pattern *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)) angeführten Beispielen deutlich, die aufzeigen, wie sich alle drei Operationen in separate »Interpreter«-*Visitor*-Objekte refaktorisieren lassen.

Das Pattern *Interpreter* (*Interpreter*) stellt durchaus mehr dar, als nur eine über eine Klassenhierarchie verteilte Operation, die das Design Pattern *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) nutzt. Evaluate wird als Interpreter verstanden, weil die BooleanExp-Klassenhierarchie als Repräsentation einer Sprache aufgefasst wird. Würde man eine ähnliche Klassenhierarchie zur Repräsentation des Zusammenbaus beweglicher Fertigungsteile zugrunde legen, wäre es dagegen unwahrscheinlich, dass Operationen wie weight und copy als Interpreter verstanden würden, obwohl sie ebenfalls über eine Klassenhierarchie verteilt wären und das Pattern *Composite* (*Kompositum*) verwenden würden – einfach weil bewegliche Fertigungsteile nicht als Sprache aufgefasst werden. Es ist also alles eine Frage der Perspektive: Würde man dazu übergehen, Grammatiken für bewegliche Fertigungsteile zu publizieren, dann würden auch die auf diesen Teilen ausgeführten Operationen als Interpretationarten der jeweiligen Sprache betrachtet werden.

Praxisbeispiele

Die Anwendung des Design Patterns *Interpreter* (*Interpreter*) ist insbesondere in mit objektorientierten Sprachen implementierten Compilern, wie zum Beispiel den Smalltalk-Compilern, weit verbreitet. **SPECTalk** nutzt dieses Pattern zur Interpretation von Formatbeschreibungen für Eingabedateien [Sza92]. Und der Constraint-Löser **QOCA** verwendet es zur Auswertung von Constraints (Konsistenzbedingungen) [HHMV92].

In seiner allgemeinen Form (d. h. als eine über eine Klassenhierarchie verteilte und auf dem Design Pattern *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) basierende Operation) wird das Pattern *Interpreter* (*Interpreter*) fast immer überall dort verwendet, wo auch das Pattern *Composite* (*Kompositum*) eingesetzt wird. Es sollte allerdings besser denjenigen Fällen vorbehalten sein, in denen die Klassenhierarchie als Repräsentation einer Sprache zu betrachten ist.

Verwandte Patterns

Composite (*Kompositum*, siehe [Abschnitt 4.3](#)): Der abstrakte Syntaxbaum ist eine Instanz des Patterns *Composite* (*Kompositum*).

Flyweight (*Fliegengewicht*, siehe [Abschnitt 4.6](#)): Zeigt die gemeinsame Nutzung von Terminalsymbolen in dem abstrakten Syntaxbaum auf.

Iterator (*Iterator*, siehe [Abschnitt 5.4](#)): Der Interpreter kann zur Traversierung der Struktur ein *Iterator*-Objekt verwenden.

Visitor (*Besucher*, siehe [Abschnitt 5.11](#)): Kann zur Kapselung des Verhaltens jedes einzelnen Knotens im abstrakten Syntaxbaum in einer einzigen Klasse verwendet werden.

5.4 Iterator (Iterator)

Objektbasiertes Verhaltensmuster

Zweck

Bereitstellung eines sequenziellen Zugriffs auf die Elemente eines aggregierten Objekts, ohne dessen zugrunde liegende Struktur offenzulegen.

Auch bekannt als

Cursor

Motivation

Ein aggregiertes Objekt wie eine Liste sollte eine Zugriffsmöglichkeit auf die in ihm enthaltenen Elemente bereitstellen, ohne dass dabei zwangsläufig auch die interne Struktur offengelegt werden muss. Des Weiteren sollte eine Liste zweckdienlich auf verschiedene Arten traversiert werden können, wobei die List-Schnittstelle jedoch nicht mit Operationen für verschiedene Traversierungsvarianten belastet werden sollte – selbst dann nicht, wenn bereits absehbar ist, welche Operationen benötigt werden. Und schließlich sollte auch die parallele Ausführung mehrerer Traversierungsvorgänge auf ein und derselben Liste möglich sein.

Das Design Pattern *Iterator* (*Iterator*) erfüllt all diese Anforderungen. Der Grundgedanke dieses Patterns ist, die Zuständigkeit für den Zugriff auf das

Listenobjekt sowie für dessen Traversierung an einen **Iterator** zu übertragen: Die Iterator-Klasse definiert eine Schnittstelle für den Zugriff auf die Elemente der Liste, und ein Iterator-Objekt ist für die Überwachung des aktuellen Elements zuständig, d. h., es ist stets darüber informiert, welche Elemente bereits traversiert wurden.

So kann eine Klasse `List` beispielsweise eine Instanz `ListIterator` besitzen und in folgender Beziehung zu ihr stehen:

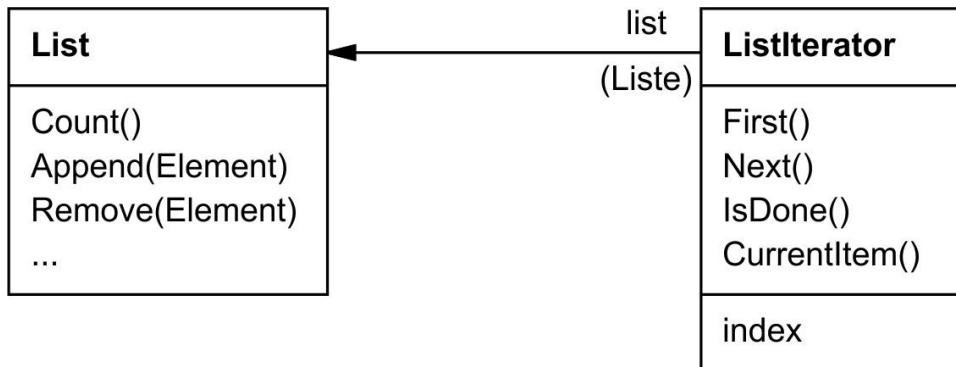


Abb. 5.15: Beziehung zwischen `List`-Klasse und `ListIterator`-Instanz

Nach der Definition der zu traversierenden `List`-Klasse kann die Klasse `ListIterator` instanziert werden, die Operationen für den sequenziellen Zugriff auf die Listenelemente bereitstellt: Die `CurrentItem`-Operation gibt das aktuelle Element aus der Liste zurück. `First` initialisiert das aktuelle Element als das erste Element. `Next` rückt das aktuelle Element auf das nachfolgende Element vor. Und `IsDone` überprüft, ob bereits über das letzte Element hinaus traversiert wurde – womit der Traversierungsvorgang abgeschlossen wäre.

Die Separierung des Traversierungsmechanismus von dem `List`-Objekt ermöglicht die Definition von Iteratoren für verschiedene Traversierungsregeln, ohne sie in der `List`-Schnittstelle einzeln aufführen zu müssen. So könnte beispielsweise eine Klasse `FilteringListIterator` den Zugriff nur auf Elemente gewähren, die bestimmte Filterbedingungen erfüllen.

Dabei ist allerdings zu beachten, dass das `Iterator`-Objekt und die Liste aneinander gekoppelt sind und der Client deshalb wissen muss, dass nicht etwa irgendeine beliebige Aggregatstruktur, sondern eine *Liste* traversiert wird – der Client ist also an eine bestimmte aggregierte Struktur gebunden. Sinnvoller wäre es jedoch, wenn die Aggregatklsasse ohne Anpassungen des Clientcodes variiert werden könnte. Und um dies zu erreichen, muss das Iteratorkonzept dahingehend erweitert werden, dass es die **polymorphe Iteration** unterstützt.

Als Beispiel soll hier angenommen werden, dass neben der `List`-Klasse auch eine `SkipList`-Implementierung vorliegt. Eine »Skip-Liste« [Pug90] ist eine probabilistische Datenstruktur, die ähnliche Merkmale aufweist wie ausbalancierte Baumstrukturen. Es soll Code geschrieben werden können, der sowohl für `List`- als auch für `SkipList`-Objekte geeignet ist.

Dazu wird eine Klasse `AbstractList` definiert, die eine einheitliche Schnittstelle zur Bearbeitung von Listen bereitstellt. Außerdem wird eine abstrakte `Iterator`-Klasse benötigt, die eine einheitliche Iterationsschnittstelle definiert. Anschließend können für die verschiedenen Listenimplementierungen jeweils eigene konkrete `Iterator`-Unterklassen definiert werden. Auf diese Weise ist die Unabhängigkeit des Iterationsmechanismus von den konkreten aggregierten Klassen gewährleistet:

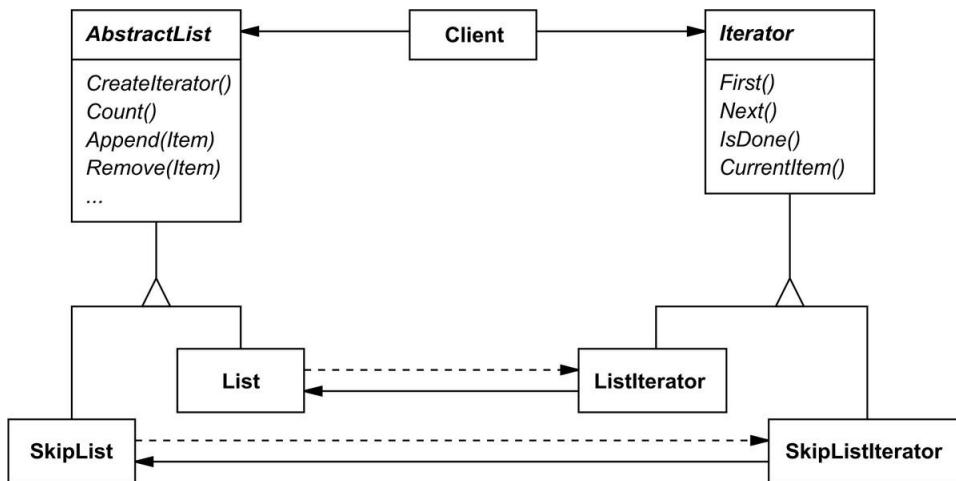


Abb. 5.16: Entkopplung des Iterationsmechanismus mithilfe der Klasse `AbstractList`

Jetzt geht es nur noch um die Frage, wie der Iterator erzeugt wird. Da der zu schreibende Code von den konkreten `List`-Klassen unabhängig sein soll, ist eine einfache Instanziierung einer spezifischen Klasse keine geeignete Lösung. Stattdessen wird den Listenobjekten selbst die Zuständigkeit für die Erzeugung ihrer entsprechenden Iteratoren zugewiesen. Zu diesem Zweck wird eine Operation wie `CreateIterator` benötigt, mit deren Hilfe die Clients Iteratorobjekte anfordern können.

Bei der Operation `CreateIterator` handelt es sich um eine Factory Method (siehe Design Pattern *Factory Method (Fabrikmethode)*, [Abschnitt 3.3](#)), die es einem Client an dieser Stelle ermöglicht, den passenden Iterator von einem Listenobjekt abzufragen. Der Factory-Method-Ansatz gestattet die Nutzung von zwei Klassenhierarchien – einer für Listen und einer zweiten für Iteratoren –, die in diesem Beispiel durch `CreateIterator` »verbunden« werden.

Anwendbarkeit

Das Design Pattern *Iterator (Iterator)* ermöglicht

- den Zugriff auf die Inhalte eines Aggregate-Objekts, ohne dessen interne Darstellung preiszugeben.
- die Unterstützung mehrerer zeitgleicher Traversierungen von aggregierten Objekten.
- die Bereitstellung einer einheitlichen Schnittstelle für die Traversierung unterschiedlicher zusammengesetzter Strukturen (d. h. zur Unterstützung der polymorphen Iteration).

Struktur

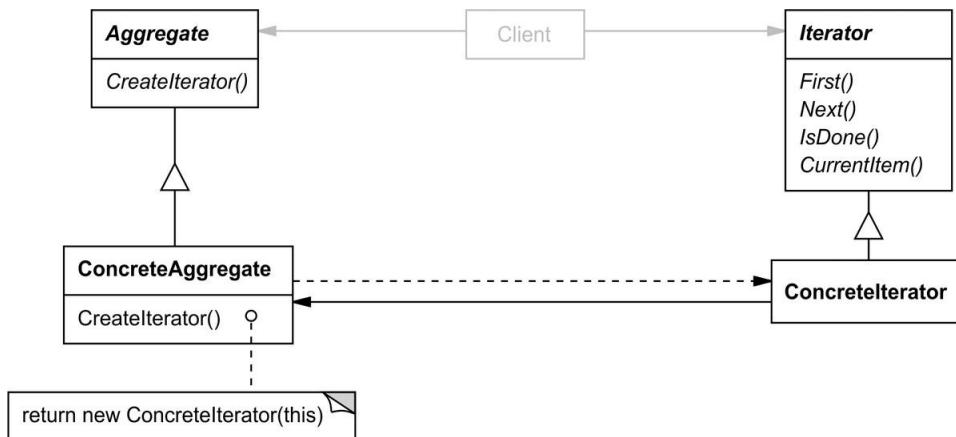


Abb. 5.17: Die Struktur des Design Patterns *Iterator (Iterator)*

Teilnehmer

- **Iterator**
 - Definiert eine Schnittstelle für den Zugriff auf die Elemente sowie für deren Traversierung.
- **ConcreteIterator**
 - Implementiert die **Iterator**-Schnittstelle.

- Verwaltet während der Traversierung die aktuelle Position des Aggregate-Objekts.
- **Aggregate**
 - Definiert eine Schnittstelle zur Erzeugung eines Iterator-Objekts.
- **ConcreteAggregate**
 - Implementiert die Schnittstelle zum Erzeugen des Iterators in der Form, dass sie eine Instanz der passenden ConcreteIterator-Klasse zurückgibt.

Interaktionen

- Die Klasse ConcreteIterator verwaltet das aktuelle Objekt in dem Aggregat und kann das nachfolgende Objekt in der Traversierung ermitteln.

Konsequenzen

Das Design Pattern *Iterator* (*Iterator*) bewirkt im Wesentlichen Folgendes:

1. *Unterstützung verschiedener Traversierungsvarianten.* Komplexe Aggregate-Objekte können auf vielfältige Art und Weise traversiert werden. So kann beispielsweise die Traversierung der Parse-Bäume bei der Codegenerierung und der semantischen Prüfung wahlweise gemäß der symmetrischen Reihenfolge oder der Hauptreihenfolge (Inorder- oder Preorder-Verfahren) stattfinden. Iteratoren erleichtern den Wechsel des Traversierungsalgorithmus, indem eine Iterator-Instanz einfach durch eine andere ersetzt wird. Zudem können zur Unterstützung weiterer Traversierungsvariationen entsprechende Iterator-Unterklassen definiert werden.
2. *Vereinfachung der Aggregatschnittstelle.* Durch die Verwendung der Traversierungsschnittstelle des Iterators ist keine weitere Schnittstelle für die Aggregate-Klassen erforderlich.
3. *Parallele Ausführung mehrerer Traversierungsvorgänge.* Da ein Iterator seinen Traversierungszustand selbst überwacht, können mehrere Traversierungsvorgänge gleichzeitig ausgeführt werden.

Implementierung

Es stehen zahlreiche Implementierungsvarianten für das Design Pattern *Iterator* (*Iterator*) zur Verfügung. Die wichtigsten von ihnen werden im Folgenden vorgestellt. Welche Vor- und Nachteile sie im Einzelfall mit sich bringen, hängt häufig von den Kontrollstrukturen der verwendeten Sprache ab. Einige Sprachen (z. B. CLU [LG86]) unterstützen das Pattern auch direkt.

1. *Steuernder Teilnehmer der Iteration.* Ein wesentlicher Entscheidungsfaktor bei der Implementierung dieses Patterns ist die Frage, welcher Teilnehmer die Iteration steuert – der Iterator oder der Client, der den Iterator nutzt. Erfolgt die Steuerung der Iteration durch den Client, wird der Iterator als **externer Iterator** bezeichnet. Wird die Iteration hingegen vom Iterator-Objekt gesteuert, spricht man von einem **internen Iterator**. Clients, die einen *externen* Iterator verwenden, müssen die Traversierung selbst vorantreiben und das jeweils nächste Element explizit vom Iterator anfordern. Im Gegensatz dazu übergibt der Client einem *internen* Iterator eine Operation, die dieser dann ausführt und auf jedes Element im Aggregat anwendet.

Hinweis

Bloch spricht in seinen Untersuchungen zu diesem Thema nicht von externen und internen, sondern von **aktiven** bzw. **passiven** Iteratoren [Blo94]. Damit bezieht er sich auf die Rolle des Clients und nicht auf den Aktivitätsgrad des Iterators.

Externe Iteratoren sind flexibler als interne Iteratoren. Zum Beispiel lassen sich zwei Objektsammlungen mithilfe von externen Iteratoren relativ leicht auf Gleichheit überprüfen, während dies mit internen Iteratoren praktisch unmöglich ist. Besonders große Schwächen offenbaren interne Iteratoren in Sprachen wie C++, die keine anonymen Funktionen, Closures oder Continuations zulassen, wie dies in Smalltalk und CLOS der Fall ist. Andererseits sind interne Iteratoren allerdings einfacher zu handhaben, weil sie auch die Iterationslogik definieren.

2. *Definition des Traversierungsalgorithmus.* Der Traversierungsalgorithmus kann nicht nur im Iterator definiert werden – sondern ebenso gut auch vom Aggregat, während der Iterator dann nur zum Speichern des Iterationszustands

dient. In dieser Variante wird der Iterator als **Cursor** bezeichnet, weil er lediglich als Pointer fungiert, der auf die aktuelle Position in dem Aggregat zeigt. Hier wird der Cursor beim clientseitigen Aufruf der `Next`-Operation des Aggregats als Argument übergeben, und die `Next`-Operation modifiziert dann den Zustand des Cursors.

Hinweis

Das Konzept des Cursors ist ein einfaches Beispiel für das Design Pattern *Memento* (*Memento*, siehe [Abschnitt 5.6](#)), dessen Implementierung größtenteils auf denselben Faktoren basiert, die auch für die Implementierung des Patterns *Iterator* (*Iterator*) gelten.

Ist dagegen der Iterator für den Traversierungsalgorithmus zuständig, können ohne Weiteres unterschiedliche Iterationsalgorithmen für ein und dasselbe Aggregat verwendet werden. Ebenso einfach ist es auch, denselben Algorithmus für verschiedene Aggregate zu nutzen. Allerdings benötigt der Traversierungsalgorithmus mitunter Zugriff auf die privaten Variablen des Aggregats – und das bedeutet wiederum, dass die Integration des Traversierungsalgorithmus in den Iterator die Kapselung des Aggregats beeinträchtigt.

3. *Robustheit des Iterators.* Die Modifizierung eines Aggregats während der Traversierung birgt einige Gefahren. Sollten Elemente zum Aggregat hinzugefügt oder daraus entfernt werden, kann es passieren, dass doppelt auf ein Element zugegriffen oder ein Element übersprungen wird. Grundsätzlich ließe sich dieses Problem zwar durch das Kopieren des Aggregats und die anschließende Ausführung der Traversierung auf der Kopie vermeiden, in der Regel erweist sich dieses Vorgehen jedoch als zu ressourcenlastig.

Ein **robuster Iterator** gewährleistet, dass die Traversierung nicht durch Ergänzungen und Löschungen von Elementen beeinträchtigt wird – und zwar ohne dass das Aggregat kopiert werden muss. Die Implementierung robuster Iteratoren kann auf vielfältige Art und Weise erfolgen. In den meisten Fällen wird der Iterator am Aggregat registriert. Wird dann ein Element hinzugefügt oder entfernt, passt das Aggregat entweder den internen Zustand seines ursprünglich erzeugten Iterators entsprechend an oder es verwaltet die Daten intern, damit eine korrekte Traversierung sichergestellt ist.

Während sich Kofler ausführlich mit der Frage der Implementierung robuster Iteratoren in ET++ beschäftigt [Kof93], beschreibt Murray deren Implementierung für die Klasse der USL-Standardkomponenten [Mur93].

4. **Zusätzliche Iterator-Operationen.** Eine minimale `Iterator`-Schnittstelle besteht aus den Operationen `First`, `Next`, `IsDone` und `CurrentItem`. Darauf hinaus können aber auch noch weitere nützliche Operationen ergänzt werden. Beispielsweise könnten geordnete Aggregate die `Previous`-Operation verwenden, die den Iterator auf das vorangehende Element zurücksetzt. Ebenso könnte für sortierte und indizierbare Objektsammlungen die `SkipTo`-Operation von Nutzen sein, um den Iterator auf einem Objekt zu positionieren, das bestimmte Kriterien erfüllt.

Hinweis

Die `Iterator`-Schnittstelle kann durch das Zusammenfassen von `Next`, `IsDone` und `CurrentItem` zu einer einzigen Operation, die zum nächsten Objekt vorrückt und dieses zurückgibt, sogar noch »minimalistischer« gestaltet werden. Nach Beendigung der Traversierung gibt sie dann einen speziellen Wert (z. B. 0) zurück, der den Abschluss der Iteration kennzeichnet.

5. **Verwendung polymorpher Iteratoren in C++.** Polymorphe Iteratoren sind an eine Bedingung geknüpft: Das `Iterator`-Objekt muss mithilfe einer Factory Method dynamisch alloziert werden. Aus diesem Grund sollten solche Iteratoren nur dann verwendet werden, wenn der Einsatz des Polymorphie-Prinzips unabdingbar ist – ansonsten sollten lieber konkrete Iteratoren genutzt werden, die auf dem Stack alloziert werden können.

Zudem bringen polymorphe Iteratoren noch einen weiteren Nachteil mit sich: Sie müssen vom Client gelöscht werden. Diese Vorgehensweise ist allerdings recht fehleranfällig, weil die Freigabe eines auf dem Heap allozierten Iterators im Anschluss an seine Verwendung häufig vergessen wird – das gilt insbesondere dann, wenn eine Operation mehrere Austrittspunkte anbietet. Und sollte eine Ausnahmebehandlung ausgelöst werden, dann wird das `Iterator`-Objekt überhaupt nicht freigegeben.

Hier bietet das Design Pattern *Proxy* (*Proxy*, siehe [Abschnitt 4.7](#)) einen Ausweg:

Statt des echten Iterators kann ein auf dem Stack alloziertes Proxy als Stellvertreter verwendet werden. Das Proxy löscht den Iterator in seinem Destruktor – und damit wird der echte Iterator gemeinsam mit dem Proxy freigegeben, sobald es den aktuellen Gültigkeitsbereich verlässt. Das Proxy sorgt also selbst im Fall einer Ausnahmebehandlung stets für eine ordentliche Speicherbereinigung. Hierbei handelt es sich um eine Anwendung der bekannten C++-Technik RAII – »Resource Allocation Is Initialization« (dt. »Ressourcenbelegung ist Initialisierung«) [ES90]. Ein passendes Beispiel dazu wird im Abschnitt »Beispielcode« beschrieben.

6. *Privilegierter Zugriff für Iteratoren.* Angesichts der starken Kopplung zwischen einem Iterator und dem Aggregat, das ihn erzeugt hat, kann er praktisch als eine Art Erweiterung des Aggregats aufgefasst werden. In C++ lässt sich diese enge Beziehung durch die Deklaration des Iterators als `friend`-Objekt seines Aggregats ausdrücken. Auf diese Weise müssen keine speziellen Operationen im Aggregat definiert werden, um den Iteratoren eine effiziente Implementierung der Traversierung zu ermöglichen.

Allerdings kann diese Form des privilegierten Zugriffs seitens der Iteratoren die Definition neuer Traversierungen erschweren, weil die Ergänzung weiterer `friend`-Objekte eine Modifizierung der Aggregatschnittstelle bedingt. Um dieses Problem zu umgehen, ist es hilfreich, die `Iterator`-Klasse mit geschützten (`protected`) Operationen für den Zugriff auf wichtige, aber nicht öffentlich zugängliche Membervariablen des Aggregats auszustatten, die dann von den `Iterator`-Unterklassen (und *nur* diesen) verwendet werden können, um privilegierten Zugriff auf das Aggregat zu erhalten.

7. *Iteratoren für Komposita.* Externe Iteratoren können in rekursiven Aggregatstrukturen wie denen des Design Patterns *Composite (Kompositum*, siehe [Abschnitt 4.3](#)) verhältnismäßig schwierig zu implementieren sein, weil sich eine Position in der Struktur über mehrere Ebenen eines verzweigten Aggregats erstrecken kann. Deshalb muss ein externer Iterator zur Überwachung des aktuellen Objekts einen Pfad durch das *Composite*-Objekt speichern. In manchen Fällen ist dann die Verwendung eines internen Iterators einfacher, weil dieser die aktuelle Position einfach dadurch aufzeichnet, dass er sich selbst rekursiv aufruft und den zugehörigen Pfad implizit auf dem Call-Stack speichert.

Wenn die Knoten in einem *composite*-Objekt eine Schnittstelle für den Wechsel zu deren Geschwister-, Eltern- und Kindobjekten besitzen, kann allerdings ein Cursor-basierter Iterator die bessere Alternative sein: Der Cursor muss nur den

aktuellen Knoten überwachen und kann sich in Bezug auf die Traversierung des Composite-Objekts auf die Schnittstelle des Knotens verlassen.

Composite-Objekte müssen häufig auf mehr als eine Art traversiert werden. Gängige Varianten sind das Preorder-, das Postorder-, das Inorder- sowie das Breadth-first-Verfahren (Hauptreihenfolge, Nebenreihenfolge, symmetrische Reihenfolge und Breitensuche). Jede dieser Traversierungsarten kann mithilfe einer eigenen Iterator-Klasse unterstützt werden.

8. *Null-Iteratoren*. Ein NullIterator-Objekt ist ein degenerierter Iterator, der die Berücksichtigung von Rahmenbedingungen erleichtert. Per Definition gilt die Traversierung beim NullIterator-Objekt stets als abgeschlossen, d. h., die Auswertung seiner IsDone-Operation gibt *immer* den Wert true zurück.

Ein NullIterator-Objekt kann die Traversierung von baumförmig strukturierten Aggregaten (wie z. B. Komposita) dadurch vereinfachen, dass das aktuelle Element an jedem Punkt der Traversierung einen Iterator für seine Kindobjekte anfordert. Während die Aggregate-Elemente daraufhin wie gewohnt einen konkreten Iterator zurückgeben, liefern die Blattobjekte eine NullIterator-Instanz zurück – und ermöglichen damit eine einheitliche Implementierung der Traversierung über die gesamte Struktur hinweg.

Beispielcode

Im Folgenden werden am Beispiel einer einfachen List-Klasse aus der in [Anhang C](#) präsentierten Basisbibliothek zwei Implementierungsvarianten für Iteratoren vorgestellt: eine für die Front-to-back-Traversierung (von vorne nach hinten) der Liste und eine weitere für die Back-to-front-Traversierung (von hinten nach vorne).

Hinweis

Die Basisbibliothek unterstützt ausschließlich die Variante der Front-to-back-Traversierung.

Neben den Verwendungsmöglichkeiten für Iteratoren wird auch aufgezeigt, wie sich eine Festlegung auf eine bestimmte Implementierung vermeiden lässt. Anschließend wird das Design so modifiziert, dass eine korrekte Lösung der Iteratoren

sichergestellt ist. Und das letzte Beispiel befasst sich schließlich mit dem Konzept des internen Iterators und den Unterschieden zu seinem externen Gegenstück.

1. **List- und Iterator-Schnittstellen.** Als Erstes soll an dieser Stelle der für die Implementierung von Iteratoren bedeutsamste Teil der List-Schnittstelle betrachtet werden (komplette Schnittstelle siehe [Abschnitt C.1](#)):

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

Die Klasse `List` stellt mit ihrer öffentlichen Schnittstelle eine hinreichend effiziente Möglichkeit zur Ausführung der Iteration in beiden Traversierungsvarianten zur Verfügung. Daher ist es hier nicht nötig, Iteratoren privilegierten Zugriff auf die zugrunde liegende Datenstruktur zu gewähren, d. h., die `Iterator`-Klassen sind keine `friend`-Klassen von `List`. Um eine transparente Nutzung der verschiedenen Traversierungsarten zu ermöglichen, wird eine abstrakte `Iterator`-Klasse definiert, die ihrerseits die `Iterator`-Schnittstelle definiert:

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;

protected:
    Iterator();
};
```

2. **Implementierung der Iterator-Unterklassen.** `ListIterator` ist eine Unterklasse von `Iterator`, die wie folgt definiert ist:

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
```

```

        virtual bool IsDone() const;
        virtual Item CurrentItem() const;

private:
    const List<Item>* _list;
    long _current;
};

```

Die Implementierung dieser Unterklasse ist einfach und unkompliziert – sie speichert das `List`-Objekt mit einem Index `_current` in der Liste:

```

template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}

```

Die Operation `First` positioniert den Iterator auf dem ersten Element:

```

template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}

```

`Next` rückt das aktuelle Element auf das nächste Element vor:

```

template <class Item>
void ListIterator<Item>::Next () {
    _current++;
}

```

Die `IsDone`-Operation prüft, ob sich der Index auf ein Element im `List`-Objekt bezieht:

```

template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
}

```

Und `CurrentItem` gibt schließlich das Element an der aktuellen Indexposition zurück. Ist die Iteration bereits abgeschlossen, wird die `IteratorOutOfBoundsException`-Ausnahmebehandlung ausgelöst:

```

template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBoundsException();
    }
}

```

```
    return _list->Get(_current);  
}
```

Die Implementierung der Unterklasse `ReverseListIterator` erfolgt nach dem gleichen Prinzip, allerdings positioniert die `First`-Operation den Index `_current` in diesem Fall am Ende der Liste, und `Next` setzt ihn dann sukzessive in absteigender Richtung bis zum ersten Element herunter.

3. *Verwendung der Iteratoren.* An dieser Stelle wird eine `List`-Klasse mit `Employee`-Objekten angenommen, die in Listenform ausgegeben werden sollen. Die `Employee`-Klasse unterstützt zu diesem Zweck eine `Print`-Operation. Nun wird eine Operation `PrintEmployees` definiert, die einen Iterator als Argument entgegennimmt und diesen verwendet, um die Liste zu traversieren und die Ausgabe auszuführen:

```
void PrintEmployees (Iterator<Employee*>& i) {  
    for (i.First(); !i.IsDone (); i.Next()) {  
        i.CurrentItem()->Print();  
    }  
}
```

Da sowohl Iteratoren für Back-to-front- als auch für Front-to-back-Traversierungen vorhanden sind, kann diese Operation zur Ausgabe der `Employee`-Objekte in beiden Sortierreihenfolgen wiederverwendet werden:

```
List<Employee*>* employees;  
// ...  
ListIterator<Employee*> forward(employees);  
ReverseListIterator<Employee*> backward(employees);  
  
PrintEmployees(forward);  
PrintEmployees(backward);
```

4. *Vermeidung der Festlegung auf eine bestimmte Listenimplementierung.* Als Nächstes gilt es zu überlegen, wie eine `SkipList`-Variante der Listenimplementierung den Iterationscode beeinflussen würde. Eine Unterklasse `SkipList` von `List` muss einen `SkipListIterator` bereitstellen, der die `Iterator`-Schnittstelle implementiert. Intern muss der `SkipListIterator` mehr als nur einen Index verwalten, um eine effiziente Iteration durchzuführen. Da er sich jedoch nach der `Iterator`-Schnittstelle richtet, kann die `PrintEmployees`-Operation auch dann verwendet werden, wenn die `Employee`-Objekte in einem `SkipList`-Objekt gespeichert sind:

```
SkipList<Employee*>* employees;  
// ...
```

```
SkipListIterator<Employee*> iterator(employees);
PrintEmployees(iterator);
```

Auch wenn dieser Ansatz funktioniert, ist es grundsätzlich besser, wenn keine bestimmte Listenimplementierung – wie in diesem Fall `SkipList` – vorgegeben würde. Deshalb kann eine Klasse `AbstractList` definiert werden, die die List-Schnittstelle für verschiedene Listenimplementierungen standardisiert. `List` und `SkipList` werden dann zu Unterklassen von `AbstractList`.

Zur Ermöglichung der polymorphen Iteration definiert die Klasse `AbstractList` eine Factory Method `CreateIterator`, die von Unterklassen überschrieben wird, um deren jeweils passende Iteratoren zurückzugeben:

```
template <class Item>
class AbstractList {
public:
    virtual Iterator<Item>* CreateIterator() const = 0;
    // ...
};
```

Als Alternative könnte auch eine allgemeine Mixin-Klasse `Traversable` angelegt werden, die die Schnittstelle zur Erzeugung eines Iterators definiert. Aggregierte Klassen können die polymorphe Iteration durch Erben von `Traversable` unterstützen.

`List` überschreibt `CreateIterator` und gibt dann ein `ListIterator`-Objekt zurück:

```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```

Damit sind nun alle Voraussetzungen erfüllt, um von einer konkreten Implementierung unabhängigen Code für die Ausgabe der `Employee`-Objekte schreiben zu können:

```
// Es ist lediglich bekannt, dass ein AbstractList-Objekt
// zur Verfügung steht

AbstractList<Employee*>* employees;
// ...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
```

```
    delete iterator;
```

5. *Sicherstellung der Speicherbereinigung von Iteratoren.* Ein wichtiger Aspekt beim Einsatz der Factory Method `createIterator` ist, dass es zwar ein neu alloziertes `Iterator`-Objekt zurückgibt – dieses muss jedoch separat bzw. von Hand gelöscht werden. Andernfalls, z. B. wenn dies vergessen wird, entsteht ein Speicherleck. Um den Clients die Arbeit zu erleichtern, wird daher eine Klasse `IteratorPtr` erzeugt, die als Proxy für einen Iterator fungiert und die Speicherbereinigung des `Iterator`-Objekts sicherstellt, sobald es den aktuellen Gültigkeitsbereich verlässt.

Die Klasse `IteratorPtr` wird immer auf dem Stack alloziert.

Hinweis

Um die Allozierung auf dem Stack beim Kompilieren zu gewährleisten, können die `new`- und `delete`-Operatoren als privat deklariert werden. Eine Implementierung ist dafür nicht erforderlich.

In C++ erfolgt der Aufruf des zugehörigen Destruktors zur Lösung des echten Iterators automatisch. `IteratorPtr` überlädt sowohl `operator ->` als auch `operator *` in der Art und Weise, dass `IteratorPtr` wie ein Pointer auf einen Iterator behandelt werden kann. Die Memberfunktionen von `IteratorPtr` sind alle inline implementiert, so dass kein Mehraufwand entstehen kann:

```
template <class Item>
class IteratorPtr {
public:
    IteratorPtr(Iterator<Item>* i): _i(i) { }
    ~IteratorPtr() { delete _i; }
    Iterator<Item>* operator->() { return _i; }
    Iterator<Item>& operator*() { return *_i; }

private:
    // Sperren der Kopier- und
    // Zuweisungsfunktionalität, um
    // Mehrfachlöschen von _i zu vermeiden

    IteratorPtr(const IteratorPtr&);
    IteratorPtr& operator=(const IteratorPtr&);

private:
```

```
    Iterator<Item>* _i;  
};
```

Mithilfe der Klasse `IteratorPtr` lässt sich der Code für die Ausgabe der `Employee`-Objekte wie folgt vereinfachen:

```
AbstractList<Employee*>* employees;  
// ...  
  
IteratorPtr<Employee*> iterator(employees->CreateIterator());  
PrintEmployees(*iterator);
```

6. *Internes ListIterator-Objekt.* Abschließend folgt an dieser Stelle noch ein Beispiel für die mögliche Implementierung einer internen oder passiven `ListIterator`-Klasse. Hier wird die Iteration vom Iterator gesteuert, der auf jedes einzelne Element eine Operation anwendet.

Das Problem ist in diesem Fall die Art der Parametrisierung des Iterators mit der Operation, die auf jedem Element ausgeführt werden soll. C++ unterstützt keine anonymen Funktionen oder Closures, wie sie andere Sprachen zur Verfügung stellen. Dennoch existieren mindestens zwei Umsetzungsmöglichkeiten: zum einen die Übergabe eines Pointers an eine (globale oder statische) Funktion und zum anderen die Unterklassenbildung. Im ersten Fall ruft der Iterator die an ihn übergebene Operation an jedem Punkt innerhalb der Iteration auf. Im zweiten Fall ruft der Iterator eine Operation auf, die zur Herbeiführung eines spezifischen Verhaltens von einer Unterklasse überschrieben wird.

Beide Verfahren sind nicht ideal: Häufig soll im Laufe der Iteration ein Zustand aufgebaut werden, wofür Funktionen nicht unbedingt besonders gut geeignet sind – denn dann müssten statische Variablen verwendet werden, um den Zustand zu speichern. Eine `Iterator`-Unterklasse bietet dagegen eine bequeme Möglichkeit, den fortlaufenden Zustand wie in einer Instanzvariablen zu speichern – allerdings bedeutet die Erstellung einer Unterklasse für jede der unterschiedlichen Traversierungsvarianten einen nicht zu unterschätzenden Mehraufwand.

Der nachfolgende Code skizziert die zweite Vorgehensweise, die auf der Unterklassenbildung basiert. Der interne Iterator ist in diesem Fall mit `ListTraverser` bezeichnet:

```
template <class Item>  
class ListTraverser {  
public:
```

```

        ListTraverser(List<Item>* aList);
        bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};

```

Die Klasse `ListTraverser` nimmt eine `List`-Instanz als Parameter entgegen. Intern verwendet sie zur Ausführung der Traversierung ein externes `ListIterator`-Objekt. `Traverse` startet die Traversierung und ruft für jedes Element die Operation `ProcessItem` auf. Der interne Iterator kann die Traversierung gegebenenfalls abbrechen, indem er `ProcessItem` zur Rückgabe von `false` veranlasst. Und `Traverse` meldet zurück, ob die Traversierung vorzeitig abgebrochen wurde:

```

template <class Item>
ListTraverser<Item>::ListTraverser (
    List<Item>* aList
) : _iterator(aList) { }

template <class Item>
bool ListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        result = ProcessItem(_iterator.CurrentItem());
        if (result == false) {
            break;
        }
    }
    return result;
}

```

Als Nächstes wird ein `ListTraverser`-Objekt für die Ausgabe der ersten 10 `Employee`-Objekte verwendet. Dazu wird eine Unterklasse von `ListTraverser` erzeugt und `ProcessItem` überschrieben. Die Anzahl der ausgegebenen `Employee`-Objekte wird in einer `_count`-Instanzvariablen mitgezählt:

```

class PrintNEmployees : public ListTraverser<Employee*> {
public:
    PrintNEmployees(List<Employee*>* aList, int n) :
        ListTraverser<Employee*>(aList),
        _total(n), _count(0) { }

```

```

protected:
    bool ProcessItem(Employee* const&);

private:
    int _total;
    int _count;
};

bool PrintNEmployees::ProcessItem (Employee* const& e) {
    _count++;
    e->Print();
    return _count < _total;
}

```

Der folgende Code demonstriert die Umsetzung der Ausgabe der ersten 10 Employee-Objekte mithilfe der Klasse PrintNEmployees:

```

List<Employee*>* employees;
// ...

PrintNEmployees pa(employees, 10)
pa.Traverse();

```

Beachtenswert ist hier, dass der Client die Iterationsschleife nicht selbst spezifiziert. Die gesamte Iterationslogik ist wiederverwendbar. Und das ist zugleich auch der wichtigste Vorteil eines internen Iterators – auch wenn er aufgrund der Notwendigkeit, eine neue Klasse definieren zu müssen, mit einem Mehraufwand verbunden ist. Die Nutzung eines externen Iterators würde im Vergleich dazu wie folgt aussehen:

```

ListIterator<Employee*> i(employees);
int count = 0;

for (i.First(); !i.IsDone(); i.Next()) {
    count++;
    i.CurrentItem()->Print();

    if (count >= 10) {
        break;
    }
}

```

Interne Iteratoren können verschiedene Iterationsarten kapseln. So kapselt FilteringListTraverser beispielsweise eine Iteration, die nur diejenigen Elemente verarbeitet, die einer Überprüfung standhalten:

```

template <class Item>
class FilteringListTraverser {

```

```

public:
    FilteringListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
    virtual bool TestItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};

```

Diese Schnittstelle ist mit Ausnahme einer zusätzlichen `TestItem`-Memberfunktion, die die Überprüfung definiert, mit der Schnittstelle von `ListTraverser` identisch. `TestItem` wird dabei zur Spezifizierung der Überprüfung von den Unterklassen überschrieben.

Und `Traverse` entscheidet schließlich auf der Grundlage des Ergebnisses der Überprüfung, ob die Traversierung fortgesetzt wird:

```

template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        if (TestItem(_iterator.CurrentItem())) {
            result = ProcessItem(_iterator.CurrentItem());
            if (result == false) {
                break;
            }
        }
    }
    return result;
}

```

In einer abgewandelten Form könnte diese Klasse `Traverse` auch so definieren, dass die Traversierung abgebrochen wird, falls mindestens ein Element der Überprüfung standhält.,

Hinweis

Bei der in den vorgenannten Beispielen vorgestellten `Traverse`-Operation handelt es sich um eine *Template Method (Schablonenmethode*, siehe [Abschnitt 5.10](#)) mit den primitiven Operationen `TestItem` und `ProcessItem`.

Praxisbeispiele

Iteratoren sind in objektorientierten Systemen durchaus gebräuchlich. Die meisten Klassenbibliotheken für Objektsammlungen bieten in der einen oder anderen Form auch Iteratoren an.

So zum Beispiel auch die beliebten **Booch-Components** [Boo94]. Sie gestatten sowohl die Implementierung von Queues fester maximaler Größe (**bounded**) als auch dynamisch wachsende (**unbounded**) Queues. Die Queue-Schnittstelle ist durch eine abstrakte Queue-Klasse definiert. Zur Unterstützung der polymorphen Iteration innerhalb der verschiedenen Queue-Implementierungen wird der Queue-Iterator im Sinne der abstrakten Queue-Klassenschnittstelle definiert. Diese Variante hat den Vorteil, dass keine Factory Method benötigt wird, um die jeweils zugehörigen Iteratoren von den Queue-Implementierungen abzufragen. Allerdings muss die Schnittstelle der abstrakten Queue-Klasse dann auch leistungsfähig genug sein, um den Iterator effizient implementieren zu können.

In **Smalltalk** müssen die Iteratoren nicht so expliziert definiert werden. Die Standard-Container-Klassen (Bag, Set, Dictionary, OrderedCollection, String etc.) definieren eine interne Iterator-Methode `do:`, die einen Block (d. h. eine Closure) als Argument annimmt. Jedes Element in dem Container wird an die lokale Variable in dem Block gebunden und dann wird der Block ausgeführt. Darüber hinaus enthält Smalltalk auch einen Satz von Stream-Klassen, die eine Iterator-ähnliche Schnittstelle unterstützen. `ReadStream` ist im Wesentlichen ein Iterator und kann als externer Iterator für alle sequenziellen Objektsammlungen verwendet werden. Für nicht-sequenzielle Container wie Set und Dictionary sind keine standardmäßigen externen Iteratoren verfügbar.

Die **ET++-Container-Klassen** [WGM88] bieten sowohl polymorphe Iteratoren als auch das zuvor beschriebene Proxy zur Speicherbereinigung. Das **Unidraw Application Framework für grafische Editoren** [VL90] nutzt Cursor-basierte Iteratoren.

ObjectWindows 2.0 [Bor94] stellt eine Klassenhierarchie von Iteratoren für Behälter zur Verfügung. Hier können verschiedene Behältertypen mit demselben Verfahren iteriert werden. Die Syntax der Objectwindow-Iteration basiert auf der Überladung des Postinkrement-Operators `++`, um die Iteration voranzutreiben.

Verwandte Patterns

Composite (Kompositum, siehe [Abschnitt 4.3](#)): Iteratoren werden häufig auf rekursive Strukturen wie Komposita angewendet.

Factory Method (Fabrikmethode, siehe [Abschnitt 3.3](#)): Polymorphe Iteratoren verwenden zur Instanziierung der passenden `Iterator`-Unterklasse Factory Methods.

Memento (Memento, siehe [Abschnitt 5.6](#)): Die Design Patterns *Memento (Memento)* und *Iterator (Iterator)* werden häufig gemeinsam genutzt. Ein *Memento*-Objekt kann den Zustand einer Iteration aufzeichnen und dann intern im *Iterator* gespeichert werden.

5.5 Mediator (Vermittler)

Objektbasiertes Verhaltensmuster

Zweck

Definition eines Objekts, das die Interaktionsweise eines Objektsatzes in sich kapselt. Das Design Pattern *Mediator (Vermittler)* begünstigt lose Kopplungen, indem es die explizite Referenzierung der Objekte untereinander unterbindet und so eine individuelle Steuerung ihrer Interaktionen ermöglicht.

Motivation

Das objektorientierte Design fordert die Verhaltensverteilung auf mehrere Objekte – dadurch können allerdings recht komplexe Objektstrukturen mit zahlreichen unmittelbaren Kopplungen entstehen. Im schlimmsten Fall kann das sogar so weit gehen, dass jedes einzelne Objekt alle anderen Objekte kennt und referenziert.

Generell begünstigt die Partitionierung eines Systems in mehrere Objekte zwar die Wiederverwendbarkeit, andererseits wird dieser Vorteil jedoch durch ausufernde gegenseitige Referenzierungen in hohem Maße wieder zunichtegemacht. Wenn viele

Kopplungen zwischen den Objekten bestehen, sinkt die Wahrscheinlichkeit, dass ein Objekt ohne die Unterstützung der anderen Objekte funktionieren kann – das System wird somit zunehmend monolithisch. Zudem kann die Verteilung des Verhaltens auf mehrere Objekte die Durchführung weitreichender Modifikationen im Systemverhalten erschweren. Und das wiederum führt in letzter Konsequenz dazu, dass zur Anpassung des Systemverhaltens zwangsläufig zahlreiche Unterklassen definiert werden müssen.

Als Beispiel wird im Folgenden die Implementierung von Dialogfenstern in eine grafische Benutzeroberfläche demonstriert. Wie in Abbildung 5.18 zu sehen, stellt ein Dialog eine Reihe von Widgets wie Schaltflächen, Menüs und Eingabefeldern zur Auswahl:

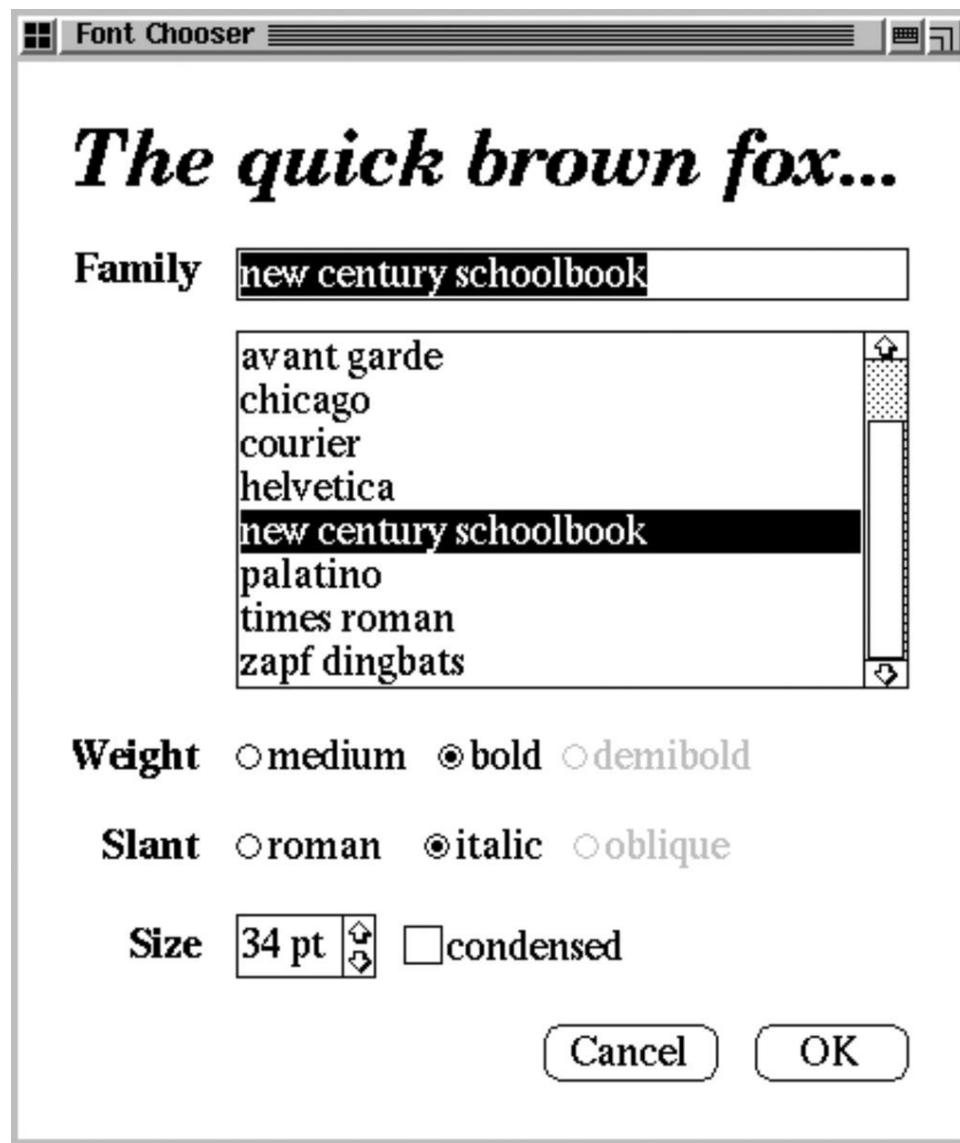


Abb. 5.18: Beispielaufbau eines Dialogfensters

Nicht selten stehen die in einem Dialog präsentierten Widgets in Abhängigkeit zueinander. So kann zum Beispiel die Aktivierbarkeit einer Schaltfläche unmittelbar davon abhängig sein, dass ein bestimmtes, zunächst leeres Eingabefeld mit Text gefüllt wird. Oder der Inhalt eines Eingabefeldes ändert sich entsprechend der in einem Listenfeld (**List Box**) getroffenen Auswahl. Möglich ist auch die umgekehrte Variante, also dass die Eingabe eines Textes in ein Eingabefeld die automatische Auswahl eines oder mehrerer Einträge in einem Listenfeld bewirkt. Und erst wenn sich Text in dem Eingabefeld befindet, werden auch andere Schaltflächen aktivierbar, die es dem User ermöglichen, eine Operation auf den Text anzuwenden, wie z. B. einen Änderungs- oder Löschvorgang.

Individuelle Dialoge bedingen normalerweise auch individuelle Abhängigkeiten zwischen den Widgets. Das bedeutet, dass selbst dann, wenn in den Dialogen gleichartige Widgets angeboten werden, trotzdem nicht einfach existierende Widget-Klassen wiederverwendet werden können. Vielmehr müssen sie so angepasst werden, dass sie dialogspezifische Abhängigkeiten widerspiegeln. Die individuelle Anpassung durch Unterklassenbildung ist allerdings recht mühsam, weil zahlreiche Klassen davon betroffen sind.

Dieser Problematik lässt sich durch die Kapselung des kollektiven Verhaltens in einem separaten Mediator-Objekt begegnen. Ein Mediator-Objekt steuert und koordiniert die Interaktionen innerhalb einer Gruppe von Objekten. Es dient als »Vermittler«, der die direkte gegenseitige Bezugnahme innerhalb einer Objektgruppe unterbindet. Stattdessen referenzieren die Objekte ausschließlich das Mediator-Objekt, wodurch sich die Anzahl der Kopplungen drastisch reduziert.

Zum Beispiel kann ein Objekt `FontDialogDirector`, wie in [Abbildung 5.19](#) gezeigt, als Mediator-Objekt für die Widgets in einem Dialogfenster fungieren. Es weiß, welche Widgets sich in einem Dialogfenster befinden und koordiniert deren Interaktionen – `FontDialogDirector` übernimmt also die Funktion einer Kommunikationsschnittstelle für Widgets:

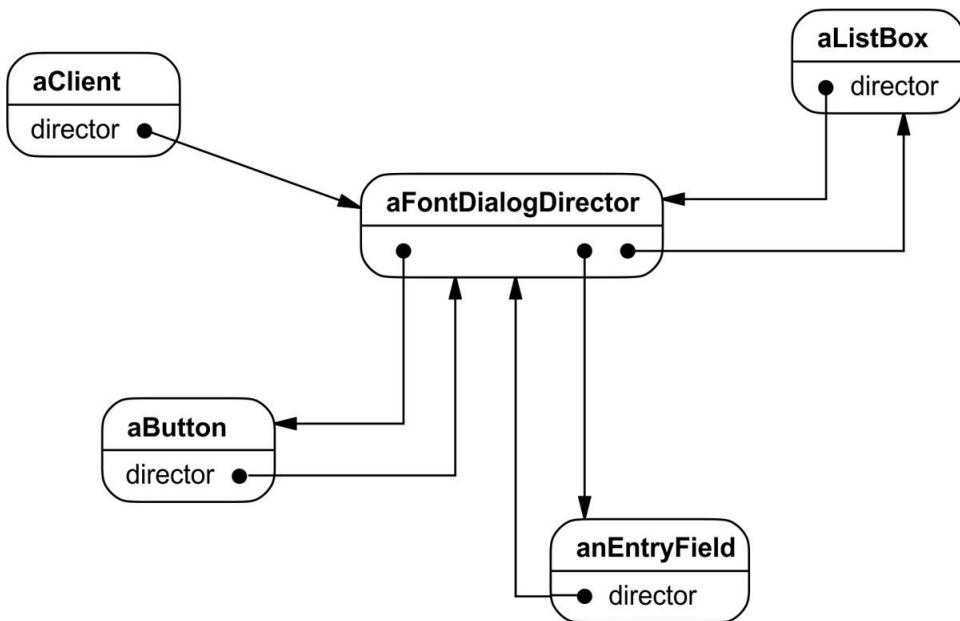


Abb. 5.19: *FontDialogDirector*-Objekt als Kommunikationsschnittstelle

Das in [Abbildung 5.20](#) dargestellte Interaktionsdiagramm veranschaulicht die Interaktionen der einzelnen Objekte beim Eintreten einer Auswahländerung in einem Listenfeld:

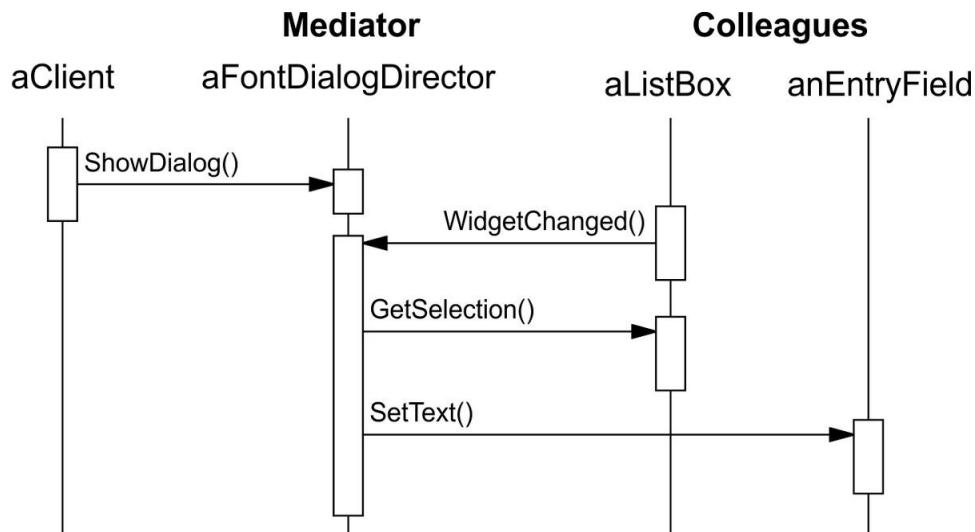


Abb. 5.20: Objektinteraktionen bei Verwendung eines Mediator-Objekts

Die Abfolge der Ereignisse, die bei der Auswahl eines Eintrags aus dem Listenfeld an ein Eingabefeld weitergeleitet werden, stellt sich wie folgt dar:

1. Das Listenfeld teilt seinem Director-Objekt mit, dass es sich geändert hat.
2. Das Listenfeld übermittelt die aktuelle Auswahl an das Director-Objekt.

3. Das Director-Objekt leitet die Auswahl an das Eingabefeld weiter.
4. Da das Eingabefeld nun Text enthält, aktiviert das Director-Objekt die Schaltfläche(n) zum Auslösen der jeweiligen Aktion(en).

Beachtenswert ist hierbei, wie das Director-Objekt zwischen dem Listenfeld und dem Eingabefeld vermittelt: Die Kommunikation zwischen den Widgets erfolgt ausschließlich indirekt, d. h. über das Director-Objekt. Sie brauchen keinerlei Kenntnis voneinander zu haben, sondern müssen lediglich das Director-Objekt kennen. Zudem kann das Verhalten, das in einer einzigen Klasse gekapselt ist, einfach durch die Erweiterung oder den Austausch dieser Klasse modifiziert oder ersetzt werden.

Die schematische Darstellung in [Abbildung 5.21](#) zeigt, wie sich die `FontDialogDirector`-Abstraktion in eine Klassenbibliothek integrieren lässt:

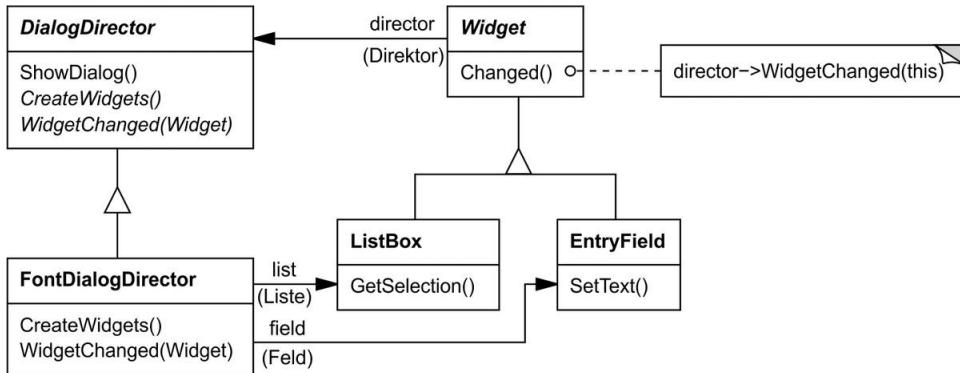


Abb. 5.21: Integration der `FontDialogDirector`-Abstraktion in eine Klassenbibliothek

Die abstrakte Klasse `DialogDirector` definiert das Gesamtverhalten eines Dialogfensters. Durch den Aufruf der Operation `ShowDialog` initiieren die Clients die Anzeige des Dialogs auf dem Bildschirm. Bei `CreateWidgets` handelt es sich um eine abstrakte Operation zur Erzeugung der Widgets in einem Dialog. Auch `WidgetChange` ist eine abstrakte Operation, die wiederum von den Widgets aufgerufen wird, um ihrem Director-Objekt mitzuteilen, dass sie sich geändert haben. Sowohl `CreateWidgets` als auch `WidgetChanged` werden von den `DialogDirector`-Unterklassen überschrieben – im ersten Fall zur Erzeugung der passenden Widgets und im zweiten Fall zur Verarbeitung der aktuell vorliegenden Änderungen.

Anwendbarkeit

Das Design Pattern *Mediator* (*Vermittler*) ist dann geeignet, wenn

- die Kommunikation innerhalb eines Objektsatzes zwar auf wohldefinierte, aber auch komplexe Art und Weise erfolgt. Die damit einhergehenden Abhängigkeiten sind unstrukturiert und schwer zu verstehen.
- sich die Wiederverwendung eines Objekts schwierig gestaltet, weil es zahlreiche andere Objekte referenziert und auch mit diesen kommuniziert.
- die Anpassbarkeit eines über mehrere Klassen verteilten Verhaltens ohne weitreichende Unterklassenbildung gewährleistet sein soll.

Struktur

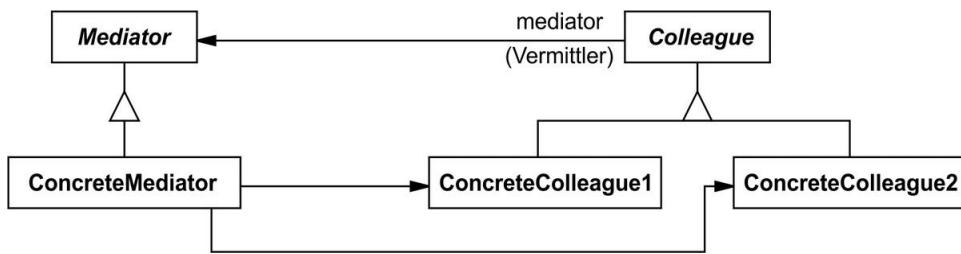


Abb. 5.22: Die Struktur des Design Patterns *Mediator* (*Vermittler*)

Eine typische Objektstruktur könnte wie folgt aussehen:

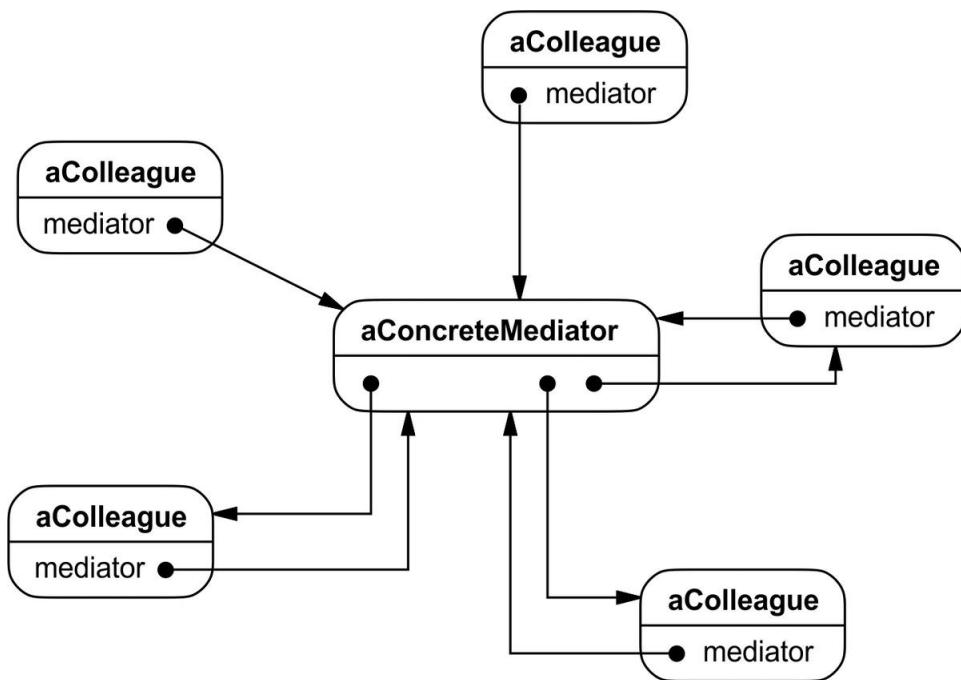


Abb. 5.23: Mögliche Objektstruktur einer Mediator-Klasse

Teilnehmer

- **Mediator** (DialogDirector)
 - Definiert eine Schnittstelle für die Kommunikation mit Colleague-Objekten.
- **ConcreteMediator** (FontDialogDirector)
 - Implementiert das kooperative Verhalten durch Koordination der Colleague-Objekte.
 - Kennt seine Colleague-Objekte und verwaltet diese.
- **Colleague-Klassen** (ListBox, EntryField)
 - Jede Colleague-Klasse kennt ihr Mediator-Objekt.
 - Sämtliche Colleague-Objekte kommunizieren nicht mit den anderen Colleague-Objekten, sondern ausschließlich mit ihrem Mediator-Objekt.

Interaktionen

- **Colleague-Objekte** senden und empfangen Requests von einem **Mediator-Objekt**. Das **Mediator-Objekt** leitet diese Requests dann an die betreffenden **Colleague-Objekte** weiter und implementiert so das kooperative Verhalten.

Konsequenzen

Das Design Pattern **Mediator (Vermittler)** bringt folgende Vor- und Nachteile mit sich:

1. *Eingeschränkte Unterklassenbildung.* Ein **Mediator-Objekt** lokalisiert Verhalten, das ansonsten über mehrere Objekte verteilt würde. Zum Ändern dieses Verhaltens ist lediglich das Ableiten von Unterklassen der Klasse **Mediator** erforderlich – **Colleague-Klassen** können dagegen unverändert wiederverwendet werden.
2. *Entkopplung von Colleague-Objekten.* Ein **Mediator-Objekt** fördert die lose Kopplung zwischen den **Colleague-Objekten**. **Colleague-** und **Mediator-Klassen** können unabhängig voneinander variiert und wiederverwendet werden.
3. *Vereinfachung der Objektprotokolle.* Ein **Mediator-Objekt** ersetzt n-zu-n-Interaktionen durch 1-zu-n-Beziehungen zwischen ihm und seinen **Colleague-Objekten**, die einfacher zu verstehen, zu verwalten und zu erweitern sind.
4. *Abstrahierung der Objektkooperation.* Durch die Handhabung des **Vermittlungsprozesses** als unabhängiges Konzept und seine Kapselung in einem Objekt richtet sich der Fokus auf die von individuellem Verhalten losgelöste Interaktion zwischen den Objekten – und das sorgt wiederum für mehr Transparenz hinsichtlich der Objektinteraktion in einem System.
5. *Zentralisierte Steuerung.* Das Design Pattern **Mediator (Vermittler)** tauscht die Komplexität der Interaktion gegen die Komplexität des **Mediator-Objekts**. Aufgrund dessen, dass ein **Mediator-Objekt** Protokolle kapselt, kann es eine höhere Komplexität erreichen als jedes individuelle **Colleague-Objekt** – und kann sich somit durchaus zu einem monolithischen Konstrukt entwickeln, dass nur schwer zu warten ist.

Implementierung

Bei der Implementierung des Design Patterns *Mediator (Vermittler)* sind folgende Aspekte zu beachten:

1. *Wegfall der abstrakten Mediator-Klasse.* Wenn die `Colleague`-Objekte ausschließlich mit einem `Mediator`-Objekt zusammenarbeiten, besteht keine Notwendigkeit für die Definition einer abstrakten `Mediator`-Klasse. Die durch die `Mediator`-Klasse ermöglichte abstrakte Kopplung gestattet `Colleague`-Objekten die Kooperation mit verschiedenen `Mediator`-Unterklassen und umgekehrt.
2. *Kommunikation zwischen colleague und Mediator.* `Colleague`-Objekte müssen beim Eintritt eines relevanten Ereignisses mit ihrem `Mediator`-Objekt kommunizieren. Eine Möglichkeit, dies zu gewährleisten, besteht darin, das `Mediator`-Objekt mithilfe des Design Patterns *Observer (Beobachter)*, siehe [Abschnitt 5.7](#)) als `Observer` zu implementieren. Dadurch verhalten sich `Colleague`-Klassen wie Subjekte, die das `Mediator`-Objekt im Fall einer Zustandsänderung entsprechend benachrichtigen und die Auswirkungen der Änderungen an die anderen `Colleague`-Objekte weiterleiten.

Ein anderer Ansatz besteht darin, eine spezielle Schnittstelle für Benachrichtigungen in der `Mediator`-Klasse zu definieren, die den `Colleague`-Objekten eine direktere Kommunikation ermöglicht. Smalltalk/V für Windows macht sich zu diesem Zweck eine Art Delegation zunutze: Während der Kommunikation mit dem `Mediator` übergibt ein `Colleague`-Objekt sich selbst als Argument und ermöglicht dem `Mediator`-Objekt damit die Absenderidentifikation. Dieses Verfahren kommt auch in dem nachfolgenden Beispielcode zum Einsatz. Die Smalltalk/V-Implementierung wird im Abschnitt »Verwandte Patterns« näher erläutert.

Beispielcode

Im Folgenden wird eine Klasse `DialogDirector` zur Implementierung des in [Abbildung 5.18](#) dargestellten Dialogfensters zur Schriftartenauswahl verwendet. Die abstrakte Klasse `DialogDirector` definiert die Schnittstelle für `Director`-Objekte:

```
class DialogDirector {  
public:  
    virtual ~DialogDirector();
```

```

    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;

protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};

```

Widget ist die abstrakte Basisklasse für Widgets. Ein Widget kennt sein Director-Objekt:

```

class Widget {
public:
    Widget(DialogDirector*);
    virtual void Changed();

    virtual void HandleMouse(MouseEvent& event);
    // ...
private:
    DialogDirector* _director;
};

```

Die Operation `Changed` ruft die `WidgetChanged`-Operation des Director-Objekts auf. Widgets rufen hingegen die Operation `widgetChanged` ihres Director-Objekts auf, um es über ein bedeutendes Ereignis zu benachrichtigen:

```

void Widget::Changed () {
    _director->WidgetChanged(this);
}

```

Die Unterklassen von `DialogDirector` überschreiben `WidgetChanged`, um die von einer Änderung betroffenen Widgets entsprechend anzupassen. Das Widget leitet wiederum eine Referenz auf sich selbst als Argument an `WidgetChanged` weiter und ermöglicht dem Director-Objekt damit die Identifizierung des geänderten Widgets. Die `DialogDirector`-Unterklassen definieren die `CreateWidgets` zur Erstellung der Widgets in dem Dialogfenster auf rein virtuelle Weise neu.

`ListBox`, `EntryField` und `Button` sind auf spezielle Elemente der Benutzeroberfläche bezogene Unterklassen von `Widget`. `ListBox` stellt eine `GetSelection`-Operation zur Verfügung, die die aktuelle Auswahl zurückgibt, und die Operation `SetText` von `EntryField` übernimmt neuen Text in das Eingabefeld:

```

class ListBox : public Widget {
public:
    ListBox(DialogDirector*);

```

```

        virtual const char* GetSelection();
        virtual void SetList(List<char*>* listItems);
        virtual void HandleMouse(MouseEvent& event);
        // ...
    };

class EntryField : public Widget {
public:
    EntryField(DialogDirector*);

    virtual void SetText(const char* text);
    virtual const char* GetText();
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

```

Jedes Auslösen des simplen Widgets Button führt zum Aufruf der Operation Changed. Dieser Vorgang findet in der Implementierung von HandleMouse statt:

```

class Button : public Widget {
public:
    Button(DialogDirector*);

    virtual void SetText(const char* text);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}

```

Die Klasse FontDialogDirector ist eine Unterklasse von DialogDirector und vermittelt zwischen den Widgets in dem Dialogfenster:

```

class FontDialogDirector : public DialogDirector {
public:
    FontDialogDirector();
    virtual ~FontDialogDirector();
    virtual void WidgetChanged(Widget*);

protected:
    virtual void CreateWidgets();

private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
}

```

```
};
```

FontDialogDirector überwacht die Widgets, die sie anzeigt. Sie definiert CreateWidgets so um, dass die Widgets erzeugt und die Referenzen darauf initialisiert werden:

```
void FontDialogDirector::CreateWidgets () {
    _ok = new Button(this);
    _cancel = new Button(this);
    _fontList = new ListBox(this);
    _fontName = new EntryField(this);

    // Ergänzt die verfügbaren Font-Namen in das Listenfeld
    // Stellt die Widgets in dem Dialogfenster zusammen
}
```

Und WidgetChanged stellt schließlich die korrekte Zusammenarbeit der Widgets sicher:

```
void FontDialogDirector::WidgetChanged (
    Widget* theChangedWidget
) {
    if (theChangedWidget == _fontList) {
        _fontName->SetText(_fontList->GetSelection());

    } else if (theChangedWidget == _ok) {
        // Wendet den Font-Wechsel an und beendet den Dialog
        // ...
    } else if (theChangedWidget == _cancel) {
        // Beendet den Dialog
    }
}
```

Die Komplexität der Operation WidgetChanged steigt proportional zur Komplexität des Dialogs an. Darüber hinaus sind umfangreiche Dialoge zwar auch aus anderen Gründen nicht erstrebenswert, die Komplexität des Mediators kann die Vorteile dieses Design Patterns jedoch auch in anderen Anwendungsformen beeinträchtigen.

Praxisbeispiele

Sowohl in **ET++** [WGM88] als auch in der Klassenbibliothek **THINK C** [Sym93b] werden Director-ähnliche Objekte als Vermittler für die in Dialogen verwendeten Widgets genutzt.

Die Anwendungsarchitektur von **Smalltalk/V für Windows** basiert auf einer Mediator-Struktur [LaL94]. In dieser Umgebung besteht eine Anwendung aus einem Fenster, das in verschiedene Bereiche (**Panes**) unterteilt ist. Die Bibliothek enthält mehrere vordefinierte Bereichsobjekte (Pane-Objekte), wie z. B. `TextPane`, `ListBox`, `Button` usw. Diese Bereiche können ohne Unterklassenbildung genutzt werden. Der Anwendungsentwickler bildet lediglich Unterklassen der Klasse `ViewManager`, die für die Koordination der einzelnen Bereiche zuständig ist. `ViewManager` repräsentiert dabei das Mediator-Objekt, und jedes Pane-Objekt kennt ausschließlich seinen `ViewManager`, der sozusagen der »Eigentümer« des Bereichs ist. Die Bereiche referenzieren sich nicht direkt gegenseitig.

Das in [Abbildung 5.24](#) dargestellte Objektdiagramm zeigt eine Momentaufnahme einer Anwendung zur Laufzeit:

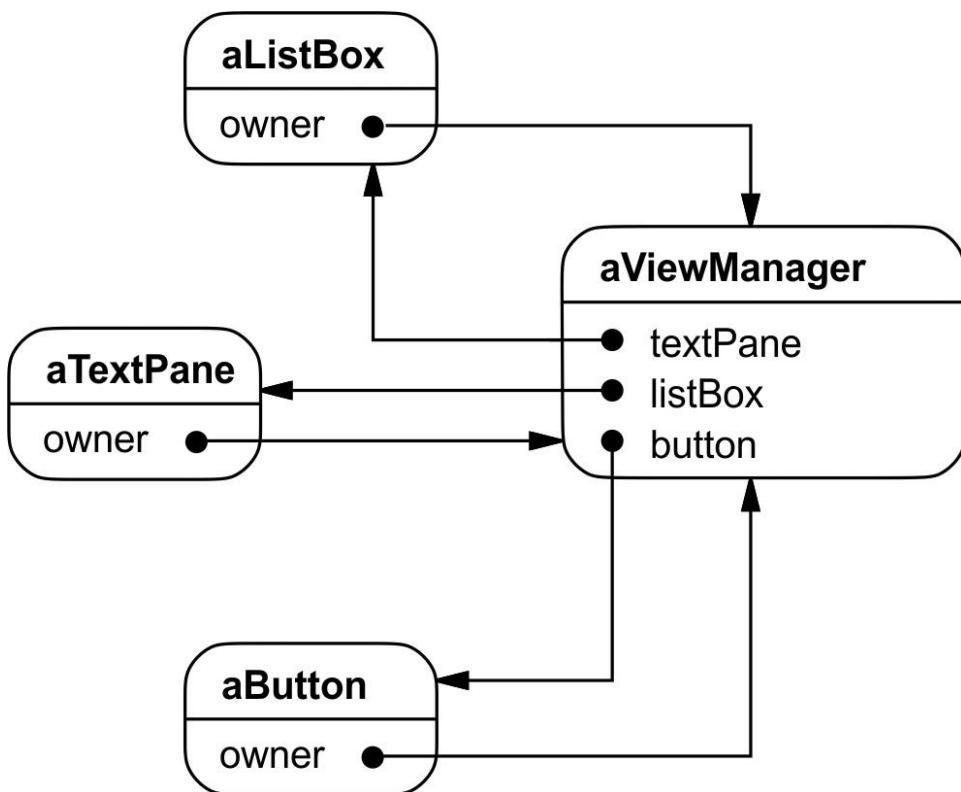


Abb. 5.24: Objektdiagramm einer Anwendung

Die Kommunikation zwischen den Pane-Objekten und dem `ViewManager` wird in **Smalltalk/V** über einen Ereignismechanismus gesteuert: Ein Bereich generiert ein Ereignis, wenn es Informationen vom Mediator-Objekt abfragen oder diesem mitteilen möchte, dass etwas Relevantes passiert ist. Ein Ereignis definiert ein Symbol (z. B. `#select`), das das Ereignis identifiziert. Dieser Selektor ist der Event Handler des Ereignisses und wird aufgerufen, sobald das Ereignis eintritt.

Der folgende Codeauszug veranschaulicht die Erzeugung eines `ListPane`-Objekts in einer `ViewManager`-Unterklasse sowie die Registrierung eines Event Handlers für das `#select`-Ereignis durch den `ViewManager`:

```
self addSubpane: (ListPane new
  paneName: 'myListPane';
  owner: self;
  when: #select perform: #listSelect:).
```

Eine weitere Anwendungsmöglichkeit für das Design Pattern *Mediator (Vermittler)* ist die Koordination komplexer Aktualisierungen. Ein Beispiel dafür ist die im Rahmen des Design Patterns *Observer (Beobachter)*, siehe [Abschnitt 5.7](#)) vorgestellte Klasse `ChangeManager`. Sobald sich ein Objekt ändert, benachrichtigt es den `ChangeManager`, der daraufhin die notwendigen Aktualisierungen koordiniert, indem er die von dem Objekt abhängigen Subjekte entsprechend informiert.

Eine ähnliche Verwendung findet das Design Pattern *Mediator (Vermittler)* auch im **Unidraw Application Framework für grafische Editoren** [VL90]. Hier wird eine Klasse namens `cSolver` genutzt, um die Constraints (Konsistenzbedingungen) der Beziehungen zwischen »Konnektoren« sicherzustellen. Objekte in grafischen Editoren können auf unterschiedliche Art und Weise in scheinbarem Zusammenhang stehen. Konnektoren sind für solche Anwendungen nützlich, in denen die Konnektivität automatisch aufrechterhalten wird, wie beispielsweise in Diagrammeditoren und Schaltkreissystemen. Die Klasse `cSolver` dient als Vermittler zwischen den Konnektoren, der die Constraints der Verbindungen löst und die Konnektorpositionen zwecks korrekter Wiedergabe aktualisiert.

Verwandte Patterns

Das Design Pattern *Facade (Fassade*, siehe [Abschnitt 4.5](#)) unterscheidet sich vom Pattern *Mediator (Vermittler)* dadurch, dass es ein Subsystem von Objekten abstrahiert, um so eine komfortablere Schnittstelle zur Verfügung zu stellen. Es besitzt ein unidirektionales Protokoll, d. h., *Facade*-Objekte senden zwar Requests an die Subsystemklassen, aber nicht umgekehrt. Im Gegensatz dazu ermöglicht das Design Pattern *Mediator (Vermittler)* kooperatives Verhalten, das *Colleague*-Objekte nicht bieten bzw. bieten können, und nutzt ein multidirektionales Protokoll.

Colleague-Objekte können mithilfe des Design Patterns *Observer (Beobachter*, siehe [Abschnitt 5.7](#)) mit dem *Mediator*-Objekt kommunizieren.

5.6 Memento (Memento)

Objektbasiertes Verhaltensmuster

Zweck

Erfassung und Externalisierung des internen Zustands eines Objekts, ohne dessen Kapselung zu beeinträchtigen, so dass es später wieder in diesen Zustand zurückversetzt werden kann.

Auch bekannt als

Token

Motivation

Mitunter ist es erforderlich, den internen Zustand eines Objekts aufzuzeichnen – so zum Beispiel bei der Implementierung von Überwachungs- und UNDO-Mechanismen, die dem User gestatten, versuchsweise ausgeführte Operationen wieder zurückzunehmen oder Fehler zu korrigieren. Damit Objekte wieder in ihren ursprünglichen Zustand zurückversetzt werden können, müssen die entsprechenden Informationen irgendwo gespeichert werden. Normalerweise kapseln die Objekte jedoch ihren Zustand oder einen Teil davon, so dass er anderen Objekten nicht zur Verfügung steht und somit auch nicht extern gespeichert werden kann. Den Zustand preiszugeben, würde die Kapselung beeinträchtigen – und dadurch könnte wiederum die Zuverlässigkeit und Erweiterbarkeit der Anwendung leiden.

Ein gutes Beispiel hierfür ist ein Grafikeditor, der es erlaubt, Verbindungen zwischen Objekten herzustellen. Verbindet ein User zwei Rechtecke mit einer Linie, bleiben sie auch dann miteinander verbunden, wenn der User eins davon verschiebt (siehe [Abbildung 5.25](#)). Der Editor sorgt dafür, dass die Linie sich ausdehnt, um die Verbindung aufrechtzuerhalten:

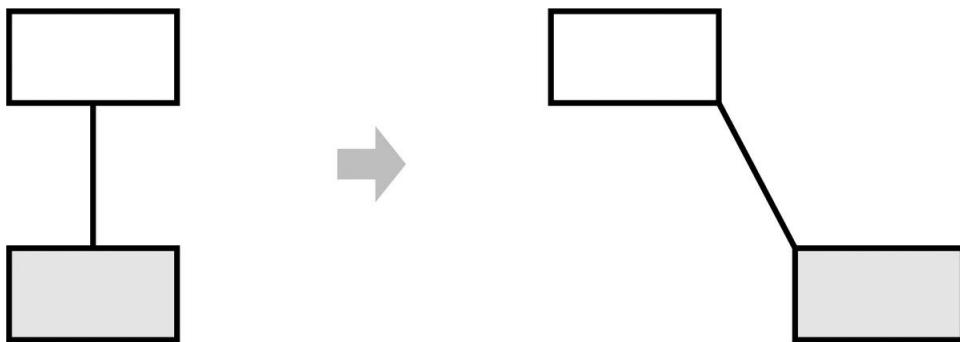


Abb. 5.25: Verschieben eines Rechtecks

Eine wohlbekannte Methode zur Aufrechterhaltung solcher Verbindungsbeziehungen zwischen Objekten ist ein **Constraint-Löser-System**, also ein System, das für die Einhaltung von Konsistenzbedingungen sorgt. Dessen Funktionalität lässt sich in einem Constraint-Löser-Objekt kapseln. In diesem Objekt werden die Verbindungen beim Anlegen in Form von sie beschreibenden mathematischen Gleichungen gespeichert. Sobald der User eine Verbindung erstellt oder das Diagramm irgendwie modifiziert, löst das Objekt diese Gleichungen. Die Resultate der Berechnungen verwendet der Constraint-Löser dann, um die Grafik unter Beibehaltung der gegebenen Verbindungen umzuordnen.

Für diese Anwendung eine UNDO-Funktion bereitzustellen, ist nicht so einfach, wie es auf den ersten Blick aussieht. Ein auf der Hand liegendes Verfahren zum Rückgängigmachen einer Bewegung wäre es, die Distanz der Verschiebung zu speichern und das Objekt in entgegengesetzter Richtung um diese Strecke zurückzubewegen. Allerdings kann dabei nicht garantiert werden, dass sämtliche Objekte dort erscheinen, wo sie sich vorher befunden haben. Angenommen, die Verbindungsgeraden verfügt über etwas Spielraum. In diesem Fall wird durch ein einfaches Zurückbewegen des Rechtecks an die ursprüngliche Position nicht unbedingt das erwünschte Ergebnis erzielt (siehe [Abbildung 5.26](#)):

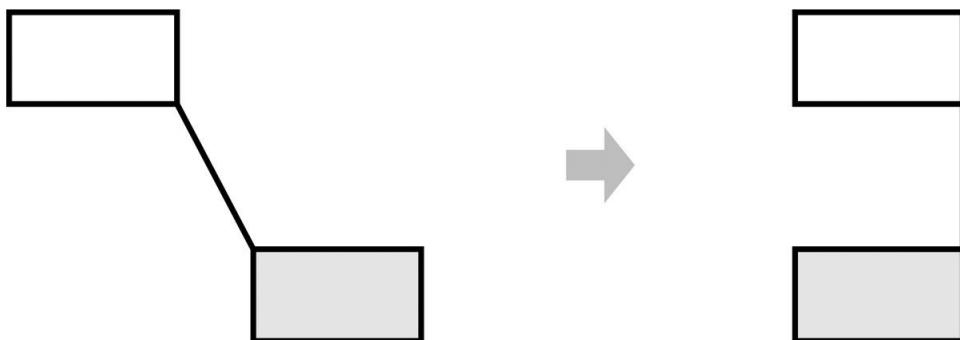


Abb. 5.26: Zurückbewegen des Rechtecks

Die öffentliche Schnittstelle des Constraint-Lösers reicht im Allgemeinen nicht aus, um eine exakte Umkehrung seiner Auswirkungen auf andere Objekte zu ermöglichen. Der Mechanismus zum Rückgängigmachen muss enger mit dem Constraint-Löser zusammenarbeiten, um den ursprünglichen Zustand wiederherzustellen – gleichzeitig sollte dabei aber auch die Preisgabe der Interna des Constraint-Lösers gegenüber dem UNDO-Mechanismus vermieden werden.

Diese Problemstellung lässt sich mit dem Design Pattern *Memento* (*Memento*) lösen. Bei einem **Memento** handelt es sich um ein Objekt, das eine Momentaufnahme des internen Zustands eines anderen Objekts speichert, das als **Urheber** (engl. *Originator*) des Mementos bezeichnet wird. Wenn der UNDO-Mechanismus den Zustand des Urhebers überprüft, ruft er bei ihm ein Memento ab, das der Urheber mit Informationen initialisiert, die seinen aktuellen Zustand beschreiben. Informationen im Memento speichern oder davon abfragen darf nur der Urheber – für andere Objekte ist das Memento uneinsehbar.

In dem soeben vorgestellten Beispiel des Grafikeditors kann der Constraint-Löser die Rolle des Urhebers einnehmen. Die nachstehende Abfolge von Ereignissen beschreibt einen UNDO-Vorgang:

1. Beim Verschieben eines Rechtecks fragt der Editor – quasi als Nebenwirkung – ein Memento beim Constraint-Löser ab.
2. Der Constraint-Löser erzeugt das Memento und liefert es in diesem Fall als eine Instanz der Klasse `SolverState` zurück. Ein solches `SolverState`-Memento besitzt Datenstrukturen, die über den aktuellen Zustand der internen Gleichungen und Variablen des Constraint-Lösers Auskunft geben.
3. Wenn der Benutzer das Verschieben später rückgängig macht, übergibt der Editor das `SolverState`-Memento wieder an den Constraint-Löser.
4. Der Constraint-Löser stellt seine internen Gleichungen und Variablen anhand der Informationen im `SolverState`-Memento wieder her, so dass sie exakt dem ursprünglichen Zustand entsprechen.

Auf diese Art und Weise ist es dem Constraint-Löser möglich, anderen Objekten die zur Wiederherstellung eines vorherigen Zustands erforderlichen Informationen anzutrauen, ohne dabei seine internen Strukturen und Darstellungen offenzulegen.

Anwendbarkeit

Der Einsatz des Design Patterns *Memento* (*Memento*) ist sinnvoll, wenn

- der Zustand eines Objekts (oder ein Teil davon) gespeichert werden soll, damit er später wiederhergestellt werden kann,
und
- eine direkte Schnittstelle zum Abrufen des Zustands Implementierungsdetails enthüllen und die Kapselung des Objekts beeinträchtigen würde.

Struktur

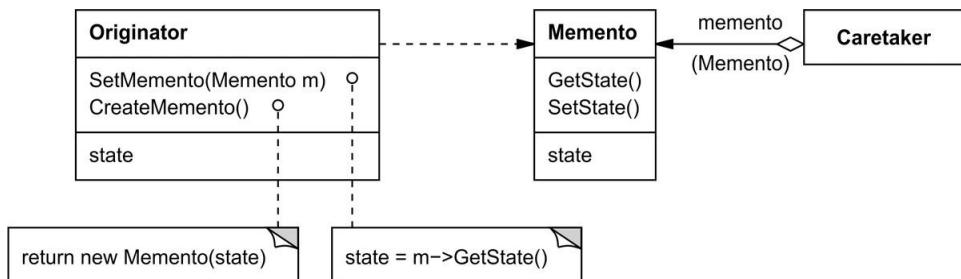


Abb. 5.27: Struktur des Design Patterns *Memento* (*Memento*)

Teilnehmer

- **Memento** (*SolverState*)
 - Speichert den internen Zustand des *originator*-Objekts. Das *Memento* speichert so viel oder so wenig vom internen Zustand des Urhebers, wie es ihm beliebt.
 - Schützt vor dem Zugriff durch andere Objekte als dem *originator*. *Mementos* besitzen im Grunde genommen zwei Schnittstellen: Dem *Caretaker*-Objekt steht nur die »schmale« Schnittstelle des *Mementos* zur Verfügung – es kann das *Memento* lediglich an andere Objekte weitergeben. Dem *originator* wird hingegen die »breite« Schnittstelle bereitgestellt, die den Zugriff auf sämtliche Daten gestattet, die zur Wiederherstellung des vorherigen Zustands erforderlich sind. Im Idealfall

ist nur demjenigen Originator, der das Memento erzeugt hat, der Zugriff auf den internen Zustand des Mementos erlaubt.

- **Originator** (ConstraintSolver)

- Erzeugt ein Memento, das eine Momentaufnahme seines aktuellen internen Zustands enthält.
- Verwendet das Memento, um seinen internen Zustand wiederherzustellen.

- **Caretaker** (UNDO-Mechanismus)

- Ist für die sichere Verwahrung des Mementos verantwortlich.
- Ändert den Inhalt eines Mementos nicht und greift auch nicht darauf zu.

Interaktionen

- Ein Caretaker ruft ein Memento von einem Originator-Objekt ab, verwahrt es für eine Weile und übergibt es dann wieder an den Originator, wie das nachstehende Interaktionsdiagramm demonstriert:

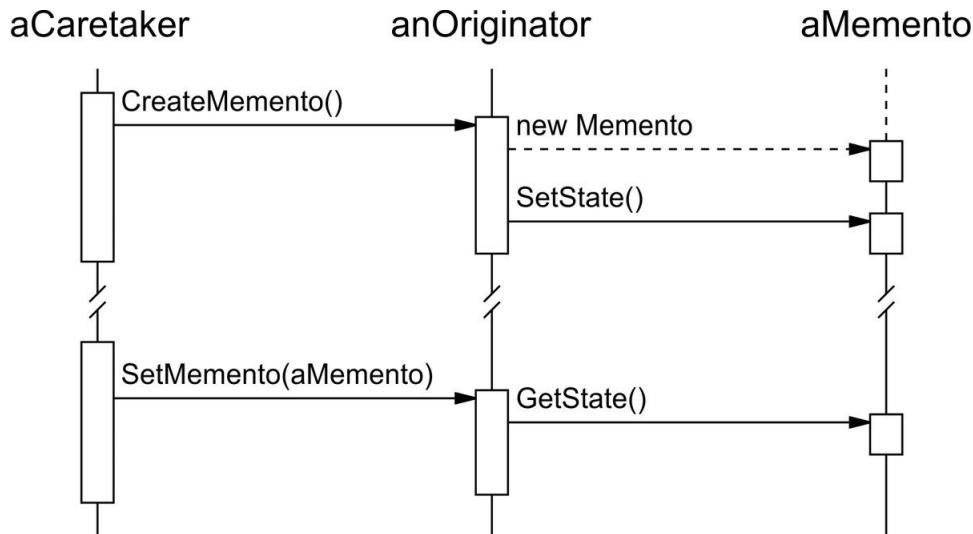


Abb. 5.28: Verwahrung und Übergabe eines Mementos

Es kommt vor, dass der Caretaker das Memento gar nicht wieder an den Urheber übergibt, weil es möglicherweise für den Urheber keine Veranlassung gibt, einen vorherigen Zustand wiederherzustellen.

- Mementos verhalten sich passiv. Ausschließlich der Originator, der ein Memento erzeugt hat, wird dessen Zustand ändern oder abrufen.

Konsequenzen

Das Design Pattern *Memento (Memento)* bringt einige Konsequenzen mit sich:

1. *Beibehaltung der Kapselungsgrenzen.* Das Pattern *Memento (Memento)* vermeidet die Preisgabe von Informationen, die ausschließlich ein Urheber verwalten sollte, die aber dessen ungeachtet außerhalb des Originator-Objekts gespeichert werden müssen. Das Pattern schirmt andere Objekte vor möglicherweise komplexen Interna des Urhebers ab und bewahrt auf diese Weise die Kapselungsgrenzen.
2. *Vereinfachung des Urhebers.* Bei anderen die Kapselungsgrenzen bewahrenden Designs speichert der Originator die Versionen seines internen Zustands, die Clients abgefragt haben. Dadurch wird die gesamte Last der Speicherverwaltung dem Urheber auferlegt. Clients, die die von ihnen abgefragten Zustände selbst verwalten, vereinfachen das Originator-Objekt und unterbinden für die Clients die Notwendigkeit, die Urheber unterrichten zu müssen, wenn ihre Arbeit erledigt ist.
3. *Die Verwendung von Mementos kann aufwendig sein.* Mementos können erheblichen Mehraufwand nach sich ziehen, wenn der Urheber große Mengen an Informationen kopieren muss, um sie im Memento zu speichern oder Clients nur oft genug Mementos abfragen oder übergeben. Wenn die Kapselung und Wiederherstellung des Urheberzustands nicht schnell und einfach möglich sein sollte, ist das Pattern vielleicht ungeeignet. Ziehen Sie in diesem Zusammenhang auch die Erläuterungen zu inkrementellen Änderungen im Abschnitt »Implementierung« in Betracht.
4. *Definition der »schmalen« und der »breiten« Schnittstelle.* In manchen Programmiersprachen kann es schwierig sein sicherzustellen, dass ausschließlich der Originator auf den Zustand des Mementos zugreifen darf.
5. *Versteckter Aufwand bei der Verwaltung von Mementos.* Ein Caretaker ist für das Löschen des von ihm verwalteten Mementos zuständig. Der Caretaker hat allerdings keine Kenntnis vom Umfang des im Memento gespeicherten Zustands. Ein ansonsten leichtgewichtiger Caretaker kann daher unter Umständen beim Speichern von Mementos eine große Speicherverwaltungslast

verursachen.

Implementierung

Bei der Implementierung des Design Patterns *Memento (Memento)* sind zwei Aspekte zu beachten:

1. *Sprachunterstützung*. Mementos besitzen zwei Schnittstellen: eine breite für den Urheber und eine schmale für alle anderen Objekte. Im Idealfall unterstützt die zur Implementierung eingesetzte Programmiersprache zwei statische Schutzniveaus. C++ ermöglicht das, indem der Urheber zu einer `friend`-Klasse des Mementos gemacht und die breite Schnittstelle des Mementos als `private` deklariert wird. Nur die schmale Schnittstelle ist öffentlich. Zum Beispiel:

```
class State;

class Originator {
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
    // ...

private:
    State* _state;
    //Interne Datenstrukturen
    // ...
};

class Memento {
public:
    // Schmale öffentliche Schnittstelle
    virtual ~Memento();

private:
    // Private Member, auf die nur der
    // Originator zugreifen kann
    friend class Originator;
    Memento();

    void SetState(State* );
    State* GetState();
    // ...

private:
    State* _state;
    // ...
};
```

2. Speichern inkrementeller Änderungen. Wenn die Erzeugung und die Rückgabe des Mementos an den Originator in einer vorhersehbaren Reihenfolge stattfinden, braucht das Memento lediglich die *inkrementellen Änderungen* am internen Zustand des Originator-Objekts zu speichern.

So können beispielsweise widerrufbare Befehle in einer Befehlshistorie Mementos verwenden, um zu gewährleisten, dass beim Rückgängigmachen exakt der Zustand vor der Befehlsausführung wiederhergestellt wird (siehe Design Pattern *Command (Befehl, Abschnitt 5.2)*). Durch die Befehlshistorie ist eine feste Reihenfolge vorgegeben, in der Befehle widerrufen oder wiederholt werden können. Mementos brauchen also nur die inkrementellen Änderungen zu speichern, die ein Befehl bewirkt, nicht jedoch den vollständigen Zustand aller betroffenen Objekte. In dem im Abschnitt »Motivation« gezeigten Beispiel braucht der Constraint-Löser nur die sich ändernden internen Strukturen zu speichern, um die Verbindungsline zwischen den Rechtecken zu erhalten – und eben nicht die kompletten Positionsdaten der Objekte.

Beispielcode

Der folgende C++-Code veranschaulicht das schon erläuterte Constraint-Löser-Beispiel. MoveCommand-Objekte (siehe *Command (Befehl, Abschnitt 5.2)*) werden dazu verwendet, Verschiebungen eines Grafikobjekts von einem Ort zum anderen zu widerrufen bzw. zu wiederholen. Der Grafikeditor ruft die Execute- oder die Unexecute-Operation auf, um ein Grafikobjekt zu verschieben bzw. um die Verschiebung rückgängig zu machen. Der Befehl speichert sein Zielobjekt, die beim Verschieben zurückgelegte Strecke sowie eine ConstraintSolverMemento-Instanz – ein Memento, das den Zustand des Constraint-Lösers enthält:

```
class Graphic;
// Basisklasse für die Grafikobjekte
// im Grafikeditor

class MoveCommand {
public:
    MoveCommand(Graphic* target, const Point& delta);
    void Execute();
    void Unexecute();

private:
    ConstraintSolverMemento* _state;
    Point _delta;
    Graphic* _target;
};
```

Die Klasse `ConstraintSolver` legt die für die Verbindungsline geltenden Beschränkungen fest. Wichtig ist ihre Memberfunktion `Solve`, die die durch die `AddConstraint`-Operation eingerichteten Beschränkungen berücksichtigt. Damit das Rückgängigmachen funktioniert, kann der `ConstraintSolver`-Zustand mittels `CreateMemento` in Form einer `ConstraintSolverMemento`-Instanz externalisiert werden. Durch den Aufruf von `SetMemento` wird ein vorheriger Zustands des Constraint-Lösers wiederhergestellt. Die `ConstraintSolver`-Klasse ist ein Singleton (siehe Design Pattern *Singleton (Singleton, Abschnitt 3.5)*).

```
class ConstraintSolver {
public:
    static ConstraintSolver* Instance();

    void Solve();
    void AddConstraint(
        Graphic* startConnection, Graphic* endConnection
    );
    void RemoveConstraint(
        Graphic* startConnection, Graphic* endConnection
    );
    ConstraintSolverMemento* CreateMemento();
    void SetMemento(ConstraintSolverMemento*);

private:
    // Nicht-triviale Zustandsbeschreibung und
    // Operationen, die sicherstellen, dass die
    // für die Verbindungsline geltenden
    // Beschränkungen eingehalten werden
};

class ConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();

private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();
    // Privater Zustand des Constraint-Lösers
};
```

Mit diesen Schnittstellen können die `MoveCommand`-Member `Execute` und `Unexecute` folgendermaßen implementiert werden:

```
void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento();
    // Memento erzeugen
    _target->Move(_delta);
    solver->Solve();
```

```

}

void MoveCommand::Unexecute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _target->Move(-_delta);
    solver->SetMemento(_state);
    // Zustand wiederherstellen
    solver->Solve();
}

```

Execute beschafft sich zunächst einmal ein ConstraintSolverMemento und verschiebt erst dann die Grafik. Unexecute verschiebt die Grafik wieder an ihren ursprünglichen Ort, stellt den vorherigen Zustand des Constraint-Lösers wieder her und weist ihn schließlich an, die geltenden Beschränkungen zu beachten.

Praxisbeispiele

Das vorangehende Codebeispiel beruht auf der **Unidraw**-Implementierung der CSolver-Klasse zur Unterstützung von Verbindungen [VL90].

In der Programmiersprache **Dylan** [App92] stellen Collection-Klassen eine Schnittstelle zum Iterieren zur Verfügung, die an das Design Pattern *Memento (Memento)* angelehnt ist. Diese Klassen kennen ein »Zustandsobjekt«, bei dem es sich um ein Memento handelt, das den Zustand der Iteration repräsentiert. Jede dieser Klassen kann den aktuellen Zustand der Iteration auf beliebige Weise abbilden – die eigentliche Darstellung bleibt den Clients vollständig verborgen. Die Herangehensweise beim Iterieren in Dylan könnte wie folgt in C++ übersetzt werden:

```

template <class Item>
class Collection {
public:
    Collection();

    IterationState* CreateInitialState();
    void Next(IterationState*);
    bool IsDone(const IterationState*) const;
    Item CurrentItem(const IterationState*) const;
    IterationState* Copy(const IterationState*) const;

    void Append(const Item&);
    void Remove(const Item&);
    // ...
};

```

`CreateInitialState` liefert ein initialisiertes `IterationState`-Objekt für die jeweilige Objektsammlung (engl. *Collection*) zurück. Mit `Next` kann in der Iteration ein Schritt weitergesprungen werden – tatsächlich wird dabei der Iterationsindex erhöht. `IsDone` gibt `true` zurück, sobald `Next` das letzte Element der Objektsammlung passiert hat. `CurrentItem` befragt das Zustandsobjekt und liefert dasjenige Element der Objektsammlung zurück, auf das es verweist. `Copy` gibt eine Kopie des Zustandsobjekts zurück. Das ist nützlich, um eine bestimmte Stelle im Ablauf einer Iteration zu kennzeichnen.

Mithilfe der `ItemType`-Klasse kann wie folgt über die Instanzen einer Objektsammlung iteriert werden:

```
class ItemType {  
public:  
    void Process();  
    // ...  
};  
  
Collection<ItemType*> aCollection;  
IterationState* state;  
  
state = aCollection.CreateInitialState();  
  
while (!aCollection.IsDone(state)) {  
    aCollection.CurrentItem(state)->Process();  
    aCollection.Next(state);  
}  
delete state;
```

Hinweis

Beachtenswert ist in diesem Beispiel, dass das Zustandsobjekt `state` nach Abschluss der Iteration gelöscht wird. Sollte `ProcessItem` allerdings eine Ausnahmebehandlung auslösen, wird `delete` nicht aufgerufen und es kommt zu einem Speicherleck. In C++ stellt dies ein Problem dar, nicht aber in Dylan, da es hier eine automatische Speichereinigung gibt. Eine Lösung für diese Problematik wird in Punkt 5 des Abschnitts »Beispielcode« bei den Ausführungen zum Design Pattern *Iterator* (*Iterator*, siehe [Abschnitt 5.4](#)) erläutert.

Eine auf Mementos beruhende Iterationsschnittstelle besitzt zwei interessante Vorteile:

1. Eine einzelne Objektsammlung kann mehrere verschiedene Iterationszustände besitzen. (Gleiches gilt für das Design Pattern *Iterator* (*Iterator*, siehe [Abschnitt 5.4](#))).
2. Es ist nicht notwendig, die Kapselung einer Objektsammlung zu beeinträchtigen, um die Iteration zu unterstützen. Das Memento wird nur von der Objektsammlung selbst konsultiert – andere Objekte haben darauf keinen Zugriff. Andere Ansätze zur Iteration machen es erforderlich, die Kapselung zu beeinträchtigen, weil die *Iterator*-Klassen zu *friend*-Klassen der *Collection*-Klassen gemacht werden müssen (siehe Design Pattern *Iterator* (*Iterator*, [Abschnitt 5.4](#))). Bei der auf Mementos beruhenden Implementierung verhält es sich genau umgekehrt: *Collection* ist eine *friend*-Klasse von *IteratorState*.

Das Constraint-Löser-Toolkit **QOCA** speichert Informationen über inkrementelle Änderungen in Mementos [HHMV92]. Die Clients erhalten ein Memento, das eine Lösung für die aktuell geltenden Beschränkungen beschreibt. Es enthält dabei nur diejenigen beschränkenden Variablen, die sich seit der letzten Lösung geändert haben. Für gewöhnlich ändert sich bei jeder neuen Lösung nur eine kleine Untermenge der einschränkenden Variablen. Diese Untermenge reicht aus, wenn der Constraint-Löser frühere Lösungen zu Rate zieht. Dazu ist es jedoch erforderlich, Mementos der dazwischenliegenden Lösungen wiederherzustellen. Mementos dürfen daher nicht in beliebiger Reihenfolge gespeichert werden. QOCA bedient sich zum Wiederherstellen vorhergehender Lösungen einer Historie.

Verwandte Patterns

Command (Befehl, siehe [Abschnitt 5.2](#)): Befehle können Mementos dazu verwenden, die Zustände widerrufbarer Operationen zu bewahren.

Iterator (Iterator, siehe [Abschnitt 5.4](#)): Mementos können – wie zuvor beschrieben – zur Iteration genutzt werden.

5.7 Observer (Beobachter)

Objektbasiertes Verhaltensmuster

Zweck

Definition einer 1-zu-n-Abhängigkeit zwischen Objekten, damit im Fall einer Zustandsänderung eines Objekts alle davon abhängigen Objekte entsprechend benachrichtigt und automatisch aktualisiert werden.

Auch bekannt als

Dependents, Publish-Subscribe

Motivation

Ein System in eine Sammlung zusammenarbeitender Klassen aufzuteilen, hat häufig den Nebeneffekt, dass es erforderlich wird, die Einheitlichkeit miteinander in Zusammenhang stehender Klassen zu bewahren. Eine enge Kopplung dieser Klassen ist nicht dazu geeignet, eine solche Einheitlichkeit zu erzielen, weil sie deren Wiederverwendbarkeit verringert.

Viele Toolkits grafischer Benutzeroberflächen trennen beispielsweise die eigentliche Darstellung der Benutzerschnittstelle von den zugrunde liegenden Anwendungsdaten [KP88, LVC89, P+88, WGM88]. Die Klassen für Anwendungsdaten und deren Darstellung lassen sich unabhängig voneinander wiederverwenden. Sie können jedoch auch zusammenarbeiten. Sowohl ein Tabellenobjekt als auch ein Säulendiagrammobjekt können im Rahmen desselben Anwendungsdatenobjekts unter Verwendung verschiedener Darstellungsformen Informationen anzeigen. Tabelle und Diagramm haben keine Kenntnis voneinander und ermöglichen auf diese Weise, nur das benötigte Objekt wiederzuverwenden. Dennoch *verhalten* sich beide Objekte, als ob sie voneinander wüssten. Wenn der User den Inhalt der Tabelle ändert, zeigt das Diagramm die Änderungen sofort an. In umgekehrter Richtung verhält es sich genauso:

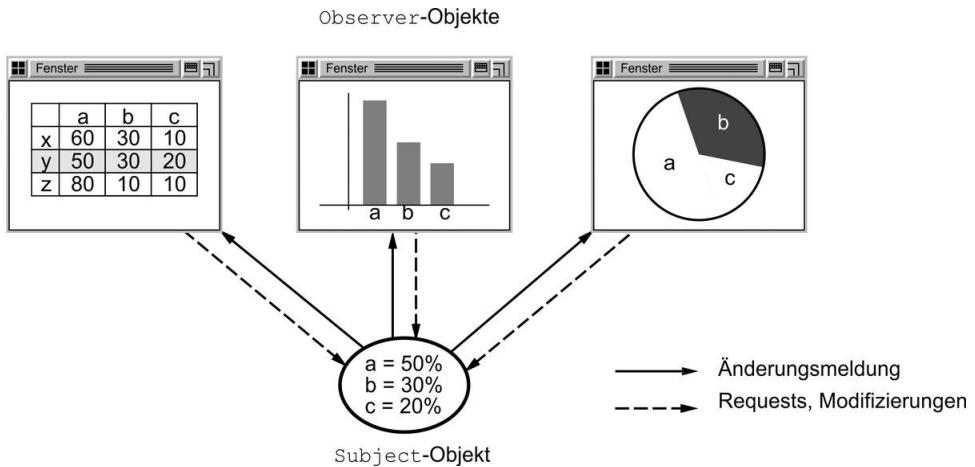


Abb. 5.29: Observer-Objekte einer Tabellenkalkulation

Dieses Verhalten lässt darauf schließen, dass Tabelle und Diagramm vom Datenobjekt abhängig sind und daher im Fall einer Zustandsänderung benachrichtigt werden müssen. Es gibt keinen Grund, die Anzahl abhängiger Objekte auf zwei zu beschränken – es könnte eine beliebige Anzahl verschiedener Benutzerschnittstellen für dieselben Daten geben.

Das Design Pattern *Observer (Beobachter)* beschreibt, wie derartige Beziehungen eingerichtet werden. Bei diesem Pattern sind das **Subjekt** und der **Observer** die entscheidenden Objekte. Ein Subjekt kann beliebig viele abhängige Observer besitzen. Bei jeder Zustandsänderung des Subjekts werden alle Observer benachrichtigt. Als Reaktion darauf wird jeder Observer das Subjekt kontaktieren, um seinen eigenen Zustand mit demjenigen des Subjekts zu synchronisieren.

Dieses Zusammenspiel ist auch unter der Bezeichnung **Publish-Subscribe** (dt. »Veröffentlichen-Abonnieren«) bekannt. Das Subjekt ist der Herausgeber der Benachrichtigungen. Es versendet diese Benachrichtigungen, ohne seine Observer überhaupt zu kennen. Die Benachrichtigungen können von einer beliebigen Anzahl von Observern abonniert werden.

Anwendbarkeit

Die Verwendung des Design Patterns *Observer (Beobachter)* empfiehlt sich in folgenden Situationen:

- Wenn eine Abstraktion zwei Aspekte besitzt und einer davon vom anderen abhängig ist. Die Kapselung dieser Aspekte in verschiedenen Objekten gestattet

ihre voneinander unabhängige Änderung und Wiederverwendung.

- Wenn eine Modifikation an einem der Objekte das Ändern anderer Objekte erfordert und nicht bekannt ist, wie viele der anderen Objekte geändert werden müssen.
- Wenn ein Objekt andere Objekte benachrichtigen können soll, ohne Annahmen darüber zu machen, um welche Objekte es sich handelt. Mit anderen Worten: Die Objekte sollen nicht eng gekoppelt sein.

Struktur

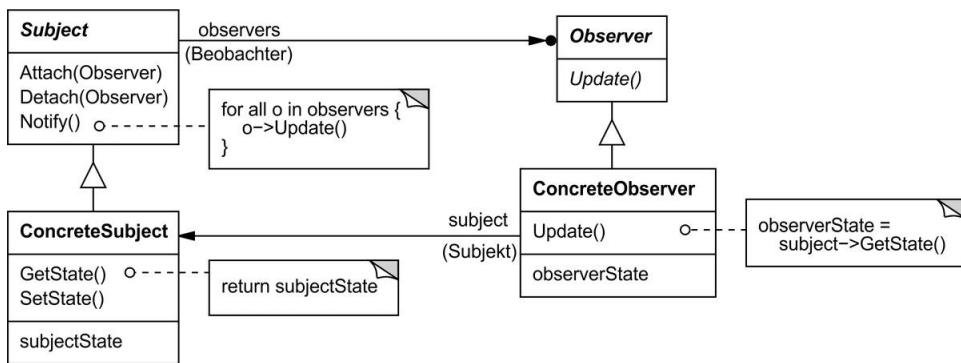


Abb. 5.30: Struktur des Design Patterns Observer (Beobachter)

Teilnehmer

- **Subject**
 - Kennt seine Observer. Ein Subjekt kann von einer beliebigen Anzahl von Observern beobachtet werden.
 - Stellt eine Schnittstelle zum Anbinden und Lösen von Observer-Objekten zur Verfügung.
- **Observer**
 - Definiert eine Aktualisierungsschnittstelle für Objekte, die bei Zustandsänderungen eines Subjekts benachrichtigt werden sollen.
- **ConcreteSubject**

- Speichert den interessierenden Zustand in `ConcreteObserver`-Objekten.
 - Sendet bei Zustandsänderungen eine Benachrichtigung an seine Observer.
- **ConcreteObserver**
 - Verwahrt eine Referenz auf ein `ConcreteSubject`-Objekt.
 - Speichert den Zustand, der mit demjenigen des Subjekts in Übereinstimmung bleiben soll.
 - Implementiert die Aktualisierungsschnittstelle für Observer, damit der eigene Zustand mit demjenigen von Subject übereinstimmt.

Interaktionen

- `ConcreteSubject` benachrichtigt seine Observer, sobald eine Zustandsänderung eintritt, die dazu führen könnte, dass die Zustände der Observer-Objekte nicht mehr mit dem eigenen Zustand übereinstimmen.
- Wenn ein `ConcreteObserver` von einer Änderung in `ConcreteSubject` unterrichtet wird, fragt er möglicherweise Informationen beim Subject ab, die er dazu verwendet, seinen eigenen Zustand mit demjenigen von Subject abzustimmen.

Das nachstehende Diagramm zeigt die Zusammenarbeit zwischen einem Subjekt und zwei Observern:

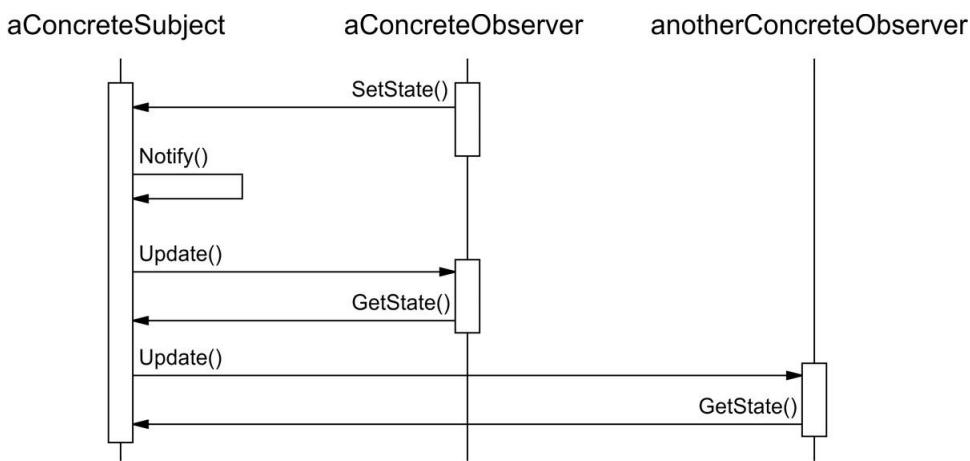


Abb. 5.31: Zusammenspiel von Subjekt und Observern

Beachtenswert ist hier, wie das **Observer**-Objekt, das die Änderungsanfrage auslöst, seine Aktualisierung aufschiebt, bis es vom Subjekt eine Benachrichtigung erhält. **Notify** wird nicht immer nur vom Subjekt aufgerufen, sondern kann auch von einem **Observer**- oder einem völlig anderen Objekt aufgerufen werden. Im Abschnitt »Implementierung« kommen noch einige andere verbreitete Varianten zur Sprache.

Konsequenzen

Das Design Pattern **Observer (Beobachter)** ermöglicht es, Subjekte und Observer unabhängig voneinander zu ändern. Subjekte lassen sich wiederverwenden, ohne auch die Observer wiederzuverwenden und umgekehrt. Es gestattet außerdem das Hinzufügen von Observern, ohne Änderungen am Subjekt oder anderen Observern vornehmen zu müssen.

Zu den weiteren Vorteilen und Verantwortlichkeiten des Patterns **Observer (Beobachter)** gehören die folgenden:

1. *Abstrakte Kopplung von Subjekt und Observer.* Dem Subjekt ist nur bekannt, dass es eine Reihe von Observern gibt, die sich nach der einfachen Schnittstelle der abstrakten **Observer**-Klasse richten. Das Subjekt hat keine Kenntnis von der konkreten Klasse irgendeines Observers. Die Kopplung zwischen Subjekten und Observern ist abstrakt und auf das Nötigste beschränkt.

Da **Subject** und **Observer** nicht eng gekoppelt sind, können sie verschiedenen Abstrahierungsschichten eines Systems angehören. Ein auf einer niedrigeren Schicht angesiedeltes Subjekt kann mit einem auf einer höheren Schicht befindlichen Observer kommunizieren und ihn benachrichtigen. Die unterschiedlichen Schichten des Systems bleiben dabei unangetastet. Wenn Subjekt und Observer in einen Topf geworfen werden, muss sich das resultierende Objekt entweder über zwei Schichten erstrecken (und die Regeln der Schichtenbildung verletzen) oder es muss sich gezwungenermaßen in einer der beiden Schichten befinden (wodurch möglicherweise die Abstrahierungsschichten gestört werden).

2. *Unterstützung von Broadcast-Kommunikation.* Anders als bei herkömmlichen Requests muss das Subjekt beim Versenden einer Benachrichtigung keinen Empfänger angeben. Die Benachrichtigung wird automatisch allen interessierten Objekten zugestellt, die sie abonniert haben. Dem Subjekt ist es vollkommen egal, wie viele Objekte an der Benachrichtigung interessiert sind.

Es ist nur dafür zuständig, seine Observer zu benachrichtigen. Das ermöglicht das jederzeitige Hinzufügen oder Entfernen von Observer-Objekten. Dabei bleibt es dem Observer überlassen, auf eine Benachrichtigung zu reagieren oder sie zu ignorieren.

3. *Unerwartete Aktualisierungen.* Da Observer keine Kenntnis von der Existenz anderer Observer besitzen, können sie dem letztendlich erforderlichen Aufwand einer Änderung gegenüber blind sein. Eine scheinbar harmlose Operation des Subjekts kann eine Lawine von Aktualisierungen bei Observern und ihren abhängigen Objekten auslösen. Darüber hinaus können Abhängigkeitskriterien, die nicht wohldefiniert sind oder nicht ordentlich verwaltet werden, zu zweifelhaften Aktualisierungen führen, deren Ursache schwer zu finden ist.

Dieses Problem wird durch die Tatsache, dass das schlichte Aktualisierungsprotokoll keine Hinweise darauf enthält, was sich beim Subjekt geändert hat, weiter verschärft. Ohne ein umfassenderes Protokoll, das den Observern dabei hilft herauszufinden, was sich geändert hat, fällt ihnen womöglich die schwierige Aufgabe zu, die Änderungen selbst zu ermitteln.

Implementierung

In diesem Abschnitt kommen verschiedene Aspekte zur Sprache, die bei der Implementierung von Abhängigkeiten eine Rolle spielen:

1. *Zuordnung von Subjekten zu ihren Observern.* Die einfachste Möglichkeit, den Überblick über die zu benachrichtigenden Observer zu behalten, besteht darin, Referenzen auf die Observer explizit im Subjekt zu speichern. Falls es viele Subjekte und nur einige wenige Observer gibt, kann eine solche Speicherung allerdings zu aufwendig sein. Eine Lösung wäre es, Speicherbedarf gegen Rechenzeit einzutauschen und einen assoziativen Lookup-Mechanismus (z. B. eine Hashtabelle) zu verwenden, um die Zuordnung von Subjekten zu Observern zu verwalten. Ein Subjekt ohne Observer hat in diesem Fall keinen zusätzlichen Speicherbedarf. Allerdings wird bei diesem Ansatz der Zugriff auf Observer aufwendiger.
2. *Observieren mehrerer Subjekte.* In manchen Situationen kann es sinnvoll sein, dass ein Observer von mehr als einem Subjekt abhängt. Eine Tabelle kann beispielsweise von mehreren Datenquellen abhängig sein. In solchen Fällen muss die Schnittstelle zur Aktualisierung erweitert werden, damit der Observer

davon Kenntnis erhält, welches Subjekt ihm eine Benachrichtigung übermittelt. Zu diesem Zweck kann sich das Subjekt bei der Aktualisierungsoperation selbst als Parameter übergeben und den Observer auf diese Weise wissen lassen, welches Subjekt er überprüfen soll.

3. *Wer löst die Aktualisierung aus?* Das Subjekt und seine Observer vertrauen darauf, dass der Mechanismus zum Übermitteln von Benachrichtigungen verlässlich funktioniert. Aber welches Objekt ruft eigentlich `Notify` auf, um die Aktualisierung auszulösen? Hierfür gibt es zwei Möglichkeiten:
 - a. Diejenigen Operationen, die Zustandsänderungen bewirken, rufen `Notify` auf, nachdem sie eine Änderung am Zustand des Subjekts vorgenommen haben. Der Vorteil dieses Ansatzes ist, dass die Clients nicht daran denken müssen, die `Notify`-Methode des Subjekts aufzurufen. Der Nachteil besteht darin, dass mehrere aufeinanderfolgende Operationen auch eine Reihe aufeinanderfolgender Aktualisierungen verursachen – und das könnte sich als ineffizient erweisen.
 - b. Die Clients sind dafür verantwortlich, `Notify` zum richtigen Zeitpunkt aufzurufen. Das hat den Vorteil, dass der Client mit dem Auslösen der Aktualisierung so lange warten kann, bis mehrere aufeinanderfolgende Zustandsänderungen abgeschlossen sind, und damit überflüssige zwischenzeitliche Aktualisierungen vermeidet. Der Nachteil besteht darin, dass die Clients zusätzlich die Verantwortung für das Auslösen der Aktualisierung übernehmen müssen. Dadurch erhöht sich die Wahrscheinlichkeit, dass Fehler auftreten, weil die Clients den Aufruf von `Notify` vergessen könnten.
4. *Verwaiste Referenzen auf gelöschte Subjekte.* Das Löschen von Subjekten sollte nicht zu verwaisten Referenzen (engl. *dangling references*) in den Observern führen. Dies lässt sich zum Beispiel erreichen, indem sichergestellt wird, dass das Subjekt die betreffenden Observer von seiner Löschung benachrichtigt, damit sie ihre Referenzen zurücksetzen. Die Observer einfach zu löschen, ist im Allgemeinen keine Lösung, weil möglicherweise auch andere Objekte auf sie verweisen oder die Observer ihrerseits andere Subjekte observieren.
5. *Vor der Benachrichtigung sicherstellen, dass der Zustand des Subjekts in sich widerspruchsfrei ist.* Es ist wichtig, dafür Sorge zu tragen, dass der Zustand des Subjekts vor dem Aufruf von `Notify` in sich widerspruchsfrei ist, weil die Observer während der Aktualisierung ihres eigenen Zustands den aktuellen Zustand des Subjekts abfragen.

Die interne Widerspruchsfreiheit kann leicht unabsichtlich verletzt werden, wenn Operationen in Unterklassen des Subjekts ererbte Operationen aufrufen. Beispielsweise wird die Benachrichtigung im folgenden Codeabschnitt ausgelöst, während der Zustand des Subjekts in sich widersprüchlich ist:

```
void MySubject::Operation (int newValue) {
    BaseClassSubject::Operation(newValue);
    // Benachrichtigung auslösen

    _myInstVar +=newValue;
    // Zustand der UnterkLASSE aktualisieren
    // (zu spät!)
}
```

Dieser Fallstrick lässt sich umgehen, indem die Benachrichtigungen aus einer *Template Method* (*Schablonenmethode*, siehe [Abschnitt 5.10](#)) in einer abstrakten Subject-Klasse heraus versandt werden. Hierzu reicht es aus, eine primitive Operation zu definieren, die von Unterklassen überschrieben werden kann, und dafür zu sorgen, dass Notify die letzte Operation in der Template Method ist. Dadurch wird sichergestellt, dass das Objekt in sich widerspruchsfrei ist, wenn Unterklassen Subject-Operationen überschreiben:

```
void Text::Cut (TextRange r) {
    ReplaceRange(r); // Wird in Unterklassen
                      // neu definiert
    Notify();
}
```

Es ist übrigens immer empfehlenswert zu dokumentieren, welche Subject-Operationen Benachrichtigungen auslösen.

6. *Vermeidung Observer-eigener Aktualisierungsprotokolle: die Push- und Pull-Modelle.* Bei Implementierungen des Design Patterns *Observer (Beobachter)* übermittelt das Subjekt oftmals zusätzliche Informationen über die Zustandsänderung, die es als Argument an *update* übergibt. Der Umfang dieser Informationen kann beträchtlich variieren.

Bei dem als **Push-Modell** bezeichneten Extremfall übermittelt das Subjekt seinen Observern ausführliche Informationen über die Zustandsänderung – ungeachtet dessen, ob sie sie überhaupt haben möchten oder nicht. Der andere Extremfall ist das **Pull-Modell**. Hier übermittelt das Subjekt nur die allernötigsten Informationen. Danach erkundigen sich die Observer ausdrücklich nach weiteren Details.

Das Pull-Modell unterstreicht die Unkenntnis des Subjekts von seinen Observern. Das Push-Modell hingegen geht davon aus, dass Subjekte gewisse Kenntnisse über den Informationsbedarf ihrer Observer besitzen. Im Push-Modell sind Observer möglicherweise weniger gut wiederverwendbar, weil Subject-Klassen Annahmen über Observer-Klassen machen, die eventuell nicht immer korrekt sind. Andererseits ist das Pull-Modell womöglich ineffizient, weil die Observer-Klassen beim Feststellen der Änderungen ohne Hilfe der Subject-Klasse auskommen müssen.

7. *Ausdrückliche Festlegung der Änderungen von Interesse.* Die Effizienz von Aktualisierungen lässt sich steigern, indem die Schnittstelle des Subjekts den Observern gestattet, sich nur für bestimmte Ereignisse von besonderem Interesse zu registrieren. Tritt ein solches Ereignis ein, benachrichtigt das Subjekt nur diejenigen Observer, die ihr Interesse daran bekundet haben. Einer Möglichkeit, dies zu unterstützen, liegt der Begriff des **Aspekts** eines Subject-Objekts zugrunde. Zur Bekundung ihres Interesses an bestimmten Ereignissen können sich Observer mittels

```
void Subject::Attach(Observer*, Aspect& interest);
```

bei ihren Subjekten registrieren. `interest` gibt dabei das interessierende Ereignis an. Zum Zeitpunkt der Benachrichtigung übermittelt das Subjekt den fraglichen Aspekt seinen Observern in Form eines Parameters der `Update`-Operation. Zum Beispiel so:

```
void Observer::Update(Subject*, Aspect& interest);
```

8. *Kapselung komplexer Aktualisierungsvorschriften.* Wenn die Abhängigkeiten zwischen Subjekten und Observern besonders kompliziert sind, wird möglicherweise ein Objekt benötigt, das diese Beziehungen verwahrt. Ein solches Objekt wird als **ChangeManager** (Änderungsmanager) bezeichnet. Seine Aufgabe ist es, den Arbeitsaufwand zu minimieren, der nötig ist, damit Observer eine Änderung ihres Subjekts widerspiegeln. Wenn an einer Operation beispielsweise mehrere gegenseitig voneinander abhängige Subjekte beteiligt sind, muss gegebenenfalls sichergestellt werden, dass die zugehörigen Observer erst dann benachrichtigt werden, wenn *alle* Subjekte geändert worden sind, um zu vermeiden, dass Observer mehrmals benachrichtigt werden.

Der Änderungsmanager ist für drei Dinge verantwortlich:

- a. Er ordnet Subjekten ihre Observer zu und stellt eine Schnittstelle zur Aktualisierung dieser Zuordnung bereit. Dadurch entfällt die

Notwendigkeit, dass Subjekte ihre Referenzen auf Observer verwahren müssen (und umgekehrt).

- b. Er legt eine bestimmte Aktualisierungsstrategie fest.
- c. Auf Anfrage eines Subjekts aktualisiert er alle abhängigen Observer.

Das nachstehende Diagramm zeigt eine einfache, auf dem Änderungsmanager beruhende Implementierung des Design Patterns *Observer* (*Beobachter*). Hier finden sich zwei spezialisierte Änderungsmanager: SimpleChangeManager ist insofern naiv, als er stets sämtliche Observer eines jeden Subjekts aktualisiert. Im Gegensatz dazu handhabt DAGChangeManager gerichtete azyklische Graphen von Abhängigkeiten zwischen Subjekten und ihren Observern. Ein DAGChangeManager ist einem SimpleChangeManager vorzuziehen, wenn ein Observer mehr als ein Subjekt observiert. In diesem Fall könnte eine Änderung in zwei oder mehr Subjekten zu überflüssigen Aktualisierungen führen. Der DAGChangeManager stellt hingegen sicher, dass der Observer nur ein Mal aktualisiert wird. Wenn mehrfache Aktualisierungen nicht zur Debatte stehen, ist SimpleChangeManager jedoch völlig ausreichend.

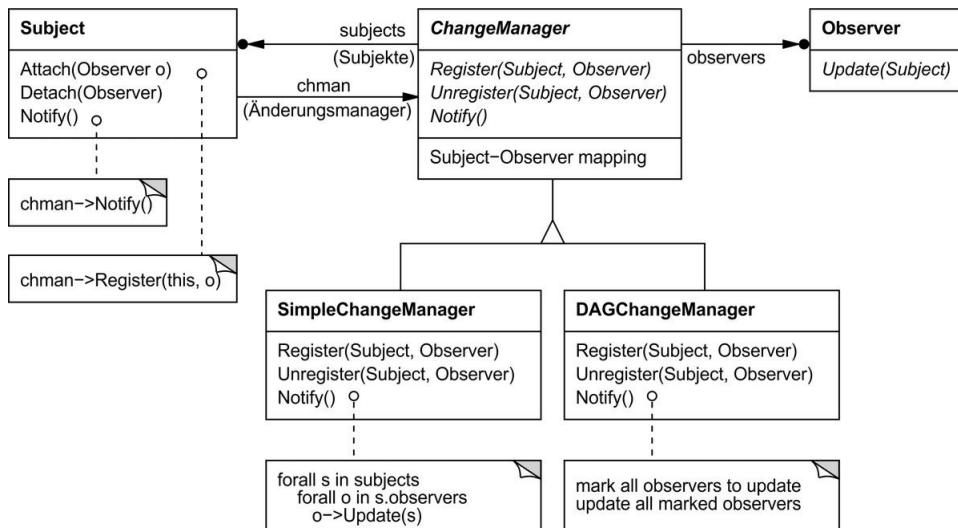


Abb. 5.32: Auf dem Änderungsmanager beruhende Implementierung des Design Patterns *Observer* (*Beobachter*)

ChangeManager ist eine Instanz des Design Patterns *Mediator* (*Vermittler*, siehe [Abschnitt 5.5](#)). Im Allgemeinen gibt es nur einen Änderungsmanager, der global zugänglich ist. Hier wäre auch das Design Pattern *Singleton* (*Singleton*, siehe [Abschnitt 3.5](#)) sinnvoll.

9. Kombination von Subject- und Observer-Klasse. In Programmiersprachen

ohne Mehrfachvererbung (wie Smalltalk) erstellte Klassenbibliotheken definieren im Allgemeinen keine getrennten Subject- und Observer-Klassen, sondern führen die Schnittstellen in einer einzelnen Klasse zusammen. Dadurch ist es möglich, auch ohne Mehrfachvererbung ein Objekt einzurichten, das sowohl als Subjekt als auch als Observer fungiert. In Smalltalk sind beispielsweise die Subject- und Observer-Schnittstellen in der Wurzelklasse Object definiert und damit für alle anderen Klassen verfügbar.

Beispielcode

Die Observer-Schnittstelle ist in einer abstrakten Klasse definiert:

```
class Subject;

class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;

protected:
    Observer();
};
```

Diese Implementierung unterstützt mehrere Subjekte pro Observer. Das der Update-Operation übergebene Subjekt erlaubt es dem Observer festzustellen, welches der Subjekte sich geändert hat, sofern er mehr als eins observiert.

Die Subject-Schnittstelle ist auf ähnliche Weise in einer abstrakten Klasse definiert:

```
class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();

protected:
    Subject();

private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
```

```

}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}

```

`ClockTimer` ist ein konkretes Subjekt zum Speichern und Bewahren der Uhrzeit. Es benachrichtigt seine Observer sekündlich. `ClockTimer` stellt eine Schnittstelle zum Abruf verschiedener Zeiteinheiten (wie etwa Stunde, Minute und Sekunde) bereit:

```

class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick();
};

```

Die `Tick`-Operation wird in regelmäßigen Zeitintervallen von einem internen Timer aufgerufen und stellt damit einen präzisen Taktgeber zur Verfügung. `Tick` aktualisiert den internen `ClockTimer`-Zustand und ruft `Notify` auf, um die Observer von der Zustandsänderung zu benachrichtigen:

```

void ClockTimer::Tick () {
    // Zustand der internen Zeitnahme aktualisieren
    // ...
    Notify();
}

```

Nun kann eine Klasse namens `DigitalClock` definiert werden, die die Uhrzeit anzeigt. Die grafische Funktionalität erbt sie von einer `Widget`-Klasse, die das Toolkit für die Benutzerschnittstelle bereitstellt. Die `Observer`-Schnittstelle wird der `DigitalClock`-Schnittstelle hinzugefügt, indem sie sie von `Observer` erbt:

```

class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
}

```

```

    virtual void Update(Subject*);
        // Überschreibt Observer-Operation

    virtual void Draw();
        // Überschreibt Widget-Operation;
        // Legt das Zeichnen der Digitaluhr fest

private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~DigitalClock () {
    _subject->Detach(this);
}

```

Bevor die `Update`-Operation die Uhrzeitanzeige zeichnet, überprüft sie, ob es sich beim benachrichtigenden Subjekt um das Subjekt der Uhr handelt:

```

void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw () {
    // Neue Werte vom Subjekt abrufen

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.

    // Uhrzeitanzeige zeichnen
}

```

Auf die gleiche Weise kann auch eine `AnalogClock`-Klasse definiert werden:

```

class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};

```

Der nachstehende Code erzeugt eine `AnalogClock` und eine `DigitalClock`, die stets die gleiche Zeit anzeigen:

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

Jedes Mal, wenn `timer` »tickt«, werden die beiden Uhren aktualisiert und zeichnen sich dementsprechend neu.

Praxisbeispiele

Das erste und wohl auch bekannteste Beispiel für das Design Pattern *Observer (Beobachter)* taucht im **Smalltalk Model/View/Controller (MVC)** auf, dem Framework der Benutzerschnittelle in der Smalltalk-Umgebung [KP88]. Die MVC-Klasse `Model` übernimmt die `Subject`-Rolle, während `view` die Basisklasse für `Observer` ist. **Smalltalk, ET++** [WGM88] und die **THINK-Klassenbibliothek** [Sym93b] stellen einen allgemeinen Mechanismus zur Handhabung von Abhängigkeiten bereit, indem die Basisklasse aller anderen Klassen in diesen Systemen um die `Subject`- und `Observer`-Schnittstellen ergänzt wird.

Verwandte Patterns

Mediator (Vermittler, siehe [Abschnitt 5.5](#)): Durch die Kapselung komplexer Aktualisierungsvorschriften fungiert der Änderungsmanager als Vermittler zwischen Subjekten und Observern.

Singleton (Singleton, siehe [Abschnitt 3.5](#)): Der Änderungsmanager kann das Design Pattern *Singleton (Singleton)* verwenden, um seine Einzigartigkeit sicherzustellen und um global zugänglich zu sein.

5.8 State (Zustand)

Objektbasiertes Verhaltensmuster

Zweck

Anpassung der Verhaltensweise eines Objekts im Fall einer internen Zustandsänderung, so dass es den Anschein hat, als hätte es seine Klasse gewechselt.

Auch bekannt als

Object for States

Motivation

In diesem Abschnitt wird eine `TCPConnection`-Klasse als Beispiel verwendet, die eine Netzwerkverbindung repräsentiert. Ein `TCPConnection`-Objekt kann sich in einem von mehreren möglichen Zuständen befinden: `Established` (verbunden), `Listening` (horchend) oder `Closed` (getrennt). Wenn ein `TCPConnection`-Objekt einen Request von anderen Objekten empfängt, antwortet es entsprechend seines aktuellen Zustands unterschiedlich. Beispielsweise hängt das Resultat eines `Open`-Requests davon ab, ob die Verbindung sich gerade im `Closed`- oder im `Established`-Zustand befindet. Das Design Pattern *State (Zustand)* beschreibt, wie `TCPConnection` in seinen verschiedenen Zuständen unterschiedliches Verhalten aufweisen kann.

Der entscheidende Gedanke bei diesem Pattern ist die Einführung einer abstrakten Klasse namens `TCPState`, die die Zustände der Netzwerkverbindung repräsentiert. Die `TCPState`-Klasse deklariert eine Schnittstelle, die allen Klassen gemeinsam ist, die einen der verschiedenen Betriebszustände darstellen. Das für einen bestimmten Zustand typische Verhalten wird durch `TCPState`-Unterklassen implementiert. So implementieren die Klassen `TCPEstablished` und `TCPClosed` beispielsweise die den `Established`- und `Closed`-Zuständen einer `TCPConnection` eigenen Verhaltensweisen:

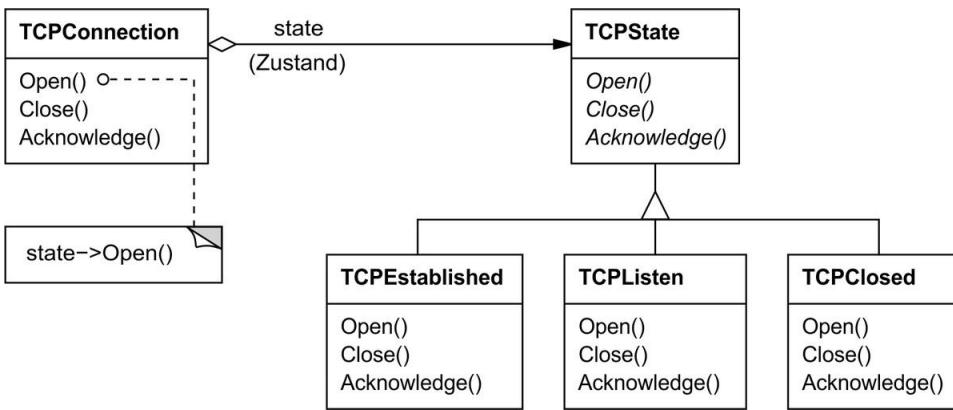


Abb. 5.33: Zustände einer TCP-Netzwerkverbindung

Die **TCPConnection**-Klasse verwahrt ein Zustandsobjekt (eine **TCPState**-Unterklasseninstanz), die den gegenwärtigen Zustand der TCP-Verbindung repräsentiert. Die **TCPConnection**-Klasse delegiert alle zustandsspezifischen Requests an dieses Zustandsobjekt. Operationen, die einem bestimmten Zustand der Verbindung eigen sind, führt die **TCPConnection**-Klasse mittels ihrer **TCPState**-Unterklasseninstanz aus.

Sobald sich der Verbindungszustand ändert, ändert das **TCPConnection**-Objekt auch das von ihm verwendete Zustandsobjekt. Wenn eine **Established**-Verbindung beispielsweise den Zustand **Closed** annimmt, ersetzt **TCPConnection** die **TCPEstablished**-Instanz durch die **TCPClosed**-Instanz.

Anwendbarkeit

Die Verwendung des Design Patterns *State (Zustand)* eignet sich in folgenden Fällen:

- Das Verhalten eines Objekts hängt von seinem Zustand ab und es muss seine Verhaltensweise zur Laufzeit in Abhängigkeit von diesem Zustand ändern.
- Es gibt Operationen, in denen umfangreiche, mehrteilige bedingte Anweisungen auftreten, die vom Zustand des Objekts abhängen. Dieser Zustand wird für gewöhnlich durch eine oder mehrere benannte Konstanten repräsentiert. Oftmals verwenden mehrere Operationen dieselben bedingten Anweisungen. Das Design Pattern *State (Zustand)* verteilt die verschiedenen Zweige der bedingten Anweisungen jeweils auf eigene Klassen. Dadurch kann der Zustand des Objekts wie ein eigenständiges Objekt behandelt werden, das sich unabhängig von anderen Objekten ändert.

Struktur

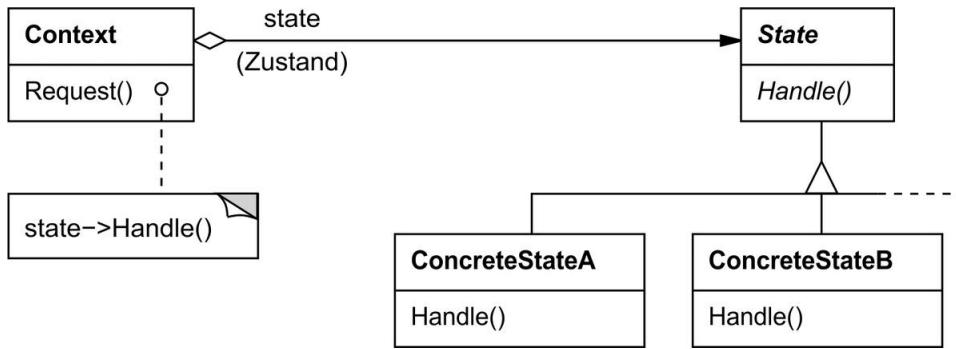


Abb. 5.34: Struktur des Design Pattern State (Zustand)

Teilnehmer

- **Context** (TCPConnection)
 - Definiert die Schnittstelle, die für Clients von Interesse ist.
 - Verwahrt eine Instanz einer **ConcreteState**-Unterklasse, die den aktuellen Zustand beschreibt.
- **State** (TCPState)
 - Definiert eine Schnittstelle zur Kapselung der Verhaltensweise, die zu einem bestimmten **Context**-Zustand gehört.
- **ConcreteState**-Unterklassen (TCPEstablished, TCPListen, TCPClosed)
 - Jede Unterklasse implementiert eine Verhaltensweise, die zu einem bestimmten **Context**-Zustand gehört.

Interaktionen

- **Context** delegiert zustandsspezifische Requests an das aktuelle **ConcreteState**-Objekt.
- Ein **Context** kann sich selbst als Argument an das **State**-Objekt übergeben, das den Request bearbeitet. Auf diese Weise kann das **State**-Objekt, falls

erforderlich, auf `Context` zugreifen.

- `Context` ist die wichtigste Schnittstelle für Clients. Die Clients können `Context` mittels eines `State`-Objekts konfigurieren. Sobald das geschehen ist, müssen die Clients nicht mehr direkt mit dem `State`-Objekt interagieren.
- Welcher der Zustände unter welchen Umständen die Oberhand gewinnt, entscheiden entweder `Context` oder die `ConcreteState`-Unterklassen.

Konsequenzen

Das Design Pattern *State (Zustand)* bringt die folgenden Konsequenzen mit sich:

1. *Es verortet zustandsspezifische Verhaltensweisen und verteilt sie auf die verschiedenen Zustände.* Das Pattern *State (Zustand)* packt alle zu einem bestimmten Zustand zugehörigen Verhaltensweisen in ein Objekt. Da sich sämtlicher zustandsspezifischer Code in einer `State`-Unterklasse befindet, können durch die Definition neuer Unterklassen leicht neue Zustände und Zustandsänderungen hinzugefügt werden.

Alternativ könnten Werte zur Kennzeichnung interner Zustände definiert werden, die `Context`-Operationen explizit überprüfen. Dann wären allerdings einander sehr ähnliche bedingte Anweisungen (oder Case-Anweisungen) über die gesamte `Context`-Implementierung verstreut. Das Hinzufügen eines neuen Zustands könnte es erforderlich machen, mehrere Operationen zu ändern, was wiederum die Wartung verkompliziert.

Das Design Pattern *State (Zustand)* umgeht diese Schwierigkeit, bringt aber möglicherweise eine andere mit sich, weil es die Verhaltensweisen verschiedener Zustände auf mehrere `State`-Unterklassen verteilt. Das erhöht die Anzahl der Klassen und ist weniger kompakt als eine einzelne Klasse. Eine solche Aufteilung ist jedoch von Vorteil, falls es sehr viele Zustände gibt, da andernfalls umfangreiche bedingte Anweisungen erforderlich wären.

Ebenso wie sehr lange Prozeduren sind auch lange bedingte Anweisungen unerwünscht. Sie sind unhandlich und führen oft dazu, dass der Code weniger deutlich erscheint, was es wiederum erschwert, sie zu modifizieren und zu erweitern. Das Pattern *State (Zustand)* bietet eine bessere Möglichkeit, den zustandsspezifischen Code zu strukturieren: Die Programmlogik, die die Zustandsänderungen festlegt, wird nicht in unflexiblen `if`- oder `switch`-

Anweisungen verpackt, sondern auf die State-Unterklassen verteilt. Die Kapselung der einzelnen Zustandsänderungen und Aktionen in einer Klasse macht aus dem Konzept eines Betriebszustands ein vollwertiges Objekt. Der Code ist dadurch zwangsläufig besser strukturiert und sein Zweck wird deutlicher.

2. *Zustandsänderungen werden deutlich hervorgehoben.* Wenn ein Objekt seinen aktuellen Zustand ausschließlich mittels intern festgelegter Werte beschreibt, gibt es keine ausdrückliche Repräsentierung der Zustandsänderungen. Sie erscheinen lediglich als Zuweisungen zu irgendwelchen Variablen. Die Einführung eigener Objekte für verschiedene Zustände hebt Zustandsänderungen deutlich hervor.

State-Objekte können außerdem den Context vor widersprüchlichen internen Zuständen abschirmen, weil Zustandsänderungen aus der Context-Perspektive atomar sind – es wird nur eine Variable neu zugewiesen (die Variable für das State-Objekt in Context), nicht mehrere.

3. *State-Objekte können gemeinsam genutzt werden.* Falls State-Objekte keine Instanzvariablen besitzen – d. h., der repräsentierte Zustand ist allein durch ihren Typ vorgegeben – können Context-Objekte ein State-Objekt gemeinsam nutzen. Auf diese Weise gemeinsam genutzte State-Objekte sind im Wesentlichen *Flyweights* (*Fliegengewicht*, siehe [Abschnitt 4.6](#)), die keinen inneren Zustand besitzen, sondern nur ein bestimmtes Verhalten zeigen.

Implementierung

Bei der Implementierung des Design Patterns *State (Zustand)* sind verschiedene Aspekte zu beachten:

1. *Wer legt Zustandsänderungen fest?* Das Pattern *State (Zustand)* gibt nicht vor, welcher der Teilnehmer die Kriterien für eine Zustandsänderung definiert. Sind die Kriterien fest vorgegeben, können sie vollständig im Context-Objekt implementiert werden. Es ist im Allgemeinen jedoch flexibler und zweckmäßiger, wenn die State-Unterklassen ihren nachfolgenden Zustand und den Zeitpunkt der Zustandsänderung selbst bestimmen können. Dazu ist es erforderlich, eine Context-Schnittstelle hinzuzufügen, die es den State-Objekten ausdrücklich erlaubt, den aktuellen Context-Zustand zu ändern.

Die Vorschriften für Zustandsänderungen in dieser Art zu dezentralisieren,

erleichtert es, sie zu modifizieren oder zu erweitern, indem neue State-Unterklassen definiert werden. Die Dezentralisierung bringt jedoch den Nachteil mit sich, dass eine State-Unterklasse Kenntnis von mindestens einer anderen haben muss, wodurch es bei solch einer Implementierung zu Abhängigkeiten zwischen den Unterklassen kommt.

2. *Eine tabellenbasierte Alternative.* In »C++ Programming Style« [Car92] beschreibt Cargill eine andere Möglichkeit, zustandsgetriebenem Code eine bessere Struktur aufzuerlegen: Er verwendet Tabellen für die Zuordnung von Eingaben zu Zustandsänderungen. Für jeden Zustand ist in einer Tabelle zu allen möglichen Eingaben der jeweils nachfolgende Zustand verzeichnet. Letztendlich werden bei diesem Ansatz die herkömmlichen bedingten Anweisungen (und im Fall des Design Patterns *State (Zustand)* virtuelle Funktionen) in das Durchsuchen einer Tabelle überführt.

Der Hauptvorteil von Tabellen ist ihre Regelhaftigkeit: Die Kriterien für Zustandsänderungen können durch das Modifizieren von Daten statt durch Umschreiben des Programmcodes geändert werden. Es gibt allerdings auch einige Nachteile:

- Das Durchsuchen einer Tabelle ist häufig weniger effizient als der Aufruf einer (virtuellen) Funktion.
- Die Vorschriften für Zustandsänderungen in einem einheitlichen, für Tabellen geeigneten Format auszudrücken, lässt die Kriterien für Zustandsänderungen weniger klar erscheinen und macht sie daher schwerer verständlich.
- Es ist für gewöhnlich schwierig, die Zustandsänderung begleitende Aktionen hinzuzufügen. Der tabellenbasierte Ansatz erfasst zwar Zustände und deren Änderungen, muss aber noch weiter ausgebaut werden, um bei jeder Zustandsänderung beliebige Berechnungen ausführen zu können.

Der wesentliche Unterschied zwischen dem tabellenbasierten Ansatz und dem Design Pattern *State (Zustand)* kann wie folgt zusammengefasst werden: Das Pattern *State (Zustand)* bildet zustandsspezifisches Verhalten ab, der tabellenbasierte Ansatz hingegen konzentriert sich darauf, Zustandsänderungen zu beschreiben.

3. *Erzeugen und Löschen von State-Objekten.* Bei der Implementierung ist es oftmals eine Überlegung wert, abzuwägen, ob man 1. State-Objekte nur bei

Bedarf erzeugt und nach Gebrauch wieder löscht oder 2. sie vorab erzeugt und niemals löscht.

Der ersten Variante ist der Vorzug zu geben, wenn die zur Laufzeit auftretenden Zustände unbekannt sind *und* Context-Objekte den Zustand nur selten ändern. Dieser Ansatz vermeidet es, nicht benötigte Objekte zu erzeugen, was von Bedeutung ist, wenn die State-Objekte große Informationsmengen speichern. Die zweite Variante ist besser geeignet, wenn Zustandsänderungen in schneller Folge auftreten. In diesem Fall ist es angebracht, das Löschen von State-Objekten zu vermeiden, weil sie schon bald wieder benötigt werden könnten. Der für die Instanziierung erforderliche Aufwand fällt vorab an, und einen Aufwand beim Löschen gibt es nicht. Dennoch kann dieser Ansatz auch unbequem sein, denn die Context-Objekte müssen Referenzen auf alle möglicherweise auftretenden Zustände speichern.

4. *Einsatz dynamischer Vererbung.* Ein verändertes Verhalten bei einem bestimmten Request könnte durch eine Änderung der Klasse eines Objekts zur Laufzeit erzielt werden – in den meisten objektorientierten Programmiersprachen ist dies jedoch nicht möglich. Zu den Ausnahmen gehören Self [US87] und weitere auf der Delegation beruhende Sprachen, die einen solchen Mechanismus bieten und daher das Design Pattern *State (Zustand)* unmittelbar unterstützen. In Self können Objekte Operationen an andere Objekte delegieren und auf diese Weise eine Form der dynamischen Vererbung bewerkstelligen. Durch die Änderung des Ziels einer Delegation zur Laufzeit wird letztendlich die Vererbungsstruktur modifiziert. Dieser Mechanismus erlaubt es, dass Objekte ihr Verhalten ändern – und das wiederum läuft darauf hinaus, dass sie ihre Klasse wechseln.

Beispielcode

Das folgende Beispiel zeigt den C++-Code für die im Abschnitt »Motivation« vorgestellte TCPConnection-Klasse. Es handelt sich hier um eine vereinfachte Version des TCP-Protokolls, die weder das vollständige Protokoll noch sämtliche Zustände von TCP-Verbindungen wiedergibt. Dieses Beispiel beruht auf dem von Lynch und Rose beschriebenen TCP-Verbindungsprotokoll [LR93].

Zunächst wird die Klasse TCPConnection definiert, die eine Schnittstelle zum Übertragen von Daten bereitstellt und Requests zum Ändern des Zustands bearbeitet:

```
class TCPOctetStream;
```

```

class TCPState;

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();
    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);

private:
    friend class TCPState;
    void ChangeState(TCPState* );
private:
    TCPState* _state;
};

```

TCPConnection verwahrt eine Instanz der TCPState-Klasse in der Membervariablen `_state`. Die Klasse TCPState kopiert die TCPConnection-Schnittstelle für Zustandsänderungen. Die verschiedenen TCPState-Operationen nehmen jeweils eine TCPConnection-Instanz als Parameter entgegen, was es TCPState erlaubt, auf Daten der TCPConnection-Klasse zuzugreifen und ihren Verbindungszustand zu ändern:

```

class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream* );
    virtual void ActiveOpen(TCPConnection* );
    virtual void PassiveOpen(TCPConnection* );
    virtual void Close(TCPConnection* );
    virtual void Synchronize(TCPConnection* );
    virtual void Acknowledge(TCPConnection* );
    virtual void Send(TCPConnection* );

protected:
    void ChangeState(TCPConnection*, TCPState* );
};

```

TCPConnection delegiert alle zustandsspezifischen Requests an die TCPState-Instanz `_state`. Außerdem stellt TCPConnection eine Operation zur Verfügung, um diese Variable auf einen neuen TCPState zu setzen. Der TCPConnection-Konstruktor initialisiert das Objekt mit dem Zustand TCPClosed (der später noch definiert wird):

```

TCPConnection::TCPConnection (){
    _state = TCPClosed::Instance();
}

```

```

}

void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}

void TCPConnection::ActiveOpen () {
    _state->ActiveOpen(this);
}

void TCPConnection::PassiveOpen () {
    _state->PassiveOpen(this);
}

void TCPConnection::Close () {
    _state->Close(this);
}

void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}

```

Die `TCPConnection`-Klasse implementiert das Standardverhalten für alle an sie delegierten Requests. Außerdem kann sie mit der `ChangeState`-Operation den Zustand einer `TCPConnection` ändern. Die Klasse `TCPState` ist als `friend`-Klasse von `TCPConnection` deklariert, damit sie über privilegierten Zugriff auf diese Operation verfügt:

```

void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }

void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}

```

Die `TCPState`-Unterklassen implementieren ein zustandsspezifisches Verhalten. Eine TCP-Verbindung kann viele verschiedene Zustände besitzen: `Established` (verbunden), `Listening` (horchend), `Closed` (getrennt) etc – und für jeden dieser Zustände existiert eine `TCPState`-Unterklasse. Auf drei dieser Unterklassen soll nun näher eingegangen werden: `TCPEstablished`, `TCPListen` und `TCPClosed`:

```
class TCPEstablished : public TCPState {
```

```

public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};

class TCPListen : public TCPState {
public:
    static TCPState* Instance();

    virtual void Send(TCPConnection*);
    // ...
};

class TCPClosed : public TCPState {
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};

```

TCPState-Unterklassen verwahren keinen lokalen Zustand, daher können sie gemeinsam genutzt werden, und es ist nur jeweils eine Instanz erforderlich. Die einzige Instanz der verschiedenen TCPState-Unterklassen wird durch die statische Instance-Operation geliefert – jede TCPState-Unterklasse wird dadurch zu einem Singleton (siehe Design Pattern *Singleton* (*Singleton*), [Abschnitt 3.5](#)).

Jede TCPState-Unterklasse implementiert ein zustandsspezifisches Verhalten für die im jeweiligen Zustand zulässigen Requests:

```

void TCPClosed::ActiveOpen (TCPConnection* t) {
    // SYN senden, SYN, ACK etc. empfangen
    ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Close (TCPConnection* t) {
    // FIN senden, ACK des FIN empfangen
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit (

```

```

TCPConnection* t, TCPOctetStream* o){
    t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
    // SYN senden, SYN, ACK etc. empfangen
    ChangeState(t, TCPEstablished::Instance());
}

```

Nach Erledigung der zustandsspezifischen Arbeiten rufen diese Operationen ihrerseits die `ChangeState`-Operation auf, um den `TCPConnection`-Zustand zu ändern. `TCPConnection` hat überhaupt keine Kenntnis vom TCP-Verbindungsprotokoll. Es sind vielmehr die `TCPState`-Unterklassen, die sämtliche Zustandsänderungen und Aktionen in TCP definieren.

Praxisbeispiele

Johnson und Zweig [JZ91] beschreiben das Design Pattern *State (Zustand)* und seine Anwendung auf das **TCP-Verbindungsprotokoll**.

Die meisten verbreiteten interaktiven Zeichenprogramme stellen Werkzeuge zum Ausführen von Operationen mittels direkter Manipulation zur Verfügung.

Beispielsweise erlaubt das Werkzeug zum Zeichnen einer Linie dem User, durch Klicken und Ziehen eine neue Linie zu erstellen. Mit einem Auswahlwerkzeug kann er verschiedene Formen (Kreis, Rechteck usw.) selektieren. Für gewöhnlich steht eine ganze Palette solcher Werkzeuge zur Auswahl. Für den User stellt sich sein Handeln so dar, als ob er eins der Werkzeuge aufnimmt und es handhabt, aber tatsächlich ändert sich mit der Auswahl eines Werkzeugs das Verhalten des Editors: Ist ein Zeichenwerkzeug aktiv, erstellt man Formen, ist das Auswahlwerkzeug aktiv, wählt man Formen aus usw. Das Design Pattern *State (Zustand)* kann dazu verwendet werden, das Verhalten des Editors in Abhängigkeit vom aktiven Werkzeug zu ändern.

Es lässt sich leicht eine abstrakte Klasse `Tool` mit Unterklassen definieren, die das für das jeweilige Werkzeug typische Verhalten implementieren. Das Zeichenprogramm verwaltet ein aktuelles `Tool`-Objekt und leitet Requests dorthin weiter. Wenn der User ein neues Werkzeug auswählt, ersetzt es das Objekt und das Verhalten des Zeichenprogramms ändert sich dementsprechend.

Die beiden Zeichenprogramm-Frameworks **HotDraw** [Joh92] und **Unidraw** [VL90] verwenden genau diese Vorgehensweise. Sie erlaubt es den Clients, auf einfache

Weise neue Werkzeugarten zu definieren. In HotDraw leitet die `DrawingController`-Klasse Requests an das aktuelle `Tool`-Objekt weiter. In Unidraw heißen die entsprechenden Klassen `viewer` und `Tool`. Das nachstehende Klassendiagramm gibt einen Überblick über die `Tool`- und `DrawingController`-Schnittstellen:

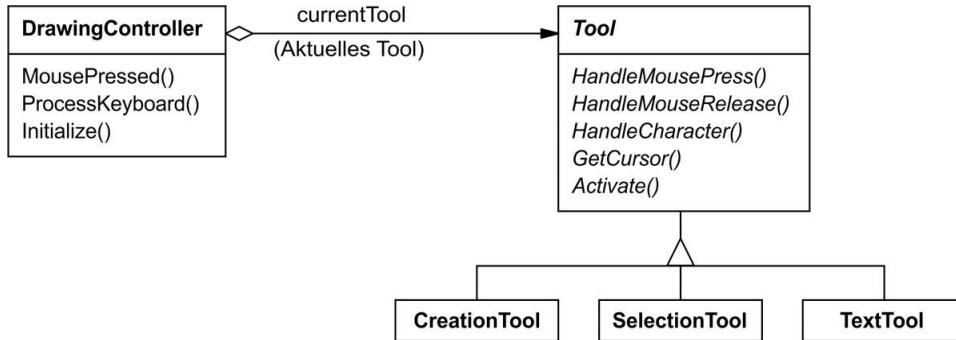


Abb. 5.35: *Tool- und DrawingController-Schnittstellen*

Die von Coplien geprägte »Umschlag-und-Brief«-Methode [Cop92] ist mit dem Pattern *State (Zustand)* verwandt. Sie beschreibt eine Technik zum Ändern der Klasse eines Objekts zur Laufzeit. Das Design Pattern *State (Zustand)* ist jedoch konkreter und konzentriert sich auf die Handhabung eines Objekts, dessen Verhalten von seinem Zustand abhängt.

Verwandte Patterns

Das Design Pattern *Flyweight (Fliegengewicht*, siehe [Abschnitt 4.6](#)) beschreibt, wann und wie sich *State*-Objekte gemeinsam nutzen lassen.

Bei *State*-Objekten handelt es sich oft um *Singletons* (siehe Design Pattern *Singleton (Singleton)*, [Abschnitt 3.5](#)).

5.9 Strategy (Strategie)

Objektbasiertes Verhaltensmuster

Zweck

Definition einer Familie von einzeln gekapselten, austauschbaren Algorithmen. Das Design Pattern *Strategy (Strategie)* ermöglicht eine variable und von den Clients unabhängige Nutzung des Algorithmus.

Auch bekannt als

Policy

Motivation

Es gibt eine ganze Reihe von Algorithmen zum Umbrechen von Fließtext in einzelne Zeilen. All diese Algorithmen fest in den Klassen, die Gebrauch davon machen, zu verankern ist aus mehreren Gründen nicht wünschenswert:

- Die Komplexität von Clients, die den Code zum Umbrechen selbst enthalten, nimmt zu. Die Clients werden dadurch umfangreicher und lassen sich schwieriger warten, insbesondere wenn sie mehrere Zeilenumbruchalgorithmen unterstützen.
- Häufig dürften unterschiedliche Algorithmen zu verschiedenen Zeitpunkten durchaus angebracht sein, grundsätzlich sollten mehrere Zeilenumbruchalgorithmen aber nur dann unterstützt werden, wenn sie auch tatsächlich alle Verwendung finden.
- Es ist schwierig, neue Algorithmen hinzuzufügen und bereits vorhandene zu ändern, wenn sie integraler Bestandteil des Clients sind.

Diese Probleme lassen sich umgehen, indem man Klassen definiert, die verschiedene Zeilenumbruchalgorithmen kapseln. Ein auf diese Art und Weise gekapselter Algorithmus wird als *Strategy (Strategie)* bezeichnet.

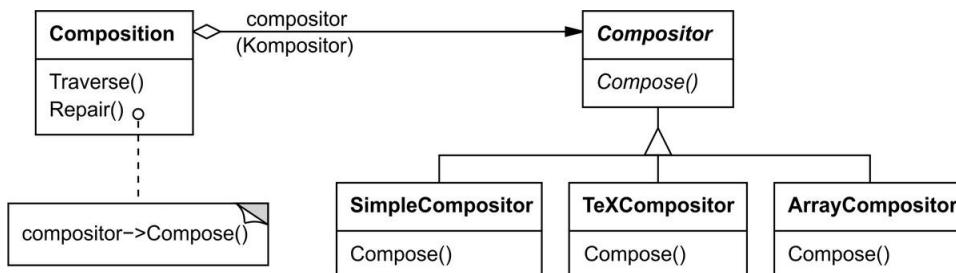


Abb. 5.36: Aufbau der *composition*-Klasse

Die hier gezeigte Composition-Klasse ist für die Verwaltung und Aktualisierung von Zeilenumbrüchen des in einer Textansicht dargestellten Textes zuständig. Die Strategien zum Umbrechen der Zeilen werden dabei nicht in der Composition-Klasse implementiert, sondern in separaten Unterklassen der abstrakten Compositor-Klasse. Die Compositor-Unterklassen implementieren verschiedene Strategien:

- **SimpleCompositor** implementiert eine einfache Strategie, die die Zeilenumbrüche einen nach dem anderen festlegt.
- **TeXCompositor** implementiert den TeX-Algorithmus zum Auffinden von Zeilenumbrüchen. Bei dieser Vorgehensweise wird versucht, die Zeilenumbrüche global zu optimieren, also jeweils für einen ganzen Absatz.
- **ArrayCompositor** implementiert eine Strategie, bei der die Zeilenumbrüche so gewählt werden, dass sich in jeder Zeile jeweils eine konstante Anzahl an Objekten befindet.

Die Composition-Klasse besitzt eine Referenz auf das Compositor-Objekt. Beim Neuformatieren ihres Textes delegiert die Composition-Klasse die Verantwortung dafür an ihr Compositor-Objekt. Dabei legt der Composition-Client fest, welcher Compositor verwendet wird, indem er den erwünschten Compositor bei der Composition-Klasse einrichtet.

Anwendbarkeit

Verwenden Sie das Design Pattern *Strategy (Strategie)*, wenn

- viele miteinander in Zusammenhang stehende Klassen sich nur in ihrem Verhalten voneinander unterscheiden. Das Pattern *Strategy (Strategie)* stellt eine Möglichkeit zur Verfügung, eine Klasse mit einer von vielen verschiedenen Verhaltensweisen auszustatten.
- verschiedene Varianten eines Algorithmus erforderlich sind. Beispielsweise könnten Sie mehrere Algorithmen einrichten, die unterschiedliche Kompromisse zwischen Speicherbedarf und Rechenzeit eingehen. Das Pattern *Strategy (Strategie)* lässt sich hierfür einsetzen, wenn die verschiedenen Varianten in Form einer Klassenhierarchie von Algorithmen implementiert sind [HO87].

- ein Algorithmus Daten verwendet, von denen der Client keine Kenntnis haben soll. Setzen Sie das Design Pattern *Strategy (Strategie)* ein, um zu vermeiden, dass komplexe, dem Algorithmus eigene Datenstrukturen enthüllt werden.
- eine Klasse viele verschiedene Verhaltensweisen aufweist und diese bei den Operationen in Form von vielfachen bedingten Anweisungen in Erscheinung treten. Statt viele bedingte Anweisungen zu verwenden, sollten miteinander in Zusammenhang stehende bedingte Anweisungen in eine eigene Strategy-Klasse verpackt werden.

Struktur

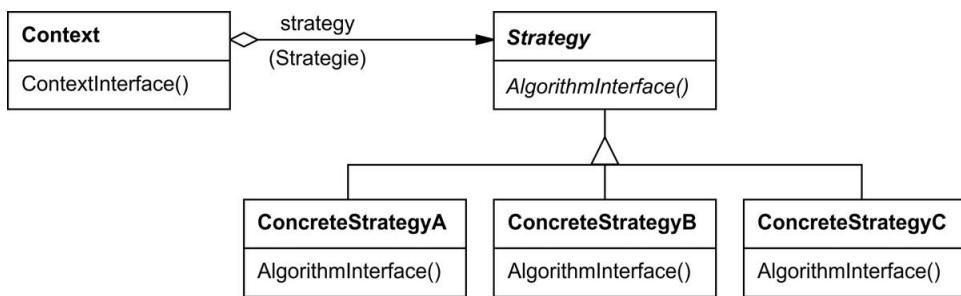


Abb. 5.37: Struktur des Design Pattern *Strategy (Strategie)*

Teilnehmer

- **Strategy (Compositor)**
 - Deklariert eine gemeinsame Schnittstelle für alle unterstützten Algorithmen. Context verwendet diese Schnittstelle, um den durch ConcreteStrategy festgelegten Algorithmus aufzurufen.
- **ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)**
 - Implementiert den Algorithmus mittels der Strategy-Schnittstelle.
- **Context (Composition)**
 - Wird mit einem ConcreteStrategy-Objekt ausgestattet.
 - Besitzt eine Referenz auf ein Strategy-Objekt.

- Kann eine Schnittstelle definieren, die es `Strategy` erlaubt, auf `Context`-Daten zuzugreifen.

Interaktionen

- `Strategy` und `Context` wechselwirken miteinander, um den ausgewählten Algorithmus zu implementieren. Beim Aufruf des Algorithmus kann `Context` alle erforderlichen Daten an `Strategy` übergeben. Alternativ kann `Context` auch sich selbst als Argument an `Strategy`-Operationen übergeben. Auf diese Weise kann `Strategy` bei Bedarf `Context` abfragen.
- `Context` leitet Requests seiner Clients an `Strategy` weiter. Normalerweise erzeugen die Clients ein `ConcreteStrategy`-Objekt und übergeben es an `Context`. Nachdem das erfolgt ist, interagieren die Clients ausschließlich mit `Context`. Oftmals gibt es eine ganze Familie von `ConcreteStrategy`-Klassen, aus der ein Client auswählen kann.

Konsequenzen

Das Pattern `Strategy` (*Strategie*) besitzt folgende Vor- und Nachteile:

1. *Familien verwandter Algorithmen.* Durch eine Hierarchie von `Strategy`-Klassen entsteht eine Familie von Algorithmen oder Verhaltensweisen, die `Context`-Objekte wiederverwenden können. Beim Ausgliedern der den verschiedenen Algorithmen gemeinsamen Funktionalität kann das Vererbungsprinzip hilfreich sein.
2. *Eine Alternative zum Erstellen von Unterklassen.* Die Vererbung bietet eine weitere Möglichkeit, eine Reihe verschiedener Algorithmen oder Verhaltensweisen zu unterstützen. Die Unterklassen einer `Context`-Klasse können direkt erstellt werden, um sie mit einem anderen Verhalten auszustatten, allerdings ist dieses Verhalten dann in der `Context`-Klasse fest verankert. Algorithmus- und `Context`-Implementierung werden hierbei vermengt, was es erschwert, `Context` zu verstehen, zu warten und zu erweitern. Und der Algorithmus kann auch nicht mehr zur Laufzeit verändert werden. Letztendlich gelangt man zu vielen zusammenhängenden Klassen, deren einziger Unterschied im verwendeten Algorithmus oder der Verhaltensweise besteht. Die Kapselung des Algorithmus in einer eigenen `Strategy`-Klasse ermöglicht

es, den Algorithmus unabhängig von Context zu ändern und erleichtert es, ihn zu wechseln, zu verstehen und zu erweitern.

3. *Strategien beseitigen das Erfordernis bedingter Anweisungen.* Das Pattern *Strategy (Strategie)* bietet bei der Auswahl eines erwünschten Verhaltens eine Alternative zu bedingten Anweisungen. Wenn verschiedene Verhaltensweisen in nur einer Klasse zusammengefasst werden, ist es schwierig, bei der Auswahl der richtigen Verhaltensweise auf bedingte Anweisungen zu verzichten. Die Kapselung des Verhaltens in einer separaten *Strategy*-Klasse beseitigt die Notwendigkeit dieser bedingten Anweisungen.

Ohne das Pattern *Strategy (Strategie)* könnte der Code zum Umbrechen von Fließtext beispielsweise so aussehen:

```
void Composition::Repair () {  
    switch (_breakingStrategy) {  
        case SimpleStrategy:  
            ComposeWithSimpleCompositor();  
            break;  
        case TexStrategy:  
            ComposeWithTeXCompositor();  
            break;  
        // ...  
    }  
    // Ergebnis mit vorhandenen Textarrangements  
    // zusammenführen, falls erforderlich  
}
```

Bei Verwendung des Patterns *Strategy (Strategie)* entfallen diese case-Anweisungen durch Delegation des Zeilenumbruchs an das *Strategy*-Objekt:

```
void Composition::Repair () {  
    _compositor->Compose();  
    // Ergebnis mit vorhandenen Textarrangements  
    // zusammenführen, falls erforderlich  
}
```

Enthält der Programmcode viele bedingte Anweisungen, ist das oftmals ein Hinweis darauf, dass das Pattern *Strategy (Strategie)* zur Anwendung kommen sollte.

4. *Eine Auswahl an Implementierungen.* Das Pattern *Strategy (Strategie)* ermöglicht verschiedene Implementierungen desselben Verhaltens. Die Auswahl einer der Strategien, die verschiedene Kompromisse zwischen Speicherbedarf und Rechenzeit eingehen, trifft der Client.

5. *Kenntnis verschiedener Strategien.* Das Pattern besitzt einen potenziellen Nachteil, nämlich dass einem Client klar sein muss, wie sich die Strategien voneinander unterscheiden, bevor er eine geeignete auswählen kann. Womöglich werden den Clients dabei auch Aspekte der Implementierung enthüllt. Aus diesem Grund sollte das Design Pattern *Strategy (Strategie)* nur dann verwendet werden, wenn die Änderung der Verhaltensweisen für die Clients von Bedeutung ist.
6. *Mehraufwand bei der Kommunikation zwischen Strategy und Context.* Die Strategy-Schnittstelle wird von allen `ConcreteStrategy`-Klassen gemeinsam genutzt, unabhängig davon, ob die darin implementierten Algorithmen einfach oder kompliziert sind. Es ist daher wahrscheinlich, dass einige der `ConcreteStrategy`-Objekte gar nicht alle ihnen über diese Schnittstelle übergebenen Informationen verwenden – simple `ConcreteStrategy`-Objekte benutzen sie möglicherweise sogar überhaupt nicht! Das bedeutet, dass `Context` mitunter Parameter erzeugt und initialisiert, die niemals verwendet werden. Sollte dies Schwierigkeiten bereiten, ist eine engere Kopplung zwischen `Strategy` und `Context` erforderlich.
7. *Erhöhte Anzahl von Objekten.* Die Verwendung des Patterns *Strategy (Strategie)* erhöht die Anzahl der Objekte einer Anwendung. Unter bestimmten Umständen lässt sich dieser Mehraufwand dadurch reduzieren, dass `Strategy`-Objekte als zustandslose Objekte implementiert werden, die `Context`-Objekte gemeinsam nutzen können. Der noch verbleibende Zustand wird durch das `Context`-Objekt verwaltet, das ihn bei jedem Request an das `Strategy`-Objekt übergibt. Gemeinsam genutzte `Strategy`-Objekte sollten ihren Zustand zwischen zwei Aufrufen nicht beibehalten. Das Design Pattern *Flyweight (Fliegengewicht*, siehe [Abschnitt 4.6](#)) beschreibt diesen Ansatz ausführlicher.

Implementierung

Beachten Sie bei der Implementierung folgende Aspekte:

1. *Definition der Strategy- und Context-Schnittstellen.* Die `Strategy`- und `Context`-Schnittstellen müssen einem `ConcreteStrategy`-Objekt effizienten Zugriff auf alle benötigten `Context`-Daten gewähren – das gilt auch in umgekehrter Richtung.

Ein Ansatz besteht darin, dass das `Context`-Objekt Daten als Parameter an `Strategy`-Operationen übergibt. Mit anderen Worten: Beförderung der Daten

zum Strategy-Objekt. `Strategy` und `Context` bleiben dadurch entkoppelt. Andererseits könnte das `Context`-Objekt Daten übergeben, die das `Strategy`-Objekt gar nicht benötigt.

Bei einer weiteren Herangehensweise übergibt das `Context`-Objekt *sich selbst* als Argument und das `Strategy`-Objekt muss die Daten ausdrücklich beim `Context`-Objekt erfragen. Alternativ speichert das `Strategy`-Objekt eine Referenz auf sein `Context`-Objekt und beseitigt damit die Notwendigkeit, überhaupt irgendwelche Daten zu übergeben. In beiden Fällen kann das `Strategy`-Objekt die erforderlichen Informationen in Erfahrung bringen. Das `Context`-Objekt muss nun allerdings eine ausgefeilte Schnittstelle zu seinen Daten zur Verfügung stellen, wodurch `Strategy`- und `Context`-Objekt enger gekoppelt sind.

Die beste Vorgehensweise ergibt sich aus den Anforderungen eines bestimmten Algorithmus und den von ihm benötigten Daten.

2. **Strategy-Objekte als Template-Parameter.** In C++ können Templates dazu verwendet werden, eine Klasse mit einem `Strategy`-Objekt auszustatten. Dieses Verfahren ist jedoch nur anwendbar, wenn das `Strategy`-Objekt 1. schon beim Kompilieren ausgewählt werden kann und 2. zur Laufzeit nicht geändert werden muss. In diesem Fall wird die entsprechende Klasse (z. B. `Context`) als Template-Klasse definiert, die eine `Strategy`-Klasse als Parameter besitzt:

```
template <class AStrategy>
class Context {
    void Operation() { theStrategy.DoAlgorithm(); }
    // ...
private:
    AStrategy theStrategy;
};
```

Bei der Instanziierung wird die Klasse dann mit einer `Strategy`-Klasse ausgestattet:

```
class MyStrategy {
public:
    void DoAlgorithm();
};

Context<MyStrategy> aContext;
```

Bei der Verwendung von Templates ist es nicht erforderlich, eine abstrakte Klasse zu definieren, die wiederum die `Strategy`-Schnittstelle definiert. Die

Verwendung von **Strategy** als Template-Parameter ermöglicht außerdem die statische Anbindung eines **Strategy**-Objekts an sein **Context**-Objekt, was möglicherweise die Effizienz erhöht.

3. *Optionale Strategy-Objekte.* Wenn es einen Sinn ergibt, *kein* **Strategy**-Objekt einzusetzen, lässt sich die **Context**-Klasse vereinfachen. Sie überprüft das Vorhandensein eines **Strategy**-Objekts, bevor sie darauf zugreift. Gibt es eins, verwendet das **Context**-Objekt es wie gewohnt. Gibt es jedoch kein **Strategy**-Objekt, zeigt das **Context**-Objekt das Standardverhalten. Der Vorteil dieses Ansatzes ist, dass sich die Clients überhaupt nicht um **Strategy**-Objekte zu kümmern brauchen, *es sei denn*, ihnen missfällt das Standardverhalten.

Beispielcode

An dieser Stelle wird der High-Level-Code des Beispiels aus dem Abschnitt »Motivation« aufgezeigt, der auf der Implementierung von **Composition**- und **Compositor**-Klassen in InterView [LCI+92] beruht.

Die **Composition**-Klasse verwaltet eine Reihe von **Component**-Instanzen, die Text und Grafikelemente eines Dokuments repräsentieren. Das **Composition**-Objekt ordnet **Component**-Objekte zeilenweise an und verwendet dazu eine **Compositor**-Unterklasseninstanz, die eine Zeilenumbruchstrategie kapselt. Jedem **Component**-Objekt ist eine ursprüngliche Größe, eine Dehnbarkeit und eine Schrumpfbarkeit zugeordnet. Die Dehnbarkeit gibt an, in welchem Ausmaß die Komponente über ihre ursprüngliche Größe hinaus wachsen kann. Die Schrumpfbarkeit beschreibt, in welchem Maß es schrumpfen kann. Das **Composition**-Objekt übergibt diese Werte an ein **Compositor**-Objekt, das sie zum Ermitteln der besten Positionen für Zeilenumbrüche heranzieht:

```
class Composition {
public:
    Composition(Compositor* );
    void Repair();

private:
    Compositor* _compositor;
    Component* _components;
    // Komponentenliste
    int _componentCount;
    // Komponentenanzahl
    int _lineWidth;
    // Zeilenbreite
    int* _lineBreaks;
```

```

    // Position der Zeilenumbrüche
    // in Komponenten
    int _lineCount;
    // Zeilenanzahl
};

```

Wenn ein neues Layout benötigt wird, fordert das **Composition**-Objekt sein **Compositor**-Objekt auf festzustellen, wo die Zeilenumbrüche platziert werden sollen. Das **Composition**-Objekt übergibt dem **compositor**-Objekt drei Arrays, die die ursprüngliche Größe sowie die Dehn- und Schrumpfbarkeit der Komponenten festlegen. Ebenfalls übergeben werden die Komponentenanzahl, die Breite der Zeile und ein Array, das das **compositor**-Objekt mit den Positionen der Zeilenumbrüche befüllt. Das **Compositor**-Objekt liefert die Anzahl der errechneten Zeilenumbrüche zurück.

Die **Compositor**-Schnittstelle erlaubt es dem **Composition**-Objekt, alle erforderlichen Daten zu übergeben. Hier handelt es sich um ein Beispiel für die »Beförderung der Daten zum **Strategy**-Objekt«:

```

class Compositor {
public:
    virtual int Compose(
        Coord natural[], Coord stretch[],
        Coord shrink[], int componentCount,
        int linewidth, int breaks[]
    ) = 0;

protected:
    Compositor();
};

```

Beachten Sie hier, dass **Compositor** eine abstrakte Klasse ist. Die verschiedenen Zeilenumbruchstrategien werden in konkreten Unterklassen definiert.

Das **Composition**-Objekt ruft in der **Repair**-Operation sein **Compositor**-Objekt auf. **Repair** initialisiert zunächst Arrays mit Werten für die ursprüngliche Größe sowie die Dehn- und Schrumpfbarkeit aller Komponenten (die Einzelheiten lassen wir hier der Bündigkeit halber weg). Danach fragt es beim **Compositor**-Objekt die Zeilenumbrüche ab und richtet die Komponenten dementsprechend ein (ebenfalls weggelassen):

```

void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
}

```

```

int* breaks;

// Einrichten von Arrays mit den
// gewünschten Werten
// ...

// Ermitteln der Zeilenumbrüche
int breakCount;
breakCount = _compositor->Compose(
    natural, stretchability, shrinkability,
    componentCount, _lineWidth, breaks
);
// Einrichten der Komponenten entsprechend
// den Zeilenumbrüchen
// ...
}

```

Als Nächstes werden die Compositor-Unterklassen betrachtet. Die SimpleCompositor-Klasse überprüft die Komponenten zeilenweise, um festzustellen, wo die Zeilenumbrüche hingehören:

```

class SimpleCompositor : public Compositor {
public:
    SimpleCompositor ();

    virtual int Compose(
        Coord natural[], Coord stretch[],
        Coord shrink[], int componentCount,
        int lineWidth, int breaks[]
    );
    // ...
};

```

Die Klasse TeXCompositor verwendet eine allgemeinere Strategie: Sie überprüft jeweils einen Absatz und berücksichtigt dabei Größe und Dehnbarkeit der Komponenten. Außerdem versucht sie, für eine ausgeglichene »Färbung« des Absatzes zu sorgen, indem sie den Leerraum zwischen den Komponenten minimiert.

```

class TeXCompositor : public Compositor {
public:
    TeXCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[],
        Coord shrink[], int componentCount,
        int lineWidth, int breaks[]

    );
    // ...
};

```

```
};
```

Die `ArrayCompositor`-Klasse teilt die Komponenten in gleichmäßigen Abständen in Zeilen auf:

```
class ArrayCompositor : public Compositor {
public:
    ArrayCompositor(int interval);

    virtual int Compose(
        Coord natural[], Coord stretch[],
        Coord shrink[], int componentCount,
        int lineWidth, int breaks[]

    );
    // ...
};
```

Diese Klassen nutzen nicht in jedem Fall alle Informationen, die `Compose` bereitstellt. Die `SimpleCompositor`-Klasse lässt die Dehnbarkeit der Komponenten außer Acht und berücksichtigt lediglich die ursprüngliche Größe. Die `TeXCompositor`-Klasse verwendet sämtliche Informationen, die `ArrayCompositor`-Klasse hingegen gar keine.

Bei der `Composition`-Instanziierung wird der zu verwendende `Compositor` übergeben:

```
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor(100));
```

Die `Compositor`-Schnittstelle ist sorgfältig dafür ausgelegt, alle Layout-Algorithmen zu unterstützen, die möglicherweise von Unterklassen implementiert werden – schließlich soll ja nicht bei jeder neuen Unterklassie die Schnittstelle geändert werden müssen, denn das erfordert auch eine Anpassung der bereits vorhandenen Unterklassen. Im Allgemeinen bestimmen die `Strategy`- und `Context`-Schnittstellen, wie nah das Design Pattern seinem Ziel kommt.

Praxisbeispiele

Sowohl **ET++** [WGM88] als auch **InterViews** verwenden das Design Pattern `Strategy` (`Strategie`) zur Kapselung der verschiedenen beschriebenen Zeilenumbruchalgorithmen.

Im **RTL-System** zur Optimierung von Compilercode [JML92] wird das Pattern *Strategy (Strategie)* zur Definition verschiedener Methoden für die Belegung von Speicherregistern (`RegisterAllocator`) und Richtlinien bezüglich des Befehlssatz-Schedulings (`RISCscheduler`, `CISCscheduler`) eingesetzt. Dieses Vorgehen bietet die nötige Flexibilität für die Ausrichtung der Optimierung auf verschiedene Rechnerarchitekturen.

Das **ET++SwapsManager-Framework für Berechnungen** ermittelt die Kosten verschiedener Finanzierungsinstrumente [EG92]. Die wesentlichen Abstrahierungen sind `Instrument` (Finanzierungsinstrument) und `YieldCurve` (Renditekurve). Die verschiedenen Instrumente sind als `Instrument`-Unterklassen implementiert. `YieldCurve` berechnet Abzinsungsfaktoren, die den gegenwärtigen Wert der zukünftigen Zahlungsströme bestimmen. Das Framework stellt eine Familie von `ConcreteStrategy`-Klassen zur Verfügung, die Zahlungsströme erzeugen, Swap-Geschäfte bewerten und Abzinsungsfaktoren berechnen. Man kann damit neue Berechnungsmodule erzeugen, indem `Instrument` und `YieldCurve` mit verschiedenen `ConcreteStrategy`-Objekten ausgestattet werden. Dieser Ansatz erlaubt es nicht nur, vorhandene `Strategy`-Implementierungen auszuwählen und miteinander zu kombinieren, sondern auch, neue zu definieren.

Die von **Booch** [BV90] eingesetzten Komponenten verwenden `Strategy`-Objekte als Argumente für Templates. Seine `Collection`-Klassen besitzen drei `Strategy`-Objekte, die verschiedene Methoden zur Speicherplatzreservierung nutzen: geregelt (`Managed`, der Speicher entstammt einem Pool), gesteuert (`Controlled`, Reservierung und Freigabe des Speichers sind durch eine Sperre geschützt) oder ungeregelt (`Unmanaged`, die normale Speicherreservierung). Diese `Strategy`-Objekte werden einer `Collection`-Klasse bei der Instanziierung als `Template`-Argumente übergeben. Beispielsweise wird eine `UnboundedCollection`-Klasse, die die normale Speicherreservierung einsetzt, durch `UnboundedCollection<MyItemType*, Unmanaged>` instanziert.

RApp ist ein System zum Entwerfen von integrierten Schaltungen [GA89, AG90]. Es muss ermitteln, wie die Drähte geführt werden sollen, die verschiedene Teilsysteme der Schaltung miteinander verbinden. Diese Routing-Algorithmen sind in RApp als Unterklassen einer abstrakten Router-Klasse definiert, bei der es sich um eine `Strategy`-Klasse handelt.

Borlands **ObjectWindows** [Bor94] verwendet `Strategy`-Objekte in Dialogfeldern, um sicherzustellen, dass der User gültige Daten eingibt. Zahlenwerte dürfen beispielsweise oftmals einen bestimmten Bereich nicht überschreiten oder ein Eingabefeld darf nur numerische Werte enthalten. Die Überprüfung eines Strings

auf Richtigkeit erfordert möglicherweise das Durchsuchen einer Tabelle.

ObjectWindows verwendet Validator-Objekte zur Kapselung von Strategy-Objekten, die solchen Überprüfungen dienen. Validator ist also ein Beispiel für ein Strategy-Objekt. Dateneingabefelder delegieren die Überprüfung an ein optionales Validator-Objekt. Der Client stattet ein Eingabefeld mit einem Validator aus, sofern eine Überprüfung erforderlich ist (ein Beispiel für ein optionales Strategy-Objekt). Wenn das Dialogfeld geschlossen wird, fordern die Eingabefelder ihre Validator-Objekte auf, die Daten zu überprüfen. Für häufige Fälle stellt die Klassenbibliothek Validator-Objekte bereit, wie etwa einen RangeValidator zum Überprüfen von Zahlen. Neue, speziell auf einen Client ausgerichtete Validator-Objekte lassen sich leicht durch Erstellen einer Validator-Unterklasse erzeugen.

Verwandte Patterns

Flyweight (Fliegengewicht, siehe [Abschnitt 4.6](#)): Strategy-Objekte sind häufig gut für das Design Pattern *Flyweight (Fliegengewicht)* geeignet.

5.10 Template Method (Schablonenmethode)

Klassenbasiertes Verhaltensmuster

Zweck

Definition der Grundstruktur eines Algorithmus in einer Operation sowie Delegation einiger Ablaufschritte an Unterklassen. Das Design Pattern *Template Method (Schablonenmethode)* ermöglicht den Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne dessen grundlegende Struktur zu verändern.

Motivation

Zur Veranschaulichung dieses Patterns wird hier ein Anwendungs-Framework

verwendet, das die Klassen Application und Document bereitstellt. Die Application-Klasse ist für das Öffnen vorhandener Dokumente zuständig, die in einem externen Format gespeichert sind, beispielsweise als Datei. Ein Document-Objekt repräsentiert den Inhalt eines Dokuments, nachdem die Datei eingelesen wurde.

Mit diesem Framework erstellte Anwendungen können Unterklassen von Application und Document erstellen, um sie an ihre speziellen Bedürfnisse anzupassen. Beispielsweise definiert ein Zeichenprogramm die Unterklassen DrawApplication und DrawDocument, ein Tabellenkalkulationsprogramm hingegen die Unterklassen SpreadsheetApplication und SpreadsheetDocument.

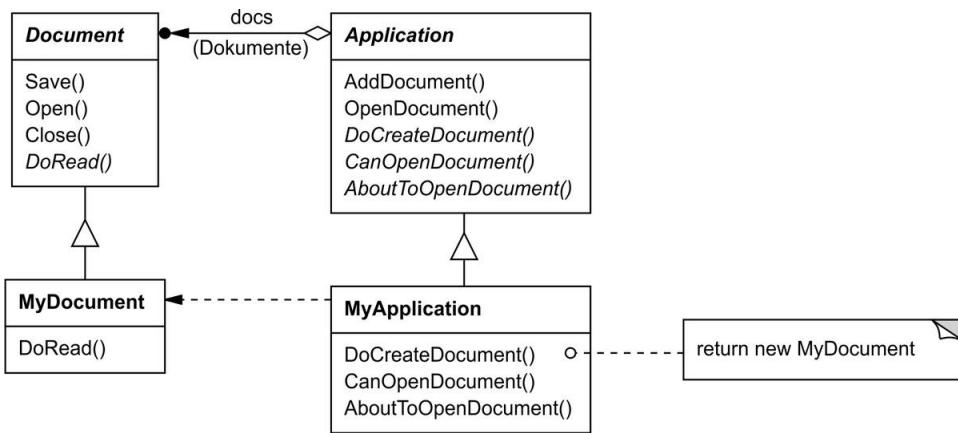


Abb. 5.38: Document- und Application-Klasse

Die abstrakte Application-Klasse definiert den Algorithmus zum Öffnen und Einlesen eines Dokuments in ihrer OpenDocument-Operation:

```

void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // Kann dieses Dokument nicht verarbeiten
        return;
    }

    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
  
```

Die OpenDocument-Klasse legt die Schritte beim Öffnen eines Dokuments fest. Sie

überprüft, ob das Dokument geöffnet werden kann, erzeugt ein anwendungseigenes Document-Objekt, fügt es dem Satz der ihr bekannten Dokumente hinzu und liest es aus der Datei ein.

OpenDocument wird als *Template Method (Schablonenmethode)* bezeichnet. Eine Schablonenmethode definiert einen Algorithmus in Form abstrakter Operationen, die von Unterklassen überschrieben werden, um ein konkretes Verhalten bereitzustellen. Diejenigen Schritte des Algorithmus, die prüfen, ob das Dokument geöffnet werden kann und das Dokument erzeugen, sind in Application-Unterklassen (`CanOpenDocument` bzw. `DoCreateDocument`) definiert. Der Schritt zum Einlesen des Dokuments ist in einer Document-Klasse (`DoRead`) definiert. Die Template Method definiert außerdem eine Operation, die Application-Unterklassen, die daran interessiert sind, darüber informiert, wenn die Anwendung im Begriff ist, das Dokument zu öffnen (`AboutToOpenDocument`).

Durch die Definition einiger der Schritte eines Algorithmus in Form abstrakter Operationen legt die Template Method zwar deren Reihenfolge fest, aber sie ermöglicht den Application- und Document-Unterklassen, diese Schritte ihren Bedürfnissen anzupassen.

Anwendbarkeit

Das Design Pattern *Template Method (Schablonenmethode)* sollte eingesetzt werden,

- um die unveränderlichen Teile eines Algorithmus einmalig zu implementieren und es den Unterklassen zu überlassen, Verhaltensweisen zu implementieren, die veränderlich sind.
- wenn die Verhaltensweisen, die Unterklassen gemeinsam haben, in eine gemeinsame Klasse ausgelagert werden sollen, um doppelten Code zu vermeiden. Hierbei handelt es sich um ein schönes Beispiel von »Refaktorierung zwecks Verallgemeinerung« (engl. *refactoring to generalize*), wie es Opdyke und Johnson [OJ93] beschrieben haben. Zunächst müssen die unterschiedlichen Funktionalitäten im vorhandenen Code identifiziert werden, dann werden sie auf neue Operationen verteilt. Abschließend wird der fragile Code durch eine Template Method ersetzt, die eine der neuen Operationen aufruft.
- um die Erweiterungen von Unterklassen zu regulieren. Dazu kann eine Template Method definiert werden, die an bestimmten Stellen einen »Hook«

(dt. »Einschubmethode«) aufruft. Dadurch sind Erweiterungen nur an diesen Stellen zugelassen.

Struktur

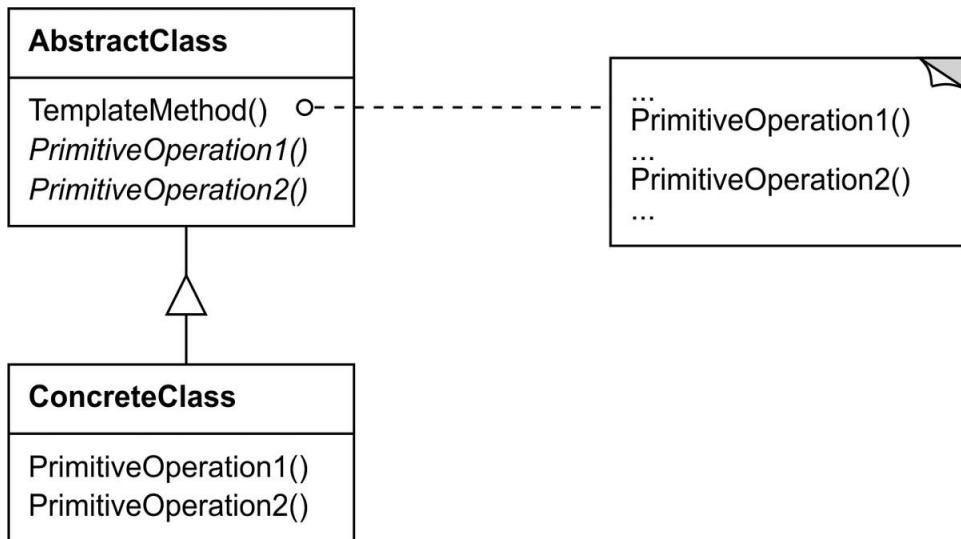


Abb. 5.39: Struktur des Design Patterns Template Method (Schablonenmethode)

Teilnehmer

- **AbstractClass** (Application)
 - Definiert abstrakte primitive Operationen, die durch konkrete Unterklassen überschrieben werden, um die einzelnen Schritte eines Algorithmus zu implementieren.
 - Implementiert eine Template Method, die die Grundstruktur eines Algorithmus beschreibt. Diese Schablonenmethode ruft sowohl primitive Operationen als auch in **AbstractClass** oder anderen Objekten definierte Operationen auf.
- **ConcreteClass** (MyApplication)
 - Implementiert die primitiven Operationen, die die einer Unterklasse zugeordneten Schritte eines Algorithmus ausführen.

Interaktionen

- ConcreteClass überlässt es AbstractClass, die unveränderlichen Schritte des Algorithmus zu implementieren.

Konsequenzen

Template Methods (Schablonenmethoden) gehören zu den grundlegenden Techniken zur Wiederverwendung von Code. Sie spielen insbesondere für Klassenbibliotheken eine wichtige Rolle, weil sie eine Möglichkeit bieten, gemeinsame Verhaltensweisen in Klassen der Bibliothek auszulagern.

Template Methods (Schablonenmethoden) führen zu einer auf den Kopf gestellten Ablaufstruktur, die manchmal auch als »Hollywood-Prinzip« bezeichnet wird, soll heißen: »Rufen Sie uns nicht an, wir rufen Sie an.« [Swe85]. Das bezieht sich darauf, dass eine Basisklasse die Operationen einer Unterklasse aufruft – und nicht umgekehrt.

Template Methods (Schablonenmethoden) rufen die folgenden Arten von Operationen auf:

- konkrete Operationen (entweder von ConcreteClass oder von Client-Klassen),
- konkrete AbstractClass-Operationen (d. h. Operationen, die für Unterklassen generell nützlich sind),
- primitive Operationen (d. h. abstrakte Operationen),
- *Factory Methods (Fabrikmethoden)*, siehe [Abschnitt 3.3](#)) und
- Hook-Operationen, die ein Standardverhalten bereitstellen, das Unterklassen bei Bedarf erweitern können. Eine Hook-Operation bewirkt standardmäßig oftmals überhaupt nichts.

Es ist wichtig, dass *Template Methods (Schablonenmethoden)* angeben, welche Operationen Hooks (die überschrieben werden können) und welche abstrakten Operationen (die überschrieben werden müssen) sind. Um eine abstrakte Klasse effektiv wiederverwenden zu können, muss der Autor einer Unterklasse wissen, welche Operationen zum Überschreiben geeignet sind.

Eine Unterklasse kann das Verhalten einer Basisklasse *erweitern*, indem sie eine Operation überschreibt und ausdrücklich die Operation der Basisklasse aufruft:

```
void DerivedClass::Operation () {
    ParentClass::Operation();
    //Erweitertes Verhalten der abgeleiteten Klasse
}
```

Leider vergisst man schnell einmal, die geerbte Operation aufzurufen. Eine solche Operation kann aber leicht in eine Template Method umgewandelt werden, um der Basisklasse zu ermöglichen, ihre Erweiterung durch Unterklassen zu regulieren. Die zugrunde liegende Idee ist, in der Template Method der Basisklasse eine Hook-Operation aufzurufen. Die Unterklassen können dann diese Hook-Operation überschreiben:

```
void ParentClass::Operation () {
    // Verhalten der Basisklasse
    HookOperation();
}
```

In ParentClass bewirkt HookOperation gar nichts:

```
void ParentClass::HookOperation () { }
```

Unterklassen überschreiben HookOperation, um das Verhalten zu erweitern:

```
void DerivedClass::HookOperation () {
    // Erweiterung der abgeleiteten Klasse
}
```

Implementierung

Hinsichtlich der Implementierung sind drei Aspekte zu beachten:

1. *C++-Zugriffssteuerung verwenden.* In C++ werden die von einer Template Method aufgerufenen Operationen als geschützte Member deklariert. Dadurch ist sichergestellt, dass sie nur von der Template Method aufgerufen werden können. Primitive Operationen, die überschrieben werden müssen, werden als rein virtuell deklariert. Die Template Method selbst sollte nicht überschrieben werden – deshalb kann sie auch zu einer nicht-virtuellen Memberfunktion gemacht werden.
2. *Minimierung primitiver Operationen.* Ein wichtiges Ziel beim Entwerfen einer

Template Method ist die Minimierung der Anzahl primitiver Operationen, die eine Unterklasse überschreiben muss, um den Algorithmus zu konkretisieren. Je mehr Operationen überschrieben werden müssen, desto lästiger wird es für den Client.

3. *Namenskonventionen*. Diejenigen Operationen, die überschrieben werden sollen, können bei der Benennung durch ein Präfix gekennzeichnet werden. Das MacApp-Framework für Macintosh-Anwendungen [App89] stellt beispielsweise allen Template Methods ein »Do-« voran: »DoCreateDocument«, »DoRead« usw.

Beispielcode

Das folgende C++-Beispiel zeigt, wie eine Basisklasse ihren Unterklassen eine Invariante auferlegen kann. Das Beispiel entstammt NeXTs AppKit [Add94]. Eine Klasse `View` gestattet das Zeichnen auf dem Bildschirm. Die `View`-Klasse erzwingt die Invariante, dass ihre Unterklassen nur dann etwas zeichnen können, nachdem die fragliche `View`-Instanz den »Fokus« erhalten hat, was es erforderlich macht, dass zuvor ein bestimmter, für das Zeichnen wichtiger Zustand (z. B. Einstellungen für Farben und Schriftarten) korrekt eingerichtet worden ist.

Um diesen Zeichnungszustand herzustellen, kann eine Template Method namens `Display` verwendet werden. Die `View`-Klasse definiert zwei konkrete Operationen, `SetFocus` und `ResetFocus`, die den Zeichnungszustand einrichten bzw. wieder aufräumen. Die `DoDisplay`-Hook-Operation der `View`-Klasse führt das eigentliche Zeichnen aus. `Display` ruft `SetFocus` vor `DoDisplay` auf, um den Zeichnungszustand einzurichten. Danach ruft `Display` `ResetFocus` auf, um den Zeichnungszustand wieder freizugeben:

```
void View::Display () {
    SetFocus();
    DoDisplay();
    ResetFocus();
}
```

Die `View`-Clients rufen jedes Mal `Display` auf, um die Invariante zu erhalten, und die `View`-Unterklassen überschreiben `DoDisplay` immer.

In der `View`-Klasse ist `DoDisplay` untätig:

```
void View::DoDisplay () { }
```

Unterklassen überschreiben `DoDisplay`, um ihre eigenen Zeichenmethoden hinzuzufügen:

```
void MyView::DoDisplay () {  
    // View-Inhalt zeichnen  
}
```

Praxisbeispiele

Das Design Pattern *Template Method (Schablonenmethode)* ist von so grundlegender Bedeutung, dass es in fast jeder abstrakten Klasse zu finden ist. **Wirfs-Brock et al.** [WW90, WBJ90] liefern eine gute Übersicht über und Erläuterungen zu Template Methods.

Verwandte Patterns

Factory Methods (Fabrikmethoden, siehe [Abschnitt 3.3](#)) werden häufig von Template Methods aufgerufen. In dem Beispiel aus dem Abschnitt »Motivation« ruft die Template Method `OpenDocument` die Factory Method `DoCreateDocument` auf.

Strategy (Strategie, siehe [Abschnitt 5.9](#)): Template Methods nutzen die Vererbung, um Teile eines Algorithmus zu ändern. Das Pattern *Strategy (Strategie)* verwendet Delegation, um den gesamten Algorithmus auszutauschen.

5.11 Visitor (Besucher)

Objektbasiertes Verhaltensmuster

Zweck

Darstellung einer auf die Elemente einer Objektstruktur anzuwendenden Operation. Das Design Pattern *Visitor (Besucher)* ermöglicht die Definition einer neuen Operation, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

Motivation

In diesem Beispiel wird ein Compiler verwendet, der Programme als abstrakte Syntaxbäume darstellt. Dazu muss er eine »statische semantische« Analyse der abstrakten Syntaxbäume durchführen, wie beispielsweise die Überprüfung, ob alle Variablen definiert sind. Außerdem muss er auch Code erzeugen. Er könnte also Operationen definieren, die eine Typüberprüfung durchführen, Code optimieren, den Programmablauf analysieren, untersuchen, ob Variablen vor ihrer Verwendung Werte zugewiesen wurden, usw. Darüber hinaus lassen sich die abstrakten Syntaxbäume zur Optimierung der Codeausgabe (engl. *pretty-printing*), zur Neustrukturierung von Programmen, zur Ausstattung des Codes mit Zusatzinformationen zwecks Überwachung des Programms (sogenannte »Code-Instrumentierung«) und zur Berechnung verschiedener weiterer Softwaremetriken eines Programms verwenden.

Die meisten dieser Operationen bedingen eine andere Handhabung der Knoten mit Zuweisungskommandos als derjenigen, die Variablenzugriffe oder arithmetische Ausdrücke enthalten. Daher existiert jeweils eine eigene Klasse für Zuweisungskommandos, eine für den Zugriff auf Variablen, eine weitere für arithmetische Ausdrücke usw. Dieser Satz von Node-Klassen hängt natürlich von der Programmiersprache ab, die kompiliert wird, er ändert sich jedoch von Sprache zu Sprache nur wenig.

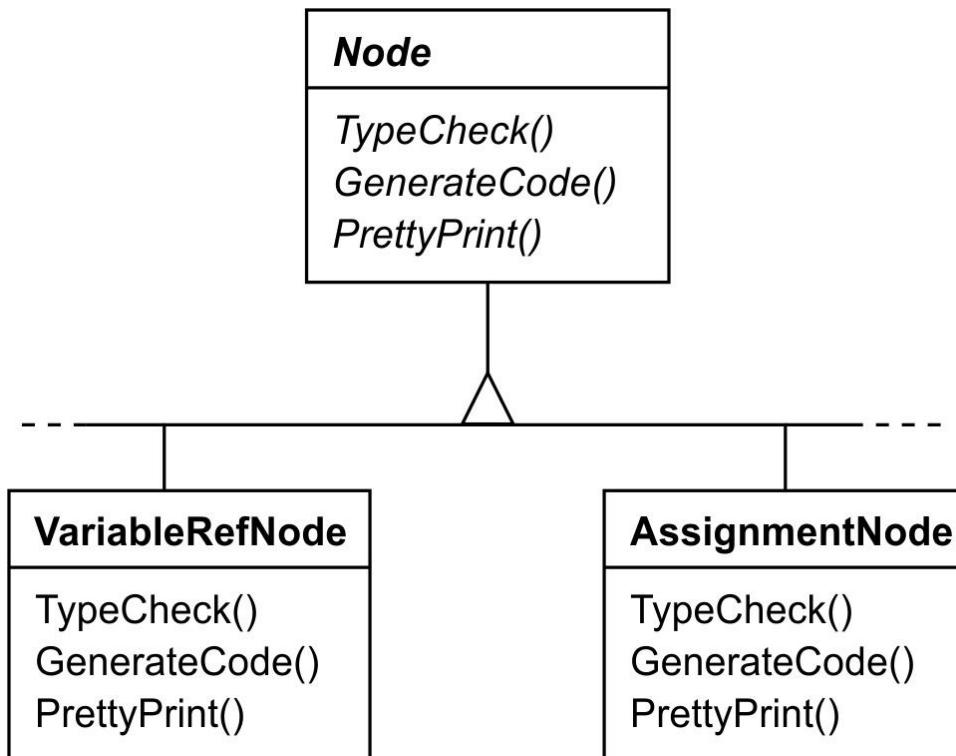


Abb. 5.40: Ein Teil der Node-Klassenhierarchie

Das Diagramm (siehe [Abbildung 5.40](#)) zeigt einen Teil der Node-Klassenhierarchie. Das Problem hierbei ist, dass die Verteilung all dieser Operationen auf verschiedene Knotenklassen zu einem System führt, das schwierig zu verstehen, zu warten und zu ändern ist. Es dürfte einige Verwirrung stiften, dass Code zur Typüberprüfung mit Code zur Quelltextformatierung und Code zur Analyse des Programmablaufs vermengt ist. Außerdem erfordert das Hinzufügen einer neuen Operation für gewöhnlich eine Neukompilierung aller drei Klassen. Besser wäre es aber, wenn jede neue Operation einzeln hinzugefügt werden könnte und die Knotenklassen von den Operationen, die auf sie angewendet werden, unabhängig wären.

Beides lässt sich erreichen, wenn die miteinander verwandten Operationen jeder Klasse in ein eigenes Objekt verpackt werden, das als *Visitor (Besucher)* bezeichnet wird, und dieses Objekt bei der Traversierung des abstrakten Syntaxbaums an dessen Elemente zu übergeben. »Akzeptiert« ein Element den Besucher, sendet es einen Request an das für seine eigene Klasse zuständige *Visitor*-Objekt. Das Element wird dabei als Argument übergeben. Das *Visitor*-Objekt führt daraufhin die Operation für das Element aus – diejenige Operation, die sich vorher in der Klasse des Elements befand.

Ein Compiler, der auf das Pattern *Visitor (Besucher)* verzichtet, könnte beispielsweise den Typ einer Prozedur überprüfen, indem er die *TypeCheck*-Operation seines abstrakten Syntaxbaums aufruft. Jeder der Knoten würde *TypeCheck* durch einen *TypeCheck*-Aufruf seiner eigenen Komponenten implementieren (siehe [Abbildung 5.40](#)). Wenn der Compiler die Typüberprüfung einer Prozedur mithilfe des Patterns *Visitor (Besucher)* durchführt, erzeugt er ein *TypeCheckingVisitor*-Objekt und übergibt es beim Aufruf der *Accept*-Operation des abstrakten Syntaxbaums als Argument. Die verschiedenen Knoten implementieren *Accept*, indem sie ihrerseits *Visitor* aufrufen: Ein Knoten mit Zuweisungskommando ruft die *visitor*-Operation *visitAssignment* auf, während eine Referenz auf eine Variable die *visitVariableReference*-Operation aufruft. Was früher einmal die *TypeCheck*-Operation der Klasse *AssignmentNode* war, ist jetzt die *visitAssignment*-Operation des *TypeCheckingVisitor*.

Damit das Pattern *Visitor (Besucher)* mehr als nur eine Typüberprüfung leisten kann, wird für alle Besucher eines abstrakten Syntaxbaums eine abstrakte Basisklasse *NodeVisitor* benötigt. Die *NodeVisitor*-Klasse muss für jede Knotenklasse eine Operation deklarieren. Eine Anwendung, die Softwaremetriken berechnen soll, kann neue *NodeVisitor*-Unterklassen anlegen und braucht ihren Knotenklassen keinen anwendungsspezifischen Code mehr hinzuzufügen. Das

Design Pattern *Visitor (Besucher)* kapselt die Operation bei jeder Kompilierungsphase in einem dieser Phase zugehörigen Visitor-Objekt.

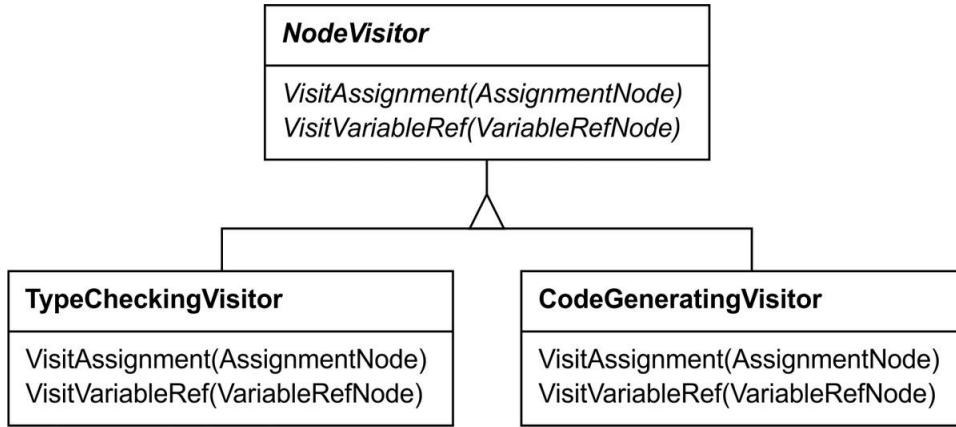


Abb. 5.41: Die *NodeVisitor*-Hierarchie

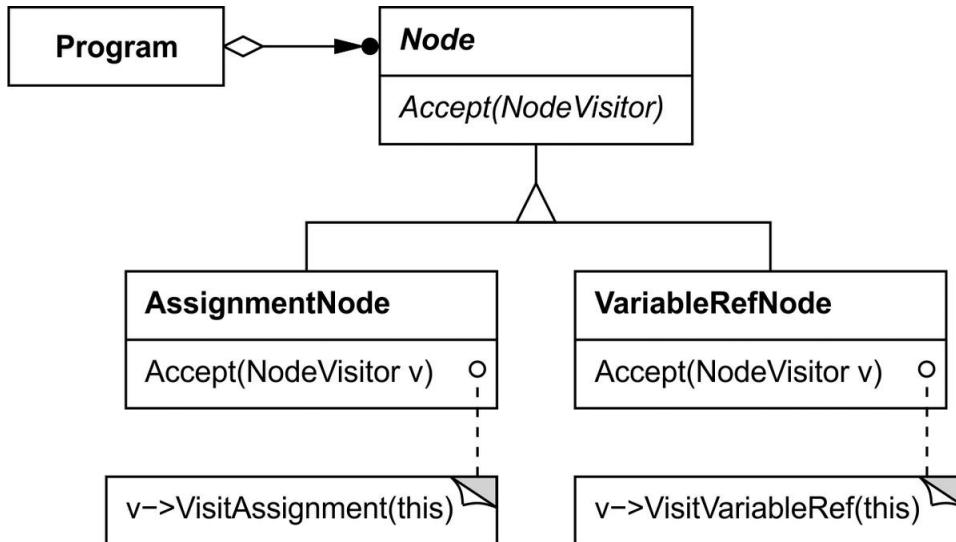


Abb. 5.42: Die *Node*-Hierarchie

Beim Design Pattern *Visitor (Besucher)* werden zwei Klassenhierarchien definiert: eine für die zu bearbeitenden Elemente (die *Node*-Hierarchie) und eine für die *Visitor*-Objekte, die Operationen für die Elemente definieren (die *NodeVisitor*-Hierarchie). Neue Operationen werden durch das Hinzufügen einer Unterklasse zur *Visitor*-Klassenhierarchie erstellt. Solange sich der vom Compiler akzeptierte Sprachbau nicht ändert (also keine neuen *Node*-Unterklassen hinzugefügt werden müssen), lässt sich neue Funktionalität einfach durch die Definition neuer *NodeVisitor*-Unterklassen ergänzen.

Anwendbarkeit

Das Design Pattern *Visitor (Besucher)* bietet sich an, wenn

- eine Objektstruktur viele Klassen von Objekten mit unterschiedlicher Schnittstelle enthält und Operationen mit diesen Objekten ausgeführt werden sollen, die von deren konkreter Klasse abhängen.
- viele individuelle und verschiedenartige Operationen mit den Objekten einer Objektstruktur durchgeführt und deren Klassen möglichst nicht mit diesen Operationen überfrachtet werden sollen. Das Pattern *Visitor (Besucher)* ermöglicht es, miteinander verwandte Operationen zusammenzuhalten, indem sie in derselben Klasse definiert werden. Wenn die Objektstruktur von mehreren Anwendungen gemeinsam genutzt wird, kann das Pattern *Visitor (Besucher)* dazu beitragen, die entsprechenden Operationen nur in diejenigen Anwendungen zu integrieren, die sie auch benötigen.
- sich die Klassen der Objektstruktur nur selten ändern, aber häufig neue Operationen dafür eingerichtet werden sollen. Das Ändern der Klassen einer Objektstruktur macht eine Neudefinition aller *Visitor*-Schnittstellen erforderlich, was potenziell sehr aufwendig sein kann. Falls sich die Klassen der Objektstruktur öfter ändern, ist es vermutlich besser, die Operationen in den betroffenen Klassen zu definieren.

Struktur

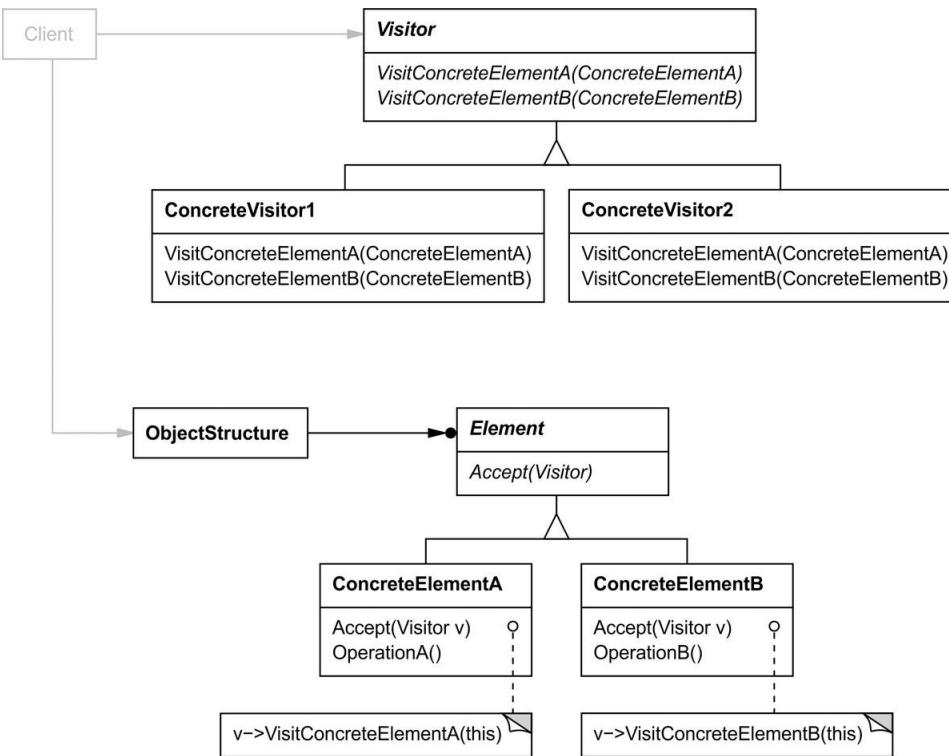


Abb. 5.43: Struktur des Design Patterns Visitor (Besucher)

Teilnehmer

- **Visitor** (NodeVisitor)
 - Deklariert eine visit-Operation für jede ConcreteElement-Klasse in der Objektstruktur. Die Klasse, die den visit-Request an das visitor-Objekt sendet, wird durch den Namen und die Signatur der Operation identifiziert. Auf diese Weise kann das visitor-Objekt die konkrete Klasse des Elements ermitteln, das es gerade besucht. Das visitor-Objekt kann dann mittels der jeweiligen Schnittstelle direkt auf das Element zugreifen.
- **ConcreteVisitor** (TypeCheckingVisitor)
 - Implementiert die vom Visitor-Objekt deklarierten Operationen. Jede Operation implementiert einen der Schritte des für die zugehörige Klasse von Objekten definierten Algorithmus.
- **Element** (Node)

- Definiert eine Accept-Operation, die ein Visitor-Objekt als Argument entgegennimmt.
- **ConcreteElement** (`AssignmentNode`, `VariableRefNode`)
 - Implementiert eine Accept-Operation, die ein Visitor-Objekt als Argument entgegennimmt.
- **ObjectStructure** (`Program`)
 - Kann seine Elemente der Reihe nach benennen.
 - Kann bei Bedarf eine High-Level-Schnittstelle bereitstellen, die es dem Visitor-Objekt gestattet, seine Elemente zu besuchen.
 - Ist entweder vom Typ *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#)) oder eine Objektsammlung (engl. *Collection*), wie z. B. eine Liste (engl. *list*) oder eine Menge (engl. *set*).

Interaktionen

- Ein Client, der das Pattern *Visitor* (*Besucher*) verwendet, muss zunächst ein `ConcreteVisitor`-Objekt erstellen und dann beim Traversieren der Objektstruktur die Elemente mit dem Visitor-Objekt aufsuchen.
- Beim Besuch eines Elements ruft dieses die seiner Klasse zugehörige Visitor-Operation auf. Das Element über gibt dieser Operation sich selbst als Argument, damit das Visitor-Objekt bei Bedarf auf seinen Zustand zugreifen kann.

Das Interaktionsdiagramm in [Abbildung 5.44](#) zeigt das Zusammenwirken von Objektstruktur, Visitor-Objekt und zwei Elementen:

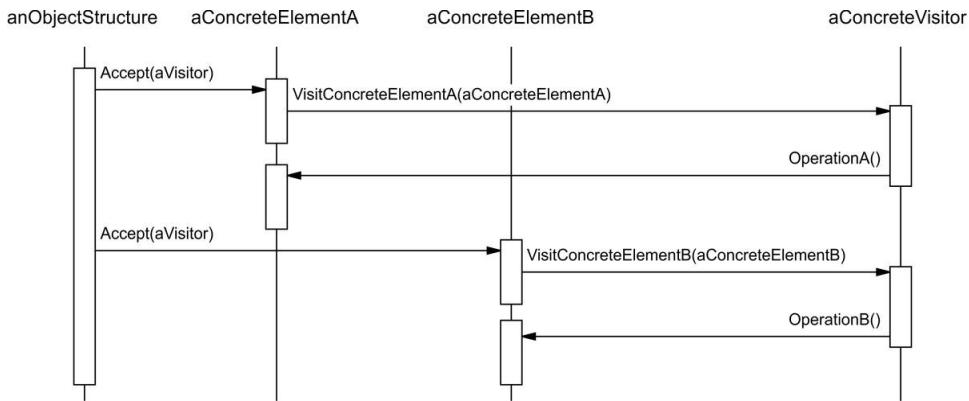


Abb. 5.44: Zusammenwirken von Objektstruktur, visitor-Objekt und zwei Elementen

Konsequenzen

1. *Visitor-Objekte erleichtern das Hinzufügen neuer Operationen.* Visitor-Objekte vereinfachen die Ergänzung von Operationen, die von den Komponenten komplexer Objekte abhängen. Zur Erstellung einer neuen Operation in einer Objektstruktur braucht einfach nur ein Visitor-Objekt hinzugefügt werden. Ist die Funktionalität hingegen auf viele verschiedene Klassen verteilt, muss bei der Definition einer neuen Operation jede einzelne Klasse geändert werden.
2. *Das Pattern Visitor (Besucher) vereint einander ähnliche Operationen und trennt einander unähnliche voneinander.* Ähnliche Verhaltensweisen werden nicht auf die Klassen der Objektstruktur verteilt, sondern werden vielmehr im Visitor-Objekt verortet. Einander unähnliche Gruppen von Verhaltensweisen werden auf jeweils eigene Visitor-Unterklassen verteilt. Das vereinfacht sowohl die Klassen zur Definition der Elemente als auch die in den Visitor-Objekten definierten Algorithmen. Möglicherweise vorhandene, dem Algorithmus eigene Datenstrukturen können im Visitor-Objekt verborgen werden.
3. *Das Hinzufügen neuer ConcreteElement-Klassen ist schwierig.* Das Design Pattern *Visitor (Besucher)* erschwert das Hinzufügen neuer Element-Unterklassen. Jedes neue `ConcreteElement` führt zu einer neuen abstrakten `Visitor`-Operation und einer entsprechenden Implementierung in jeder `ConcreteVisitor`-Klasse. Mitunter kann die `Visitor`-Klasse eine Standardimplementierung bereitstellen, die an die meisten `ConcreteVisitor`-Objekte vererbbar ist, aber das ist eher die Ausnahme als die Regel.

Die zentrale Frage in Bezug auf die Entscheidung, ob das Design Pattern *Visitor (Besucher)* geeignet ist, lautet daher, ob es in den meisten Fällen wahrscheinlich ist, dass die für eine Objektstruktur erstellten Algorithmen oder die diese Struktur bildenden Klassen geändert werden müssen. Die *visitor*-Klassenhierarchie kann schwierig zu warten sein, wenn häufig neue *ConcreteElement*-Klassen hinzugefügt werden. In derartigen Fällen ist es vermutlich sinnvoller, die Operationen in den Klassen zu definieren, aus denen die Struktur aufgebaut ist. Wenn die *Element*-Klassenhierarchie ausgereift ist, aber immer wieder Operationen hinzugefügt oder Algorithmen geändert werden, ist das Pattern *Visitor (Besucher)* bei der Handhabung dieser Anpassungen hilfreich.

4. *Klassenhierarchie-übergreifende Besuche.* Ein *Iterator* (siehe Design Pattern *Iterator (Iterator)*, [Abschnitt 5.4](#)) kann die Objekte einer Struktur bei deren Traversierung aufsuchen, indem es deren Operationen aufruft. Bei Objektstrukturen mit unterschiedlichen Elementtypen funktioniert das jedoch nicht. Die in [Abschnitt 5.4](#) unter »Beispielcode« (Punkt 1) definierte *Iterator*-Schnittstelle kann beispielsweise nur auf Objekte vom Typ *Item* zugreifen:

```
template <class Item>
class Iterator {
    // ...
    Item currentItem() const;
};
```

Das bedeutet, dass alle Objekte, die das *Iterator*-Objekt aufsuchen kann, die gemeinsame Basisklasse *Item* besitzen.

Diese Einschränkung gilt für ein *Visitor*-Objekt nicht. Es kann auch Objekte aufsuchen, die keine gemeinsame Basisklasse besitzen. Der *Visitor*-Schnittstelle können Objekte beliebigen Typs hinzugefügt werden.

Beispielsweise brauchen im folgenden Code *MyType* und *YourType* in keiner Weise durch Vererbung miteinander verknüpft zu sein:

```
class Visitor {
public:
    // ...
    void VisitMyType(MyType* );
    void VisitYourType(YourType* );
};
```

5. *Ansammeln von Zuständen.* *Visitor*-Objekte können beim Aufsuchen jedes einzelnen Elements einer Objektstruktur die verschiedenen Zustände

zusammensammeln. Ohne visitor-Objekte würde der Zustand in Form zusätzlicher Argumente an die für die Traversierung zuständigen Operationen übergeben werden oder die Argumente würden als globale Variablen in Erscheinung treten.

6. *Verletzung der Kapselung.* Der Ansatz des Design Patterns *Visitor (Besucher)* geht davon aus, dass die `ConcreteElement`-Schnittstelle so leistungsfähig ist, dass visitor-Objekte ihre Aufgabe auch erledigen können. Das führt oft dazu, dass das Pattern die Bereitstellung öffentlich zugänglicher Operationen erzwingt, die auf den internen Zustand eines Elements zugreifen und dadurch möglicherweise die Kapselung verletzen.

Implementierung

Jede Objektstruktur besitzt eine ihr zugehörige visitor-Klasse. Diese abstrakte visitor-Klasse deklariert eine `VisitConcreteElement`-Operation für jede `ConcreteElement`-Klasse der Objektstruktur. Jede `Visit`-Operation der visitor-Klasse deklariert ihr Argument als ein bestimmtes `ConcreteElement`-Objekt, das es dem visitor-Objekt erlaubt, direkt auf die `ConcreteElement`-Schnittstelle zuzugreifen. `ConcreteVisitor`-Klassen überschreiben die einzelnen `visit`-Operationen, um das visitor-spezifische Verhalten der entsprechenden `ConcreteElement`-Klasse zu implementieren.

Die visitor-Klasse kann in C++ wie folgt deklariert werden:

```
class Visitor {
public:
    virtual void VisitElementA(ElementA* );
    virtual void VisitElementB(ElementB* );

    // usw. mit den anderen konkreten Elementen

protected:
    Visitor();
};
```

Jede `ConcreteElement`-Klasse implementiert eine `Accept`-Operation, die wiederum die entsprechende `visit...-`Operation bei dem für das `ConcreteElement`-Objekt zuständigen visitor aufruft. Die letztendlich aufgerufene Operation hängt also sowohl von der `Element`- als auch von der `Visitor`-Klasse ab.

Hinweis

An dieser Stelle könnte man auch eine Funktionsüberladung verwenden, um den Operationen denselben einfachen Namen (wie etwa `visit`) zu geben, denn sie unterscheiden sich ja bereits durch die übergebenen Parameter. Eine solche Überladung hat Vor-und Nachteile: Einerseits unterstreicht sie die Tatsache, dass beide Operationen dieselbe Analyse umfassen, wenngleich mit unterschiedlichen Parametern. Andererseits wird bei der Lektüre des Codes weniger deutlich, was aufseiten des Aufrufers vor sich geht. Letzten Endes ist die Funktionsüberladung also auch ein wenig Geschmackssache.

Die konkreten Elemente sind wie folgt deklariert:

```
class Element {  
public:  
    virtual ~Element();  
    virtual void Accept(Visitor&) = 0;  
  
protected:  
    Element();  
};  
  
class ElementA : public Element {  
public:  
    ElementA();  
    virtual void Accept(Visitor& v) {  
        v.VisitElementA(this);  
    }  
};  
  
class ElementB : public Element {  
public:  
    ElementB();  
    virtual void Accept(Visitor& v) {  
        v.VisitElementB(this);  
    }  
};
```

Eine `CompositeElement`-Klasse könnte `Accept` auf diese Weise implementieren:

```
class CompositeElement : public Element {  
public:  
    virtual void Accept(Visitor&);  
private:  
    List<Element*>* _children;  
};
```

```

void CompositeElement::Accept (Visitor& v) {
    ListIterator<Element*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Accept(v);
    }
    v.VisitCompositeElement(this);
}

```

Hier treten zwei weitere Implementierungsaspekte bei der Anwendung des Design Patterns *Visitor (Besucher)* in Erscheinung:

1. *Doppelte Verteilung*. Im Grunde ermöglicht das Pattern *Visitor (Besucher)* das Hinzufügen von Operationen zu einer Klasse, ohne sie dadurch zu verändern. Das Pattern erreicht dies durch die Anwendung einer Technik, die als »doppelte Verteilung« (engl. *double-dispatch*) bezeichnet wird. Dabei handelt es sich um ein wohlbekanntes Verfahren, das einige Programmiersprachen sogar direkt unterstützen (z. B. CLOS). Sprachen wie C++ und Smalltalk unterstützen dagegen lediglich eine »einfache Verteilung« (engl. *single-dispatch*).

In Sprachen mit einfacher Verteilung bestimmen zwei Kriterien, welche Operation einen Request bearbeitet: der Name des Requests und der Typ des Empfängers. Die Operation, die beispielsweise ein `GenerateCode`-Request aufruft, hängt vom Typ des befragten `Node`-Objekts ab. In C++ bewirkt der `GenerateCode`-Request bei einer `VariableRefNode`-Instanz den Aufruf von `VariableRefNode::GenerateCode` (der den Code für den Verweis auf eine Variable erzeugt). Wird der `GenerateCode`-Request an eine `AssignmentNode`-Instanz gesendet, wird stattdessen `AssignmentNode::GenerateCode` aufgerufen (wodurch der Code für eine Zuweisung erzeugt wird). Die ausgeführte Operation hängt also sowohl von der Art des Requests als auch vom Typ des Empfängers ab.

»Doppelte Verteilung« bedeutet einfach nur, dass die ausgeführte Operation von der Art des Requests und den Typen von *zwei* Empfängern abhängt. `Accept` ist eine Operation mit doppelter Verteilung. Ihre Bedeutung hängt von zwei Typen ab: dem des `Visitor`- und dem des `Element`-Objekts. Die doppelte Verteilung ermöglicht es `Visitor`-Objekten, bei jeder Klasse von Elementen verschiedene Operationen anzufordern.

Hierbei handelt es sich um das Schlüsselkonzept des Patterns *Visitor (Besucher)*: Die ausgeführte Operation ist sowohl vom Typ des `Visitor`-Objekts als auch vom Typ des besuchten Elements abhängig. Anstatt die

Operationen statisch in die Schnittstelle des Elements einzubinden, können mehrere Operationen in einem `Visitor`-Objekt zusammengelegt und `Accept` verwendet werden, um die Anbindung zur Laufzeit vorzunehmen. Das Erweitern der Schnittstelle eines Elements läuft darauf hinaus, statt lauter neuer Unterklassen des Elements eine neue `Visitor`-Schnittstelle zu definieren.

Hinweis

Wenn es schon eine *doppelte* Verteilung gibt, warum dann nicht auch eine *drei-* oder *vierfache*? Tatsächlich handelt es sich bei der doppelten Verteilung nur um einen Sonderfall der »mehreren Verteilung« (engl. *multiple-dispatch*), bei der die Operation anhand einer beliebigen Anzahl von Typen ausgewählt wird. (CLOS unterstützt tatsächlich auch die mehrfache Verteilung.) In Programmiersprachen, die doppelte oder mehrfache Verteilung unterstützen, gibt es einen geringeren Bedarf für das Design Pattern *Visitor* (*Besucher*).

2. *Wer ist für die Traversierung der Objektstruktur verantwortlich?* Ein `visitor`-Objekt muss jedes einzelne Element der Objektstruktur aufsuchen. Nun stellt sich die Frage: Wie gelangt es dort hin? Die Zuständigkeit für die Traversierung könnte an drei Stellen implementiert werden: in der Objektstruktur, im `visitor`-Objekt oder in einem eigenen `Iterator`-Objekt (siehe Design Pattern *Iterator* (*Iterator*), [Abschnitt 5.4](#)).

Häufig ist die Objektstruktur für die Iteration verantwortlich. Eine Objektsammlung durchläuft einfach ihre Elemente und ruft bei jedem die `Accept`-Operation auf. Ein Composite-Objekt (siehe Design Pattern *Composite (Kompositum)*, [Abschnitt 4.3](#)) traversiert die eigenen Elemente üblicherweise, indem jede `Accept`-Operation die Kindelemente durchläuft und bei jedem `Accept` rekursiv aufruft.

Eine weitere Möglichkeit zum Aufsuchen der Elemente ist die Verwendung eines Iterators. In C++ könnte ein interner oder ein externer Iterator genutzt werden, je nachdem, was verfügbar oder schneller ist. In Smalltalk wird für gewöhnlich ein interner Iterator eingesetzt, der `do:` und einen Block verwendet. Da interne Iteratoren durch die Objektstruktur implementiert werden, bedeutet die Verwendung eines solchen Iterators mehr oder weniger, die Zuständigkeit für die Iteration der Objektstruktur zuzuweisen. Der größte Unterschied besteht

darin, dass ein interner Iterator keine doppelte Verteilung bewirkt: Es wird eine `Visitor`-Operation mit einem `Element` als Argument statt einer `Element`-Operation mit dem `visitor` als Argument aufgerufen. Es ist jedoch leicht möglich, das Pattern `Visitor (Besucher)` mit einem internen Iterator zu nutzen, sofern die `visitor`-Operation die `Element`-Operation aufruft, ohne dabei Rekursion einzusetzen.

Man könnte den Traversierungsalgorithmus sogar im `visitor`-Objekt implementieren, allerdings würde das zur Duplizierung des Traversierungscodes bei allen `ConcreteVisitor`-Objekten der zugehörigen `ConcreteElement`-Objekte führen. Der Hauptgrund, die Verantwortung für die Traversierung dem `visitor`-Objekt zu übertragen, ist die Implementierung einer besonders komplizierten Traversierung, die von den Resultaten der Operationen abhängt, die mit der Objektstruktur durchgeführt werden. Ein Beispiel für einen solchen Fall folgt im nächsten Abschnitt.

Beispielcode

Da die Design Patterns `Visitor (Besucher)` und `Composite (Kompositum, siehe Abschnitt 4.3)` gewöhnlich miteinander verknüpft sind, wird hier die im Beispielcode des [Abschnitts 4.3](#) definierte `Equipment`-Klasse zur Illustration des Patterns `Visitor (Besucher)` verwendet. Es dient an dieser Stelle zur Definition von Operationen zur Berechnung des Materialbestands und der Gesamtkosten für ein Gerät. Die `Equipment`-Klassen sind so einfach, dass der Einsatz des Patterns `Visitor (Besucher)` eigentlich gar nicht notwendig ist, aber sie erleichtern es zu erkennen, was bei der Implementierung des Patterns zu beachten ist.

Hier ist noch einmal die `Equipment`-Klasse des Design Patterns `Composite (Kompositum, siehe Abschnitt 4.3)`, die um eine `Accept`-Operation erweitert wurde, damit sie mit dem Pattern `Visitor (Besucher)` zusammenarbeiten kann:

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Accept(EquipmentVisitor&);
```

```

protected:
    Equipment(const char* );
private:
    const char* _name;
};

```

Die `Equipment`-Operationen liefern die Attribute eines Geräts zurück, wie z. B. Leistungsaufnahme und Preis. Die Operationen werden entsprechend der Geräteart (wie etwa Gehäuse, Laufwerke und Platinen) in Unterklassen neu definiert.

Die abstrakte Klasse für alle `Visitor`-Objekte der `Equipment`-Klasse besitzt für jede `Equipment`-Unterkelas eine virtuelle Funktion. Standardmäßig bewirken all diese virtuellen Funktionen gar nichts:

```

class EquipmentVisitor {
public:
    virtual ~EquipmentVisitor();

    virtual void VisitFloppyDisk(FloppyDisk* );
    virtual void VisitCard(Card* );
    virtual void VisitChassis(Chassis* );
    virtual void VisitBus(Bus* );

    // usw. für weitere konkrete Equipment-Unterklassen

protected:
    Equipmentvisitor();
};

```

Die `Equipment`-Unterklassen definieren `Accept` im Prinzip auf dieselbe Weise: `Accept` ruft diejenige `EquipmentVisitor`-Operation auf, die zu der Klasse gehört, die den `Accept`-Request empfangen hat, und zwar so:

```

void FloppyDisk::Accept (EquipmentVisitor& visitor) {
    visitor.VisitFloppyDisk(this);
}

```

`Equipment`-Objekte, die andere `Equipment`-Objekte enthalten (insbesondere `CompositeEquipment`-Unterklassen im Pattern *Composite (Kompositum*, siehe [Abschnitt 4.3](#))), implementieren `Accept`, indem sie über ihre Kindobjekte iterieren und bei jedem einzelnen von ihnen `Accept` aufrufen. Danach wird wie gewohnt die `Visit`-Operation aufgerufen. `Chassis::Accept` könnte beispielsweise alle im Gehäuse befindlichen Objekte wie folgt durchlaufen:

```

void Chassis::Accept (EquipmentVisitor& visitor) {

```

```

    for (
        ListIterator<Equipment*> i(_parts);
        !i.IsDone();
        i.Next()
    ) {
        i.CurrentItem()->Accept(visitor);
    }
    visitor.VisitChassis(this);
}

```

In `EquipmentVisitor`-Unterklassen werden spezielle Algorithmen der `Equipment`-Struktur definiert. Das `PricingVisitor`-Objekt errechnet den Preis der `Equipment`-Struktur. Es ermittelt die Nettopreise der einfacheren Geräte (z. B. Disketten) und die Rabattpreise für die aus mehreren Komponenten zusammengesetzten Geräte (z. B. Gehäuse und Busse):

```

class PricingVisitor : public EquipmentVisitor {
public:
    PricingVisitor();

    Currency& GetTotalPrice();

    virtual void VisitFloppyDisk(FloppyDisk* );
    virtual void VisitCard(Card* );
    virtual void VisitChassis(Chassis* );
    virtual void VisitBus(Bus* );
    // ...

private:
    Currency _total;
};

void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _total += e->NetPrice();
}

void PricingVisitor::VisitChassis (Chassis* e) {
    _total += e->DiscountPrice();
}

```

`PricingVisitor` errechnet so die Gesamtkosten aller Knoten in der `Equipment`-Struktur. Beachtenswert ist hier, dass das `PricingVisitor`-Objekt die Festlegung der Bepreisung einer `Equipment`-Klasse der jeweils zugehörigen Memberfunktion überlässt. Außerdem lassen sich die Preisvorschriften einer `Equipment`-Struktur durch einfaches Ändern der `PricingVisitor`-Klasse modifizieren.

Eine `Visitor`-Klasse zum Ermitteln des Materialbestands kann folgendermaßen definiert werden:

```

class Inventory-Visitor : public EquipmentVisitor {
public:
    InventoryVisitor();
    Inventory& GetInventory();

    virtual void VisitFloppyDisk(FloppyDisk* );
    virtual void VisitCard(Card* );
    virtual void VisitChassis(Chassis* );
    virtual void VisitBus(Bus* );
    // ...

private:
    Inventory _inventory;
};

```

Die `InventoryVisitor`-Klasse beinhaltet die Gesamtkosten aller Gerätetypen in der Objektstruktur. Sie verwendet eine `Inventory`-Klasse, die eine Schnittstelle zum Hinzufügen von Geräten festlegt (auf deren Definition hier nicht weiter eingegangen wird):

```

void InventoryVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _inventory.Accumulate(e);
}

void InventoryVisitor::VisitChassis (Chassis* e) {
    _inventory.Accumulate(e);
}

```

Und so wird ein `InventoryVisitor`-Objekt zusammen mit einer `Equipment`-Struktur eingesetzt:

```

Equipment* component;
InventoryVisitor visitor;

component->Accept(visitor);
cout << "Inventory "
    << component->Name()
    << visitor.GetInventory();

```

Im Folgenden wird aufgezeigt, wie das Smalltalk-Beispiel für das Design Pattern *Interpreter* (*Interpreter*, siehe [Abschnitt 5.3](#)) mithilfe des Patterns *Visitor* (*Besucher*) implementiert werden kann. Dieses Beispiel ist wie das vorhergehende so kurz und knapp, dass mit dem Einsatz des Patterns *Visitor* (*Besucher*) nur wenig erreicht wird, aber es demonstriert die Verwendungsweise sehr schön. Außerdem zeigt es eine Situation auf, in der das `visitor`-Objekt für die Iteration verantwortlich ist.

Die Objektstruktur eines regulären Ausdrucks soll aus vier Klassen bestehen, die

allesamt die Methode `accept:` besitzen, die das Visitor-Objekt als Argument entgegennimmt. In der `SequenceExpression`-Klasse sieht die `accept:-Methode` so aus:

```
accept: aVisitor
    ^ aVisitor visitSequence: self
```

In der `RepeatExpression`-Klasse versendet die `accept:-Methode` eine `visitRepeat:-Nachricht`. In der `AlternationExpression`-Klasse verschickt sie eine `visitAlternation:-Nachricht` und in der `LiteralExpression`-Klasse schließlich eine `visitLiteral:-Nachricht`.

Die vier Klassen müssen außerdem Zugriffsfunktionen für das Visitor-Objekt bereitstellen. In der `SequenceExpression`-Klasse sind das `expression1` und `expression2`, in der `AlternationExpression`-Klasse `alternative1` und `alternative2`, in der `RepeatExpression`-Klasse ist es `repetition` und in der `LiteralExpression`-Klasse `components`.

Die `ConcreteVisitor`-Klasse heißt `REMatchingVisitor`. Sie ist für die Traversierung verantwortlich, weil der Traversierungsalgorithmus unregelmäßig vorgeht. Die größte Unregelmäßigkeit ist, dass ein `RepeatExpression`-Objekt seine Komponenten mehrmals traversiert. Die `REMatchingVisitor`-Klasse besitzt die Instanzvariable `inputState`. Ihre Methoden entsprechen im Wesentlichen den `match:-Methoden` der `Expression`-Klassen beim Design Pattern *Interpreter* (*Interpreter*, siehe [Abschnitt 5.3](#)), mit der Ausnahme, dass das Argument namens `inputState` durch den abzugleichenden `Expression`-Knoten ersetzt wird. Diese Methoden geben jedoch nach wie vor denjenigen Satz von Eingabestreams zurück, auf den der Ausdruck zur Beschreibung des aktuellen Zustands passen würde:

```
visitSequence: sequenceExp
    inputState := sequenceExp expression1 accept: self.
    ^ sequenceExp expression2 accept: self.

visitRepeat: repeatExp
    | finalState |
    finalState := inputState copy.
    [inputState isEmpty]
        whileFalse:
            [inputState := repeatExp repetition accept: self.
             finalState addAll: inputState].
    ^ finalState

visitAlternation: alternateExp
    | finalState originalState |
    originalState := inputState.
```

```

finalState := alternateExp alternative1 accept: self.
inputState := originalState.
finalState addAll: (alternateExp alternative2 accept: self).
^ finalState

visitLiteral: literalExp
| finalState tStream |
finalState := Set new.
inputState
do:
    [:stream | tStream := stream copy.
    (tStream nextAvailable:
        literalExp components size
    ) = literalExp components
        ifTrue: [finalState add: tStream]
].
^ finalState

```

Praxisbeispiele

Der **Smalltalk-80-Compiler** besitzt eine `Visitor`-Klasse namens `ProgramNodeEnumerator`, die vornehmlich für Algorithmen zur Analyse von Quellcode verwendet wird. Sie wird nicht zur Codeerzeugung oder zur Optimierung der Quelltextausgabe eingesetzt, obwohl das durchaus möglich wäre.

IRIS Inventor [Str93] ist ein Toolkit zum Entwickeln von 3-D-Grafikanwendungen. Inventor stellt eine dreidimensionale Szene als eine Hierarchie von Knoten dar, von denen ein jeder entweder ein geometrisches Objekt oder aber eins der Attribute eines solchen Objekts repräsentiert. Operationen wie das Zeichnen einer Szene oder die Zuordnung von Eingabeereignissen bedingen verschiedenste Traversierungsarten dieser Hierarchie. Inventor erledigt diese Aufgabe mittels `Visitor`-Objekten, die hier als »Actions« bezeichnet werden. Es gibt `Visitor`-Objekte zum Zeichnen, zur Handhabung von Ereignissen, zum Suchen und Archivieren oder zum Ermitteln von Begrenzungsrahmen (engl. *bounding boxes*).

Inventor implementiert in C++ einen Mechanismus zur Bereitstellung der doppelten Verteilung, um das Hinzufügen neuer Knoten zu erleichtern. Dieser Mechanismus beruht auf zur Laufzeit vorhandene Typinformationen und einer zweidimensionalen Tabelle, in der die Zeilen `Visitor`-Objekte und die Spalten `Node`-Objekte darstellen. In den Zellen der Tabelle werden Zeiger auf Funktionen gespeichert, die `Visitor`- und `Node`-Klassen miteinander verknüpfen.

Der Begriff »Visitor« wurde von Mark Linton in der Spezifikation des X-

Konsortiums zum **Fresco-Application-Toolkit** geprägt.

Verwandte Patterns

Composite (Kompositum, siehe [Abschnitt 4.3](#)): Das Design Pattern *Visitor (Besucher)* kann eingesetzt werden, um Operationen auf eine durch das Pattern *Composite (Kompositum)* definierte Objektstruktur anzuwenden.

Interpreter (Interpreter, siehe [Abschnitt 5.3](#)): Das Design Pattern *Visitor (Besucher)* kann verwendet werden, um die Interpretation auszuführen.

5.12 Weitere Erläuterungen zu den Verhaltensmustern

5.12.1 Variieren der Kapselung

Viele Verhaltensmuster beschäftigen sich mit dem Variieren der Kapselung. Wenn sich ein bestimmter Aspekt eines Programms häufig ändert, definieren solche Verhaltensmuster ein Objekt, das diesen Aspekt kapselt. Andere Programmteile können dann mit diesem Objekt zusammenarbeiten, wenn sie in irgendeiner Weise von diesem Aspekt abhängig sind. Für gewöhnlich definiert solch ein Pattern eine abstrakte Klasse, die das kapselnde Objekt beschreibt, von dem es auch seine Bezeichnung ableitet. Zum Beispiel kapselt

- ein Strategy-Objekt einen Algorithmus (siehe Design Pattern *Strategy (Strategie)*, [Abschnitt 5.9](#)),
- ein State-Objekt ein zustandsabhängiges Verhalten (siehe Design Pattern *State (Zustand)*, [Abschnitt 5.8](#)),
- ein Mediator-Objekt das Protokoll zwischen Objekten (siehe Design Pattern *Mediator (Vermittler)*, [Abschnitt 5.5](#)), und
- ein Iterator-Objekt die Art und Weise, wie der Zugriff auf die Komponenten eines aggregierten Objekts erfolgt und wie diese traversiert werden (siehe Design Pattern *Iterator (Iterator)*, [Abschnitt 5.4](#)).

Hinweis

Diese Vorgänge sind nicht auf Verhaltensmuster beschränkt. Die Patterns *Abstract Factory* (*Abstrakte Fabrik*, siehe [Abschnitt 3.1](#)), *Builder* (*Erbauer*, siehe [Abschnitt 3.2](#)) und *Prototype* (*Prototyp*, siehe [Abschnitt 3.4](#)) kapseln Informationen über die Erzeugung von Objekten. *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#)) kapselt Zuständigkeiten, die einem Objekt hinzugefügt werden können. Und das Design Pattern *Bridge* (*Brücke*, siehe [Abschnitt 4.2](#)) trennt eine Abstrahierung von seiner Implementierung und ermöglicht es, sie unabhängig voneinander zu variieren.

Diese Patterns beschreiben diejenigen Aspekte eines Programms, die sich wahrscheinlich ändern werden. Dabei kennen die meisten Patterns zwei Arten von Objekten: das neue Objekt, das den fraglichen Aspekt kapselt, und das bereits vorhandene Objekt, das vom neuen Gebrauch macht. Normalerweise wäre die Funktionalität neuer Objekte integraler Bestandteil der vorhandenen Objekte – wäre da nicht das Pattern. Der Code eines *Strategy*-Objekts beispielsweise wäre vermutlich in dessen *Context*-Objekt zu finden, und der Code eines *State*-Objekts wäre unmittelbar in dessen *Context*-Objekt implementiert.

Aber nicht alle Verhaltensmuster teilen die Funktionalität in dieser Weise auf. Das Pattern *Chain of Responsibility* (*Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) kann beispielsweise mit einer beliebigen Anzahl von Objekten umgehen (daher auch der Begriff »Chain«, also »Kette«), die alle schon im System vorhanden sein können.

Das Design Pattern *Chain of Responsibility* (*Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) veranschaulicht einen weiteren Unterschied zwischen den verschiedenen Verhaltensmustern: Statische Kommunikationsverbindungen zwischen den Klassen gibt es nicht bei allen Verhaltensmustern. *Chain of Responsibility* (*Zuständigkeitskette*) schreibt die Kommunikation zwischen einer nach oben offenen Anzahl von Objekten vor. Andere Design Patterns ziehen dazu Objekte heran, die als Argumente übergeben werden.

5.12.2 Objekte als Argumente

Einige Patterns setzen ein Objekt ein, das *immer* als Argument verwendet wird. Eins davon ist *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)). Als Argument einer polymorphen *Accept*-Operation des besuchten Objekts dient das *visitor*-Objekt selbst – es wird jedoch nie als Bestandteil solcher Objekte betrachtet, obwohl die herkömmliche Alternative zu diesem Pattern die Verteilung des *visitor*-Codes auf die Klassen der Objektstruktur wäre.

Andere Patterns definieren Objekte, die als »magische Tokens« fungieren sollen und herumgereicht werden. Zu einem späteren Zeitpunkt können sie dann zu diesem Zweck herangezogen werden. Sowohl das Design Pattern *Command* (*Befehl*, siehe [Abschnitt 5.2](#)) als auch *Memento* (*Memento*, siehe [Abschnitt 5.6](#)) sind dieser Kategorie zuzuordnen. Bei *Command* (*Befehl*) handelt es sich bei dem Token um einen Request – bei *Memento* (*Memento*) repräsentiert es den internen Zustand eines Objekts zu einem bestimmten Zeitpunkt. In beiden Fällen kann das Token eine komplexe interne Darstellung besitzen, die der Client jedoch nicht wahrnimmt. Aber auch hier gibt es Unterschiede: Beim Design Pattern *Command* (*Befehl*) spielt Polymorphismus eine wichtige Rolle, denn die Ausführung des Command-Objekts ist eine polymorphe Operation. Im Gegensatz dazu ist die *Memento*-Schnittstelle so beschränkt, dass ein Memento nur als Wert übergeben werden kann. Es ist daher wahrscheinlich, dass es seinen Clients gar keine polymorphen Operationen anbietet.

5.12.3 Kommunikation: Kapseln oder Verteilen?

Die Design Patterns *Mediator* (*Vermittler*, siehe [Abschnitt 5.5](#)) und *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)) konkurrieren miteinander. Der Unterschied zwischen den beiden besteht darin, dass Letzteres die Kommunikation durch den Einsatz von *Observer*- und *Subject*-Objekten verteilt, während ein *Mediator*-Objekt die Kommunikation zwischen anderen Objekten kapselt.

Beim Pattern *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)) gibt es kein einzelnes Objekt das eine Konsistenzbedingung kapselt. Stattdessen müssen *Observer* und *Subjekt* zusammenarbeiten, um der Bedingung zu genügen. Die Art der Kommunikation wird dadurch bestimmt, wie *Observer* und *Subjekte* miteinander verknüpft sind: Einzelne *Subjekte* besitzen normalerweise viele *Observer*, und mitunter ist der *Observer* eines *Subjekts* selbst *Subjekt* eines anderen *Observers*. Das Design Pattern *Mediator* (*Vermittler*, siehe [Abschnitt 5.5](#)) zentralisiert die Kommunikation, anstatt sie zu verteilen. Es weist die Verantwortung dafür, einer Bedingung zu genügen, unmittelbar dem *Mediator* zu.

In der Regel ist es einfacher, wiederverwendbare *Observer* und *Subjekte* zu erzeugen als wiederverwendbare *Mediator*-Objekte. Das Pattern *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)) begünstigt die Aufteilung von Code und die Entkopplung von *Observer* und *Subjekt*, was zu feiner unterteilten Klassen führt, die besser zur Wiederverwendung geeignet sind.

Andererseits ist der Informationsfluss beim *Mediator* (*Vermittler*, siehe [Abschnitt 5.5](#)) leichter verständlich als beim *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)).

Observer und Subjekte werden normalerweise kurz nach ihrer Erzeugung miteinander verknüpft, und später im Programm ist diese Verbindung schwer erkennbar. Ist man mit dem Design Pattern *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)) vertraut, wird klar, dass es von Bedeutung ist, wie Observer und Subjekte miteinander verbunden sind und nach welchen Verbindungen man suchen muss. Die mit dem Pattern *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)) einhergehenden Umwege machen ein System allerdings schwerer verständlich.

In Smalltalk können Observer mittels Requests parametrisiert werden, um auf den Subject-Zustand zuzugreifen. Sie sind daher noch besser wiederverwendbar als in C++. Das lässt das Pattern *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)) in Smalltalk attraktiver erscheinen als *Mediator* (*Vermittler*, siehe [Abschnitt 5.5](#)) – in Fällen, in denen ein C++-Programmierer *Mediator* verwendet, wird ein Smalltalk-Programmierer daher oft *Observer* bevorzugen.

5.12.4 Absender und Empfänger entkoppeln

Wenn zusammenarbeitende Objekte direkt aufeinander verweisen, werden sie dadurch voneinander abhängig – und das kann negative Auswirkungen auf die Schichtenbildung und die Wiederverwendbarkeit eines Systems haben. Die Design Patterns *Command* (*Befehl*, siehe [Abschnitt 5.2](#)), *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)), *Mediator* (*Vermittler*, siehe [Abschnitt 5.5](#)) und *Chain of Responsibility* (*Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) nehmen die Entkopplung von Absendern und Empfängern in Angriff, gehen dabei aber verschiedene Kompromisse ein.

Das Pattern *Command* (*Befehl*, siehe [Abschnitt 5.2](#)) unterstützt das Entkoppeln durch Verwendung eines Command-Objekts, das die Verbindung zwischen Absender und Empfänger definiert:

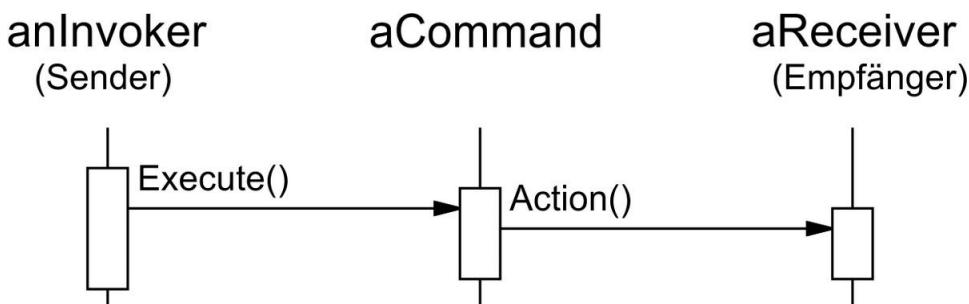


Abb. 5.45: Entkopplung mittels command-Objekt

Das Command-Objekt stellt eine einfache Schnittstelle zum Verfassen von Requests bereit (nämlich die Execute-Operation). Die Absender-Empfänger-Beziehung in einem eigenen Objekt zu definieren ermöglicht es dem Absender, mit verschiedenen Empfängern zusammenzuarbeiten. Der Absender bleibt von den Empfängern entkoppelt, wodurch er leicht wiederverwendbar ist. Außerdem kann das Command-Objekt wiederverwendet werden, um einen Empfänger mit verschiedenen Absendern zu parametrisieren. Das Pattern *Command (Befehl)*, siehe [Abschnitt 5.2](#)) benötigt eigentlich eine Unterklasse für jede Absender-Empfänger-Beziehung, es beschreibt jedoch auch Implementierungsverfahren, die ohne Unterklassen auskommen.

Das Pattern *Observer (Beobachter)*, siehe [Abschnitt 5.7](#)) entkoppelt Absender (Subjekte) und Empfänger (Observer) durch die Definition einer Schnittstelle zum Mitteilen von Änderungen in den Subjekten. Die Anbindung von Absendern an Empfänger ist beim Pattern *Observer (Beobachter)*, siehe [Abschnitt 5.7](#)) etwas lockerer als beim Pattern *Command (Befehl)*, siehe [Abschnitt 5.2](#)), denn ein Subjekt kann mehrere Observer besitzen, deren Anzahl sich zur Laufzeit außerdem ändern kann.

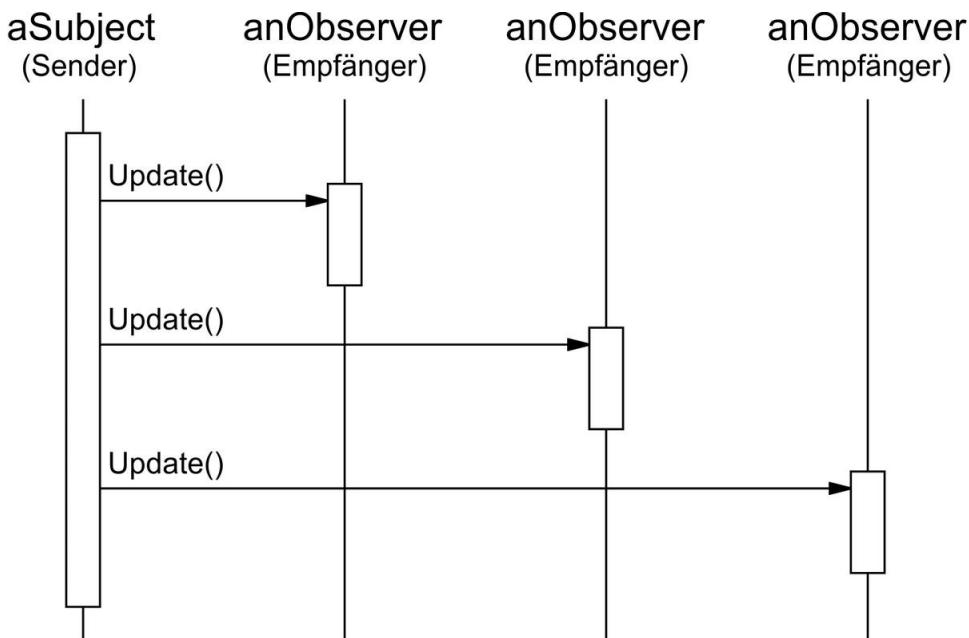


Abb. 5.46: Entkopplung beim Design Pattern *Observer*

Die Subject- und observer-Schnittstellen sind beim Design Pattern *Observer (Beobachter)*, siehe [Abschnitt 5.7](#)) für das Mitteilen von Änderungen ausgelegt – insofern ist das Pattern *Observer (Beobachter)* am besten zum Entkoppeln von Objekten geeignet, wenn Abhängigkeiten zwischen ihnen bestehen.

Das Design Pattern *Mediator* (*Vermittler*, siehe [Abschnitt 5.5](#)) entkoppelt Objekte dadurch, dass sie über ein Mediator-Objekt nur indirekt aufeinander verweisen:

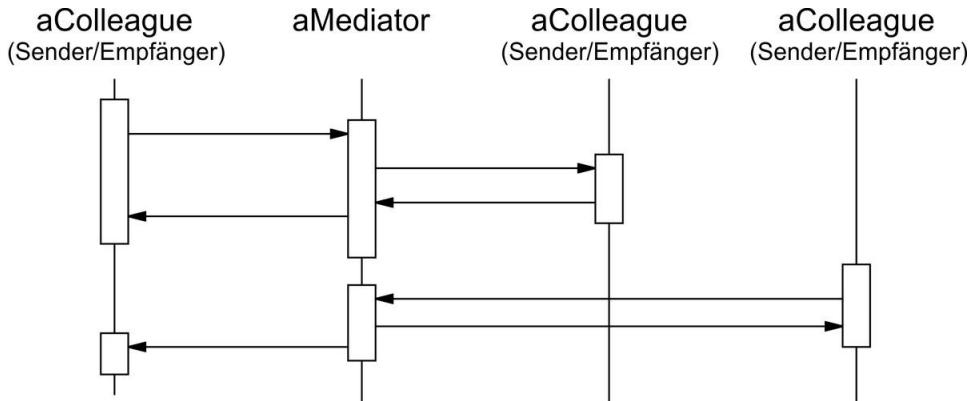


Abb. 5.47: Entkopplung durch ein Mediator-Objekt

Ein Mediator-Objekt vermittelt Requests zwischen Colleague-Objekten und zentralisiert deren Kommunikation. Infolgedessen können Colleague-Objekte nur über die Mediator-Schnittstelle miteinander kommunizieren. Da diese Schnittstelle unveränderlich ist, muss das Mediator-Objekt gegebenenfalls einen eigenen Mechanismus zum Weiterleiten von Requests implementieren, um flexibel zu bleiben. Dabei ist es möglich, Requests zu kodieren und Argumente zu bündeln, so dass Colleague-Objekten ein unbeschränkter Satz von Operationen zur Verfügung steht.

Das Design Pattern *Mediator* (*Vermittler*, siehe [Abschnitt 5.5](#)) kann die Notwendigkeit verringern, Unterklassen erstellen zu müssen, da es das Kommunikationsverhalten in einer Klasse zusammenfasst, anstatt es auf Unterklassen zu verteilen. Allerdings beeinträchtigen Ad-hoc-Mechanismen zur Request-Weiterleitung oft die Typsicherheit.

Das Design Pattern *Chain of Responsibility* (*Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) schließlich entkoppelt den Absender vom Empfänger, indem ein Request an eine Kette potenzieller Empfänger weitergegeben wird:

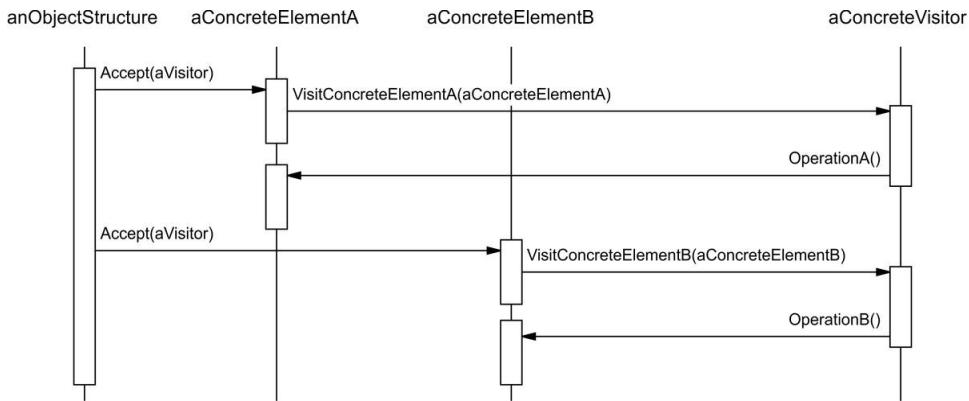


Abb. 5.48: Weitergabe eines Requests

Da die Schnittstelle zwischen Absender und Empfänger unveränderlich ist, erfordert das Pattern *Chain of Responsibility* (*Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) möglicherweise einen eigenen Mechanismus zu Request-Weiterleitung. Daher weist es hinsichtlich der Typsicherheit dieselben Nachteile auf wie das Pattern *Mediator* (*Vermittler*, siehe [Abschnitt 5.5](#)). Das Design Pattern *Chain of Responsibility* (*Zuständigkeitskette*) ist gut dazu geeignet, Absender und Empfänger zu entkoppeln, wenn die Zuständigkeitskette bereits Bestandteil der Struktur des Systems ist und eins von mehreren Objekten den Request möglicherweise handhaben kann. Dass die Kette leicht geändert oder erweitert werden kann, bietet zusätzliche Flexibilität.

5.12.5 Zusammenfassung

Von wenigen Ausnahmen abgesehen, ergänzen und stärken Verhaltensmuster einander. Eine der Klassen in einer Zuständigkeitskette wird wahrscheinlich wenigstens eine Anwendung des Patterns *Template Method* (*Schablonenmethode*, siehe [Abschnitt 5.10](#)) enthalten. Die Schablonenmethode kann primitive Operationen verwenden, um festzustellen, ob ein Objekt einen Request bearbeiten soll und dasjenige Objekt auswählen, an das der Request weitergeleitet wird. Die Zuständigkeitskette kann das Pattern *Command* (*Befehl*, siehe [Abschnitt 5.2](#)) verwenden, um Requests als Objekte zu repräsentieren. Ein *Interpreter* (*Interpreter*, siehe [Abschnitt 5.3](#)) kann das Pattern *State* (*Zustand*, siehe [Abschnitt 5.8](#)) zum Festlegen von Parsing-Kontexten einsetzen. Ein *Iterator* (*Iterator*, siehe [Abschnitt 5.4](#)) ist in der Lage, Objektsammlungen zu traversieren, und das Pattern *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)) ermöglicht es, Operationen auf die Elemente einer solchen Sammlung anzuwenden.

Verhaltensmuster arbeiten auch gut mit anderen Design Patterns zusammen. Ein System, das beispielsweise das Pattern *Composite* (*Kompositum*, siehe [Abschnitt 4.3](#))

einsetzt, könnte das Pattern *Visitor* (*Besucher*, siehe [Abschnitt 5.11](#)) dazu benutzen, Operationen auf Komponenten des Kompositums anzuwenden. Es könnte eine *Chain of Responsibility* (*Zuständigkeitskette*, siehe [Abschnitt 5.1](#)) verwenden, um Komponenten den Zugriff auf globale Eigenschaften über deren Elternobjekte zu gewähren. Das Pattern *Decorator* (*Dekorierer*, siehe [Abschnitt 4.4](#)) könnte Anwendung finden, um diese Eigenschaften bei einem Teil des Kompositums zu überschreiben. Zur Verknüpfung einer Objektstruktur mit einer weiteren könnte auch das Pattern *Observer* (*Beobachter*, siehe [Abschnitt 5.7](#)) zum Einsatz kommen, und das Pattern *State* (*Zustand*, siehe [Abschnitt 5.8](#)) würde es ermöglichen, dass Komponenten ihr Verhalten ändern, sobald sich auch ihr Zustand ändert. Das Kompositum selbst könnte unter Anwendung des beim Design Pattern *Builder* (*Erbauer*, siehe [Abschnitt 3.2](#)) verfolgten Ansatzes erzeugt werden und ließe sich von anderen Teilen des Systems als *Prototype* (*Prototyp*, siehe [Abschnitt 3.4](#)) handhaben.

Gut durchdachte objektorientierte Systeme sind eben genau das: Sie enthalten mehrere Design Patterns – aber nicht, weil ihre Designer sie notwendigerweise unter diesen Aspekten betrachtet haben. Die Ausgestaltung auf der Ebene des *Patterns*, nicht auf der einer Klasse oder eines Objekts, ermöglicht es, mit geringerem Aufwand dieselben Synergieeffekte zu erzielen.

Kapitel 6: Schlusswort der Autoren

Der ein oder andere Leser dieses Buches mag vielleicht zu dem Schluss kommen, dass damit möglicherweise nicht wirklich viel erreicht ist: Wir präsentieren weder bislang nie dagewesene Algorithmen oder Programmietechniken, noch beschreiben wir eine bahnbrechende neue methodische Vorgehensweise für das Systemdesign oder erfinden die Designtheorie neu. Dieses Buch dokumentiert lediglich bereits existierende Designs. Dadurch könnte man zu dem Urteil gelangen, dass es zwar als akzeptabler Leitfaden für Einsteiger in die objektorientierte Programmierung taugt, erfahrenen objektorientierten Designern aber nicht sonderlich viel Neues zu bieten hat.

Wir hoffen allerdings, dass unsere Leser das etwas anders sehen. Die Katalogisierung von Design Patterns ist eine wichtige Maßnahme. Sie ermöglicht eine standardisierte Namensgebung und Definition der von uns angewendeten Techniken. Ohne die eingehende Untersuchung und Erfassung der in Softwareprogrammen eingesetzten Design Patterns wird es uns nur schwer gelingen, sie zu verbessern oder neue Patterns zu entwickeln.

Dieses Buch ist erst der Anfang. Es dokumentiert einige der von den Experten des objektorientierten Designs am häufigsten genutzten Design Patterns – die bislang jedoch nur durch Mund-zu-Mund-Propaganda oder die intensive Analyse vorhandener Systeme Verbreitung fanden. Und da schon die früheren Versionen dieses Buches viele Entwickler dazu veranlasst haben, die von ihnen verwendeten Patterns zu dokumentieren, hoffen wir, dass die vorliegende Fassung noch mehr Softwaredesigner motivieren wird, es ihnen gleichzutun. Uns ist daran gelegen, mit diesem Werk den Grundstein für eine Bewegung zur Dokumentation des Expertenwissens erfahrener Softwareentwickler zu legen.

Am Ende dieses Kapitels wird abschließend erörtert, welchen Einfluss bzw. welche Auswirkungen die Design Patterns unserer Meinung nach haben werden, in welchem Zusammenhang sie mit anderen Bereichen der Designarbeit zu sehen sind und wie Sie sich selbst an der Identifizierung und Katalogisierung von Patterns betätigen können.

6.1 Was kann man von Design Patterns erwarten?

Die in diesem Buch vorgestellten Design Patterns können sich in Abhängigkeit von Ihrer persönlichen Erfahrung als Entwickler in unterschiedlicher Art und Weise auf Ihre objektorientierte Designarbeit auswirken.

6.1.1 Ein gemeinsames Designvokabular

Von Programmierexperten durchgeführte Studien zu konventionellen Programmiersprachen haben gezeigt, dass Wissen und Erfahrung nicht einfach auf syntaktischer Ebene, sondern in größeren konzeptuellen Strukturen wie Algorithmen, Datenstrukturen und Idiomen [AS85, Cop92, Cur89, SS86] sowie Planungsstrategien zum Erreichen einer bestimmten Zielsetzung [SE84] organisiert sind. Softwaredesignern ist weniger an der für die Aufzeichnung ihres Designs verwendeten Notation gelegen, sondern vielmehr daran, zu versuchen, die aktuelle Designaufgabe mit Planungsverfahren, Algorithmen, Datenstrukturen und Idiomen in Übereinstimmung zu bringen, die sie in der Vergangenheit erlernt haben.

Die namentliche Bezeichnung von Algorithmen und Datenstrukturen ist im Bereich der Informatik zwar allgemein üblich, andere Musterarten werden jedoch oftmals nicht so eindeutig benannt. Design Patterns bieten Softwareentwicklern die Möglichkeit, ein gemeinsames Vokabular zur Kommunikation, Dokumentation und Diskussion von Designalternativen zu nutzen. Sie tragen dazu bei, die Komplexität der Systeme so weit zu reduzieren, dass sie auf einer höheren Abstraktionsebene diskutiert werden können, als dies auf der Basis der Designnotation oder der Programmiersprache möglich ist. Design Patterns heben das Niveau der gestalterischen und kommunikativen Möglichkeiten für den Austausch mit Kollegen und anderen Entwicklern.

Nach der Lektüre der in diesem Buch vorgestellten Design Patterns wird sich Ihr Designvokabular sehr schnell ändern: Sie werden sich direkt, sprich namentlich auf die Design Patterns beziehen und feststellen, dass Sie Aussagen wie »Lasst uns hier den Observer benutzen« oder »Aus diesen Klassen sollten wir Strategy-Objekte machen« ganz selbstverständlich in Ihren täglichen Sprachgebrauch aufnehmen.

6.1.2 Eine Dokumentations- und Lernhilfe

Wer mit den in diesem Buch beschriebenen Design Patterns vertraut ist, dem wird das Verständnis existierender Systeme leichter fallen – denn sie werden in den meisten großen objektorientierten Systemen angewendet. So klagen beispielsweise

Einsteiger in die objektorientierte Programmierung häufig darüber, dass ihnen die Anwendung des Vererbungsprinzips in den Systemen, mit denen sie arbeiten, verwirrend erscheint und der Programmablauf für sie schwer nachzuvollziehen ist – und das liegt größtenteils in ihrem mangelnden Wissen über die in solchen Systemen genutzten Design Patterns begründet. Aber nicht nur in dieser Hinsicht ist die Kenntnis der Design Patterns zweifellos für ein besseres Verständnis existierender objektorientierter Systeme hilfreich.

Da sie Lösungsmöglichkeiten für allgemeine Problemstellungen im Softwaredesign bereitstellen, tragen die Design Patterns auch dazu bei, Sie zu einem besseren Designer zu machen. Wer erst einmal lange genug mit objektorientierten Systemen gearbeitet hat, wird die Patterns höchstwahrscheinlich irgendwann auch von allein erkennen und verinnerlichen – die Informationen in diesem Buch werden diesen Prozess jedoch deutlich beschleunigen.

Sie werden feststellen, dass es sehr viel einfacher ist, ein System zu verstehen, wenn es anhand der Design Patterns beschrieben wird, die darin enthalten sind. Im anderen Fall muss das Design üblicherweise zunächst einmal mithilfe des Reverse-Engineering-Verfahrens analysiert werden, damit die verwendeten Strukturen aufgedeckt und kommuniziert werden können. Ein gemeinsames Vokabular hat hier den Vorteil, dass nicht erst das gesamte Design Pattern ausführlich beschrieben werden muss – Sie nennen es einfach beim Namen und können dann davon ausgehen, dass Ihr Gegenüber genau weiß, was es damit auf sich hat. Und selbst wenn sich Ihr Gesprächspartner nicht mit den Patterns auskennen sollte, kann er sie nachschlagen und sich entsprechend schnell informieren – das funktioniert immer noch einfacher, als das gesamte System analysieren zu müssen.

Wir verwenden die vorgestellten Patterns auch in unseren eigenen Designs und sie haben sich für uns als unschätzbar wertvoll erwiesen – auch wenn wir sie nur auf relativ simple Art einsetzen: Wir nutzen sie zur Benennung von Klassen, als Denkanstoß für weiterführende Überlegungen zu guten Designs sowie deren Kommunikation an Dritte und zur Beschreibung und Erläuterung von Designs in Form einer Aneinanderreihung der angewendeten Patterns [BJ94]. Weitere und anspruchsvollere Anwendungsmöglichkeiten zu finden, wie z. B. in Pattern-basierten CASE-Tools oder Hypertextdokumenten, fällt nicht schwer – doch auch ohne solche fortgeschrittenen Tools sind Design Patterns eine große Hilfe.

6.1.3 Eine Ergänzung zu existierenden Methoden

Objektorientierte Designmethoden sollen gute Designs fördern, Einsteigern in den

Bereich der Designentwicklung vermitteln, wie man gute Designs erstellt, und den Designprozess als solchen standardisieren. Eine Designmethode definiert in aller Regel einen Satz von (meist grafischen) Notationen für die Modellierung verschiedener Aspekte eines Designs sowie einen Regelsatz, der beschreibt, wie und wann eine Notation zu verwenden ist. Generell beschreiben Designmethoden Problemstellungen, die während der Designarbeit in Erscheinung treten, wie sie sich lösen lassen und wie ein Design zu bewerten ist. Dennoch waren sie bisher nicht in der Lage, die Erfahrung von Designexperten zu erfassen.

Wir glauben, dass unsere Design Patterns einen wichtigen Teil dessen darstellen, was den objektorientierten Designmethoden fehlt. Die in diesem Buch vorgestellten Patterns demonstrieren den Einsatz primitiver Techniken wie Objekterzeugung, Vererbung und Polymorphie. Sie zeigen auf, wie sich ein System mit einem Algorithmus, einem Verhalten, einem Zustand oder den Objekten, die es erzeugen soll, parametrisieren lässt. Design Patterns ermöglichen eine detailliertere Erläuterung des »Warum« eines Designs – und nicht bloß eine Aufzeichnung der Resultate der vom Entwickler getroffenen Entscheidungen. Die Abschnitte »Anwendbarkeit«, »Konsequenzen« und »Implementierung« in den Beschreibungen der einzelnen Design Patterns bieten wertvolle Hilfestellung für die Entscheidungsfindung des Entwicklers.

Besonders nützlich sind Design Patterns, wenn es darum geht, ein Analysemodell zu einer Implementierung weiterzuentwickeln. Trotz vielfacher Behauptungen, dass der Übergang von der objektorientierten Analyse zum Design einfach umzusetzen sei, stellt sich dies in der Praxis ganz anders dar. Ein flexibles und wiederverwendbares Design wird in aller Regel auch Objekte enthalten, die im Analysemodell noch gar nicht vorgesehen waren. Ebenso beeinflussen auch die Programmiersprache und die verwendeten Klassenbibliotheken das Design. Und oftmals ist eine Umgestaltung der Analysemodelle zur Gewährleistung ihrer Wiederverwendbarkeit nicht zu vermeiden. Viele der in diesem Katalog vorgestellten Design Patterns befassen sich mit diesen Problemen – deshalb heißen sie ja auch *Design Patterns*.

Eine vollwertige Designmethode braucht allerdings mehr Mustervarianten als nur Design Patterns. So könnten zum Beispiel auch Analysemuster, Gestaltungsmuster für Benutzeroberflächen oder performancesteigernde Muster verwendet werden. Die Design Patterns sind jedoch in jedem Fall ein wesentlicher Bestandteil, der bislang gefehlt hat.

6.1.4 Zielsetzungen für Refactorings

Eins der häufigsten Probleme bei der Entwicklung wiederverwendbarer Software ist die sogenannte Reorganisation bzw. das **Refactoring** [OJ90]. Design Patterns dienen in dieser Hinsicht als Entscheidungshilfe für die Art der Reorganisation eines Designs und können zudem den Aufwand für das später durchzuführende Refactoring mindern.

Objektorientierte Software durchläuft im Rahmen ihres Lebenszyklus mehrere Phasen. Brian Foote unterscheidet in diesem Zusammenhang die Phasen **Prototyping**, **Expansion** und **Konsolidierung** [Foo92].

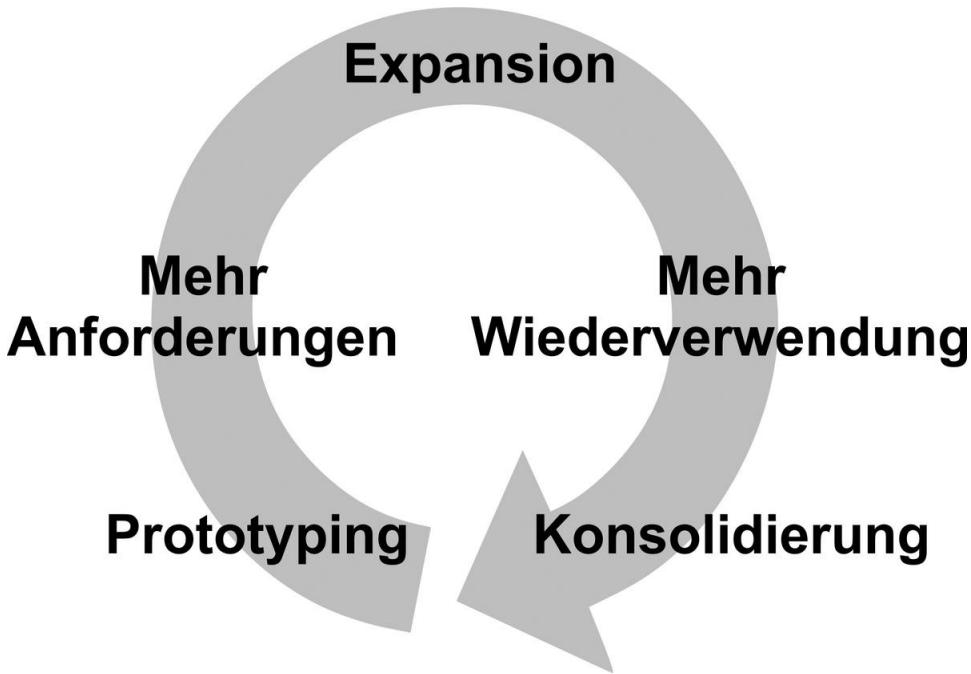


Abb. 6.1: Lebenszyklus-Phasen objektorientierter Software

Die *Prototyping-Phase* ist von vielen verschiedenen Aktivitäten geprägt, die die Software mittels Rapid Prototyping und inkrementeller Anpassungen zum Leben erweckt, bis sie einen ersten grundlegenden Bestand an Anforderungen erfüllt und die Einsatzreife erreicht. An diesem Punkt besteht die Software normalerweise aus Klassenhierarchien, die die Eigenheiten des ursprünglichen Problembereichs recht genau widerspiegeln. Hier kommt hauptsächlich die White-Box-Wiederverwendung durch Vererbung zum Tragen.

Sobald die Software die Einsatzreife erreicht hat und in Betrieb genommen wird, wird ihre weitere Evolution von zwei widersprüchlichen Erfordernissen bestimmt: Zum einen muss sie weiteren Anforderungen genügen – zum anderen muss sie umfassender wiederzuverwenden sein. Neue Anforderungen resultieren üblicherweise in neuen Klassen und Operationen, möglicherweise sogar neuen

Klassenhierarchien. Die Software durchläuft also eine *Expansionsphase*, um neuen Anforderungen entsprechen zu können. Das kann allerdings nicht endlos so weitergehen, und so wird sie schließlich irgendwann zwangsläufig zu unflexibel und instabil, um weitere Anpassungen zu verkraften: Die Klassenhierarchien werden nicht mehr einem einzigen Problembereich angehören, sondern viele Problembereiche widerspiegeln – und die Klassen werden zahlreiche unzusammenhängende Operationen und Instanzvariablen definieren.

Damit trotzdem eine Weiterentwicklung möglich ist, muss die Software in einem als *Refactoring* bezeichneten Prozess reorganisiert werden. In dieser Phase entstehen meist die Frameworks. Darüber hinaus umfasst das Refactoring auch die Aufteilung der Klassen in spezifische und allgemein verwendbare Komponenten, die Neupositionierung von Operationen in der Klassenhierarchie sowie die Rationalisierung der Klassenschnittstellen. Diese *Konsolidierungsphase* bringt zahlreiche neue Objektarten hervor, häufig durch die Zerlegung existierender Objekte sowie die Anwendung der Objektkomposition statt des Vererbungsprinzips. Die Black-Box-Wiederverwendung ersetzt somit die White-Box-Wiederverwendung. Der fortlaufende Bedarf im Hinblick auf die Erfüllung immer mehr Anforderungen gepaart mit dem Bedarf an mehr Wiederverwendung treibt die objektorientierte Software durch wiederholte Phasen der Expansion und Konsolidierung – Expansion, wenn neue Anforderungen erfüllt werden müssen, und Konsolidierung, wenn die Software universeller einsatzbar wird.

Dieser Zyklus ist unvermeidbar. Gute Designer sind sich allerdings der Anpassungen bewusst, die durch Refactorings nötig werden können. Und gute Designer wissen auch, welche Klassen- und Objektstrukturen dazu beitragen können, Refactorings zu vermeiden – ihre Designs halten auch veränderten Anforderungen stand. Eine gründliche Anforderungsanalyse zeigt auf, welche Erfordernisse sich im Laufe des Lebenszyklus der Software sehr wahrscheinlich ändern werden – und ein gutes Design wird sich demgegenüber als robust behaupten können.

Die in diesem Buch vorgestellten Design Patterns erfassen viele der Strukturen, die sich durch das Refactoring ergeben. Wenn sie schon frühzeitig im Designprozess angewendet werden, lassen sich spätere Refactorings verhindern. Aber selbst wenn erst nach der Fertigstellung des Systems erkennbar wird, wie sich ein Pattern einsetzen lässt, kann es auch dann noch Wege aufzeigen, wie das System in geeigneter Weise angepasst werden kann – und damit stellen Design Patterns auch Refactoring-Ziele dar.

6.2 Eine kleine Kataloggeschichte

Die Anfänge des in diesem Buch beschriebenen Design-Pattern-Katalogs reichen bis zur Doktorarbeit von Erich Gamma [Gam91, Gam92] zurück: Etwa die Hälfte der vorgestellten Patterns fanden bereits in dieser Arbeit Erwähnung. Als 1991 die OOPSLA-Konferenz stattfand, waren sie schon offiziell in einem unabhängigen Katalog zusammengefasst. Inzwischen hatte sich Richard Helm zu ihm gesellt, um an der Ausarbeitung des Katalogs mitzuwirken. Bald darauf schloss sich auch John Vlissides an und bis zur OOPSLA 1992 war das Team schließlich durch den Beitritt von Ralph Johnson komplett.

Gemeinsam arbeiteten wir hart daran, den Katalog für die Veröffentlichung auf der ECOOP 1993 vorzubereiten, merkten aber schon bald, dass ein 90-Seiten-Wälzer nicht akzeptiert werden würde. Also erstellten wir eine Zusammenfassung des Katalogs und reichten diese Version ein, die dann auch angenommen wurde. Und kurz danach fassten wir den Entschluss, den Katalog als Buch zu veröffentlichen.

Hinweis

Die *OOPSLA (Object-Oriented Programming, Systems, Languages & Applications)* war eine von der *Association for Computing Machinery (ACM)* organisierte jährlich stattfindende US-Fachkonferenz zum Thema objektorientierte Softwareentwicklung, die seit 2010 unter dem Namen *SPLASH* fortgeführt wird.

Die von der *Association Internationale pour les Technologies Objets (AITO)* organisierte *ECOOP (European Conference on Object-Oriented Programming)* ist die ebenfalls jährlich stattfindende Schwesterkonferenz in Europa.

Wir haben unsere Pattern-Namen im Laufe der Zeit immer wieder ein wenig verändert: Aus »Wrapper« (Umwickler) wurde »Decorator« (Dekorierer), aus »Glue« (Leim) wurde »Facade« (Fassade), »Solitaire« wurde zu »Singleton«, und »Walker« (Wanderer) wurde zu »Visitor« (Besucher). Einige Design Patterns wurden komplett verworfen, weil sie uns nicht bedeutsam genug erschienen. Ansonsten hat sich die im Katalog erfasste Pattern-Sammlung seit 1992 allerdings nur wenig verändert – die Design Patterns selbst haben sich jedoch enorm weiterentwickelt.

Tatsächlich ist das eigentliche Entdecken eines Patterns der einfache Teil. Wir alle waren aktiv an der Errichtung objektorientierter Systeme beteiligt und stellten fest, dass die *Erkennung* von Mustern im Grunde recht simpel ist, wenn man nur genug Systeme untersucht – sie zu *beschreiben*, ist dagegen eine deutlich schwierigere Angelegenheit.

Wenn man Systeme errichtet und sich mit der Planung dessen beschäftigt, was man da baut, wird man in allem, was man tut, Patterns erkennen. Eine unmissverständliche Beschreibung dieser Muster zu entwickeln, damit sie von Außenstehenden verstanden werden und sie begreifen können, warum sie wichtig sind, ist allerdings in der Tat recht schwierig. Als wir den Katalog in den frühen Entstehungsstadien verschiedenen Fachleuten vorlegten, war ihnen seine Bedeutung sofort klar – wirklich verstanden haben die Patterns jedoch nur diejenigen Experten, die sie bereits verwendet hatten.

Da eine der wichtigsten Zielsetzungen dieses Buches lautete, Einsteigern die Kunst des objektorientierten Designs nicht nur näher zu bringen, sondern zu vermitteln, stand außer Frage, dass der Katalog verbessert werden musste. Also erweiterten wir den durchschnittlichen Umfang einer Pattern-Beschreibung von weniger als zwei auf mehr als zehn Seiten, indem wir jeweils ausführliche Motivationsbeispiele und Beispielcode hinzunahmen. Außerdem führten wir Untersuchungen zu den individuellen Vor- und Nachteilen der einzelnen Patterns sowie den verschiedenen Implementierungsmöglichkeiten durch und ergänzten auch die hierbei gewonnenen Erkenntnisse. Auf diese Weise waren die Design Patterns einfacher zu erlernen.

Im darauffolgenden Jahr nahmen wir dann noch eine weitere wegweisende Änderung vor: Wir widmeten uns mehr den Problemstellungen, die ein Design Pattern zu lösen vermag. Am einfachsten lässt sich ein Pattern als Problemlösung betrachten – als eine Technik, die adaptiert und immer wieder angewendet werden kann. Schwieriger ist es dagegen zu erkennen, wann der Einsatz eines Patterns *angemessen* ist, denn dazu müssen nicht nur die lösbareren Problemstellungen, sondern auch der Kontext, in dem das Pattern die beste Lösung bietet, ermittelt und beschrieben werden. Im Allgemeinen ist es einfacher zu erkennen, *was* getan wird als *warum* – im Fall eines Design Patterns ist aber gerade das »Warum« der entscheidende Faktor für die Identifizierung des Problems, das es lösen kann. Nicht minder bedeutsam ist es auch, den Zweck eines Patterns zu kennen, weil dadurch die Auswahl des passenden Patterns überhaupt erst möglich ist – und es trägt natürlich auch dazu bei, das Design existierender Systeme besser und schneller zu verstehen. Der Autor eines Design Patterns muss die Problemstellung, die sich mithilfe des Patterns lösen lässt, in jedem Fall eindeutig bestimmen und charakterisieren – auch im Nachhinein, wenn das Problem bereits gelöst sein sollte.

6.3 Die Pattern-Gemeinde

Wir sind natürlich nicht die Einzigen, die an der Katalogisierung der von Experten verwendeten Muster in Buchform interessiert sind. Vielmehr sind wir Teil einer großen Community, deren Interesse sich auf Design Patterns im Allgemeinen und softwarebezogene Patterns im Besonderen richtet. Der Architektur- und Systemtheoretiker Christopher Alexander war der Erste, der das Auftreten wiederkehrender Muster in Bauwerken und Stadtplanungen eingehend untersuchte und eine »Muster-Sprache« entwickelte, um sie generieren und umsetzen zu können. Seine Arbeit hat uns immer wieder inspiriert – und deswegen ist es nur richtig und in jedem Fall lohnenswert, einen Vergleich zu seiner Arbeit ebenso wie zu den Erkenntnissen anderer Autoren zu ziehen, die sich mit softwarebezogenen Patterns beschäftigen.

6.3.1 Christopher Alexanders »Muster-Sprache«

Unsere Arbeit weist viele Gemeinsamkeiten mit der von Christopher Alexander auf. Die Ausgangsbasis ist immer die Beobachtung existierender Systeme und deren Untersuchung auf das Vorkommen von Mustern. Wie Alexander liefern auch wir in diesem Buch schablonenartige Beschreibungen der Patterns, wenngleich sich die Schablonen an sich deutlich voneinander unterscheiden. Und schließlich bedienen wir uns ebenfalls einer natürlichen, nicht-formalen Sprache und führen zahlreiche Beispiele an. Darüber hinaus wird jedes Pattern eingehend erklärt und nicht nur beschrieben.

Andererseits gibt es aber auch zahlreiche Punkte, in denen sich unsere Arbeit von der Christopher Alexanders unterscheidet:

1. Bauwerke werden schon seit Tausenden von Jahren von Menschenhand errichtet, so dass es zahllose klassische Beispiele gibt, auf die man sich beziehen kann. Im Vergleich dazu reicht die Geschichte der Softwaresysteme erst relativ kurze Zeit zurück – und nur wenige dieser Systeme können tatsächlich als »Klassiker« bezeichnet werden.
2. Alexander gibt eine Reihenfolge vor, in der seine Muster angewendet werden sollen – das ist in diesem Buch nicht der Fall.
3. Alexanders Muster sind vorwiegend auf die Problemstellungen ausgerichtet, für die sie geeignet sind. Der in diesem Buch vorliegende Design-Pattern-

Katalog konzentriert sich dagegen weitaus mehr auf die Lösungen.

4. Alexander stellt die Behauptung auf, dass seine Muster komplett Bauwerke generieren können. Wir behaupten nicht, dass unsere Design Patterns gleich vollständige Programme erzeugen können.

Wenn Alexander bekräftigt, dass sich ein Haus allein durch die sukzessive Anwendung seiner Muster entwerfen ließe, verfolgt er damit ähnliche Ziele wie die objektorientierten Designmethodiker, die Schritt-für-Schritt-Anweisungen für die Entwicklung eines Designs vorgeben. Alexander bestreitet nicht die Notwendigkeit von Kreativität: Einige seiner Muster setzen ein umfassendes Verständnis der Lebensgewohnheiten der Menschen voraus, die die so entstehenden Bauwerke nutzen werden – und sein Glaube an die »Poesie« des Designs impliziert einen Grad an Expertenwissen, der über die Muster-Sprache hinausgeht [AIS+77, Seite XLII, »Die Poesie der Sprache«]. Seine Beschreibung zur Art und Weise, wie Patterns Designs generieren, geht jedoch davon aus, dass eine Muster-Sprache den Designprozess deterministisch und wiederholbar machen kann.

Alexanders Sichtweise hat uns dazu inspiriert, uns auf die Vor- und Nachteile eines Designs zu konzentrieren – die verschiedenen »Kräfte«, die sich bei der Gestaltung eines Designs hilfreich auswirken. In der Folge haben wir uns eingehender damit befasst, die Anwendbarkeit und die Konsequenzen unserer Patterns zu verstehen, und uns weniger mit der Definition einer formalen Repräsentation der Patterns auseinandergesetzt. Natürlich könnte eine solche Repräsentation wahrscheinlich die Automatisierung von Patterns ermöglichen, andererseits ist es gegenwärtig wichtiger, den Wirkungsraum der Design Patterns zu erkunden, als sie zu formalisieren.

Aus Alexanders Sicht bilden die in diesem Buch beschriebenen Design Patterns keine Muster-Sprache. Schon allein die schiere Vielfalt der entwickelten Softwaresysteme lässt nur schwer erkennen, wie man einen »kompletten« Pattern-Satz bereitstellen könnte, der schrittweise Instruktionen für das Designen einer Anwendung vorgibt. Für bestimmte Arten von Anwendungen, wie z. B. die Berichterstellung oder Formularsysteme, wäre das gegebenenfalls möglich – unser Katalog stellt jedoch lediglich eine Sammlung von verwandten Patterns dar. Wir können nicht behaupten, dass es sich um eine »Muster-Sprache« handelt.

Um genau zu sein, halten wir es für unwahrscheinlich, dass es *jemals* eine vollständige Muster-Sprache für Software geben wird. Es ist aber durchaus denkbar, eine Muster-Sprache zu entwickeln, die *umfassender* ist als unser Katalog – in der Frameworks und deren Anwendung [Joh92] ebenso berücksichtigt werden müssten

wie Patterns für das Design von Benutzeroberflächen [BJ94], Analysemuster [Coa92] und alle anderen Aspekte, die zur Softwareentwicklung gehören. Design Patterns bilden lediglich einen Teil einer größeren, umfassenderen Muster-Sprache für Software.

6.3.2 Patterns in Softwaresystemen

Unsere ersten gemeinsamen Erfahrungen haben wir im Studium der Softwarearchitektur auf einem von Bruce Anderson geleiteten Workshop während der OOPSLA 1991 gemacht, der ausschließlich der Entwicklung eines Handbuchs für Softwarearchitekten gewidmet war. (Verglichen mit dem vorliegenden Buch wäre »Enzyklopädie der Softwarearchitektur« wohl eine treffendere Bezeichnung als »Architekturhandbuch«.) Diesem ersten Workshop folgten eine Reihe von Meetings, darunter auch die erste Konferenz zum Thema Muster-Sprachen in der Programmierung, die im August 1994 stattfand. Diese Konferenz begründete eine Community von an der Dokumentation von Expertenwissen im Bereich der Softwareprogrammierung interessierten Entwicklern.

Natürlich hatten sich zuvor bereits andere dieses Ziel gesetzt: Donald Knuths »*The Art of Computer Programming*« [Knu73] war einer der ersten Versuche, softwarebezogenes Wissen zu katalogisieren, wenngleich er sich auf die Beschreibung von Algorithmen beschränkte. Trotzdem erwies sich das Vorhaben als zu umfangreich, als dass es abgeschlossen werden konnte. Auch die »*Graphics Gems*«-Reihe [Gla90, Arv91, Kir92] ist im Prinzip eine Art Wissenskatalog zum Thema Software, obwohl sie sich ebenfalls hauptsächlich auf Algorithmen konzentriert. Das *Domain Specific Software Architecture Program* des US-Verteidigungsministeriums [GM92] befasst sich hingegen schwerpunktmäßig mit dem Sammeln von Architekturdaten. Die wissensbasiert arbeitende Fraktion der Software Engineering Community versucht softwarebezogenes Wissen allgemein zu repräsentieren. Und es gibt noch viele weitere Gruppierungen, die zumindest teilweise dieselben Ziele verfolgen wie wir.

Auch James Copliens »*Advanced C++: Programming Styles and Idioms*« [Cop92] hat uns beeinflusst. Die in seinem Buch beschriebenen Patterns sind meist C++-orientierter als die hier vorgestellten Design Patterns. Außerdem enthält sein Werk viele Patterns niedriger Abstraktionsebene. Dennoch ergeben sich einige Überschneidungen, auf die in den Pattern-Beschreibungen entsprechend hingewiesen wird. Jim ist ein aktives Mitglied der Pattern Community und arbeitet unter anderem an Mustern, die die Rolle des Menschen in Unternehmen für

Softwareentwicklung beschreiben.

Darüber hinaus gibt es noch zahlreiche andere Quellen für Pattern-Beschreibungen. Zum Beispiel war Kent Beck einer der Ersten aus der Software Community, die Christopher Alexanders Arbeit Beachtung schenkten. 1993 begann er, eine Kolumne über Smalltalk-Patterns für das Fachmagazin »*The Smalltalk Report*« zu schreiben. Peter Coad beschäftigt sich ebenfalls mit der Katalogisierung von Patterns. In seinen frühen Arbeiten befasste er sich allerdings hauptsächlich mit Analysemustern [Coa92].

6.4 Eine Einladung

Welche Maßnahmen können Sie ergreifen, wenn Sie sich für Patterns interessieren? Die oberste Regel lautet: Nutzen Sie sie und versuchen Sie weitere Design Patterns aufzuspüren, die zu Ihrem Designstil passen. Es steht zu erwarten, dass in den nächsten Jahren immer mehr Bücher und Artikel zu diesem Thema erscheinen werden, so dass Ihnen reichlich Quellmaterial zur Verfügung stehen wird, um neue Patterns zu entdecken. Entwickeln Sie außerdem Ihr eigenes Pattern-Vokabular und wenden Sie es konsequent an, z. B. wenn Sie sich mit anderen Entwicklern oder Kollegen über Ihre Designs austauschen, wenn Sie Ihre Designs planen und sie dokumentieren.

Darüber hinaus sollten Sie ein kritischer Konsument sein. Der Design-Pattern-Katalog ist das Ergebnis harter Arbeit – und zwar nicht nur von unserer Seite, sondern auch vonseiten Dutzender von Reviewern, die uns ihr Feedback geliefert haben. Sollten Sie ein Problem feststellen oder weiterführende Erklärungen für nötig halten, dann kontaktieren Sie uns. Dasselbe gilt im Übrigen auch für jeden anderen Pattern-Katalog: Geben Sie den Autoren Ihr Feedback! Eine der großartigen Eigenschaften von Design Patterns besteht darin, dass sie die Designentscheidungen aus dem Bereich der vagen Intuition herausleiten und es den Autoren ermöglichen, die damit einhergehenden Vor- und Nachteile explizit zu benennen. Dadurch lassen sich Schwachpunkte in den Patterns wesentlich leichter feststellen und diskutieren. Nutzen Sie diese Möglichkeit!

Und zum Dritten: Erfassen Sie die von Ihnen verwendeten Patterns und dokumentieren Sie sie. Machen Sie sie zu einem festen Bestandteil Ihrer Designdokumentation und zeigen Sie sie anderen Leuten. Sie müssen nicht in einem Forschungslabor arbeiten, um Design Patterns zu entdecken. Tatsächlich ist das Auffinden relevanter Patterns ohne eigene praktische Erfahrung sogar nahezu

unmöglich. Schreiben Sie deshalb ruhig einen eigenen Pattern-Katalog – aber stellen Sie sicher, dass Ihnen Dritte helfen, sie in die richtige Form zu bringen!

6.5 Ein abschließender Gedanke

In den besten Designs kommen gleich mehrere Design Patterns zur Anwendung, die sich ergänzen und ineinandergreifen, um ein größeres Ganzes in der Form zu erschaffen, wie es der US-amerikanische Architekturtheoretiker Christopher Alexander in seinem Buch »*Eine Muster-Sprache*« [AIS+77, Seite XLII] beschreibt:

»Das Gleiche gilt für Muster-Sprachen. Man kann Gebäude errichten, indem man Muster locker kombiniert. Ein so entstandenes Gebäude ist eine Zusammenstellung von Mustern. Aber es hat keine Dichte, keine Tiefe. Man kann aber auch Muster so zusammenfügen, daß viele, viele Muster einander im selben physischen Raum überlagern: das Gebäude ist sehr dicht; es enthält viele Bedeutungen auf kleinem Raum, und durch diese Dichte wird es tief.«

Anhang A: Glossar

Abstrakte Klasse

Die Hauptaufgabe einer abstrakten Klasse besteht in der Definition einer Schnittstelle. Sie delegiert ihre Implementierung teilweise oder komplett an Unterklassen und kann nicht instanziert werden.

Abstrakte Kopplung

Wenn eine Klasse A eine Referenz auf eine abstrakte Klasse B enthält, spricht man davon, dass A eine *abstrakte Kopplung* zu B aufweist, weil A auf einen *Objekttyp*, nicht aber auf ein konkretes Objekt verweist.

Abstrakte Operation

Eine Operation, die eine Signatur deklariert, aber nicht implementiert. In C++ entspricht eine abstrakte Operation einer *rein virtuellen Memberfunktion* (engl. *pure virtual member function*).

Aggregationsbeziehung

Die Beziehung zwischen einem Aggregat und seinen *Teilen* bzw. *Bestandteilen*, sprich Unterobjekten. Eine Klasse definiert diese Beziehung für ihre Instanzen (z. B. aggregierte Objekte).

Aggregiertes Objekt

Ein aus Unterobjekten zusammengesetztes Objekt. Diese Unterobjekte werden als *Teile* bzw. *Bestandteile* des Aggregats bezeichnet und unterliegen dessen Zuständigkeit.

Assoziationsbeziehung

Eine Klasse, die auf eine andere Klasse verweist, steht in einer *Assoziationsbeziehung* zu dieser Klasse.

Basisklasse

Eine *Basisklasse* ist die Klasse, von der eine andere Klasse erbt. In Smalltalk oder C++ wird eine solche vererbende Klasse auch als *Superklasse* (engl. *Superclass*), *Elternklasse* (engl. *Parent Class*) oder *Vorfahrenklasse* (engl. *Ancestor Class*) bezeichnet.

Black-Box-Wiederverwendung

Eine auf der Objektkomposition basierende Form der Wiederverwendung. Die Analogie zur »Black Box« führt daher, dass zusammengesetzte Objekte auch untereinander keine internen Details preisgeben.

Delegation

Ein Implementierungsmechanismus, der einem Objekt die *Weitergabe* bzw. *Delegation* eines Requests an ein anderes Objekt gestattet. Der *Delegate* – das Delegationsobjekt, dem die Aufgabe übertragen wird – bearbeitet den Request stellvertretend für das Objekt, das ihn ursprünglich in Empfang genommen hat.

Design Pattern

Ein *Design Pattern* ermöglicht die systematische Benennung, Zielsetzung und Erläuterung eines allgemeinen Designs, das sich eines in objektorientierten Systemen wiederkehrenden Designproblems annimmt. Es beschreibt die Problemstellung, die passende Lösung, die Situation, wann die Lösung anwendbar ist, sowie deren Konsequenzen. Außerdem liefert es auch Hinweise und Beispiele zur Implementierung. Die Lösung besteht aus einer allgemeinen Anordnung von Objekten und Klassen und ist auf die Behebung einer Problemstellung in einem bestimmten Kontext ausgerichtet sowie entsprechend implementiert.

Destruktor

In C++ versteht man unter einem *Destruktor* eine Operation, die automatisch aufgerufen wird, wenn ein Objekt im Begriff ist, gelöscht zu werden.

Dynamisches Binden

Die Assoziation zwischen einem Request und einem Objekt sowie einer seiner Operationen zur Laufzeit. In C++ werden ausschließlich virtuelle Funktionen dynamisch gebunden.

Elternklasse

Siehe *Basisklasse*.

Empfänger

Das Zielobjekt eines Requests.

Framework

Ein Satz kooperierender Klassen, die ein Gerüst für wiederverwendbares Design für eine bestimmte Art von Software ergeben. Ein *Framework* dient als architektonische Richtlinie, um das Design in abstrakte Klassen zu unterteilen und deren Zuständigkeiten und Interaktionen zu definieren. Softwareentwickler können das Framework durch Unterklassenbildung und Zusammenführung von Instanzen der Framework-Klassen an eine bestimmte Anwendung anpassen.

Friend-Klasse

In C++ versteht man unter einer *Friend-Klasse* eine Klasse, die dieselben Zugriffsrechte auf die Operationen und Daten einer Klasse besitzt wie diese Klasse selbst.

Kapselung

Das Konzept des Verbergens einer Darstellung und Implementierung in einem Objekt. Die Darstellung eines gekapselten Objekts ist weder sichtbar, noch kann direkt von außerhalb darauf zugegriffen werden. Die einzige Zugriffs- und Modifikationsmöglichkeit besteht in der Ausführung geeigneter Operationen.

Klasse

Eine *Klasse* definiert die Schnittstelle und Implementierung eines Objekts. Sie spezifiziert die interne Darstellung des Objekts und legt fest, welche Operationen es ausführen kann.

Klassendiagramm

Die schematische Darstellung einer Klasse, ihrer internen Struktur, der zugehörigen Operationen sowie der statischen Beziehungen zwischen ihnen.

Klassenoperation

Eine auf eine Klasse, nicht aber auf ein individuelles Objekt ausgerichtete Operation. In C++ werden *Klassenoperationen* als *statische Memberfunktionen* (engl. *static member functions*) bezeichnet.

Konkrete Klasse

Eine Klasse, die keine abstrakten Operationen enthält und instanziert werden kann.

Konstruktor

In C++ versteht man unter einem *Konstruktor* eine Operation, die bei der Instanziierung einer Klasse automatisch aufgerufen wird und die neue Instanz initialisieren kann.

Kopplung

Die *Kopplung* beschreibt das Ausmaß der wechselseitigen Abhängigkeit unterschiedlicher Softwarekomponenten.

Instanzvariable

Eine Variable, die einen Teil der Objektdarstellung definiert. In C++ werden solche Variablen als *Datenmember* bezeichnet.

Interaktionsdiagramm

Die schematische Darstellung der zwischen Objekten ausgetauschten Requests.

Metaklasse

In Smalltalk sind Objekte Klassen. Eine *Metaklasse* ist die Klasse eines Klassenobjekts.

Mixin-Klasse

Eine Klasse, deren Zweck darin besteht, unter Anwendung des Vererbungsprinzips mit anderen Klassen kombiniert zu werden. *Mixin-Klassen* sind in der Regel abstrakte Klassen.

Objekt

Eine Laufzeitentität, die sowohl die Daten als auch die Prozeduren zusammenfasst, die auf der Grundlage dieser Daten arbeiten.

Objektdiagramm

Die schematische Darstellung einer bestimmten Objektstruktur zur Laufzeit.

Objektkomposition

Die Zusammensetzung bzw. *Komposition* von Objekten, um ein komplexeres Verhalten zu erzielen.

Objektreferenz

Ein Verweis auf einen Wert, der ein anderes Objekt identifiziert.

Operation

Die Daten eines Objekts können ausschließlich durch dessen Operationen manipuliert werden. Sobald ein Objekt einen Request erhält, führt es eine Operation aus. In C++ werden Operationen als *Memberfunktionen* bezeichnet, in Smalltalk wird der Begriff *Methode* verwendet.

Parametrisierter Typ

Ein Typ, der einige der ihn konstituierenden Typen unspezifiziert lässt. Die unspezifizierten Typen werden zum Zeitpunkt ihrer Verwendung als Parameter bereitgestellt. In C++ werden parametisierte Typen als *Templates* bezeichnet.

Polymorphie

Die *Fähigkeit*, Objekte zur Laufzeit durch Objekte mit übereinstimmender Schnittstelle zu ersetzen.

Private Vererbung

In C++ bezeichnet die *private Vererbung* eine Klasse, die ausschließlich zum Zweck

ihrer Implementierung geerbt wird.

Protokoll

Beschreibt die Erweiterung des Konzepts einer Schnittstelle um die zulässigen Request-Abfolgen.

Request

Ein Objekt führt nach dem Erhalt eines *Requests* von einem anderen Objekt die angeforderte Operation aus. Wird häufig auch als **Nachricht (Message)** oder **Anfrage** bezeichnet.

Schnittstelle

Die Menge aller von den Operationen eines Objekts definierten Signaturen. Die Schnittstelle beschreibt die Gesamtheit der Requests, auf die ein Objekt reagieren kann.

Signatur

Die *Signatur* einer Operation definiert deren Namen, ihre Parameter und ihren Rückgabewert.

Subsystem

Eine unabhängige Gruppe von Klassen, die zur Wahrnehmung diverser Zuständigkeiten zusammenarbeiten.

Subtyp

Ein Typ ist ein *Subtyp* eines anderen Typs, wenn seine Schnittstelle die Schnittstelle

des betreffenden anderen Typs enthält.

Supertyp

Der übergeordnete Elterntyp, von dem ein Typ erbt.

Toolkit

Eine Sammlung von Klassen und Schnittstellen, die nützliche Funktionalität zur Verfügung stellen, aber keinen unmittelbaren Einfluss auf die Definition des Anwendungsdesigns haben.

Typ

Der Name einer bestimmten Schnittstelle.

Überschreiben (Overriding)

Die Neudefinition einer (von einer *Basisklasse* geerbten) Operation in einer Unterkategorie.

Unterkategorie

Eine Klasse, die von einer übergeordneten *Basisklasse* erbt. In C++ werden *Unterklassen* häufig auch als **abgeleitete Klassen** bezeichnet.

Vererbung

Eine Beziehung, die eine Entität auf der Basis einer anderen Entität definiert. Die **Klassenvererbung** definiert eine neue Klasse auf der Grundlage einer oder mehrerer *Basisklassen*. Diese *Unterkategorie*, in C++ auch **abgeleitete Klasse** genannt, erbt sowohl ihre Schnittstelle als auch ihre Implementierung von ihren Basisklassen.

Bei der Klassenvererbung werden also die Schnittstellenvererbung und die Implementierungsvererbung miteinander kombiniert. Und auch hier gilt: Die **Schnittstellenvererbung** definiert eine neue Schnittstelle auf der Basis einer oder mehrerer bereits existierender Schnittstellen. Die **Implementierungsvererbung** definiert eine neue Implementierung auf der Grundlage einer oder mehrerer bereits vorhandener Implementierungen.

White-Box-Wiederverwendung

Eine auf der Klassenvererbung basierende Wiederverwendungstechnik. Eine Unterklasse erbt die Schnittstelle und Implementierung ihrer Basisklasse zum Zweck der Wiederverwendung, kann gegebenenfalls aber auch auf ansonsten private Bereiche der *Basisklasse* zugreifen.

Anhang B: Notationshinweise

In diesem Buch werden wichtige Konzepte in Form von Diagrammen dargestellt. Einige davon sind eher informeller Art und werden als Bildschirmfotos oder schematische Darstellungen entsprechender Objektbäume präsentiert. Die Pattern-Beschreibungen enthalten zur Veranschaulichung der Beziehungen und Interaktionen zwischen Klassen und Objekten jedoch auch formalere Notationen, die im Folgenden ausführlich erläutert werden.

Insgesamt werden drei verschiedene Arten von Diagrammen verwendet:

1. Ein **Klassendiagramm** stellt Klassen, deren Struktur sowie deren statische Beziehungen zueinander dar.
2. Ein **Objektdiagramm** bildet eine bestimmte Objektstruktur zur Laufzeit ab.
3. Ein **Interaktionsdiagramm** veranschaulicht den Austausch von Requests zwischen Objekten.

Jedes Design Pattern wird zumindest durch ein Klassendiagramm dokumentiert. Die anderen Notationen kommen lediglich bedarfsweise in Ergänzung der Ausführungen zum Einsatz. Alle Klassen- und Objektdiagramme basieren auf der **Object Modeling Technique (OMT)** [RBP+91, Rum94]. Die Interaktionsdiagramme wurden hingegen aus **Objectory** [JCJO92] und der **Booch-Methode** [Boo94] übernommen. Ihre Notationen stellen sich folgendermaßen dar:

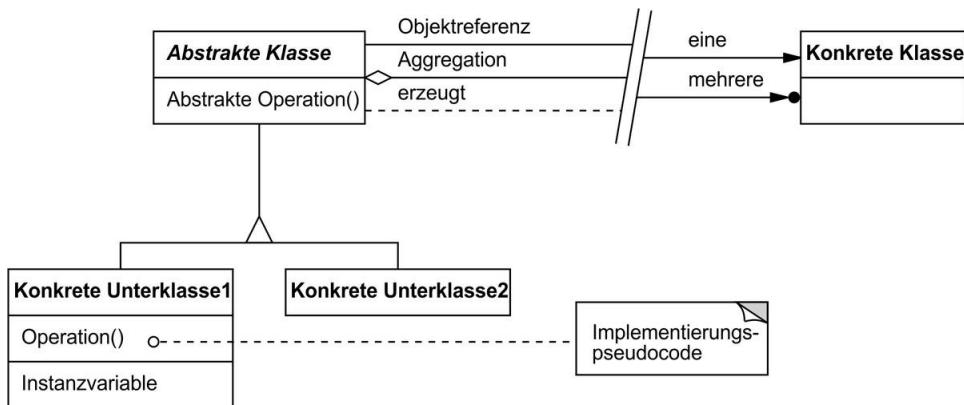


Abb. B.1: Notation für Klassendiagramme

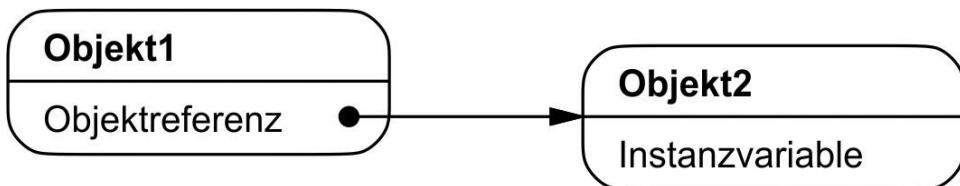


Abb. B.2: Notation für Objektdiagramme

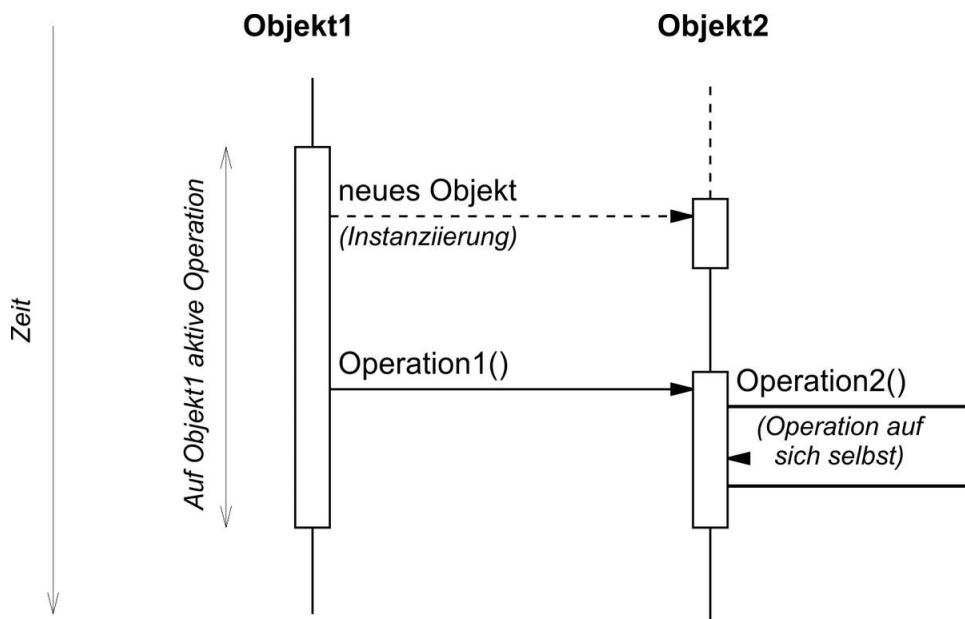


Abb. B.3: Notation für Interaktionsdiagramme

Hinweis

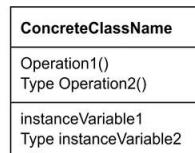
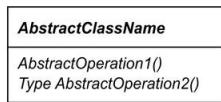
Die Object Modeling Technique weist große Ähnlichkeit mit der **Unified Modeling Language** (Vereinheitlichte Modellierungssprache, kurz **UML**) auf.

B.1 Klassendiagramme

[Abbildung B.4a](#) zeigt die OMT-Notation für abstrakte und konkrete Klassen. Eine Klasse wird im obersten umrahmten Kasten mit fettgedrucktem Namen angezeigt. Unter dem Klassennamen sind die zentralen Operationen der Klasse angegeben. Und darunter befinden sich wiederum die möglichen zugehörigen Instanzvariablen.

Hinweis

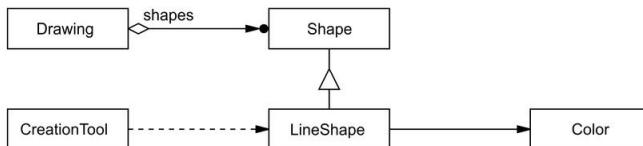
Während sich der Begriff »Objektdiagramm« in der OMT auf Klassendiagramme bezieht, werden in diesem Buch ausschließlich Darstellungen von Objektstrukturen als »Objektdiagramme« bezeichnet.



(a) Abstrakte und konkrete Klassen



(b) Teilnehmende Client-Klasse (links) und implizite Client-Klasse (rechts)



(c) Beziehungen zwischen Klassen



(d) Pseudocode-Annotation

Abb. B.4: Notationen für Klassendiagramme

Typinformationen sind optional anzugeben. Im Rahmen dieses Buches wird die C++-Konvention eingehalten, die vorsieht, dass der Typname dem Operationsnamen (zur Kennzeichnung des Rückgabetyps), der Instanzvariablen oder dem aktuellen Parameter vorangestellt wird. Kursiv gedruckte Bezeichnungen bedeuten, dass es sich um abstrakte Klassen oder Operationen handelt.

In manchen Fällen ist es hilfreich zu wissen, an welchen Stellen in einem Design Pattern Client-Klassen Teilnehmerklassen referenzieren. Wenn ein Pattern eine Client-Klasse als einen seiner Teilnehmer umfasst (d. h., dem Client wurde in dem

Pattern eine Zuständigkeit zugewiesen), erscheint der Client als gewöhnliche Klasse. Das ist z. B. im Design Pattern *Flyweight* (*Fliegengewicht*, siehe [Abschnitt 4.6](#)) der Fall. Wenn das Pattern keine teilnehmende `client`-Klasse enthält (d. h., Clients haben innerhalb des Patterns keine Zuständigkeiten), das Einbinden des Clients aber klarstellen würde, welche Teilnehmer des Patterns mit Clients interagieren, dann wird die `client`-Klasse wie in [Abbildung B.4b](#) ausgegraut dargestellt. Ein Beispiel hierfür liefert das Design Pattern *Proxy* (*Proxy*, siehe [Abschnitt 4.7](#)). Ein ausgegrauter Client liefert außerdem die Bestätigung, dass der Client nicht versehentlich in der Auflistung der Teilnehmer vergessen wurde.

[Abbildung B.4c](#) veranschaulicht die diversen Beziehungen zwischen den Klassen. In der OMT-Notation wird die Klassenvererbung durch ein weißes Dreieck gekennzeichnet, das die UnterkLASSE (hier: `LineShape`) mit ihrer BasiskLASSE (hier: `Shape`) verbindet. Eine Objektreferenz, die eine Aggregations- oder Teil-Ganzes-Beziehung repräsentiert, ist als eine durchgezogene Pfeillinie mit einer Raute am Anfangspunkt dargestellt. Der Pfeil weist dabei auf die aggregierte Klasse (z. B. `Shape`). Eine Pfeillinie ohne Raute symbolisiert eine *Kennt*-Beziehung (z. B. enthält `LineShape` eine Referenz auf ein `Color`-Objekt, das mit anderen Formen geteilt werden kann). Zur Unterscheidung einer Referenz von anderen Referenzen kann zusätzlich auch der Referenzname auf der Linie angegeben sein.

Hinweis

Zwischen den Klassen definierte Assoziationsbeziehungen werden in der OMT als einfache Linien zwischen den Klassenkästen gekennzeichnet. Assoziationen sind bidirektional. Sie stellen zwar während der Analyse ein probates Modellierungsinstrument dar, sind jedoch im Prinzip zu abstrakt, um die Beziehungen in Design Patterns adäquat auszudrücken – einfach weil Assoziationen im Rahmen des Designs auf Objektreferenzen und Pointer abgebildet werden müssen. Objektreferenzen sind dagegen grundsätzlich intrinsisch gerichtet und daher für die Beziehungen, die im Zusammenhang mit den in diesem Buch vorgestellten Design Patterns von Bedeutung sind, besser geeignet. Zum Beispiel hat `Drawing` Kenntnis von den `Shape`-Klassen, während Letztere jedoch nicht wissen, in welcher `Drawing`-Klasse sie sich befinden. Diese Art von Beziehung lässt sich nicht allein durch Assoziationen ausdrücken.

Ebenfalls hilfreich ist es darzustellen, welche Klassen andere Klassen instanziieren. Da die OMT dies nicht unterstützt, wird in diesem Buch eine gestrichelte Pfeillinie

verwendet, um solche »Erzeugt«-Beziehungen kenntlich zu machen. Der Pfeil weist auf die Klasse, die instanziert wird. In [Abbildung B.4c](#) ist zu sehen, dass die Klasse `CreationTool` `LineShape`-Objekte erzeugt.

Die OMT definiert außerdem einen gefüllten Kreis, um »mehr als eins« anzuzeigen. Erscheint dieser Kreis am Kopfende einer Referenz, bedeutet dies, dass mehrere Objekte referenziert oder aggregiert werden.

Und schließlich wurde die OMT-Notation noch um Pseudocode-Annotationen erweitert, um die Skizzierung von Operationsimplementierungen zu ermöglichen. [Abbildung B.4d](#) zeigt die Pseudocode-Annotation der `Draw`-Operation der `Drawing`-Klasse.

B.2 Objektdiagramme

Ein Objektdiagramm zeigt ausschließlich Instanzen. Es stellt eine Momentaufnahme der Objekte in einem Design Pattern dar. Die Objekte sind mit »`aName`« bezeichnet, wobei »Name« die Klasse des Objekts angibt. Das Symbol für ein Objekt (in leicht modifizierter Form aus der Standard-OMT übernommen) ist ein Kasten mit abgerundeten Ecken, in dem der Objektname durch eine Trennlinie von möglichen Objektreferenzen separiert ist. Die von dem Objekt ausgehenden Pfeile kennzeichnen die referenzierten Objekte (siehe [Abbildung B.5](#)):

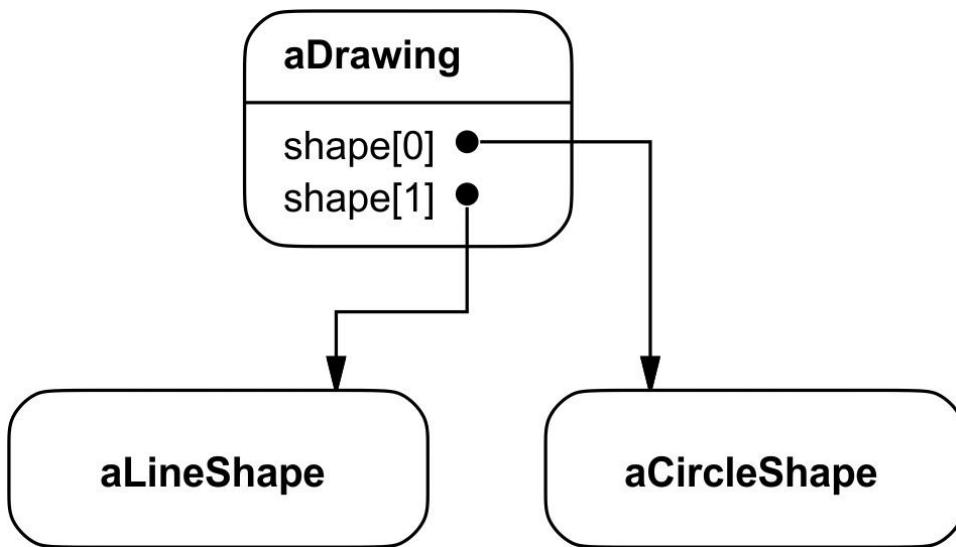


Abb. B.5: Notation für Objektdiagramme

B.3 Interaktionsdiagramme

Ein Interaktionsdiagramm veranschaulicht die Abfolge der Request-Ausführung zwischen den Objekten. Das in [Abbildung B.6](#) dargestellte Interaktionsdiagramm zeigt die Ergänzung eines Shape-Objekts zu Drawing:

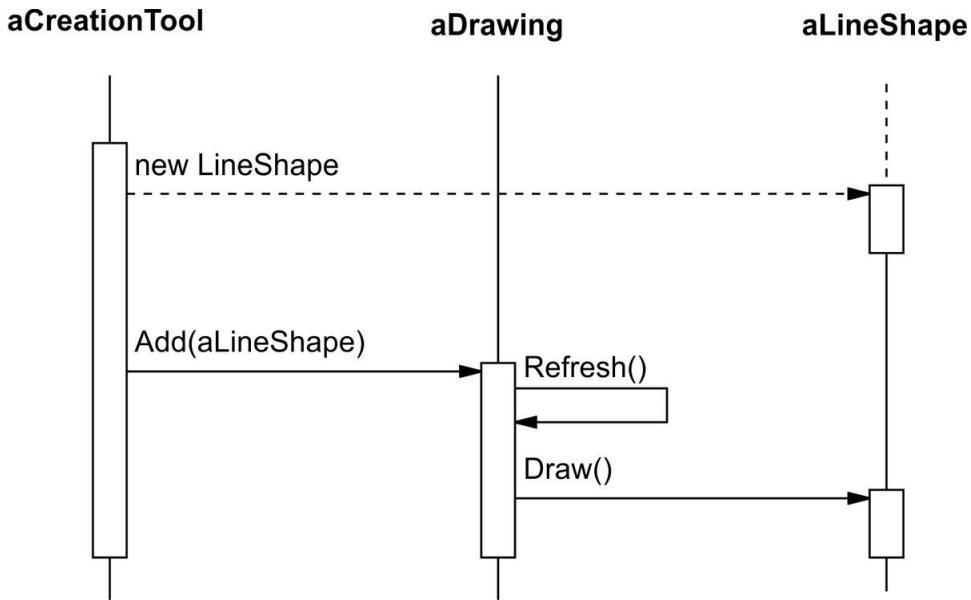


Abb. B.6: Notation für Interaktionsdiagramme

Die Zeitschiene in einem Interaktionsdiagramm verläuft von oben nach unten. Eine durchgezogene vertikale Linie gibt die Lebensdauer eines bestimmten Objekts an. Die Namenskonvention für Objekte entspricht der in Objektdiagrammen: Dem Klassennamen wird der Buchstabe »a« vorangestellt (z. B. ashape). Wenn das Objekt nicht vor Beginn der Zeitaufzeichnung in dem Diagramm instanziert wird, wird bis zum Zeitpunkt der Erzeugung eine gestrichelte Linie verwendet.

Ein vertikales Rechteck gibt an, dass ein Objekt aktiv ist bzw. einen Request bearbeitet. Die Operation kann auch Requests an andere Objekte weiterleiten. Dies wird durch einen horizontalen Pfeil angezeigt, der auf das empfangende Objekt weist. Oberhalb des Pfeils ist die Bezeichnung des Requests angegeben. Ein Request zur Erzeugung eines Objekts wird durch eine gestrichelte Pfeillinie gekennzeichnet. Richtet sie ein Request an das sendende Objekt, wird dies durch einen auf das Objekt zurückweisenden Pfeil dargestellt.

In [Abbildung B.6](#) ist zu sehen, dass der erste Request von aCreationTool die Erzeugung von aLineShape anweist. Später wird aLineShape mithilfe der Operation Add zu aDrawing hinzugefügt, woraufhin aDrawing einen Refresh-Request an sich

selbst sendet. Beachtenswert ist hier, dass aDrawing im Rahmen der Refresh-Operation auch einen Draw-Request an aLineShape ausgibt.

Anhang C: Fundamentale Klassen

Dieser Anhang dokumentiert die grundlegenden Klassen, die in den C++-Codebeispielen zu einigen der in diesem Buch vorgestellten Design Patterns verwendet werden. Sie sind ganz bewusst einfach und minimalistisch gehalten. Dabei handelt es sich um folgende Klassen:

- **List**

Eine geordnete Objektliste.

- **Iterator**

Die Schnittstelle für den sukzessiven Zugriff auf die Objekte eines Aggregats.

- **ListIterator**

Ein Iterator zum Traversieren einer **List**-Klasse.

- **Point**

Ein zweidimensionaler Punkt.

- **Rect**

Ein an den Koordinatenachsen ausgerichtetes Rechteck.

Einige der neueren C++-Standardtypen sind möglicherweise nicht in allen Compilern verfügbar. Sollte ein Compiler insbesondere den Typ `bool` nicht bereitstellen, kann dieser wie folgt manuell definiert werden:

```
typedef int bool;
const int true = 1;
const int false = 0;
```

C.1 Die Klasse List

Das **List**-Klassentemplate stellt einen grundlegenden Container zur Speicherung einer geordneten Objektliste zur Verfügung. **List** speichert Elemente als Werte, so

dass sie sowohl mit Built-in-Typen als auch mit Klasseninstanzen funktionieren. `List<int>` deklariert zum Beispiel eine Liste von `int`-Objekten. Die meisten Design Patterns verwenden die Klasse `List` jedoch zum Speichern von Pointern auf Objekte, wie beispielsweise `List<Glyph*>`. Auf diese Weise kann sie für heterogene Listen verwendet werden.

Komfortablerweise bietet die `List`-Klasse auch Synonyme für Stack-Operationen an, die Code, in dem die Klasse für Stacks genutzt wird, auch ohne die Definition einer weiteren Klasse expliziter macht:

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);
    List(List&);
    ~List();
    List& operator=(const List&);

    long Count() const;
    Item& Get(long index) const;
    Item& First() const;
    Item& Last() const;
    bool Includes(const Item&) const;

    void Append(const Item&);
    void Prepend(const Item&);

    void Remove(const Item&);
    void RemoveLast();
    void RemoveFirst();
    void RemoveAll();

    Item& Top() const;
    void Push(const Item&);
    Item& Pop();
};
```

Diese Operationen werden in den nachfolgenden Abschnitten genauer erläutert.

Erzeugung, Destruktion, Initialisierung und Zuweisung

- `List(long size)`

Initialisiert die Liste. Der Parameter `size` liefert einen Hinweis für die ursprüngliche Anzahl der Elemente.

- `List(List&)`

Überschreibt den Standard-Kopierkonstruktor, damit die Memberdaten korrekt initialisiert werden.

- `~List()`

Gibt die internen Datenstrukturen, nicht jedoch die Elemente der Liste frei. Da diese Klasse nicht für die Unterklassenbildung gedacht ist, besitzt sie keinen virtuellen Destruktor.

- `List& operator=(const List&)`

Implementiert die Zuweisungsoperation für die korrekte Zuordnung der Memberdaten.

Zugriff

Die folgenden Operationen ermöglichen den grundlegenden Zugriff auf die Elemente der Liste:

- `long Count() const`

Gibt die Anzahl der in der Liste enthaltenen Objekte zurück.

- `Item& Get(long index) const`

Gibt das Objekt an dem übergebenen Index zurück.

- `Item& First() const`

Gibt das erste Objekt in der Liste zurück.

- `Item& Last() const`

Gibt das letzte Objekt in der Liste zurück.

Hinzufügen

- `void Append(const Item&)`

Fügt das Argument als letztes Element zu der Liste hinzu.

- `void Prepend(const Item&)`

Fügt das Argument als erstes Element in die Liste ein.

Entfernen

- `void Remove(const Item&)`

Entfernt das betreffende Element aus der Liste. Für diese Operation muss der Typ der Listenelemente den Vergleichsoperator == unterstützen.

- `void RemoveFirst()`

Entfernt das erste Element aus der Liste.

- `void RemoveLast()`

Entfernt das letzte Element aus der Liste.

- `void RemoveAll()`

Entfernt alle Elemente aus der Liste.

Stack-Schnittstelle

- `Item& Top() const`

Gibt das oberste Element zurück (wenn die Liste als Stack behandelt wird).

- `void Push(const Item&)`

Legt das Element auf den Stack.

- `Item& Pop()`

Nimmt das oberste Element vom Stack.

C.2 Iterator

Iterator ist eine abstrakte Klasse, die eine Traversierungsschnittstelle für Aggregate definiert:

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

Die Operationen wirken sich folgendermaßen aus:

- `virtual void First()`

Positioniert den Iterator auf das erste Objekt im Aggregat.

- `virtual void Next()`

Positioniert den Iterator auf das nächstfolgende Objekt.

- `virtual bool IsDone() const`

Gibt true zurück, wenn keine weiteren Objekte mehr folgen.

- `virtual Item CurrentItem() const`

Gibt das Objekt an der aktuellen Position in der Objektsammlung zurück.

C.3 ListIterator

ListIterator implementiert die Iterator-Schnittstelle zur Traversierung von List-Objekten. Sein Konstruktor nimmt eine Traversierungsliste als Argument entgegen:

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
```

```

    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
};
```

C.4 Point

Die Klasse `Point` repräsentiert einen Punkt in einem zweidimensionalen kartesischen Koordinatensystem. Sie unterstützt elementare Vektorarithmetik. Die Koordinaten eines `Point`-Objekts sind wie folgt definiert:

```
typedef float Coord;
```

Die Operationen der Klasse `Point` sind selbsterklärend:

```

class Point {
public:
    static const Point Zero;

    Point(Coord x = 0.0, Coord y = 0.0);

    Coord X() const; void X(Coord x);
    Coord Y() const; void Y(Coord y);

    friend Point operator+(const Point&, const Point&);
    friend Point operator-(const Point&, const Point&);
    friend Point operator*(const Point&, const Point&);
    friend Point operator/(const Point&, const Point&);

    Point& operator+=(const Point&);
    Point& operator-=(const Point&);
    Point& operator*=(const Point&);
    Point& operator/=(const Point&);

    Point operator-();

    friend bool operator==(const Point&, const Point&);
    friend bool operator!=(const Point&, const Point&);

    friend ostream& operator<<(ostream&, const Point&);
    friend istream& operator>>(istream&, Point&);
};
```

Die statische Memberfunktion `Zero` repräsentiert `Point(0, 0)`.

C.5 Rect

Rect repräsentiert ein an den Achsen eines Koordinatensystems ausgerichtetes Rechteck. Es ist durch einen Ursprungspunkt und seine Ausmaße (sprich Breite und Höhe) definiert. Die Rect-Operationen sind selbsterklärend:

```
class Rect {  
public:  
    static const Rect Zero;  
  
    Rect(Coord x, Coord y, Coord w, Coord h);  
    Rect(const Point& origin, const Point& extent);  
  
    Coord Width() const; void Width(Coord);  
    Coord Height() const; void Height(Coord);  
    Coord Left() const; void Left(Coord);  
    Coord Bottom() const; void Bottom(Coord);  
  
    Point& Origin() const; void Origin(const Point&);  
    Point& Extent() const; void Extent(const Point&);  
  
    void MoveTo(const Point&);  
    void MoveBy(const Point&);  
  
    bool IsEmpty() const;  
    bool Contains(const Point&) const;  
};
```

Die statische Memberfunktion Zero entspricht dem Rechteck `Rect(Point(0, 0), Point(0, 0))`.

Anhang D: Quellenverzeichnis

- [Add94] Addison-Wesley, Reading, MA. *NEXTSTEP General Reference: Release 3, Volumes 1 und 2*, 1994.
- [AG90] D.B. Anderson und S. Gossain. Hierarchy evolution and the software lifecycle. In *TOOLS '90 Conference Proceedings*, S. 41 – 50, Paris, Juni 1990. Prentice Hall.
- [AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King und Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.
- [App89] Apple Computer, Inc., Cupertino, CA. *Macintosh Programmers Workshop Pascal 3.0 Reference*, 1989.
- [App92] Apple Computer, Inc., Cupertino, CA. *Dylan. An object-oriented dynamic language*, 1992.
- [Arv91] James Arvo. *Graphics Gems II*. Academic Press, Boston, MA, 1991.
- [AS85] B. Adelson und E. Soloway. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11(11):1351 – 1360, 1985.
- [BE93] Andreas Birrer und Thomas Eggenschwiler. Frameworks in the financial engineering domain: An experience report. In *European Conference on Object-Oriented Programming*, S. 21 – 35, Kaiserslautern, Deutschland, Juli 1993. Springer-Verlag.

- [BJ94] Kent Beck und Ralph Johnson. Patterns generate architectures. In *European Conference on Object-Oriented Programming*, S. 139 – 149, Bologna, Italien, Juli 1994. Springer-Verlag.
-
- [Boo94] Grady Booch. *Object-Oriented Analysis und Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1994. Second Edition.
-
- [Bor81] A. Borning. The programming language aspects of ThingLab – a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages und Systems*, 3(4):343 – 387, Oktober 1981.
-
- [Bor94] Borland International, Inc., Scotts Valley, CA. *A Technical Comparison of Borland ObjectWindows 2.0 und Microsoft MFC 2.5*, 1994.
-
- [BV90] Grady Booch und Michael Vilot. The design of the C++ Booch components. In *Object-Oriented Programming Systems, Languages und Applications Conference Proceedings*, S. 1 – 11, Ottawa, Kanada, Oktober 1990. ACM Press.
-
- [Cal93] Paul R. Calder. *Building User Interfaces with Lightweight Objects*. PhD thesis, Stanford University, 1993.
-
- [Car89] J. Carolan. Constructing bullet-proof classes. In *Proceedings C++ at Work '89*. SIGS Publications, 1989.
-
- [Car92] Tom Cargill. *C++ Programming Style*. Addison-Wesley, Reading, MA, 1992.
-
- Roy H. Campbell, Nayeem Islam, David Raila und Peter Madeany. Designing and implementing Choices: An object-oriented system

[CIRM93] in C++. *Communications of the ACM*, 36(9):117 – 126, September 1993.

[CL90] Paul R. Calder und Mark A. Linton. Glyphs: Flyweight objects for user interfaces. In *ACM User Interface Software Technologies Conference*, S. 92 – 101, Snowbird, UT, Oktober 1990.

[CL92] Paul R. Calder und Mark A. Linton. The object-oriented implementation of a document editor. In *Object-Oriented Programming Systems, Languages und Applications Conference Proceedings*, S. 154 – 165, Vancouver, British Columbia, Kanada, Oktober 1992. ACM Press.

[Coa92] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152 – 159, September 1992.

[Coo92] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Object-Oriented Programming Systems, Languages und Applications Conference Proceedings*, S. 1 – 15, Vancouver, British Columbia, Kanada, Oktober 1992. ACM Press.

[Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA, 1992.

[Cur89] Bill Curtis. Cognitive issues in reusing software artifacts. In Ted J. Biggerstaff und Alan J. Perlis, Hrsg., *Software Reusability, Volume II: Applications and Experience*, S. 269 – 287. Addison-Wesley, Reading, MA, 1989.

[dCLF93] Dennis de Champeaux, Doug Lea und Penelope Faure. *Object-Oriented System Development*. Addison-Wesley, Reading, MA, 1993.

- [Deu89] L. Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In Ted J. Biggerstaff und Alan J. Perlis, Hrsg., *Software Reusability, Volume II: Applications und Experience*, S. 57 – 71. Addison-Wesley, Reading, MA, 1989.
-
- [Ede92] D. R. Edelson. Smart pointers: They're smart, but they're not pointers. In *Proceedings of the 1992 USENIX C++ Conference*, S. 1 – 19, Portland, OR, August 1992. USENIX Association.
-
- [EG92] Thomas Eggenschwiler und Erich Gamma. The ET++SwapsManager: Using object technology in the financial engineering domain. In *Object-Oriented Programming Systems, Languages und Applications Conference Proceedings*, S. 166 – 178, Vancouver, British Columbia, Kanada, Oktober 1992. ACM Press.
-
- [ES90] Margaret A. Ellis und Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.
-
- [Foo92] Brian Foote. A fractal model of the lifecycles of reusable objects. *OOPSLA '92 Workshop on Reuse*, Oktober 1992. Vancouver, British Columbia, Kanada.
-
- [GA89] S. Gossain und D.B. Anderson. Designing a class hierarchy for domain representation and reusability. In *TOOLS '89 Conference Proceedings*, S. 201 – 210, CNIT Paris – La Defense, Frankreich, November 1989. Prentice Hall.
-
- [Gam91] Erich Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge*. Dissertation, Universität Zürich, Institut für Informatik, 1991.
-
- [Gam92] Erich Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge*. Springer-Verlag, Berlin, 1992.

- [Gla90] Andrew Glassner. *Graphics Gems*. Academic Press, Boston, MA, 1990.
- [GM92] M. Graham und E. Mettala. The Domain-Specific Software Architecture Program. In *Proceedings of DARPA Software Technology Conference*, 1992, S. 204 – 210, April 1992. Ebenfalls erschienen in *CrossTalk, The Journal of Defense Software Engineering*, S. 19 – 21, 32, Oktober 1992.
- [GR83] Adele J. Goldberg und David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.
- [HHMV92] Richard Helm, Tien Huynh, Kim Marriott und John Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, S. 1 – 22, Champéry, Schweiz, Oktober 1992. Ebenfalls verfügbar als IBM Research Division Technical Report RC 18524 (79392).
- [HO87] Daniel C. Halbert und Patrick D. O'Brien. Object-oriented development. *IEEE Software*, 4(5):71 – 79, September 1987.
- [ION94] IONA Technologies, Ltd., Dublin, Ireland. *Programmer's Guide for Orbix, Version 1.2*, 1994.
- [JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson und Gunnar Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, Wokingham, England, 1992.
- [JF88] Ralph E. Johnson und Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22 – 35Juni/Juli 1988.

- [JML92] Ralph E. Johnson, Carl McConnell und J. Michael Lake. The RTL system: A framework for code optimization. In Robert Giegerich und Susan L. Graham, Hrsg., *Code Generation – Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, S. 255 – 274, Dagstuhl, Deutschland, 1992. Springer-Verlag.
-
- [Joh92] Ralph Johnson. Documenting frameworks using patterns. In *Object-Oriented Programming Systems, Languages und Applications Conference Proceedings*, S. 63 – 76, Vancouver, British Columbia, Kanada, Oktober 1992. ACM Press.
-
- [JZ91] Ralph E. Johnson und Jonathan Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, 4(11):22 – 35, November 1991.
-
- [Kir92] David Kirk. *Graphics Gems III*. Harcourt, Brace, Jovanovich, Boston, MA, 1992.
-
- [Knu73] Donald E. Knuth. *The Art of Computer Programming, Volumes 1, 2 und 3*. Addison-Wesley, Reading, MA, 1973.
-
- [Knu84] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Reading, MA, 1984.
-
- [Kof93] Thomas Kofler. Robust iterators in ET++. *Structured Programming*, 14:62 – 85, März 1993.
-
- [KP88] Glenn E. Krasner und Stephen T. Pope. A cookbook for using the modelview controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26 – 49, August/September 1988.
-
- Wilf LaLonde. *Discovering Smalltalk*. Benjamin/Cummings,

- [LaL94] Redwood City, CA, 1994.
-
- [LCI+92] Mark Linton, Paul Calder, John Interrante, Steven Tang und John Vlissides. *InterViews Reference Manual*. CSL, Stanford University, 3.1 edition, 1992.
-
- [Lea88] Doug Lea. libg++, the GNU C++ library. In *Proceedings of the 1988 USENIX C++ Conference*, S. 243 – 256, Denver, CO, Oktober 1988. USENIX Association.
-
- [LG86] Barbara Liskov und John Guttag. *Abstraction and Specification in Program Development*. McGraw-Hill, New York, 1986.
-
- [Lie85] Henry Lieberman. There's more to menu systems than meets the screen. In *SIGGRAPH Computer Graphics*, S. 181 – 189, San Francisco, CA, Juli 1985.
-
- [Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, S. 214 – 223, Portland, OR, November 1986.
-
- [Lin92] Mark A. Linton. Encapsulating a C++ library. In *Proceedings of the 1992 USENIX C++ Conference*, S. 57 – 66, Portland, OR, August 1992. ACM Press.
-
- [LP93] Mark Linton und Chuck Price. Building distributed user interfaces with Fresco. In *Proceedings of the 7th X Technical Conference*, S. 77 – 87, Boston, MA, Januar 1993.
-
- [LR93] Daniel C. Lynch und Marshall T. Rose. *Internet System Handbook*. Addison-Wesley, Reading, MA, 1993.

- [LVC89] Mark A. Linton, John M. Vlissides und Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8 – 22, Februar 1989.
-
- [Mar91] Bruce Martin. The separation of interface und implementation in C++. In *Proceedings of the 1991 USLNIX C++ Conference*, S. 51 – 63, Washington, D.C., April 1991. USENIX Association.
-
- [McC87] Paul McCullough. Transparent forwarding: First steps. In *Object-Oriented Programming Systems, Languages und Applications Conference Proceedings*, S. 331 – 341, Orlando, FL, Oktober 1987. ACM Press.
-
- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.
-
- [Mur93] Robert B. Murray. *C++ Strategies and Tactics*. Addison-Wesley, Reading, MA, 1993.
-
- [OJ90] William F. Opdyke und Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA Conference Proceedings*, S. 145 – 161, Marist College, Poughkeepsie, NY, September 1990. ACM Press.
-
- [OJ93] William F. Opdyke und Ralph E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of the 21st Annual Computer Science Conference (ACM CSC '93)*, S. 66 – 73, Indianapolis, IN, Februar 1993.
-
- [P+88] Andrew J. Palay et al. The Andrew Toolkit: An overview. In *Proceedings of the 1988 Winter USENIX Technical Conference*, S. 9 – 21, Dallas, TX, Februar 1988. USENIX Association.

- [Par90] ParcPlace Systems, Mountain View, CA. *ObjectWorks\Smalltalk Release 4 Users Guide*, 1990.
-
- [Pas86] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, S. 341 – 346, Portland, OR, Oktober 1986. ACM Press.
-
- [Pug90] William Pugh. Skiplists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668 – 676, Juni 1990.
-
- [RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy und William Loreson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
-
- [Rum94] James Rumbaugh. The life of an object model: How the object model changes during development. *Journal of Object-Oriented Programming*, 7(1):24 – 32, März/April 1994.
-
- [SE84] Elliot Soloway und Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595 – 609, September 1984.
-
- [Sha90] Yen-Ping Shan. MoDE: A UIMS for Smalltalk. In *ACM OOPSLA/ECOOP '90 Conference Proceedings*, S. 258 – 268, Ottawa, Ontario, Kanada, Oktober 1990. ACM Press.
-
- [Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented languages. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, S. 38 – 45, Portland, OR, November 1986. ACM Press.
-
- [SS86] James C. Spohrer und Elliot Soloway. Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7):624 –

-
- [SS94] Douglas C. Schmidt und Tatsuya Suda. The Service Configurator Framework: An extensible architecture for dynamically configuring concurrent, multi-service network daemons. In *Proceeding of the Second International Workshop on Configurable Distributed Systems*, S. 190 – 201, Pittsburgh, PA, März 1994. IEEE Computer Society.
-
- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1991. Second Edition.
-
- [Str93] Paul S. Strauss. IRIS Inventor, a 3D graphics toolkit. In *Object-Oriented Programming Systems, Languages und Applications Conference Proceedings*, S. 192 – 200, Washington, D.C., September 1993. ACM Press.
-
- [Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.
-
- [Sut63] I.E. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, MIT, 1963.
-
- [Swe85] Richard E. Sweet. The Mesa programming environment. *SIGPLAN Notices*, 20(7):216 – 229, Juli 1985.
-
- [Sym93a] Symantec Corporation, Cupertino, CA. *Bedrock Developer's Architecture Kit*, 1993.
-
- [Sym93b] Symantec Corporation, Cupertino, CA. *THINK Class Library Guide*, 1993.

- [Sza92] Duane Szafron. SPECTalk: An object-oriented data specification language. In *Technology of Object-Oriented Languages and Systems (TOOLS 8)*, S. 123 – 138, Santa Barbara, CA, August 1992. Prentice Hall.
-
- [US87] David Ungar und Randall B. Smith. Self: The power of simplicity. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, S. 227 – 242, Orlando, FL, Oktober 1987. ACM Press.
-
- [VL88] John M. Vlissides und Mark A. Linton. Applying object-oriented design to structured graphics. In *Proceedings of the 1988 USENIX C++ Conference*, S. 81 – 94, Denver, CO, Oktober 1988. USENIX Association.
-
- [VL90] John M. Vlissides und Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237 – 268, Juli 1990.
-
- [WBJ90] Rebecca Wirfs-Brock und Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, 33(9): 104 – 124, 1990.
-
- [WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson und Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, NJ, 1990.
-
- [WGM88] Andre Weinand, Erich Gamma und Rudolf Marty. ET++ – An object-oriented application framework in C++. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, S. 46 – 57, San Diego, CA, September 1988. ACM Press.