# Software Engineering

Grundlagen, Menschen, Prozesse, Techniken

# 16 Entwurf

*Im Anfang schuf Gott Himmel und Erde; die Erde aber war wüst und wirr, Finsternis lag über der Urflut, und Gottes Geist schwebte über dem Wasser. Gott sprach: Es werde Licht. Und es ward Licht.*

Die Bibel, Erstes Buch Mose, 1, 1-3

In der Genesis ist die Schaffung einer Ordnung aus dem *Tohuwabohu* der erste Schritt der Schöpfung. Die Ordnung, die Struktur, ist die Voraussetzung für alle weiteren Schritte. Vor der Ordnung gibt es nur den Geist und das Chaos. Die in der Schöpfung entstandenen Strukturen wie Tag und Nacht oder Himmel und Erde sind absolut stabil. Damit wird – ganz unabhängig vom religiösen Gehalt – die überragende Bedeutung der Strukturierung deutlich. Wer ein komplexes System schafft, legt Strukturen fest, die lange Zeit gültig bleiben. Darum gilt das Entwerfen auch in der Software-Entwicklung als die zentrale Tätigkeit. Fehler im Entwurf bleiben uns oft über Jahrzehnte erhalten.

Gerade von einem Kapitel über den Entwurf kann man einen ordentlichen Entwurf erwarten, also eine leicht verständliche, übersichtliche und tragfähige Struktur. Leider widersetzt sich dieses Thema jedoch einer eleganten Strukturierung. Hier nützt – wie auch an vielen anderen Stellen dieses Buches – der Blick hinüber zu den Architekten: Ein Haus kann man aus Lehm, Holz, Stein oder vielen anderen Materialien bauen, man kann es flach oder hoch, rund oder eckig machen. Diese Möglichkeiten lassen sich nicht durch eine geschlossene Entwurfslehre fassen. Das Gleiche gilt beim Software-Entwurf: Je nach Art der zu entwickelnden Software und je nach der gewählten Entwicklungsstrategie spielen ganz unterschiedliche Überlegungen eine Rolle.

Nach einigen Betrachtungen zur Bedeutung des Entwurfs (Abschnitt 16.1) führen wir Begriffe ein, die im Kontext des Software-Entwurfs wichtig sind (Abschnitt 16.2). In Abschnitt 16.3 präsentieren wir allgemeingültige Prinzipien des Entwurfs. Danach stellen wir in den Abschnitten 16.4 und 16.5 wichtige Architektur- und Entwurfsmuster vor. Ansätze zur Wiederverwendung von Komponenten und Architekturen werden in Abschnitt 16.6 diskutiert. In Abschnitt 16.7 beschreiben wir Techniken, um Anwendungssoftware zu entwerfen. Eine kurze Betrachtung über die Beurteilung der Entwurfsqualität (Abschnitt 16.8) schließt das Kapitel ab.

## 16.1 Ziele und Bedeutung des Entwurfs

Wenn ein Software-Entwickler eine sehr große, komplexe Software realisieren soll, steht er zunächst vor dem Problem, dass er die vielen Facetten des Systems und ihre Wechselwirkungen nicht überblicken kann; er muss die Software also in seiner Vorstellung gliedern. Wenn damit überschaubare Einheiten entstanden sind, muss er zweckmäßige Lösungsstrukturen festlegen. Schließlich muss er die sehr zahlreichen Bestandteile seiner Software so organisieren, dass er den Überblick behält.

Mit dem Entwurf verfolgen wir also drei Ziele, die sich nicht scharf gegeneinander abgrenzen lassen:

1. Gliederung des Systems in überschaubare (handhabbare) Einheiten
2. Festlegung der Lösungsstruktur
3. Hierarchische Gliederung

Als Software-Architektur (siehe Definition in Abschnitt 16.2.4) bezeichnen wir vor allem das Resultat der Gliederung (Schritt 1), die eine Lösungsstruktur auf höchster Ebene festlegt (Schritt 2).

### 16.1.1 Die Gliederung in überschaubare Einheiten

Man vergleiche unsere Situation mit der Aufgabe, ein großes Bauwerk an einen anderen Standort zu versetzen. Dazu muss man das Bauwerk zumindest so weit zerlegen, dass die einzelnen Teile handhabbar werden. Was das konkret bedeutet, hängt natürlich auch von der Ausrüstung (Kran, Fahrzeuge) und von den eigenen Fähigkeiten und Erfahrungen ab. Beim Entwurf ist es ganz ähnlich.

Es ist also nicht zu erwarten, dass die Spezifikation zu Beginn der Entwicklung vollständig vorliegt und der Entwickler sie auch vollständig kennt und versteht, bevor er einen Entwurf anfertigt. Vielmehr verhilft oft erst der Entwurf zum notwendigen Verständnis. Daher arbeiten erfahrene Entwickler mit Standardstrukturen (Abschnitt 16.4), die sie für eine große Klasse von Aufgaben einsetzen. Die Wiederverwendung vorhandener Komponenten hat eine ähnliche Wirkung wie die Verwendung einer Standardstruktur.

### 16.1.2 Die Festlegung der Lösungsstruktur

Die *Struktur* eines Gegenstands ist die Menge der Beziehungen zwischen seinen Teilen; ein Gegenstand, der nicht aus (erkennbaren) Teilen aufgebaut ist, heißt *unstrukturiert* oder *amorph*. Die Struktur ist also ein Aspekt des Gegenstandes, der von den materiellen und quantitativen Aspekten abstrahiert.

Die bemerkenswerteste Eigenschaft von Strukturen ist ihre oft überraschende Stabilität. Viele Strukturen des täglichen Lebens überdauern die Materie, an die

sie gebunden scheinen. So zeigen viele Fossilien nur noch die Gestalt, die Struktur der Organismen, deren Atome in Jahrmillionen vollständig ausgetauscht wurden. Ebenso kennen wir in der Gesellschaft viele stabile Strukturen, z. B. Dörfer, Vereine, Gesetzeswerke, Spielregeln, die sich über Jahrhunderte erhalten, ohne dass man auf irgendein Detail zeigen könnte, das unverändert geblieben ist: Das Konkrete ist vergänglich, das Abstrakte ist stabil.

Das gilt auch für Software. Wenn in einem großen System immer wieder Komponenten verändert oder ausgetauscht werden, ist nach einigen Jahren kaum noch eine Zeile aus dem Code des ursprünglichen Systems erhalten, aber die Struktur ist völlig unverändert. Da die Software-Struktur großen Einfluss auf die Brauchbarkeit (Effektivität), vor allem aber auf die Wartbarkeit hat, ist die Wahl einer sinnvollen Struktur die wichtigste (technische) Entscheidung der ganzen Software-Entwicklung.

### 16.1.3 Die hierarchische Gliederung

Ein Mensch kann nur Systeme überblicken, die aus wenigen Teilen bestehen; die Psychologie gibt dazu die Zahl sieben an (Miller, 1956), d. h., bis zu (etwa) sieben Gegenstände können ohne Weiteres erfasst werden. Sind es mehr, so muss man die Gegenstände nacheinander betrachten und ihren Zusammenhang systematisch entwickeln. Betrachtet man Deklarationen und Anweisungen als Bestandteile eines Programms, dann bestehen schon kleinere Software-Systeme aus Tausenden dieser Komponenten. Sie müssen gruppiert, also hierarchisch organisiert werden, um für uns überschaubar zu sein.

Simon (1962) zeigt mit seiner Uhrmacher-Analogie, dass uns nur die Abstraktion ermöglicht, sehr komplexe Objekte herzustellen. Von zwei gleich kompetenten Uhrmachern, die beide ihre Uhren aus jeweils hundert Teilen zusammensetzen, wird einer sehr erfolgreich, während der andere scheitert. Der Erfolgreiche hat die Teile jeweils in Baugruppen zu zehn Teilen vormontiert, die am Schluss zusammengefügt werden; der andere setzt in einem einzigen Schritt alle hundert Teile zusammen. Stört ein Kunde, der eine Uhr bestellen oder abholen will, die Montage, so verliert der eine die Arbeit von wenigen Minuten, der andere von Stunden, und es gelingt ihm schließlich nicht mehr, auch nur eine einzige Uhr fertigzustellen.

Darum kann man nur kleine Programme direkt auf der Grundlage der Anforderungen codieren; alle anderen müssen schrittweise entwickelt werden.

Da man bei der Vorbereitung eines Hausbaus ähnlich vorgeht, bietet sich die *Architektur-Metapher* an. So, wie die Architektur eines Gebäudes die Formen und Beziehungen der Hauptbestandteile, also der Wände, Decken, Dachflächen, Türen und Fenster, vorgibt oder beschreibt, so macht die Software-Architektur Aussagen über die Gestalt, die Funktionen und Beziehungen der Software-Komponenten.

Durch die sinnvolle Gliederung (und nur durch sie) entsteht die Möglichkeit zur Abstraktion.

### 16.1.4 Die Entwicklungsrichtung

Ist die Verwendung einer Standardarchitektur nicht möglich oder nicht sinnvoll, muss der Entwurf sukzessive entstehen. Dabei kann man vier verschiedene Vorgehensrichtungen unterscheiden, die mit den Schlagwörtern *top-down*, *bottom-up*, *outside-in* und *inside-out* bezeichnet werden. Die mit dieser Wortwahl verbundene räumliche Vorstellung ist in Abbildung 16–1 dargestellt.

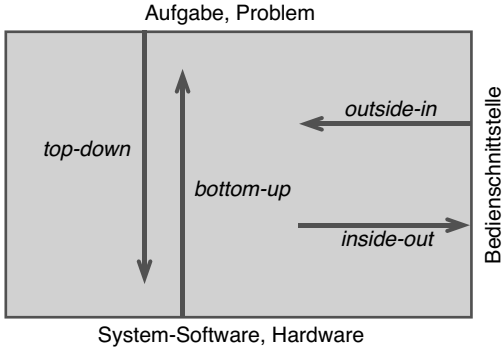



***Abb. 16–1*** *Vorgehensrichtungen beim Software-Entwurf*

Geht man von der (abstrakten) Aufgabe zur konkreten, detaillierten Lösung des Problems, so bezeichnet man das als *Top-down-Entwicklung*. Dabei zerlegt man die Aufgabe rekursiv in Teilaufgaben, bis die elementare Ebene (der Befehlsvorrat der Programmiersprache und die Betriebssystemaufrufe) erreicht ist. Man bezeichnet dieses Verfahren nach Niklaus Wirth (1971) auch als *Schrittweise Verfeinerung*. Wenn auf der Ebene der Implementierung keine ungewöhnlichen Probleme zu erwarten sind, ist die Top-down-Entwicklung der übliche Ansatz.

Diesem analytischen Vorgehen steht die Entwicklung *bottom-up* gegenüber, bei der der Entwickler die Befehle so lange synthetisiert (kombiniert), bis eine Gesamtlösung entstanden ist. Wer auf einer exotischen Hardware implementiert, wird wahrscheinlich bottom-up beginnen, also zunächst den virtuellen Rechner realisieren, der die benötigten Operationen anbietet.

Geht man von der Bedienoberfläche aus in Richtung auf die Datenstrukturen und Algorithmen vor, so erfolgt die Entwicklung *outside-in*. Das ist sinnvoll, wenn die Schnittstelle feststeht, die Implementierung aber offen ist. Sehr oft entsteht diese Situation, wenn die Kunden bereits mit einem Prototyp der Bedienoberfläche einverstanden waren.

Beginnt man umgekehrt bei den »Innereien« des Systems und arbeitet von dort zur Bedienoberfläche hin, so handelt es sich um eine *Inside-out-Entwicklung*. Sie ist typisch für eine Situation, in der ein bestehendes (Informations-)System um eine neue Funktion erweitert wird, z. B. eine (nur vage spezifizierte) grafische Ausgabe.

In der Praxis lassen sich die Ansätze nicht in Reinform anwenden. Um den Weg zur Lösung zu finden, muss man in jedem Fall beides kennen, den Ausgangspunkt und den Zielpunkt. Die zweckmäßige Vorgehensrichtung hängt von den konkreten Randbedingungen ab. Wenn bestimmte Punkte hart vorgegeben sind, etwa die Datenbank in einem bestehenden System oder eine präzise Spezifikation der Funktionalität, dann ist es zweckmäßig, von diesem Fixpunkt auszugehen, sich also vom Vorgegebenen zum Gestaltbaren hin zu bewegen. Da Entwickler in der Regel mit der Programmiersprache und dem Betriebssystem souverän umgehen können, ist diese Seite für sie gestaltbar, sie arbeiten also top-down.

### 16.1.5 Der Entwurf in der Praxis

In der Praxis wird dem Entwurf nicht immer die Aufmerksamkeit zuteil, die seiner Bedeutung angemessen wäre. Allgemeine, den Entwicklern bekannte Grundsätze für die Gestaltung der Systeme werden als ausreichende Festlegung der Architektur betrachtet, ein Entwurf findet erst auf Detailebene statt. Die Architektur wird in dieser Situation natürlich auch nur unzureichend dokumentiert.

Sie sollte aber dauerhaft und prüfbar dokumentiert werden, denn sonst ist es unmöglich, sie zu diskutieren und zu prüfen und so das Risiko einer schlechten Architektur zu verringern. Außerdem kann nur eine dokumentierte Architektur von mehreren Personen parallel implementiert werden. Ob für die Beschreibung eine wohldefinierte Sprache eingesetzt wird oder eine frei gewählte Notation, typischerweise aus natürlicher Sprache und einfachen Grafiken, hängt vom Projekt und von den Fähigkeiten und Möglichkeiten der Beteiligten ab.

Im Laufe der Wartung besteht die Gefahr, dass die Architektur durch eine Vielzahl kleiner Modifikationen schleichend korrumpiert wird und dadurch nicht nur Klarheit und Verständlichkeit einbüßt, sondern sich auch von ihrer Beschreibung entfernt, in der die Änderungen meist nicht nachgeführt werden. Damit verliert die Architekturbeschreibung ihren Wert für die Wartung der Software.

Cyril Northcote Parkinson (1909–1993)[1] hat festgestellt, dass der Aufwand für Entscheidungen in Organisationen keineswegs proportional zum Risiko ist, sondern oft eher reziprok: Details werden von Fachleuten ausführlich erörtert und systematisch geklärt, aber die wirklich wichtigen Fragen werden nicht diskutiert, weil die einen nichts davon verstehen, die anderen keine Hoffnung haben,

1 Cyril Northcote Parkinson hat in seinen sehr britischen und höchst unterhaltsamen Büchern das Wuchern der Verwaltungen beschrieben. Auf Deutsch ist vor allem »Parkinsons Gesetz« bekannt (Original: The Pursuit of Progress. John Murray, London, 1958).

die Probleme verständlich zu machen; darum wird die Vorlage einfach abgenickt, auch wenn es gute Gründe gibt, sie abzulehnen. Ähnliches gilt auch für den Architekturentwurf. Obwohl seine Bedeutung allgemein anerkannt ist, wird in der Praxis sehr oft eher zufällig irgendeine Struktur ungeprüft akzeptiert oder von einem Vorgängersystem übernommen und anschließend in Code gegossen.

Dies ist insbesondere deshalb problematisch, weil sich die statische Struktur später nur noch mit hohem, meist unangemessenem Aufwand verändern lässt. Was als romanische Basilika gebaut wurde, wird niemals zu einer freitragenden Messehalle. Die Systemfunktion stellt die Anforderungen an die statische Struktur. Die Funktion kann aber später trotzdem meist relativ leicht verändert werden.

## 16.2 Begriffe

Im Bauwesen kann man auf viele wohlverstandene Begriffe wie »Tür« und »Dach« zurückgreifen, im Software Engineering fehlen uns nur allzu oft klare Begriffe. Die Literatur bietet eine Reihe von Definitionen an, die aber kein konsistentes System ergeben. Wir müssen also auswählen, was zusammenpasst, oder selbst definieren, was wir brauchen.

### 16.2.1 System

Informatiker weichen gern auf das Wort »System« aus, wenn sie keinen treffenderen Ausdruck finden. Die meisten Definitionen, die für den Systembegriff angegeben werden, führen über verwandte Begriffe wie Konfiguration und Komponente wieder zum System zurück, sind also zyklisch. Wir müssen uns anscheinend damit abfinden, dass wir über den intuitiven Systembegriff nicht hinauskommen, so wie er beispielsweise im IEEE-Standard 1471 (2000) eingeführt wird:

> **system** — A collection of components organized to accomplish a specific function or set of functions.
>
> IEEE Std 1471 (2000)

Leider ist der Systembegriff nicht sehr aussagekräftig, ein System besteht typischerweise – aber nicht zwangsläufig – aus mehreren miteinander vernetzten Komponenten, es ist (physisch oder logisch) von seiner Umgebung unterscheidbar, und es gibt einen – meist funktionalen – Aspekt, unter dem das System eine Einheit darstellt.

Ein Software-System ist demnach ein System, dessen Komponenten Software sind. Genauer gesagt sind die Komponenten einzelne Software-Einheiten oder bestehen aus Software-Einheiten. Wir können also definieren:

Ein **Software-System** ist eine Menge von Software-Einheiten und ihren Beziehungen, wenn sie gemeinsam einem bestimmten Zweck dienen. Dieser Zweck ist im Allgemeinen komplex, er schließt neben der Bereitstellung eines ausführbaren Programms (oder auch mehrerer) gewöhnlich die Organisation, Verwendung, Erhaltung und Weiterentwicklung ein.

Die Software-Einheiten werden im Kontext der Konfigurationsverwaltung definiert (siehe Abschnitt 20.1.1).

### 16.2.2 Komponente und Modul

Der Systembegriff stützt sich auf den Komponentenbegriff, der leider ähnlich schwammig ist, insbesondere, seit er im Zusammenhang mit der komponentenbasierten Software-Entwicklung gesehen wird. Das IEEE-Glossar enthält eine recht allgemeine Definition dieses Begriffs.

**component** — One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components.

IEEE Std 610.12 (1990)

Taylor, Medvidovic und Dashofy definieren den Begriff im Kontext von Software-Architekturen wie folgt:

A **software component** is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context.

Taylor, Medvidovic, Dashofy (2010)

Eine Komponente ist somit ein Bestandteil eines Systems und ein identifizierbares Element der Architektur.

Eine Komponente bietet ihrer Umgebung eine Menge von Diensten an, die über eine wohldefinierte Schnittstelle genutzt werden können (*provided interface*). Ihre internen Operationen und Daten macht sie unzugänglich (*information hiding*, siehe Abschnitt 16.3.3). Damit eine Komponente wiederverwendet werden kann, muss zudem angegeben werden, welche Dienste anderer Komponenten in Anspruch genommen werden (*required interface*).

Eine Komponente kann die Rolle eines Dienstanbieters (*Anbieter* oder *Server*) oder die eines Dienstnutzers (*Kunde* oder *Client*) übernehmen, auch beide Rollen zugleich. Zwischen Anbieter und Kunde besteht eine (gerichtete) *Benutzt-Beziehung*. Sie erlaubt es dem Kunden, alle Dienste und Daten zu benutzen, die der Anbieter zur Verfügung stellt. Eine Komponente ist auch ein typisches »deliverable« für einen Programmierer oder für ein Programmierteam, also der Gegenstand eines Arbeitspakets.

Um Missverständnisse zu vermeiden, sprechen manche Autoren statt von einer Komponente von einem *Software-Element*, *Software-Baustein* oder auch *Architekturbaustein*.

Komponenten sind atomar oder werden verfeinert, d. h., eine Komponente kann weitere Komponenten enthalten. Beispiele für atomare Komponenten sind Klassen oder Module. Auch der Modulbegriff ist nicht klar. Eine sehr allgemeine Definition enthält die IEEE-Norm:

**module** — (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from an assembler, compiler, linkage editor, or executive routine.

(2) A logically separable part of a program.

IEEE Std 610.12 (1990)

Wir wollen diese Definition präziser fassen und definieren:

Ein **Modul** ist eine Menge von Operationen und Daten, die nur so weit von außen sichtbar sind, wie dies die Programmierer explizit zugelassen haben[2].

### 16.2.3 Schnittstelle

Wo Komponenten zusammenarbeiten sollen, müssen sie zueinanderpassen. Hierfür ist die Metapher der Schnittstelle geläufig: Schneidet man einen Gegenstand, beispielsweise einen Apfel, in zwei Teile, so entstehen zwei Oberflächen, die spiegelsymmetrisch sind. Was durch einen Schnitt von selbst entsteht, die beiden exakt zueinanderpassenden Oberflächen, muss in der Technik hergestellt werden. Das englische Wort »interface« ist darum der Realität näher, wir konstruieren keine Schnittstellen, sondern *Verbindungsstellen*.

Eine sehr allgemeine Definition für »interface« geben Bachmann et al.:

An **interface** is a boundary across which two independent entities meet and interact or communicate with each other.

Bachmann et al. (2002)

Im Kontext von Komponenten wollen wir diese Definition konkretisieren und definieren:

**Schnittstelle** — Die Grenze zwischen zwei kommunizierenden Komponenten. Die Schnittstelle einer Komponente stellt die Leistungen der Komponente für ihre Umgebung zu Verfügung und/oder fordert Leistungen, die sie aus der Umgebung benötigt.

Das bedeutet auch: Eine Schnittstelle *tut nichts und kann nichts*. Ein Adapter (synonym: Konnektor), der dazu dient, nicht zueinanderpassende Komponenten zu verbinden, *ist* keine Schnittstelle, sondern *hat* selbst Schnittstellen.

Damit wir die Schnittstelle einer Komponente nutzen können, benötigen wir detaillierte Informationen, beispielsweise über die Struktur der Schnittstelle. Diese werden in der *Schnittstellenbeschreibung* dokumentiert. Sie enthält Angaben zu folgenden Aspekten:

---

2 Wir gebrauchen hier »Modul« korrekt als Neutrum, mit dem Plural »Module«. »Der Modul« (Plural »die Moduln«) ist ein Begriff der Mechanik und der Mathematik.

- *Syntax und Art der Kommunikation:* So müssen beispielsweise Name, Anordnung und Parametertypen der angebotenen Operationen angegeben werden.
- *Funktionale Merkmale:* Dazu zählen die Dienste, die eine Komponente anbietet und die sie benötigt (z. B. die Rückgabe des Sinus-Wertes für den einzigen Parameter). Vor- und Nachbedingungen können angegeben werden, um die Dienste genauer zu beschreiben (Abschnitt 17.6.1). Zusätzlich muss das Verhalten im Fehlerfall spezifiziert werden.
- *Allgemeine Qualitätsangaben:* Da die syntaktischen und funktionalen Merkmale nicht immer ausreichen, um zu klären, ob die Komponente in eine bestimmte Umgebung passt, muss eine Schnittstellenbeschreibung auch nichtfunktionale Anforderungen enthalten. Beispielsweise kann die Antwortzeit oder die Genauigkeit des Resultats kritisch sein. Das gilt natürlich in beiden Richtungen, wenn die Komponente Dienste der Umgebung beansprucht.

Um die syntaktischen Merkmale präzise zu beschreiben, verwenden wir formale Sprachen, sogenannte *Interface Definition Languages* (IDLs). Bekannte Beispiele dafür sind die CORBA IDL und die Web Service Description Language (WSDL). Im einfachsten Fall können Schnittstellen direkt in der verwendeten Programmiersprache formuliert werden. So erlaubt es die Programmiersprache JAVA, Schnittstellen explizit (als *interfaces*) zu beschreiben. Die funktionalen und qualitativen Merkmale müssen natürlichsprachlich formuliert werden.

### 16.2.4 Entwurf und Software-Architektur

Das Wort »Entwurf« (oder »Architekturentwurf«) ist doppeldeutig, wie die nachfolgende Definition zeigt:

**design** — (1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component.

(2) The result of the process in (1).

IEEE Std 610.12 (1990)

Wir sprechen immer dann vom Entwurf, wenn die Tätigkeit, deren Resultat die Architektur ist, im Vordergrund steht. Das entspricht der Bedeutung (1) der IEEE-Definition.

Software-Architektur wird von vielen Autoren definiert (vgl. die Webseiten des SEI, SEI-Architecture, o. J.). Im IEEE-Standard 1471 (2000) finden wir:

**architecture** — The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.

IEEE Std 1471 (2000)

Bass, Clements und Kazman geben eine Definition des Software-Architekturbegriffs an, die in etwa konform zur oben zitierten ist und aus unserer Sicht die wesentlichen Aspekte enthält. Sie definieren:

> The **software architecture** of a program or computing system is the structure or structures of the system which comprise software elements, the externally visible properties of those elements, and the relationships among them.
>
> Bass, Clements, Kazman (2003)

Eine Software-Architektur besteht demnach aus Komponenten (auch »software elements« genannt) und ihren Beziehungen, insbesondere auch zur Umgebung, in verschiedenen Strukturen. So wie ein Gebäude eine tragende Struktur, ein Rohrleitungs- und ein Stromleitungsnetz enthält, so hat eine Software beispielsweise eine statische Struktur und eine Verteilungsstruktur der Komponenten. Die verschiedenen Strukturen werden durch Modelle beschrieben.

Da die Projektbeteiligten an sehr unterschiedlichen Informationen über die Architektur interessiert sind, führt der IEEE-Standard 1471 (2000) die Konzepte »Perspektive« (viewpoint) und »Sicht« (view) ein. Eine Sicht zeigt die Architektur eines Systems aus einer bestimmten Perspektive. Eine Perspektive fasst Anliegen und Informationsbedürfnisse zusammen, die durch die Interessen von Projektbeteiligten bestimmt sind. So ist beispielsweise der Projektleiter daran interessiert, wie die Entwicklung der Komponenten auf die Teams aufgeteilt werden kann. Die Betreiber des Systems hingegen möchten typischerweise wissen, auf welchen Rechnern das System installiert werden soll und welche Kommunikationsprotokolle vorgesehen sind. Entsprechende Sichten auf die Architektur liefern die geforderten spezifischen Informationen. Per definitionem fügt eine Sicht selbst keine neue Information zur Architektur hinzu, sondern fasst Informationen zusammen, die möglicherweise in verschiedenen Strukturen oder Modellen der Architektur enthalten sind.

In der Literatur gibt es verschiedene Vorschläge für Mengen von Perspektiven und Sichten, um Architekturen umfassend zu beschreiben; wegweisend ist das »4+1 View Model of Architecture« von Philippe Kruchten (1995). Vier Perspektiven haben sich in der Praxis bewährt:

- Die Sichten der *Systemperspektive* stellen dar, wie das zu entwickelnde System in die Umgebung eingebettet ist und mit welchen anderen Systemen es *wie* interagiert. Dadurch sind die Systemgrenzen und die Schnittstellen zu den Nachbarsystemen festgelegt.
- Die Sichten der *statischen Perspektive* zeigen die zentralen Komponenten der Architektur, ihre Schnittstellen und Beziehungen. In der Regel muss diese Zerlegung hierarchisch verfeinert werden, bis die einzelnen Komponenten so weit modularisiert sind, dass sie codiert werden können.
- Die Sichten der *dynamischen Perspektive* stellen dar, wie die Komponenten zur Laufzeit zusammenarbeiten (Kontrollfluss, zeitliche Abhängigkeiten), sie zeigen also das Systemverhalten. Dabei beschränkt man sich auf die Modellierung der wichtigen Interaktionen. Die Frage, inwieweit dynamische Aspekte zur Software-Architektur gehören, wird unterschiedlich beantwortet. Eine Architektur hat einen Zweck, sie ist für eine bestimmte *Funktion* geschaffen.

Diese Funktion diktiert die zentrale Anforderung, und die Architektur sollte gegen diese Anforderung geprüft werden. Trotzdem ist die Funktion nicht *Teil* der Architektur.

- Die Sichten der *Verteilungsperspektive* zeigen, wie die Komponenten der statischen Sicht auf Infrastruktur- und Hardware-Einheiten abgebildet werden. Sie stellen aber auch dar, wie die Entwicklung, der Test etc. auf die Teams der Entwicklerorganisation aufgeteilt werden.

Die Wahl der Perspektiven und Sichten hängt natürlich von den Interessen und dem Informationsbedarf der Projektbeteiligten ab. Sind diese bekannt, dann kann festgelegt werden, welche Architekturinformationen in den unterschiedlichen Sichten enthalten sein sollen.

Zusammenfassend stellen wir fest: Die Architektur einer Software ist die Gesamtheit ihrer Strukturen auf einer bestimmten Abstraktionsebene; wo nichts anderes gesagt ist, geht es um die statische Struktur der höchsten Gliederungsebene, also um die Systemarchitektur.

### 16.2.5 Architekturbeschreibung

Jede Architektur muss beschrieben werden, damit sie kommuniziert, bewertet und als Vorlage für die Codierung verwendet werden kann. Diese Beschreibung ist ein Modell der Architektur. Sie kann als Vorbild, d. h. als Bauplan für die Software, oder als Abbild der Architektur bei der Weiterentwicklung der Software genutzt werden. Verwenden wir ein Architekturmodell als Vorbild, dann bezeichnet man es auch als *Soll-Architektur*. Ist ein Architekturmodell das Abbild, dann nennen wir es *Ist-Architektur*. Der IEEE-Standard 1471 (2000) (siehe dazu Maier, Emery, Hilliard, 2001) stellt die Architekturbeschreibung in den Mittelpunkt der Betrachtung und definiert diese sehr allgemein wie folgt: »A collection of products to document an architecture.« Ellis et al. geben eine Definition für Architekturbeschreibung und Sichten an, die uns sinnvoll scheint:

> An **architectural description** is a model – document, product or other artifact – to communicate and record a system's architecture. An architectural description conveys a set of views each of which depicts the system by describing domain concerns.
>
> Ellis et al. (1996)

Um Architekturen zu beschreiben, können spezielle Architekturbeschreibungssprachen (Architecture Description Language, ADL) verwendet werden. Diese bieten eine Notation in Form einer (meist grafischen) Syntax. Die dazugehörende Semantik ist jedoch leider nicht immer präzise definiert. Mit einer bestimmten ADL kann man typischerweise eine Sicht oder mehrere Sichten einer Architektur modellieren. Medvidovic und Rosenblum definieren diesen Begriff wie folgt:

> An **ADL** for software applications focuses on the high-end structure of the overall application rather than the implementation details of any specific source module. ADLs

> provide both a concrete syntax and a conceptual framework for modelling a software system's conceptual architecture.
>
> Medvidovic, Rosenblum (1997)

Beginnend mit der Publikation von DeRemer und Kron (1976) hat das Software Engineering eine Reihe von Architekturbeschreibungssprachen hervorgebracht, die allerdings kaum eingesetzt wurden. Erst in den letzten Jahren hat sich die Sprache UML für die Beschreibung objektorientierter Architekturen durchgesetzt. An UML gibt es viel – begründete – Kritik (Parnas: »Undefined Modeling Language«), und nicht alle Aspekte der Architektur lassen sich in UML darstellen. Mangels einer Alternative sollte man UML trotzdem nutzen, wo es möglich ist.

### 16.2.6 Zusammenfassung

In den vorangegangenen Abschnitten haben wir zentrale Begriffe des Software-Entwurfs eingeführt; sie bilden einen Teil der Fachsprache eines Software-Architekten. Damit man leichter erkennt, wie die Begriffe zusammenhängen, haben wir sie in einem Begriffsmodell angeordnet.

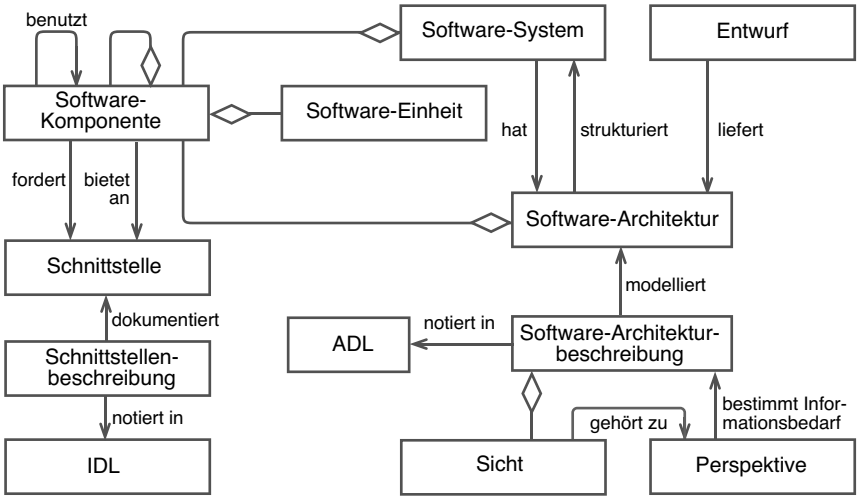



***Abb. 16–2*** *Begriffe des Software-Entwurfs*

## 16.3 Prinzipien des Architekturentwurfs

Wenn wir über die Eigenschaften einer Architektur sprechen, so stellen wir uns das System vor, das mit dieser Architektur entstehen soll. Die Software-Architektur ist also gut, wenn die an die Software gestellten funktionalen und nichtfunktionalen Anforderungen erfüllt werden können. Die Einhaltung der Anforderungen sollte zudem möglichst einfach zu prüfen sein. Beispielsweise erlaubt eine Architektur, die alle Abhängigkeiten vom Betriebssystem in einer Komponente konzentriert, eine leichte Überprüfung der Portabilitätsanforderungen.

Es gibt keine Entwurfsmethode, die uns diese Fragen allgemeingültig beantwortet und eine gute Software-Architektur garantiert, aber wir kennen Entwurfsprinzipien, die sich bewährt haben und die ein Software-Architekt beachten und anwenden sollte. Sie helfen ihm, die folgenden Fragen zu klären:

- Nach welchen Kriterien soll das System in Komponenten unterteilt werden?
- Welche Aspekte sollen in Komponenten zusammengefasst werden?
- Welche Dienstleistungen sollen Komponenten nach außen an ihrer Schnittstelle anbieten, welche Aspekte müssen geschützt sein?
- Wie sollen die Komponenten miteinander interagieren?
- Wie sollen Komponenten strukturiert und verfeinert werden?

Nachfolgend werden wichtige Entwurfsprinzipien erläutert.

### 16.3.1 Modularisierung

Wie in Abschnitt 16.2.4 definiert wurde, gliedert die Architektur eine Anwendung in sinnvolle und überschaubare Bestandteile, die Komponenten. Eine Komponente ist in der Regel intern strukturiert, sie besteht entweder aus weiteren Komponenten oder aus Modulen. Das Aufteilen des Systems in seine Komponenten wird als Modularisierung bezeichnet.

**modular decomposition** — The process of breaking a system into components to facilitate design and development; an element of modular programming.

IEEE Std 610.12 (1990)

Eng verbunden damit ist der Begriff der Modularität.

**modularity** — The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

IEEE Std 610.12 (1990)

Modularität ist demnach eine Eigenschaft der Architektur. Sie ist hoch, wenn es dem Architekten gelungen ist, das System so in Komponenten zu zerteilen, dass diese möglichst unabhängig voneinander verändert und weiterentwickelt werden können.

Parnas hat wegweisende Arbeiten zur Modularisierung geleistet (Parnas, 1972). Folgende Ziele werden danach mit dem modularen Entwurf angestrebt:

- Die Struktur jedes Moduls soll einfach und leicht verständlich sein.
- Die Implementierung eines Moduls soll austauschbar sein; Information über die Implementierung der anderen Module ist dazu nicht erforderlich. Die anderen Module sind vom Austausch nicht betroffen.
- Die Module sollen so entworfen sein, dass wahrscheinliche Änderungen ohne Modifikation der Modulschnittstellen durchgeführt werden können.
- Große Änderungen sollen sich durch eine Reihe kleiner Änderungen realisieren lassen. Solange die Schnittstellen der Module nicht verändert sind, soll es möglich sein, alte und neue Modulversionen miteinander zu testen.

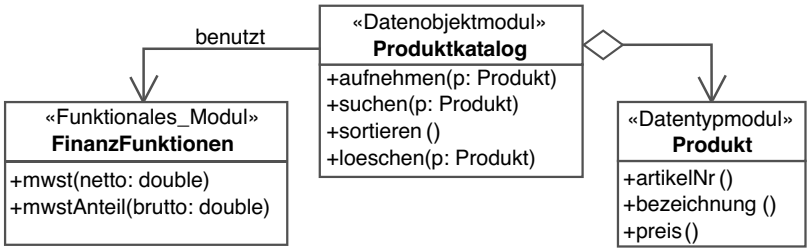



***Abb. 16–3*** *Modularten am Beispiel*

Nach den Entwurfsentscheidungen, durch die sie entstanden sind, kann man verschiedene Modularten unterscheiden (Abb. 16–3). In Nagl (1990) wird die folgende Klassifikation vorgeschlagen:

- *Funktionale Module* gruppieren Berechnungen, die logisch zusammengehören. Sie haben kein »Gedächtnis«, also keinen variablen Zustand, d. h., die Wirkungsweise der an der Schnittstelle angebotenen Berechnungsfunktionen ist nicht abhängig vom vorherigen Programmablauf. Beispiele für solche Module sind Sammlungen mathematischer Funktionen oder Transformationsfunktionen.
- *Datenobjektmodule* realisieren das Konzept der Datenkapselung. Ein Datenobjektmodul versteckt dazu Art und Aufbau der Daten und stellt an seiner Schnittstelle Operationen zur Verfügung, um die gekapselten Daten zu manipulieren. Beispiele sind Module, die globale Konfigurationsdaten kapseln, und Module, die gemeinsam benutzte Datenverwaltungen realisieren (z. B. ein Produktkatalog). Auch jede Datenbank ist ein Datenobjektmodul.
- *Datentypmodule* implementieren, wie der Name sagt, einen benutzerdefinierten Datentyp in Form eines Abstrakten Datentyps. Dadurch ist es möglich, beliebig viele Exemplare des Datentyps zu erzeugen und zu benutzen. Beispiele sind Module für die Datentypen Kunde oder Produkt.

Bei einem objektorientierten Entwurf entspricht die Klasse dem Datentypmodul. Ein Datenobjektmodul entspricht einer Klasse, die lediglich Klassenmethoden anbietet und von der keine Objekte erzeugt werden können. Mit dem Entwurfsmuster Singleton (siehe Abschnitt 16.5) können ebenfalls Datenobjektmodule realisiert werden. Funktionale Module sind beim objektorientierten Entwurf eher die Ausnahme (z. B. die Klasse `Math` der JAVA-Klassenbibliothek), da Klassen so angelegt sind, dass sie Daten und dazugehörende Operationen kapseln.

### 16.3.2 Kopplung und Zusammenhalt

Der Architekt eines Konzertsaals bemüht sich, den Saal so zu bauen, dass die akustische Kopplung nach außen sehr gering ist, damit kein Lärm hinein- oder – etwa bei Rockkonzerten – hinausdringt. Innerhalb des Saals soll dagegen der akustische Zusammenhalt hoch sein, damit auch auf den billigen Plätzen noch das leiseste Pianissimo hörbar ist.

Analog versucht der Software-Architekt, die Module so zu entwerfen, dass die (inter-modulare) Kopplung (d. h. die Breite und Komplexität der Schnittstellen zwischen den Modulen) möglichst gering, der (intra-modulare) Zusammenhalt (d. h. die Verwandtschaft zwischen den Bestandteilen eines Moduls) möglichst hoch wird.

Man kann die Module auch mit Knödeln vergleichen, die beim Kochen dazu neigen, zusammenzukleben oder zu zerfallen. Das ideale Modul klebt nicht (geringe Kopplung) und zerfällt nicht (hoher Zusammenhalt).

| Stufe | Kopplung zwischen Prozeduren / Modulen | erreichbar für |
|---|---|---|
| Einbruch | Der Code kann verändert werden | ??? |
| volle Öffnung | Auf alle Daten, z. B. auf globale Daten in einem C-Programm, kann zugegriffen werden | (Module) Prozeduren |
| selektive Öffnung | Bestimmte Variablen sind zugänglich, global oder durch expliziten Export und Import | Module, Prozeduren |
| Prozedurkopplung | Die Prozeduren verschiedener Module sind nur durch Parameter oder Funktionen gekoppelt | Module (Prozeduren) |
| Funktionskopplung | Die Prozeduren verschiedener Module sind nur durch Wertparameter und Funktionsresultate gekoppelt | Module (Prozeduren) |
| keine Kopplung | Es besteht keine Beziehung zwischen den Modulen, der Zugriff ist syntaktisch unmöglich | Module |

***Tab. 16–1*** *Stufen der Kopplung (von stark = schlecht nach schwach = gut)*

Dieses Prinzip wurde mit Structured Design (Stevens, Myers, Constantine, 1974) publiziert, es geht daher von einer FORTRAN- oder COBOL-Implementierung mit zweistufiger Hierarchie aus (System und Subroutines). Die Tabellen 16–1 (Kopplung) und 16–2 (Zusammenhalt) basieren auf Macro und Buxton (1987, S. 170).

Die Tabellen beziehen sich teilweise auf Probleme, die in modernen Programmiersprachen gar nicht auftreten können. Beispielsweise ist der Einbruch, die Code-Veränderung, nur auf primitivster Assembler-Ebene möglich.

| Stufe | Zus.halt innerhalb einer Prozedur / eines Moduls | erreichbar für |
|---|---|---|
| kein Zusammenhalt | Die Zusammenstellung ist zufällig | Module |
| Ähnlichkeit | Die Teile dienen z. B. einem ähnlichen Zweck (alle Operationen auf Matrizen oder zur Fehlerbehandlung) | Module |
| zeitliche Nähe | Die Teile werden zur selben Zeit ausgeführt (Initialisierung, Abschluss) | Module, Prozeduren |
| gemeinsame Daten | Die Teile sind durch gemeinsame Daten verbunden, auf die sie nicht exklusiven Zugriff haben | Module, Prozeduren |
| Hersteller/ Verbraucher | Der eine Teil erzeugt, was der andere verwendet | Module, Prozeduren |
| einziges Datum (Kapselung) | Die Teile realisieren alle Operationen, die auf einer gekapselten Datenstruktur möglich sind | Module, Prozeduren |
| einzige Operation | Operation, die nicht mehr zerlegbar ist | Prozeduren |

***Tab. 16–2*** *Stufen des Zusammenhalts (von schwach = schlecht nach stark = gut)*

Eine hohe Kopplung bewirkt, dass Korrekturen und Änderungen über mehrere Einheiten »verschmiert« sind, die Wartung ist entsprechend aufwendig und unsicher. Eine Einheit mit geringem Zusammenhalt könnte ohne Nachteil weiter aufgespalten werden und wäre dann leichter und sicherer zu verstehen, zu korrigieren und zu warten. Geringe Kopplung und hoher Zusammenhalt bewirken also gleichermaßen hohe Lokalität und damit gute Wartbarkeit.

Wenn wir modulare Programme betrachten, müssen wir die Aussagen über Programmkomponenten auf die Prozeduren oder auf die Module übertragen. Es liegt nahe, von den Modulen geringe Kopplung, von den Prozeduren hohen Zusammenhalt zu fordern. Die linke Seite der Abbildung 16–4 zeigt schematisch ein solches Programmsystem.

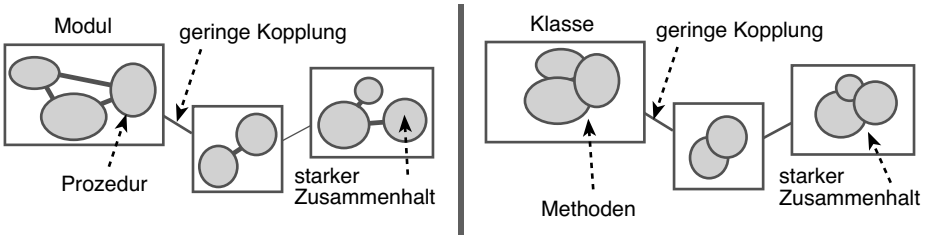



***Abb. 16–4*** *Kopplung und Zusammenhalt, links im modularen, rechts im objektorientierten Programm*

Dass die Kopplung zwischen Prozeduren im selben Modul nicht gering sein kann, ist klar, denn alle Prozeduren greifen auf die lokalen Variablen des Moduls zu. Natürlich ist auch der Zusammenhalt zwischen den Prozeduren eines Moduls geringer als der Zusammenhalt innerhalb einer Prozedur. Wenn wir diese Aussagen auf objektorientierte Programme übertragen wollen, können wir nicht einfach »Modul« durch »Klasse« und »Prozedur« durch »Methode« ersetzen (rechte Seite der Abb. 16–4). Denn während Prozeduren noch relativ unabhängig von anderen Prozeduren betrachtet (implementiert, analysiert, geprüft) werden können, sind die Methoden einer Klasse sehr eng miteinander verknüpft. Darum betrachten wir in objektorientierten Programmen sowohl bei der Kopplung als auch beim Zusammenhalt die Klassen. Sie sollten untereinander nur lose gekoppelt sein und jeweils aus Methoden bestehen, die starken Zusammenhalt haben.

Kopplung und Zusammenhalt kann man messen, wenn die Definitionen syntaktisch konkretisiert, also in Metriken umgesetzt werden; das ist für objektorientierte Programme versucht worden (siehe Abschnitt 14.4.6).

### 16.3.3 Information Hiding

Vor allem im militärischen Bereich gilt seit Langem das Prinzip, nur so viel Wissen weiterzugeben, wie der Adressat benötigt, um seine Funktion auszuüben (»Need-to-know-Prinzip«). Auf diese Weise wird die Gefahr vermindert, dass Informationen denen zugetragen werden, die sie (aus Sicht des Urhebers) missbrauchen.

Parnas (1972) hat dieses Prinzip im Software Engineering eingeführt. Auch hier geht es darum, den Missbrauch von Information zu verhindern. Dabei wird dem Empfänger keineswegs feindliches Verhalten unterstellt; schon der – weitverbreitete und kaum auszuschließende – leichtfertige Umgang mit Informationen schafft ein großes Problem.

Ein Programmierer, dessen Programm sehr oft bestimmte Daten benötigt, wird versuchen, den Zugriff sehr effizient zu implementieren. Er könnte beispielsweise ausnutzen, dass die Informationen auf bestimmten Positionen in einem Feld gespeichert sind. Sein Programm greift also direkt auf die Daten zu. Das hat (geringe) Vorteile, bis eines Tages die Speicherung der Daten verändert wird (weil der Umfang der Daten stark gewachsen ist, weil weitere Details gespeichert werden sollen o. Ä.). In diesem Moment hat die Lösung große Nachteile, denn plötzlich versagt ein Programm, das zuvor scheinbar in Ordnung war. Ein Programmierer darf also niemals Informationen über die Details der Datenorganisation ausnutzen. Andernfalls kann das Programm nicht mehr verändert werden. Aus diesem Grund ist es praktisch kaum möglich, in einem großen, traditionell strukturierten Programm die Datenstrukturen einer veränderten Situation anzupassen; die Zahl der notwendigen Änderungen ist unüberschaubar, und die betroffenen Stellen des Programms sind kaum zu lokalisieren.

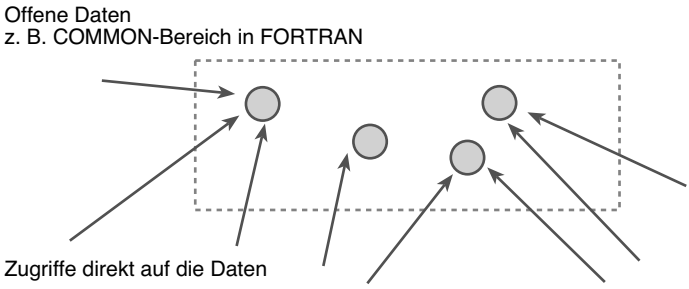



***Abb. 16–5*** *Situation ohne Kapselung von Daten*

Die Idee des »Information Hiding« besteht darin, dem Programmierer Informationen, die er nicht verwenden darf, gar nicht erst zugänglich zu machen.

**information hiding** — A software development technique in which each module's interfaces reveal as little as possible about the module's inner workings, and other modules are prevented from using information about the module that is not in the module's interface specification.

IEEE Std 610.12 (1990)

Ein Modul stellt durch Operationen an seiner Schnittstelle nur genau das zur Verfügung, was seine Kundenmodule (die Module, die seine Leistungen in Anspruch nehmen) wirklich benötigen. Ein Kundenmodul greift also nicht direkt auf eine Variable zu, sondern ruft eine Operation auf, die den gewünschten Effekt hat, z. B. einen Wert liefert (Abb. 16–6).

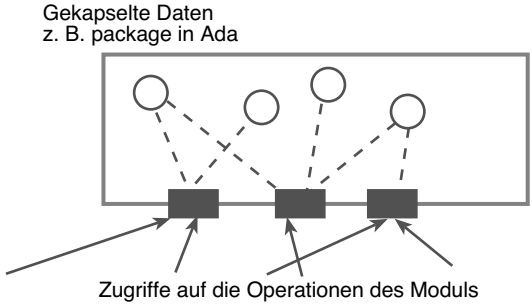



***Abb. 16–6*** *Modul mit gekapselten Daten*

Wird die Darstellung der Daten verändert, so müssen nur diese Operationen angepasst werden; die Kundenmodule sind – außer vielleicht durch eine veränderte Antwortzeit – nicht betroffen, solange die Schnittstelle des Moduls unverändert bleibt. Verheimlicht wird also die *Darstellung* der Information, nicht die (abstrakte) Information selbst, die ein Kundenmodul braucht. Die Zugriffe von außen erfolgen ausschließlich indirekt über die Operationen, die als Vermittler dienen. Die Bezeichnung »Information Hiding« ist darum unglücklich, denn sie

suggeriert, dass *wichtige* Information zurückgehalten wird. Tatsächlich wird nur eigentlich belanglose Information geschützt. Wenn wir eine Telefonnummer wählen, wissen wir über die Leitungen für unser Gespräch nichts. Darum kann es uns auch gleichgültig sein, ob wir beim nächsten Gespräch über andere Leitungen verbunden werden.

Der Schutz, der durch das Information Hiding erzielt wird, ist natürlich nur partiell, verhindert werden vor allem unbeabsichtigte Fehler. Sabotage ist weiterhin möglich, denn auch die vorgesehenen Operationen können sinnvoll oder falsch verwendet werden. Aber auch bei Sabotage erleichtert das Information Hiding die Diagnose; da alle Zugriffe über Operationen geschehen, können dort Protokoll- und Kontrollmechanismen eingebaut werden. Wenn Fehler in den Zugriffsoperationen liegen, ist die Fehlersuche wesentlich erleichtert, weil der zu untersuchende Code klar begrenzt ist.

Die folgende Analogie soll die Wirkungsweise und den Nutzen des Information Hiding verdeutlichen: Ein Bankkonto liegt voll in der Kontrolle der Bank, nur diese kann tatsächlich Veränderungen daran vornehmen. Der Kunde hat keinen direkten Zugang zum Kassenraum, er beauftragt die Bankangestellten, die Transaktionen durchzuführen, die er in Anspruch nehmen will. Auf diese Weise ist sichergestellt, dass der Kunde keine unzulässigen Transaktionen ausführt. Ändert sich die interne Organisation der Bank, so müssen nur die Angestellten davon unterrichtet werden, der Kunde ist nicht betroffen.

Auch bei Banken kann es vorkommen, dass ein Angestellter aus der Kasse Geld stiehlt. Das Verfahren schafft also keine totale Sicherheit. Wenn sich aber zeigt, dass Geld fehlt, muss man nur die Angestellten überprüfen, nicht die Kunden, denn diese hatten keine Möglichkeit, an das Geld zu kommen. Damit ist die Suche nach dem Täter relativ einfach. Das Prinzip der Geldverwaltung in der Bank verhindert allerdings nicht, dass der Kunde sein Geld unsinnig verwendet oder sich mit Krediten übernimmt. Wenn aber nach den Personen hinter verdächtigen Transaktionen gefahndet wird, ist die auf der wohlorganisierten Abwicklung basierende Buchführung der Banken äußerst nützlich.

#### Vor- und Nachteile des Information Hiding

Man kann sich über die Vor- und Nachteile von Sicherheitsgurten im Auto unterhalten; jeder vernünftige Mensch wird zustimmen, dass die Vorteile bei Weitem überwiegen. Ähnlich ist es beim Information Hiding. Die großen Vorteile sind:

- Weil die Datenstrukturen und die dazu gehörenden Zugriffsoperationen zusammen in einem einzigen Modul abgelegt werden, ist die Lokalität des Programms wesentlich verbessert. Damit sinkt die Gefahr unbeabsichtigter Fehlmanipulationen deutlich. Die Datenstrukturen können erheblich einfacher geändert werden als bei unkontrolliertem Zugriff.

- Die Zugriffe können überwacht werden, ohne dass man in die tieferen Abstraktionsebenen eindringen, also einen Debugger verwenden muss. Ein Debugger ist ein mächtiges, aber gefährliches Werkzeug, weil es den Benutzer veranlasst, das Programm auf sehr tiefem Abstraktionsniveau zu verstehen und zu bearbeiten. Wir setzen ihn als letztes Mittel ein, als Ultima Ratio, wenn wir keine andere Wahl haben, beispielsweise, wenn wir einen Compiler-Fehler vermuten.

Die (in der Regel akzeptablen) Nachteile des Information Hiding sind:

- Der Aufruf der vermittelnden Operationen kostet unvermeidlich Rechenzeit, das Programm ist dadurch weniger effizient. Parnas selbst hat 1972 darauf hingewiesen; die technische Entwicklung hat diesen Einwand entkräftet. Grundsätzlich wäre es auch möglich, die Effizienzeinbußen durch optimierende Compiler ganz zu vermeiden, denn wenn eine Funktion nur einen ohnehin verfügbaren Wert liefert, kann sie auch durch einen direkten Zugriff ersetzt werden. Das wäre für den Programmierer unsichtbar und damit unschädlich.
- Bei konsequentem Information Hiding entstehen sehr viele Module (leicht einige Hundert), und der Überblick, der gerade durch das Information Hiding geschaffen oder verbessert werden sollte, leidet. Parnas und seine Mitarbeiter haben nach einigen Erfahrungen mit dem Information Hiding vorgeschlagen, die Module in einem Modul-Handbuch (*module guide*) zu charakterisieren (Parnas, Clements, Weiss, 1985).

Werden Module nach dem Prinzip des Information Hiding entworfen, dann ist es natürlich geboten, dieses Prinzip auch bei der Codierung der Module beizubehalten. In Abschnitt 17.4 zeigen wir, wie Information Hiding im Programmcode realisiert wird.

### 16.3.4 Die Trennung von Zuständigkeiten

Die Trennung von Zuständigkeiten (*separation of concerns*) ist ein grundlegendes Prinzip im Software Engineering. Für den Software-Entwurf ist es von zentraler Bedeutung, denn jede Komponente sollte nur für einen ganz bestimmten Aufgabenbereich zuständig sein. Komponenten, die gleichzeitig mehrere Aufgaben abdecken, sind oft unnötig komplex. Das erschwert das Verständnis und damit die Wartung und Weiterentwicklung und verhindert, dass diese Komponenten wiederverwendet werden können.

Es gibt unterschiedliche Kriterien, um Zuständigkeiten zu trennen. Beim objektorientierten Entwurf sind die Daten und die Operationen darauf das Hauptkriterium: Zusammengehörende Daten und Operationen werden in einer Klasse zusammengefasst und von anderen Klassen getrennt. Oft werden auch spe-

zielle funktionale Aspekte (*Features*), beispielsweise das Drucken, das Speichern oder die Visualisierung, als eigenständige Komponenten modelliert (Abb. 16–7).

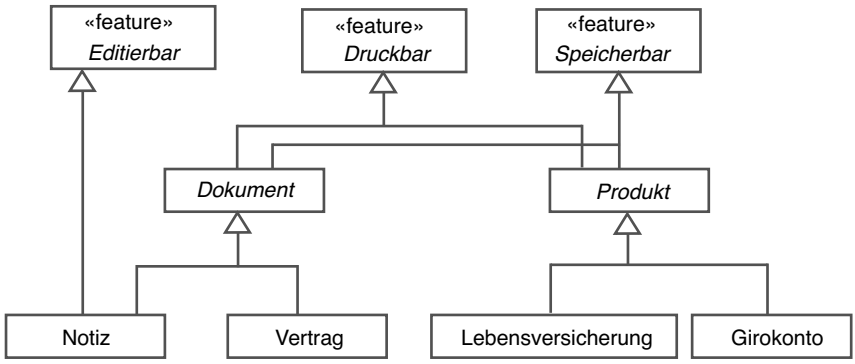



***Abb. 16–7*** *Klassen modellieren getrennte Zuständigkeiten*

Weitere wichtige Regeln zur Trennung von Zuständigkeiten sind:

- Trenne fachliche und technische Komponenten. Diese Regel ist die konsequente Weiterführung der Überlegungen zur idealen Technologie (Abschnitt 15.7.2).
- Ordne Funktionalitäten, die variabel sind oder später erweitert werden müssen, eigenen Komponenten zu.

Auch die Trennung von Funktion und Interaktion ist eine Ausprägung dieses Prinzips. Alle interaktiven Programme enthalten neben den Komponenten, die die Funktionalität realisieren, auch Komponenten für die Interaktion. Die Trennung der Zuständigkeiten bedeutet hier, dass Interaktion und Funktionalität strikt gegeneinander abgeschottet werden. Dadurch wird es möglich, die Funktion oder die Interaktion zu verändern, ohne den anderen Teil des Programms zu gefährden.

Besonders wenn Anwendungen von der Bedienoberfläche her entworfen und implementiert werden (also outside-in, Abschnitt 16.1.4), wird die Funktionalität oft unterschätzt und darum nicht sauber separiert. Das führt unweigerlich zu Problemen (Bäumer et al., 1996); Programme, in denen Funktionalität und Interaktion durchmischt sind, machen die Wartung unnötig schwierig. Die Trennung von Interaktion und Funktion ist darum ein zentrales Entwurfsprinzip für die Konstruktion interaktiver Anwendungen. Im Model-View-Controller-Architekturmuster ist sie exemplarisch umgesetzt (Abschnitt 16.4.4).

### 16.3.5 Die hierarchische Gliederung

Die Komponenten einer Architektur müssen so lange verfeinert werden, bis sie so einfach sind, dass wir sie spezifizieren und realisieren können. Die hierarchische Gliederung ist ein bewährtes Vorgehen, um Komplexität zu reduzieren.

Eine *Hierarchie* ist eine Struktur von Elementen, die durch eine (hierarchiebildende) Beziehung in eine Rangfolge gebracht werden. Entsteht dadurch eine Baumstruktur, dann spricht man von einer *Monohierarchie*, d. h., jedes Element der Hierarchie außer dem Wurzelelement besitzt genau ein übergeordnetes Element. Kann ein Element mehrere übergeordnete Elemente haben, dann handelt es sich um eine *Polyhierarchie*; die entsprechende Struktur ist ein azyklischer gerichteter Graph (Wikipedia, o. J.). Im Kontext des Software-Entwurfs verwenden wir drei Arten von Hierarchien:

- Eine *Aggregationshierarchie* gliedert ein System oder eine Komponente in ihre Bestandteile; sie wird auch »Ganzes-Teile-Hierarchie« genannt. Die Beziehung zwischen dem Ganzen und seinen Teilen ist von der Art *besteht-aus* (in der Gegenrichtung *ist-Teil-von*). Wir entwerfen Aggregationshierarchien immer auf der höchsten Ebene, der Ebene der Systemarchitektur. Sie teilen das System in Komponenten auf, die hierarchisch weiter verfeinert werden. Aggregationshierarchien sind in der Regel Monohierarchien.
- Eine *strenge Schichtenhierarchie* ordnet Komponenten (die dann als Schichten bezeichnet werden) derart, dass jede Schicht genau auf einer darunterliegenden Schicht aufbaut und die Basis für genau eine darüberliegende Schicht bildet. Eine strenge Schichtenhierarchie ist damit eine strengere Form einer Monohierarchie. Bei einer *nicht strengen Schichtenhierarchie* kann eine Schicht auf vielen darunterliegenden Schichten aufbauen, sie bilden eine Polyhierarchie. Schichtenarchitekturen werden in Abschnitt 16.4.1 beschrieben.
- Eine *Generalisierungshierarchie* ordnet Komponenten nach Merkmalen (Funktionen und Attributen), indem fundamentale, gemeinsame Merkmale mehrerer Komponenten in einer universellen Komponente zusammengefasst werden. Davon abgeleitete, spezialisierte Komponenten übernehmen diese Merkmale und fügen spezielle hinzu. Damit werden die fundamentalen Merkmale nur einmal definiert. Generalisierungshierarchien können als Mono- oder als Polyhierarchien auftreten. Der objektorientierte Entwurf stellt Generalisierungshierarchien von Klassen und Schnittstellen in den Vordergrund, da diese mithilfe der Vererbung direkt in einer objektorientierten Programmiersprache implementiert werden können.

Abbildung 16–8 zeigt links eine Architektur, die alle oben beschriebenen Hierarchiearten enthält. Die Aggregationshierarchie ist implizit durch die Topologie der Elemente angegeben (System A besteht aus den Schichten S1 und S2, die Schicht S1 aus den Komponenten K1, K2 und K3). Während die Schichtenhierarchie ebenfalls durch die Topologie angegeben wird, sind die Generalisierungsbezie-

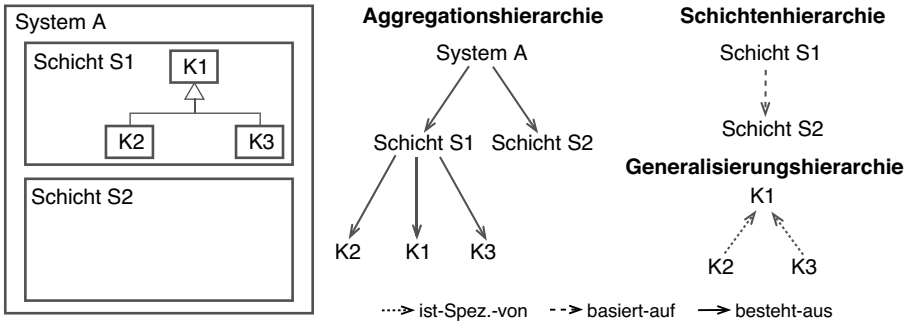



***Abb. 16–8*** *Beispiele für Hierarchien*

hungen zwischen den Komponenten K1 und K2 sowie K1 und K3 explizit dargestellt. Auf der rechten Seite sind die Elemente der Architektur, alle Beziehungen und die dadurch entstandenen Hierarchien abgebildet. Wie man sieht, kann ein Element in mehreren Hierarchien enthalten sein. So ist beispielsweise die Komponente K2 Element der Aggregations- und der Generalisierungshierarchie.

Wir brauchen alle Arten von Hierarchien. Auch wenn wir objektorientiert entwerfen, können wir nicht auf Aggregations- und Schichtenhierarchien verzichten.

### 16.3.6 Das Offen-geschlossen-Prinzip

Die in Abschnitt 16.3 vorgestellten Entwurfsprinzipien sind universell nutzbar, also auch bei einem objektorientierten Entwurf. Ein weiteres wichtiges Entwurfsprinzip ist das Offen-geschlossen-Prinzip; es kann faktisch aber nur bei einer objektorientierten Vorgehensweise angewendet werden.

Wer eine Komponente verwenden möchte, obwohl sie nicht exakt den Anforderungen entspricht, muss sie anpassen. Eine Komponente, die diese Veränderung zulässt, heißt (nach Meyer, 1997) *offen*.

Wer eine Komponente bereits verwendet, ist umgekehrt vor allem daran interessiert, dass sich die Schnittstelle und die Funktion der angebotenen Operationen nicht mehr verändern; denn jede Änderung muss in den Kundenkomponenten nachvollzogen werden. Das ist nicht nur kostspielig, sondern auch extrem fehlerträchtig. Meyer nennt eine solche Komponente *geschlossen*. Wir streben darum an, Komponenten so zu entwerfen, dass sie sowohl geschlossen, also stabil benutzbar, als auch offen für Erweiterungen sind.

Offensichtlich sind diese beiden Eigenschaften nicht miteinander vereinbar, wenn imperative Programmiersprachen zur Codierung verwendet werden. Benutzen wir jedoch eine objektorientierte Programmiersprache, dann ist es dank der Vererbung möglich, beide Ziele zu erreichen.

Betrachten wir dazu das folgende Beispiel (siehe Abb. 16–9): Die Klasse A wurde so entworfen, dass sie die Bedürfnisse der bekannten Kundenklassen B, C

und D abdeckt. Im Zuge der Weiterentwicklung des Programms werden zwei neue Klassen E und F entworfen. Sie könnten ebenfalls die Klasse A benutzen, wenn diese einige zusätzliche neue Funktionen anböte. Damit die existierenden Kundenklassen der Klasse A nicht von dieser Modifikation betroffen sind, leiten wir von A eine neue Klasse A' ab. A' hat alle Merkmale von A und zusätzlich die von E und F benötigten Funktionen.

Um Entwürfe zu erstellen, die dem Offen-geschlossen-Prinzip genügen, müssen die richtigen Abstraktionen gefunden und die veränderbaren und erweiterbaren Aspekte explizit modelliert werden. Die verwendeten Wörter (»offen«, »geschlossen«) sind unglücklich gewählt, denn sie deuten einen Gegensatz an, der nicht besteht. Sinnvoller wäre ein Begriffspaar wie »erweiterbar« und »vollständig«.

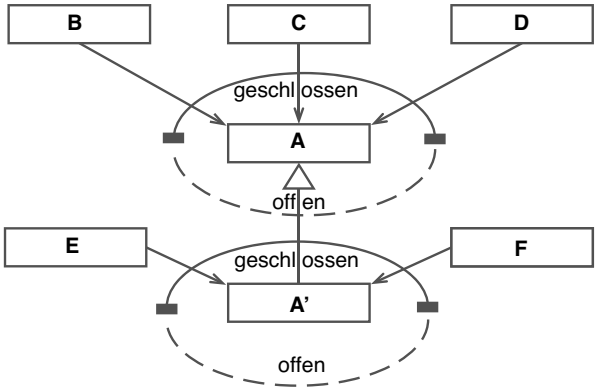



***Abb. 16–9*** *Offen-geschlossen-Konstruktion*

### 16.3.7 Das Dependency-Inversion-Prinzip

Wie in Abschnitt 16.3.2 beschrieben, wollen wir Komponenten so entwerfen, dass sie einen hohen Zusammenhalt haben und lose gekoppelt sind. Eine Komponente A ist an eine Komponente B gekoppelt, wenn sie deren Dienste verwendet. Abbildung 16–10 zeigt dies an einem einfachen Beispiel.

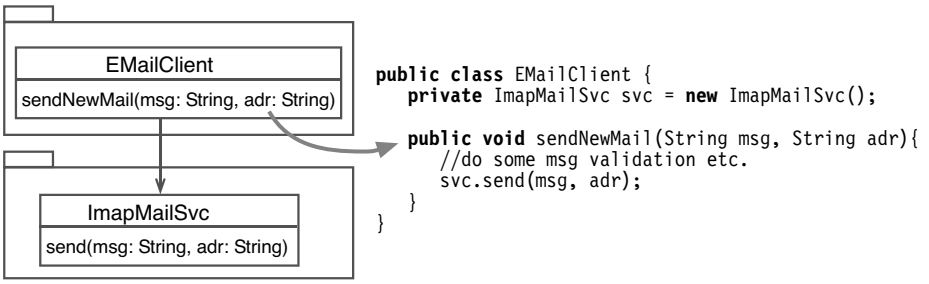



***Abb. 16–10*** *Starke Kopplung durch direkten Zugriff auf angebotene Dienste*

Die Klasse `EMailClient` verwendet ein Objekt der Klasse `ImapMailSvc`, um eine E-Mail zu versenden. Beide Klassen werden dadurch stark aneinandergekoppelt; die Klasse `EMailClient` kann nicht ohne die Klasse `ImapMailSvc` wiederverwendet werden.

Man reduziert die Kopplung zwischen zwei Komponenten, wenn man eine Schnittstelle dazwischen legt. Diese Schnittstelle wird von der Kundenkomponente vorgegeben und abstrahiert von den konkreten Diensten der Anbieterkomponente. Diese muss die geforderte Schnittstelle implementieren. Abbildung 16–11 zeigt diese Struktur.

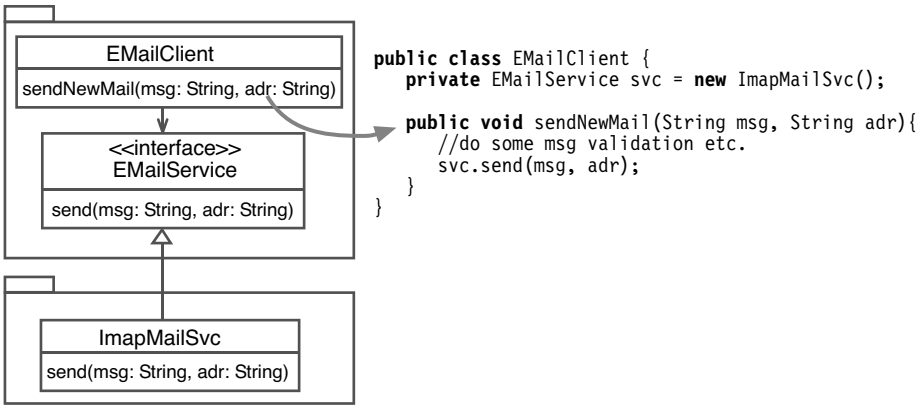



***Abb. 16–11*** *Entkopplung von Komponenten durch eine gemeinsame Schnittstelle*

Im Beispiel basiert die Kundenklasse `EMailClient` jetzt nur noch auf der eigenen Schnittstelle `EMailService` und nicht mehr direkt auf der Klasse `ImapMailSvc`. Diese implementiert die Schnittstelle; basiert also auch auf ihr. Durch diese Konstruktion wird die Kopplung nicht nur reduziert, sie wird sogar umgedreht, weil nun die Anbieterkomponente (`ImapMailSvc`) auf der Kundenkomponente (`EMailClient`) basiert.

Robert Martin hat diese Art der Entkopplung als »Dependency-Inversion-Principle« (Martin, 2003) bezeichnet. Dieses Prinzip setzt die folgenden Regeln um:

a) Eine Komponente einer höheren Ebene (Kundenkomponente) sollte nicht von einer Komponente einer niedrigeren Ebene (Anbieterkomponente) abhängen.
b) Beide Komponenten sollten über eine gemeinsame Schnittstelle (Abstraktion) miteinander interagieren.
c) Die Schnittstelle sollte nicht auf Basis der vorhandenen Dienste der Anbieterkomponente, sondern aus Sicht der Kundenkomponente gestaltet sein.
d) Die von der Anbieterkomponente zur Verfügung gestellten Dienste sollten sich an der geforderten Schnittstelle orientieren.

Wichtig ist, dass die gewählten Schnittstellen möglichst stabil sind. Sind Änderungen an der Schnittstelle notwendig, dann sind alle Anbieterkomponenten davon betroffen, die diese implementieren.

Das Dependeny-Inversion-Prinzip ist fundamental, um flexible und erweiterbare Architekturen zu entwerfen. Das in Abschnitt 16.4.2 vorgestellte Port-Adapter-Architekturmuster basiert auf der Anwendung dieses Prinzips.

### 16.3.8 Die Prinzipien im Zusammenhang

Die hier vorgestellten Entwurfsprinzipien stehen miteinander in Beziehung und ein Software-Architekt wendet typischerweise gleichzeitig mehrere dieser Prinzipien an. Die konzeptuelle Grundlage dieser – und vieler anderer – Entwurfsprinzipien ist die Abstraktion. Indem wir Abstraktionen bilden, konzentrieren wir uns auf das Wesentliche und blenden das Unwesentliche aus. Es ist eine wichtige Fähigkeit jedes guten Architekten, in Abstraktionen zu denken und Abstraktionen zu bilden.

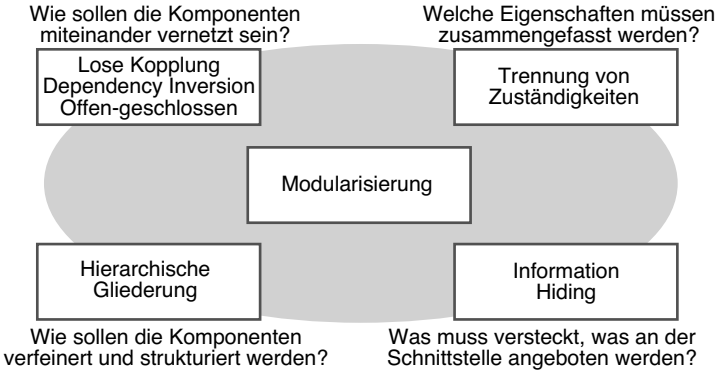



***Abb. 16–12*** *Entwurfsprinzipien und ihr Zusammenhang*

Abbildung 16–12 zeigt die beschriebenen Entwurfsprinzipien und ihre Zusammenhänge. Die Modularisierung steht im Zentrum, die anderen Prinzipien tragen dazu bei, den Prozess der Modularisierung effektiv zu unterstützen.

## 16.4 Architekturmuster

Wenn sich bestimmte Lösungsstrukturen bewährt haben, kann man sie dokumentieren und später erneut verwenden. Der Architektur-Theoretiker Christopher Alexander (1936–2022) hat solche Muster in einem Buch zusammengestellt (Alexander et al., 1977). Dieses Konzept wurde auf die Software übertragen, wir sprechen ebenfalls von *Architekturmustern*. Sie sind auf der Ebene der Systemar-

chitektur definiert und stellen Vorlagen dar, nach denen konkrete Architekturen entworfen werden können. Buschmann et al. definieren:

> An **architectural pattern** expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
>
> Buschmann et al. (1996)

Die Verwendung eines Architekturmusters impliziert bestimmte Merkmale der Software (Aufbau, Erweiterbarkeit, Kommunikation etc.). Wenn wir ein Architekturmuster verwenden, legen wir damit die Strukturen auf der obersten Ebene der Architektur fest. Die Wahl eines Architekturmusters ist darum eine zentrale Entwurfsentscheidung. Ob das vorgesehene Architekturmuster geeignet ist, hängt davon ab, ob seine speziellen Randbedingungen und Voraussetzungen erfüllt sind.

Shaw und Garlan (1996) sowie Buschmann et al. (1996) beschreiben detailliert einen Katalog von Architekturmustern. Wir beschränken uns hier darauf, einige häufig anwendbare Architekturmuster vorzustellen.

### 16.4.1 Das Schichten-Architekturmuster

In seiner Arbeit »The Structure of the THE Multiprogramming System« nutzt Edsger W. Dijkstra Software-Schichten, um ein Betriebssystem übersichtlich zu strukturieren (Dijkstra, 1968a). Er zerlegt die Funktionalität des Betriebssystems in fünf Schichten, wobei jede Schicht einen virtuellen Rechner (*virtual machine*) bildet, der der darüberliegenden Schicht definierte Dienste in gekapselter Form zur Verfügung stellt. Software-Schichten haben die folgenden Eigenschaften:

- Sie gruppieren logisch zusammengehörende Komponenten.
- Sie stellen Dienstleistungen zur Verfügung, die an der (oberen) Schnittstelle einer Schicht angeboten werden.
- Sie basieren nur auf den Dienstleistungen darunterliegender Schichten.

Schichten bauen immer aufeinander auf. Sie bilden eine Hierarchie, in der nur Abwärts- und keine Aufwärtsbeziehungen erlaubt sind.

Wenn die Dienstleistungen einer Schicht nur von Komponenten der direkt darüberlegenden Schicht benutzt werden dürfen, spricht man von einer *strengen Schichtenarchitektur.* Züllighoven (2005) bezeichnet Schichten, die nur mit ihren Nachbarschichten kommunizieren, als *protokollbasierte Schichten*. Eine solche Schicht verbirgt die unter ihr liegenden Schichten und definiert ein Protokoll, das nur der darüberliegenden Schicht zur Verfügung steht.

Dürfen die Komponenten einer Schicht auf alle Komponenten der darunterliegenden Schichten zugreifen, dann ist das eine schwächere Form der Schichtung, die als *nicht strenge Schichtenarchitektur* bezeichnet wird. Abbildung 16–13 zeigt schematisch eine strenge und eine nicht strenge Schichtenarchitektur.

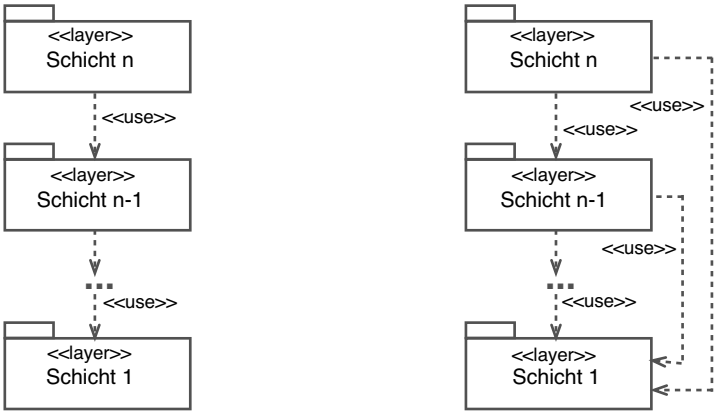



***Abb. 16–13*** *Strenge und nicht strenge Schichtenarchitektur (UML-Paketdiagramm)*

Ein schichtenbasierter Entwurf hat folgende Vorteile:

- Die Schichten sind lose gekoppelt. Aber auch diese Kopplung ist durch die Kapselung der Komponenten gering. Änderungen wirken sich darum meist nur lokal (innerhalb einer Schicht) aus.
- In Schichtenarchitekturen können keine zyklischen Abhängigkeiten zwischen den Komponenten verschiedener Schichten entstehen, da nur Abwärtsbeziehungen erlaubt sind. Das erleichtert das Verständnis und die Wartung.
- Einzelne Schichten sind leicht austauschbar, wenn die Schnittstellen nicht oder nur in sehr geringem Umfang angepasst werden müssen.

#### Das Drei-Schichten-Architekturmuster

Viele interaktive Software-Systeme sind aus den folgenden drei protokollbasierten Schichten aufgebaut: *Datenhaltungs-*, *Anwendungs-* und *Präsentationsschicht* (siehe Abb. 16–14). Dieser Aufbau ist ebenfalls ein Architekturmuster. Er spiegelt in seiner Struktur die grundsätzlichen Aufgaben wider, die bei interaktiven Software-Systemen gelöst werden müssen. Die Präsentationsschicht realisiert die Bedienoberfläche; sie stellt Informationen dar und steuert die Interaktion des Benutzers mit dem System. Dazu kommuniziert die Präsentationsschicht mit der Anwendungsschicht.

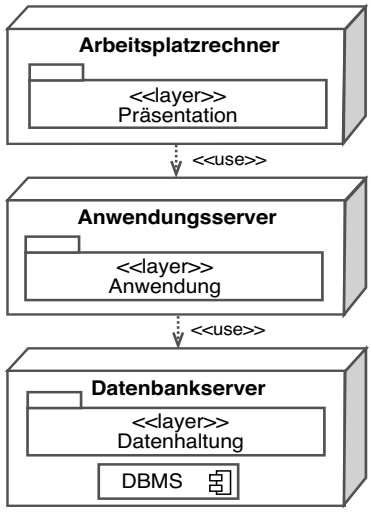



***Abb. 16–14*** *Drei-Schichten-Architektur mit physischer Verteilung (UML-Verteilungsdiagramm)*

In der Anwendungsschicht sind alle Komponenten zusammengefasst, die die fachliche Funktionalität realisieren. Sie sind so konstruiert, dass sie keine Information über die Präsentation enthalten. Um Daten zu manipulieren oder an die Präsentationsschicht weiterzuleiten, greifen die Komponenten der Anwendungsschicht auf die Datenhaltungsschicht zu. Diese sorgt dafür, dass die Daten dauerhaft (persistent) gespeichert werden, meist in einer Datenbank.

Wie die Daten geladen und gespeichert werden, ist ausschließlich der Datenhaltungsschicht bekannt; der Anwendungsschicht ist diese Information verborgen.

Ist eine Software in protokollbasierte Schichten gegliedert, so lässt sie sich relativ leicht auf verschiedene Rechner verteilen, wie das Beispiel in Abbildung 16–14 zeigt.

#### Das Architekturmuster der technischen und fachlichen Schichten

Eric Evans (2004) hat eine erweiterte Schichtung vorgeschlagen, die sich ebenfalls an den technischen Aufgaben orientiert, die bei der Implementierung von Anwendungssoftware gelöst werden müssen. Dieses Muster definiert jedoch vier technische Schichten (Abbildung 16–15):

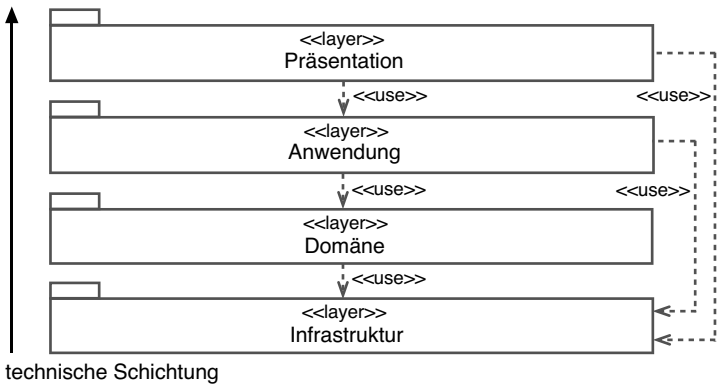



***Abb. 16–15*** *Muster der technischen Schichten*

- Die *Infrastrukturschicht* enthält Komponenten, die für die Implementierung der anderen Schichten benötigt werden. Beispiele sind Komponenten zur Datenhaltung, zur Anbindung von Druckern, Logging-Komponenten oder eine GUI-Bibliothek.
- Die *Domänenschicht* fasst die Komponenten zusammen, die die fachlichen Entitäten und Dienste des Anwendungsbereichs implementieren, der durch die Software unterstützt wird.
- Darauf aufbauend werden in der *Anwendungsschicht* die notwendigen Geschäftsprozesse oder Use Cases durch eigene Dienste realisiert.

- Mithilfe dieser Dienste werden die Funktionen der Anwendung dem Benutzer in der *Präsentationsschicht* zur Verfügung gestellt.

Dieses Muster verfeinert das Drei-Schichten-Architekturmuster, indem dessen Anwendungsschicht in zwei Schichten aufgeteilt wird, die Domänen- und die gleichnamige Anwendungsschicht. Dadurch werden die zentralen Komponenten eines Anwendungsbereichs von der Implementierung der darauf basierenden Geschäftsprozesse getrennt und nur lose daran gekoppelt. Weiterhin übernimmt die Infrastrukturschicht nicht nur die Zuständigkeit für die Datenhaltung, sondern stellt weitere notwendige Infrastrukturkomponenten zur Verfügung.

Dieses Architekturmuster bildet die Basis, um Anwendungssysteme nach »Domain-Driven Design« zu entwerfen. Wir stellen diesen Ansatz in Abschnitt 16.7.2 vor.

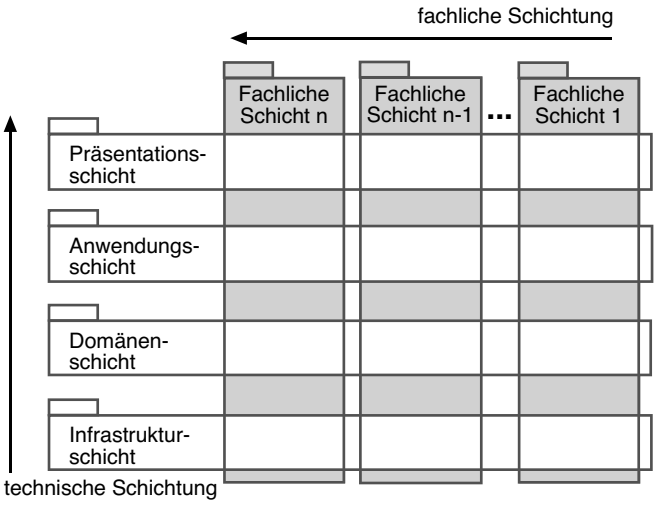



***Abb. 16–16*** *Muster der technischen und fachlichen Schichten (vereinfacht nach Lilienthal, 2020)*

Carola Lilienthal (2020) erweitert dieses Muster, indem orthogonal zu den technischen Schichten sogenannte fachliche Schichten eingeführt werden. Jede fachliche Schicht implementiert eine (fachliche) Funktionalität des Anwendungsbereichs. Da diese oft aufeinander aufbauen, müssen sie entsprechend angeordnet sein. So können beispielsweise die Dienste einer grundlegenden fachlichen Schicht »Kundenverwaltung« von einer übergeordneten Schicht »Kreditgeschäft« benutzt werden, nicht aber umgekehrt. Abbildung 16–16 zeigt schematisch die daraus resultierende Musterarchitektur. Wir erkennen, dass jede fachliche Schicht, also jede fachliche Anwendung, intern aus den vier technischen Schichten besteht.

### 16.4.2 Das Port-Adapter-Architekturmuster

Schichtenarchitekturen haben verschiedene Vorteile, da Schichten aufeinander aufbauen, lose gekoppelt sind und zyklische Abhängigkeiten zwischen den Komponenten der Schichten nicht auftreten können.

Bei einer Drei-Schichten-Architektur hängt die fachlich zentrale Schicht, die Anwendungsschicht, jedoch von den Spezifika der Datenhaltungsschicht ab. Das führt in der Regel dazu, dass die Anwendungsschicht Komponenten enthält, um die fachlichen Entitäten in ein datenbankspezifisches Format zu konvertieren. Beide Schichten werden dadurch sehr stark gekoppelt.

Man kann diese Kopplung reduzieren, indem man das »Dependency-Inversion-Prinzip« anwendet (siehe Abschnitt 16.3.7). Dazu wird in der Anwendungsschicht eine Schnittstelle für Datenzugriffsoperationen vorgegeben, auf die sich der Entwurf und die Implementierung der fachlichen Komponenten intern abstützt. Die Datenhaltungsschicht ist dafür verantwortlich, diese Schnittstelle entsprechend zu implementieren. Abbildung 16–17 zeigt die Abhängigkeiten zwischen den drei Schichten an einem Beispiel[3].

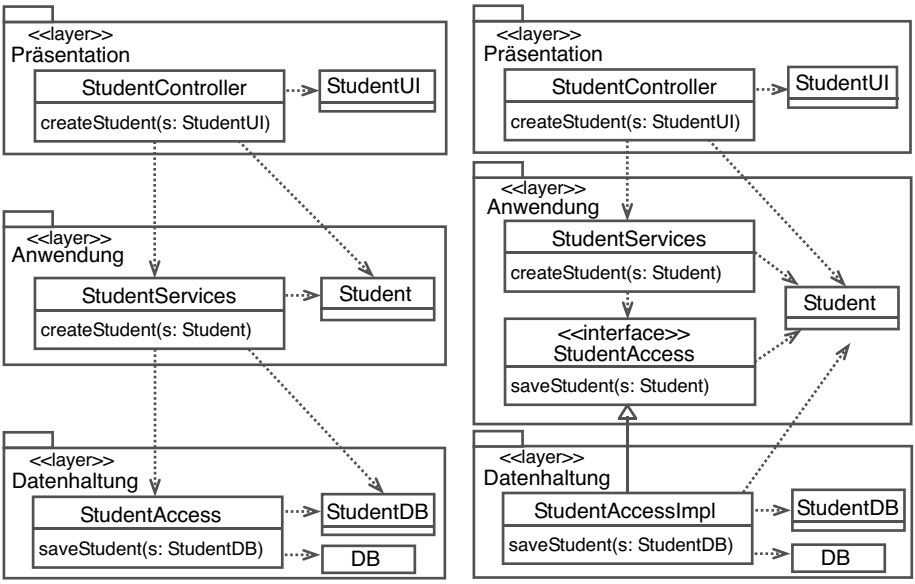



***Abb. 16–17*** *Umkehrung der Abhängigkeit zwischen Schichten durch Schnittstellen*

Das linke Diagramm zeigt eine klassische Drei-Schichten-Architektur. Die Klasse `StudentServices` implementiert Funktionen (z. B. `createStudent`), um die Daten mithilfe der Dienste, die die Klasse `StudentAccess` der Datenhaltungsschicht

3 Dieser Abschnitt und das gezeigte Beispiel basieren auf einem Blog-Eintrag von Michael Scharhag (Scharhag, 2021).

anbietet, zu speichern und aus der Datenbank zu laden. Im rechten Diagramm enthält die Anwendungsschicht eine Schnittstelle (`StudentAccess`), die alle notwendigen Datenzugriffsoperationen festlegt. Die Klasse `StudentAccessImpl` implementiert diese Schnittstelle und ist Teil der Datenhaltungsschicht.

Durch diese Konstruktion wird nicht nur erreicht, dass die Anwendungsschicht nicht mehr von den Diensten der Datenhaltungsschicht abhängt, sondern dass sich die Abhängigkeitsbeziehung zwischen diesen Schichten umdreht. Die Datenhaltungsschicht muss die Schnittstelle implementieren, die die Anwendungsschicht vorgibt; sie konvertiert die Entitäten in das datenbankspezifische Format.

Damit wird die Anwendungsschicht zum zentralen Baustein der Architektur, sowohl die Komponenten der Präsentations- als auch der Datenhaltungsschicht basieren auf ihr.

Wenn man diese Konstruktion auf allgemeine Komponenten überträgt, erhält man eine Architektur, bei der sich technisch benötigte Komponenten um die zentrale Anwendungskomponente gruppieren. Diese nutzen entweder deren Dienstleistungen, sind also Kunde der Anwendungskomponente, oder stellen benötigte Dienste durch Implementierungen der vorgegebenen Schnittstellen als Anbieter zur Verfügung. Dieses Muster wird als *Hexagonale Architektur* oder auch als *Port-Adapter-Architekturmuster* bezeichnet, das Alistair Cockburn (2005) vorgestellt hat.

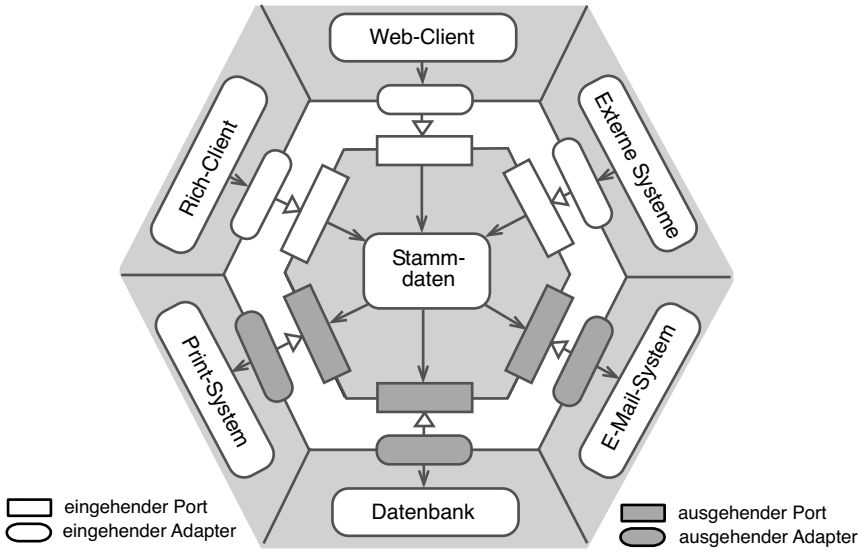



***Abb. 16–18*** *Typische Darstellung des Port-Adapter-Musters als Hexagon (nach Lilienthal, 2020)*

Abbildung 16–18 zeigt ein Beispiel für die Anwendung dieses Musters[4]. Die Anwendungskomponente, die Komponente »Stammdaten« liegt im Zentrum. Sie

ist unabhängig von den technischen Kundenkomponenten, beispielsweise von einem Web-Client, und von technischen Anbieterkomponenten, z. B. einer Datenbank oder einem E-Mail-System. Die Kommunikation mit den technischen Komponenten wird durch Ports und Adapter realisiert. Ports definieren die Schnittstellen der Anwendungskomponente, Adapter implementieren diese Ports. Es werden zwei Arten von Ports und Adapter unterschieden:

- Ein *eingehender Port* ist eine Schnittstelle, die die Anwendungskomponente Kundenkomponenten zur Verfügung stellt.
- Ein *eingehender Adapter* implementiert einen eingehenden Port als Teil der Anwendungskomponente.
- Ein *ausgehender Port* ist eine Schnittstelle, die die Anwendungskomponente Anbieterkomponenten vorgibt und auf der die eigene Implementierung basiert.
- Ein *ausgehender Adapter* ist Teil einer Anbieterkomponente und implementiert einen ausgehenden Port.

Durch diese Konstruktion kann eine Anwendungskomponente entkoppelt von den sie umgebenden technischen Komponenten entwickelt und auch getestet werden. Die Kopplung zwischen den Komponenten ist geringer als bei klassischen Schichtenarchitekturen.

Das Port-Adapter-Architekturmuster ist im Web gut dokumentiert. In der Literatur wird es auch als Onion-Architektur oder von Robert Martin als Clean-Architektur bezeichnet (Martin, 2018). Wie das Port-Adapter-Architekturmuster technisch angewendet werden kann, beschreibt beispielsweise Vierira (2021) für JAVA-Anwendungen.

### 16.4.3 Das Pipe-Filter-Architekturmuster

Die Verbindung der Verarbeitungsfunktionen durch sogenannte *Pipes* gehört zu den Charakteristika des Betriebssystems UNIX. Eine Pipe verbindet zwei Verarbeitungsschritte so miteinander, dass das Ergebnis des ersten Schritts im folgenden Schritt als Eingabe verwendet wird. Dieses Konstruktionsmuster kann man für Anwendungen wie folgt verallgemeinern:

Das **Pipe-Filter-Architekturmuster** dient dazu, eine Anwendung zu strukturieren, die Daten auf einem virtuellen Fließband verarbeitet.

Die einzelnen Verarbeitungsschritte werden in den sogenannten *Filtern* realisiert. Ein Filter verbraucht und erzeugt Daten. Pipes leiten die Ergebnisse des Filters an die nachfolgenden Filter weiter. Das erste Filter bekommt seine Daten aus der *Datenquelle*, das letzte liefert sie an die *Datensenke*. Soll eine Anwendung als

4 Das Port-Adapter-Muster wird häufig durch ein Hexagon visualisiert. Natürlich kann es aber weniger oder mehr als sechs Port-Adapter-Paare geben.

Pipe-Filter-Architektur konstruiert werden, dann muss man zuerst die einzelnen Verarbeitungsschritte identifizieren. Anschließend sind Datenformate für je zwei durch eine Pipe verbundene Filter festzulegen. Kann man ein einheitliches Datenformat für alle Filter festlegen, dann können diese auch leicht rekombiniert werden.

Wie das UNIX-Betriebssystem arbeiten auch viele Compiler nach dem Pipe-Filter-Architekturmuster. Die Verarbeitungsschritte, also die Filter, sind die lexikalische Analyse, die Syntaxanalyse, die semantische Analyse und die Code-Erzeugung (siehe Abb. 16–19).

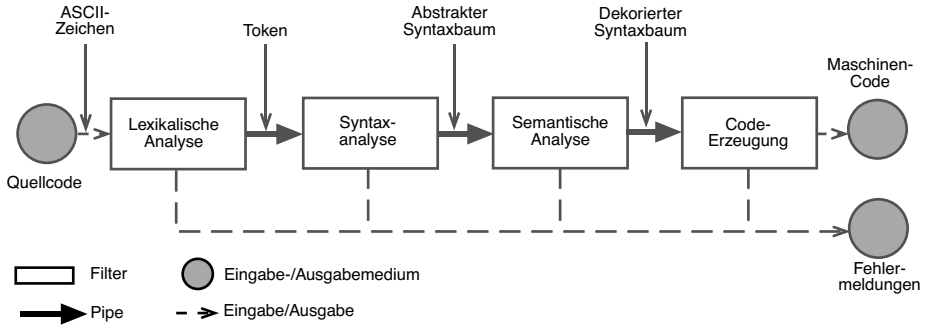



***Abb. 16–19*** *Pipe-Filter-Architektur eines Compilers*

Dieses Architekturmuster hat die folgenden Vorteile:

- Ein existierendes Filter kann einfach durch eine neue Komponente ersetzt werden, da es relativ einfache Schnittstellen hat (Ein- und Ausgang).
- Nutzen mehrere Filter das gleiche Datenaustauschformat (z.B. ASCII-Dateien), dann können sie durch Rekombination neue Verarbeitungseinheiten realisieren. Diese Eigenschaft kann beispielsweise genutzt werden, um Systemprototypen zu erstellen.

Dem stehen die folgenden Nachteile gegenüber:

- Wird aus Gründen der Flexibilität ein einheitliches Datenaustauschformat definiert, dann müssen die einzelnen Filter eventuell das Format anpassen, also Datenkonversionen durchführen. Dies kann die Effizienz merklich beeinträchtigen.
- Die Filter benutzen keine globalen Daten. Um trotzdem systematisch mit Fehlersituationen umzugehen, wird eine gemeinsame Strategie für alle Filter benötigt. Diese muss festlegen, wie die Daten im Fehlerfall weiterverarbeitet werden sollen. Im schlimmsten Fall muss die Verarbeitung abgebrochen und vollständig wiederholt werden.

### 16.4.4 Das Model-View-Controller-Architekturmuster

Die erste architektonisch saubere Trennung von Interaktion und Funktion wurde beim Bau der SMALLTALK-Entwicklungsumgebung umgesetzt und als *Model-View-Controller*-Paradigma (MVC) bezeichnet (Krasner, Pope, 1988). Nach der in Abschnitt 16.2 eingeführten Terminologie ist MVC ein Architekturmuster.

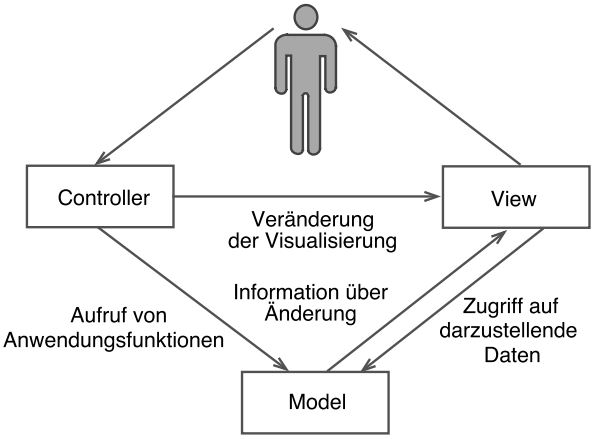



***Abb. 16–20*** *Interaktion zwischen den MVC-Komponenten*

MVC gliedert eine interaktive Anwendung in drei Komponenten:

- Das *Model*[5] realisiert die fachliche Funktionalität der Anwendung. Es kapselt die Daten der Anwendung, stellt jedoch Zugriffsoperationen zur Verfügung, mit denen die Daten abgefragt und verändert werden können.
- Eine *View* (Ansicht) präsentiert dem Benutzer die Daten. Sie verwendet die Zugriffsoperationen des Models. Zu einem Model kann es für unterschiedliche Darstellungen derselben Daten beliebig viele Views geben.
- Jeder View ist ein *Controller* zugeordnet. Dieser empfängt die Eingaben des Benutzers und reagiert darauf. Muss nach der Interaktion des Benutzers die Visualisierung geändert werden (z. B. nach der Selektion eines Textes), dann informiert der Controller die Views entsprechend. Wählt der Benutzer – beispielsweise über einen Menüeintrag – eine Funktion der Anwendung aus, dann ruft der Controller diese beim Model auf.

Abbildung 16–20 zeigt, wie die drei Komponenten miteinander verbunden sind: View und Model sind allen Komponenten bekannt, der Controller ist es nicht. Das MVC-Architekturmuster führt zu einer losen Kopplung zwischen den Komponenten und setzt das Entwurfsprinzip der Trennung von Zuständigkeiten um.

Wenn die Daten der Anwendung durch die Interaktion des Benutzers verändert werden, müssen die betroffenen Views darauf reagieren. Dazu sieht MVC

5 Wir bleiben hier bei den englischen Originalbegriffen, um Missverständnisse zu vermeiden.

einen Mechanismus vor, mit dem Veränderungen des Models den Views mitgeteilt werden. Das Model verwaltet dazu eine »Kundenliste« (*Register*), in der sich alle Views, die das Model darstellen, eintragen müssen. Ändert sich der Zustand des Models, so werden alle registrierten Views informiert; sie können dann reagieren und ihre Darstellung aktualisieren. Dieser Ablauf wird auch als *Change-update*-Mechanismus bezeichnet.

Abbildung 16–21 zeigt eine objektorientierte Modellierung der MVC-Komponenten. Sie sind durch Objektattribute miteinander verbunden. Das Model bietet neben den anwendungsspezifischen Operationen auch Operationen an, mit denen sich Views an- und abmelden können. Die private Operation `propagiereAenderung` dient dazu, allen Views eine Zustandsänderung des Models mitzuteilen (sie zu *propagieren*).

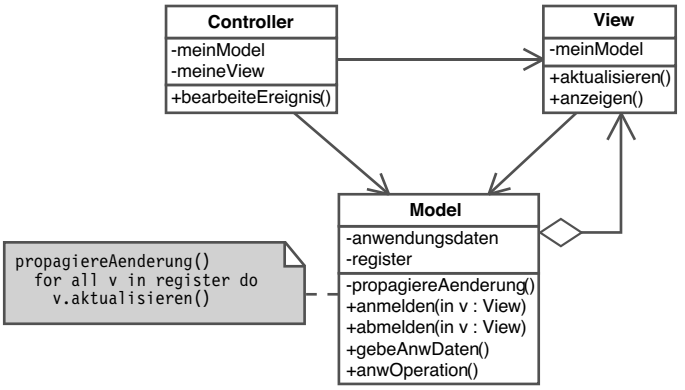



***Abb. 16–21*** *Objektorientiertes MVC-Modell*

Abbildung 16–22 zeigt, welche Nachrichten zwischen den MVC-Komponenten verschickt werden, wenn sich der Zustand des Models ändert.

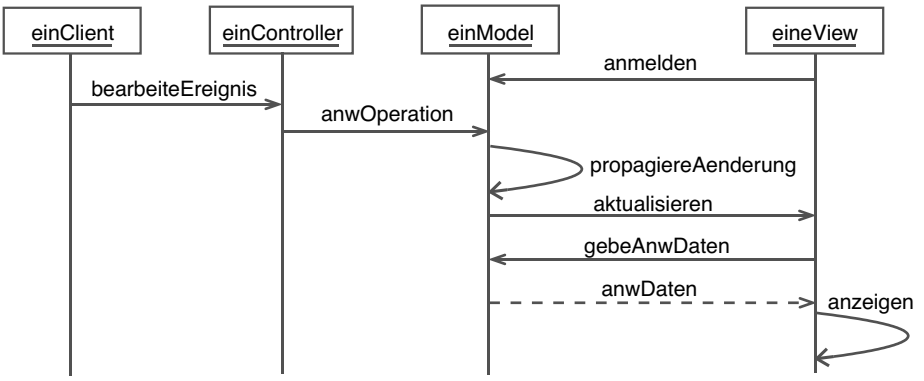



***Abb. 16–22*** *Nachrichtenfluss beim Change-update-Mechanismus (Sequenzdiagramm)*

Das MVC-Architekturmuster bietet die folgenden Vorteile:

- Es ist möglich, für dasselbe Model mehrere View-Controller-Paare vorzusehen. Aufgrund der sauberen Entkopplung vom Model können View-Controller-Paare sogar zur Laufzeit hinzugefügt oder weggenommen werden.
- Der Change-update-Mechanismus führt dazu, dass das Model in allen Views immer aktuell visualisiert wird.

Das Architekturmuster hat aber auch Schwächen. Wenn sich die Daten, die das Model verwaltet, sehr oft und sehr schnell ändern, kann das dazu führen, dass die Views diese Veränderungen nicht mehr schnell genug anzeigen können, weil jedes Mal die Daten vom Model erfragt werden müssen.

### 16.4.5 Das Plug-in-Architekturmuster

In Zeiten der imperativen Programmierung waren Software-Systeme abgeschlossen, d. h., jede Erweiterung war eine aufwendige, fehlerträchtige Wartungsarbeit und nur denen möglich, die Zugang zum Quellcode, zur Dokumentation und ggf. zu bestimmten Entwicklungssystemen hatten. Das Plug-in-Architekturmuster bietet allen Benutzern die Möglichkeit, ein System an dafür vorgesehenen Punkten zu erweitern, ohne es zu modifizieren; es setzt somit das Offen-geschlossen-Prinzip auf der Ebene von Anwendungen um.

Eine Anwendung, die nach diesem Architekturmuster aufgebaut ist, besteht aus einem Kern (auch *Host* = Wirt genannt), der durch sogenannte Plug-ins um neue Funktionen erweitert werden kann. Der Host definiert spezielle Schnittstellen, die sogenannten *Erweiterungspunkte*, auf die ein Plug-in Bezug nehmen kann (Abb. 16–23).

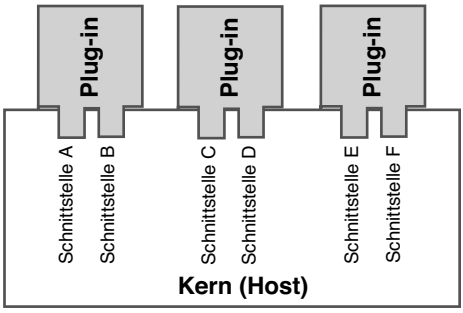



***Abb. 16–23*** *Schema einer Plug-in-Architektur*

Damit ein Plug-in im Kontext des Hosts ausgeführt werden kann, muss es gemäß den vom Host vorgegebenen technischen Konventionen realisiert sein, der Code muss entsprechend abgelegt und das Plug-in beim Host registriert sein. Beim Start des Hosts werden die aktuell vorhandenen Plug-ins identifiziert und entweder

sofort oder nach Bedarf geladen. Das Plug-in kann selbst ein Host sein, d. h., Plug-ins können Erweiterungspunkte für weitere Plug-ins definieren.

ECLIPSE ist eine Anwendung, die konsequent dieses Muster umsetzt (siehe Gamma, Beck, 2004); Werkzeuge wie THUNDERBIRD oder FIREFOX sind weitere Beispiele.

### 16.4.6 Das Microservice-Architekturmuster

In den letzten Jahren wurde mit den sogenannten »Microservices« ein neuer Komponententyp von Software-Architekturen eingeführt. Insbesondere Martin Fowler hat diesen Architekturansatz populär gemacht (Fowler, Lewis, 2014 und 2015). Software, die aus Mircoservices besteht, hat eine Microservice-Architektur, sie selbst wird als Microservice-basierte Software bezeichnet.

Es gibt leider keine allgemein akzeptierte Definition, die klärt, was ein Microservice und eine Microservice-Architektur genau sind. Jedoch gibt es eine Reihe von Merkmalen, die Microservices charakterisieren. Wolff (2018b) nennt die folgenden:

- Microservices sind das Ergebnis der Modularisierung von Anwendungen, d. h., eine Anwendung besteht auf Ebene der Systemarchitektur aus Microservices, die miteinander interagieren.
- Microservices realisieren fachliche Funktionen und stellen diese den Kunden zur Verfügung.
- Microservices speichern und verwalten die eigenen Daten.
- Microservices kommunizieren ausschließlich über das Netzwerk miteinander und nutzen dazu Protokolle, die die lose Kopplung unterstützen, oft HTTP-basiert.
- Mircoservices werden als eigenständige Prozesse ausgeführt, häufig in Form sogenannter Container.
- Microservices können unabhängig von anderen ausgeliefert und in Betrieb genommen werden.

Wir stellen fest, dass Microservices sowohl Komponenten der Systemarchitektur als auch Einheiten des Software-Deployments sind. Dies unterscheidet sie von vielen anderen Architekturkomponenten, beispielsweise von Schichten oder Modulen.

Diese oben genannten Eigenschaften schaffen jedoch keine Ordnung oder eine Verteilung von Aufgaben, wie dies bei anderen Architekturmustern der Fall ist. Das muss durch einen geeigneten Zuschnitt der Microservices erreicht werden. Wir behandeln den Entwurf von Anwendungssoftware, dessen Ergebnisse Kandidaten für Microservices sein können, in Abschnitt 16.7.

Trotzdem hat dieses Architekturmuster Vorteile, die für moderne und große Anwendungssoftware erstrebenswert sind. Dazu zählen:

- Ein Microservice kann einen anderen Microservice nur über eine Netzwerk-Kommunikation verwenden. Benutzt-Beziehungen zwischen Microservices sind demnach nicht Code-basiert, sie können nicht durch einen einfachen Code-Import etabliert werden. Das verhindert auf der einen Seite, dass Benutzt-Beziehungen leichtfertig angelegt werden. Auf der anderen Seite können die Benutzt-Beziehungen nicht aus dem Code abgeleitet werden.
- Microservices können recht einfach ersetzt werden, da jeder Microservice eine kleine abgeschlossene Komponente ist, die separat ausgeliefert und betrieben werden kann. Dies erleichtert es, neue Versionen von Microservices zu integrieren und zu nutzen oder einen Microservice durch eine neue Implementierung zu ersetzen.
- Da Microservices in eigenen Prozessen ausgeführt werden, können mehrere Instanzen eines Microservice parallel betrieben werden, falls das notwendig ist. Eine Last-Skalierung kann also dediziert realisiert werden und betrifft nicht die gesamte Software.
- Die Entwicklung und Wartung von Microservices kann gut auf Teams verteilt werden; ein Team realisiert einen Microservice oder mehrere. Da Microservices abgeschlossene Komponenten sind, können viele Entscheidungen, beispielsweise für die genutzte Technologie oder Werkzeuge, teamintern gefällt werden, da andere Teams davon nicht betroffen sind.

Diesen Vorteilen stehen folgende Herausforderungen gegenüber:

- Eine Microservice-basierte Software ist immer ein verteiltes System. Verteilte Systeme sind software-technisch gesehen immer komplex. Dazu tragen die Kommunikationsmechanismen bei, die verteilte Systeme erfordern, aber auch die Mechanismen, die notwendig sind, damit das verteilte System robust gegen Ausfälle ist.
- Da jeder Microservice seine eigenen Datenbestände verwaltet, gibt es keine gemeinsam genutzten Datenbanken. Dementsprechend werden spezielle Mechanismen benötigt, um die Integrität und Konsistenz von Daten über die Grenzen von Microservices zu gewährleisten.
- Jeder Microservice braucht eine eigene angepasste Entwicklungsumgebung, die die Teams aufsetzen und pflegen müssen. Insbesondere muss die Entwicklungsumgebung die Teams dabei unterstützen, dass neue Versionen der Microservices automatisiert ausgeliefert und in Betrieb genommen werden können; dies leisten sogenannte Continuous-Deployment-Pipelines (siehe Abschnitt 20.5.4).

Diese Vorteile und Herausforderungen müssen bewertet und abgewogen werden, wenn Software auf der Basis von Microservices entwickelt werden soll.

In der Literatur findet man bewährte Lösungsansätze für die genannten Probleme und viele Muster, die im Kontext einer Microservice-basierten Entwicklung genutzt werden können. Chris Richardson (2019) beschreibt wichtige Mus-

ter, unter anderem das Gateway-Muster. Ein Gateway übernimmt in einer Microservice-Architektur die Aufgabe, die Anfragen der Kunden an die richtigen Microservices weiterzuleiten. Abbildung 16–24 zeigt schematisch ein Beispiel für eine Microservice-Architektur, die eine Gateway-Komponente enthält.

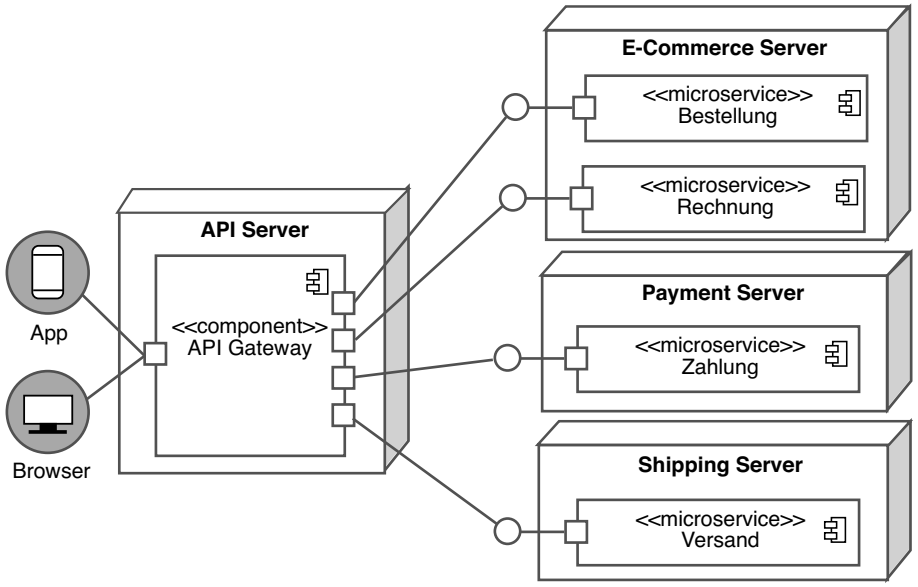



***Abb. 16–24*** *Schema einer Microservice-Architektur (UML-Verteilungsdiagramm)*

Der Ansatz, Software durch funktionale Module, durch Services, zu strukturieren, ist nicht neu. So wurde unter dem Namen »Service-Oriented Architecture« (SOA) ein unternehmensweiter Ansatz vorgestellt, um die gesamte Anwendungslandschaft eines Unternehmens auf Basis von Services zu strukturieren. Eine gute Einführung in SOA findet sich in Erl (2016). Das Microservice-Architekturmuster ist für einzelne Anwendungen konzipiert. Es ermöglicht, eine Anwendung in kleine Komponenten zu zerlegen, die unabhängig voneinander geändert, skaliert und verwaltet werden können. Es wird nicht festgelegt, wie die Anwendungen einer Anwendungslandschaft miteinander kommunizieren.

Eberhard Wolff beschreibt in seinen Büchern detailliert die Microservice-basierte Entwicklung von Software aus technischer und organisatorischer Sicht (Wolff, 2018a und 2018b).

## 16.5 Entwurfsmuster

Architekturmuster sind Muster auf der Ebene der Systemarchitektur. Ähnliche Muster gibt es auch für den Entwurf der Komponenten; sie werden *Entwurfsmuster* (*design patterns*) genannt. Entwurfsmuster gehen auf die Arbeit von Erich

Gamma zurück, der die Architektur eines Rahmenwerks für Editoren (ET++) durch einen Katalog von Entwurfsmustern beschreibt (Gamma, 1992). Daraus und aus den Entwurfserfahrungen anderer entstanden zwanzig Entwurfsmuster (Gamma et al., 1995). Auch andere Autoren haben Kataloge mit Entwurfsmustern vorgelegt (z. B. Buschmann et al., 1996).

Entwurfsmuster befassen sich im Gegensatz zu Architekturmustern mit kleineren Einheiten. Sie bieten Lösungen für gängige Entwurfsprobleme auf der Ebene des Feinentwurfs von Komponenten an. Architektur- und Entwurfsmuster dienen dazu, das Wissen über guten Software-Entwurf zu bewahren und wiederholt zu verwenden. Entwurfsmuster werden unabhängig von einer bestimmten Programmiersprache formuliert, sie greifen aber meist auf objektorientierte Konzepte zurück, implizieren damit also eine objektorientierte Codierung. Beispiele für Entwurfsmuster sind das Beobachter- und das Adapter-Muster.

Die Väter der Entwurfsmuster, Gamma, Helm, Johnson und Vlissides, charakterisieren Entwurfsmuster wie folgt:

> *Design patterns ... are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.*
>
> Gamma et al. (1995)

Wir lehnen uns an dieses Verständnis von Entwurfsmustern an und definieren:

Ein **Entwurfsmuster** beschreibt das Schema einer Lösung für ein bestimmtes, in verschiedenen konkreten Zusammenhängen wiederkehrendes Entwurfsproblem.

Ein Entwurfsmuster bietet eine generische Lösung für ein häufig auftretendes Entwurfsproblem an. Um es zu verwenden, muss man wissen, welches Problem das ist und wie man das Entwurfsmuster im Kontext einer konkreten Architektur ausprägen kann. Die Urheber eines Entwurfsmusters müssen diese beiden Informationen mitliefern.

Bevor wir einige wichtige Entwurfsmuster vorstellen, wollen wir an einem Beispiel zeigen, wie ein Entwurfsmuster eingesetzt werden kann.

### 16.5.1 Ein einführendes Beispiel

Im folgenden Beispiel, übernommen aus Shalloway und Trott (2002), soll eine Anwendung zur elektronischen Verkaufsabwicklung realisiert werden. Im ersten Entwurf sehen wir dafür zwei Klassen vor (Abb. 16–25): Die Klasse `Auftragsmanagement` dient dazu, eingehende Aufträge (zunächst nur aus Deutschland) anzunehmen, zu prüfen und zu verwalten. Die Aufträge werden an Objekte der

Klasse `Auftragsabwicklung` übergeben, die die Aufträge ausführen. Um die anfallenden Steuern zu ermitteln, enthält diese Klasse die Methode `berechneSteuer`.

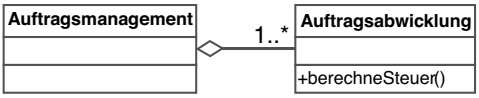



***Abb. 16–25*** *Auftragsabwicklungssystem – Architektur A*

Die Anwendung soll nun erweitert werden, damit auch Aufträge aus Österreich und der Schweiz abgewickelt werden können. Die naheliegende Implementierung dieser neuen Anforderung zeigt die Architektur B in Abbildung 16–26.

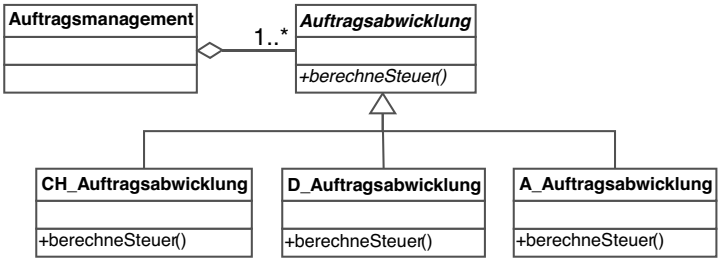



***Abb. 16–26*** *Auftragsabwicklungssystem – Architektur B*

Wir haben von der – jetzt abstrakten – Klasse `Auftragsabwicklung` drei länderspezifische Klassen abgeleitet, damit darin die geerbte Methode `berechneSteuer` entsprechend den jeweiligen Vorschriften zur Steuerberechnung realisiert wird. Wenn noch weitere Länder hinzukommen, kann die Klassenhierarchie erweitert werden. Das `Auftragsmanagement` entscheidet je nach Herkunftsland des Auftrags, welchem Objekt dieser Auftrag zur Abwicklung übermittelt werden muss.

Die Hierarchie der Klassen zur Auftragsabwicklung in Architektur B haben wir nur gebildet, weil die Steuerberechnung von Land zu Land variiert. Wir können diesen variablen Aspekt aber auch explizit durch eine eigene Klasse (`SteuerRechner`) modellieren, die von der Klasse `Auftragsabwicklung` benutzt wird. Die verschiedenen Steuerberechnungsvorschriften werden in Unterklassen der Klasse `SteuerRechner` realisiert. Die Auftragsabwicklungsobjekte können dann die Berechnung an die speziell dafür vorgesehenen Steuerberechnungsobjekte delegieren. Abbildung 16–27 zeigt das Klassendiagramm der neuen Architektur C.

Die Architektur C hat den Vorteil, dass die Steuerberechnungen nach dem Prinzip der Trennung von Zuständigkeiten separat modelliert werden. Wenn sie sich ändern, können die notwendigen Anpassungen lokal durchgeführt werden. Die Klassen haben zudem einen besseren Zusammenhalt, da Auftragsabwicklung und Steuerberechnung nicht in einer Klasse gemischt werden.

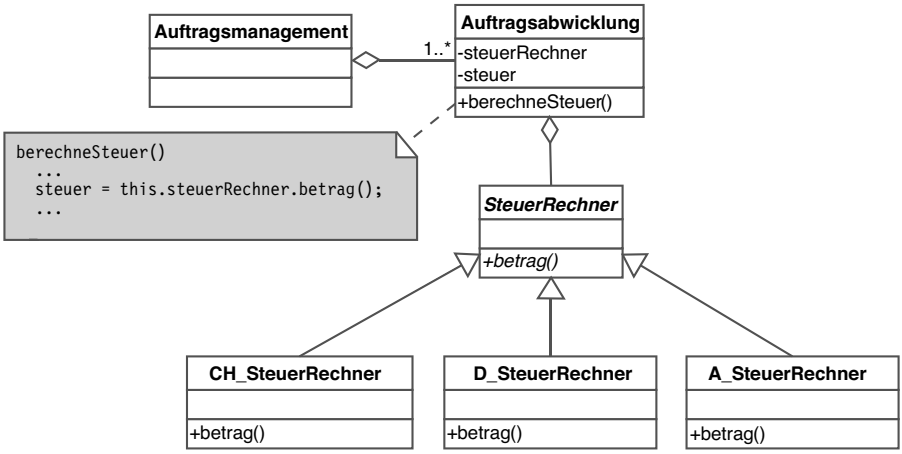
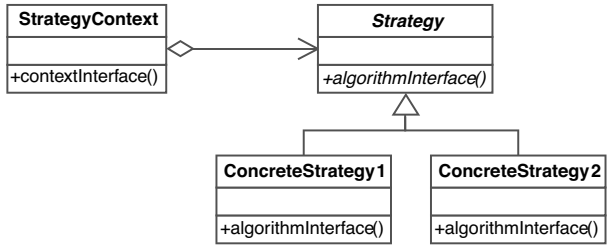



***Abb. 16–27*** *Auftragsabwicklungssystem – Architektur C*

In Architektur C ist das Entwurfsmuster *Strategie* eingesetzt. Entwurfsmuster beschreiben eine Problemlösung, indem sie angeben, welche Rollen die beteiligten Klassen übernehmen, wofür sie verantwortlich sind und wie sie interagieren. Zur Darstellung der Entwurfsmuster werden Text sowie Klassen- oder Objektdiagramme verwendet. Tabelle 16–3 beschreibt das Strategie-Muster.

| **Strategie (Strategy)** | |
|---|---|
| Problem | Verwandte Klassen unterscheiden sich lediglich dadurch, dass sie gleiche Aufgaben teilweise durch verschiedene Algorithmen lösen. |
| Lösung | Die Klassen werden nicht – wie üblich – in einer Vererbungshierarchie angeordnet. Stattdessen wird eine Klasse erstellt, die alle gemeinsamen Operationen definiert (*StrategyContext*). Die Signaturen der Operationen, die unterschiedlich zu implementieren sind, werden in einer weiteren Klasse zusammengefasst (*Strategy*). Die Rolle Strategy legt somit fest, über welche Schnittstelle die verschiedenen Algorithmen genutzt werden. Von dieser Klasse wird für jede Implementierungsalternative eine konkrete Unterklasse abgeleitet (*ConcreteStrategy*). Die Klasse mit der Rolle StrategyContext benutzt konkrete Strategy-Objekte, um die unterschiedlich implementierten Operationen per Delegation auszuführen. |
| Struktur |  |



***Tab. 16–3*** *Das Entwurfsmuster »Strategie«*

In unserem Beispiel (Abb. 16–28) kapselt die Klasse `SteuerRechner` den Aspekt der Steuerberechnung. Die verschiedenen länderspezifischen Vorschriften (Strategien) werden in Unterklassen implementiert. Objekte dieser Klassen werden dazu verwendet, die Steuern zu berechnen. Übertragen wir die Struktur des Strategie-Musters auf unseren Entwurf, dann erhalten wir die Zuordnung der Musterrollen zu den Klassen in Abbildung 16–28.

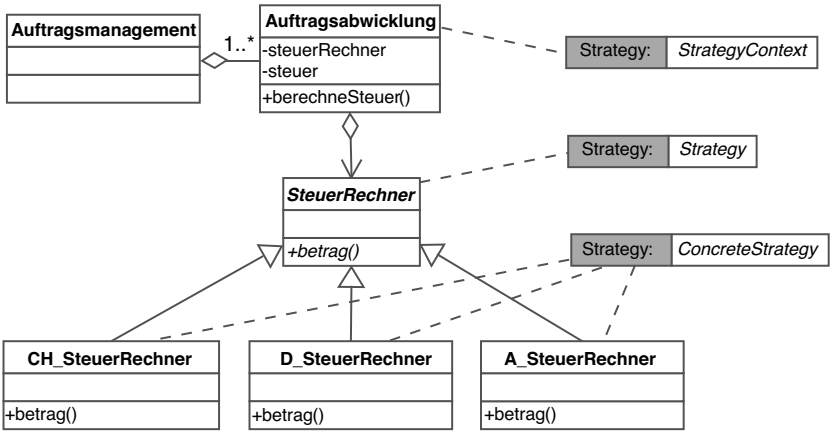



***Abb. 16–28*** *Zuordnung zwischen den Musterrollen und Klassen*

- Die Klasse `Auftragsabwicklung` besetzt die Rolle *StrategyContext*.
- Die Klasse `SteuerRechner` spielt die Rolle *Strategy*.
- Jede der Klassen `CH_SteuerRechner`, `D_SteuerRechner` und `A_SteuerRechner` hat eine Rolle als *ConcreteStrategy*.

Wie man an diesem Beispiel sieht, verschafft uns der Einsatz eines Entwurfsmusters nicht nur eine erprobte und tragfähige Architektur, sondern auch einen griffigen Namen, um diese Architektur zu benennen. Wenn wir verstanden haben, wie ein Teil der Architektur, der nach dem Strategie-Muster konstruiert ist, aufgebaut ist und wie die einzelnen Klassen zusammenarbeiten, müssen wir nicht mehr alle Einzelheiten ergründen und nachvollziehen. Entwurfsmuster schaffen also auch eine Sprache, um Entwürfe zu kommunizieren und zu dokumentieren.

### 16.5.2 Einige Entwurfsmuster

Nachfolgend stellen wir einige der in Gamma et al. (1995) beschriebenen Muster vor, die häufig beim Software-Entwurf genutzt werden können. Wir ordnen sie nach Einsatzbereichen, beschreiben für jedes Muster kurz das Problem, das es löst, und geben ein Beispiel an. Für eine detaillierte Beschreibung verweisen wir auf die Originalarbeiten und auf die weiterführende Literatur (z. B. Shalloway, Trott, 2002).

### Muster zum Verwalten von Objekten

| **Einzelstück (Singleton)** | |
|---|---|
| *Problem* | Von einer Klasse darf nur genau ein global verfügbares Objekt erzeugt werden. |
| *Beispiel* | Es darf nur einen Drucker-Spooler im System geben. |
| **Memento** | |
| *Problem* | Der Zustand eines Objekts muss so archiviert werden, dass es wieder in diesen Zustand zurückversetzt werden kann, ohne das Prinzip der Kapselung zu verletzen. |
| *Beispiel* | Ein Undo-Mechanismus soll realisiert werden. |

### Muster zur Anbindung vorhandener Klassen oder Komponenten

| **Adapter** | |
|---|---|
| *Problem* | Eine Klasse soll verwendet werden, obwohl ihre Methoden-Signaturen nicht passen und auch nicht verändert werden können. |
| *Beispiel* | In einem Editor soll eine Klasse, die eine einfache Rechtschreibprüfung realisiert, durch eine zugekaufte Klasse ersetzt werden. |
| **Vermittler (Mediator)** | |
| *Problem* | Objekte, die auf komplexe Art miteinander interagieren müssen, dürfen nur lose aneinander gekoppelt sein und müssen sich leicht austauschen lassen. |
| *Beispiel* | Das Aussehen und der Zustand verschiedener Interaktionsmittel (Menüs, Knöpfe, Eingabefelder) einer grafischen Bedienoberfläche sollen in jedem Interaktionszustand konsistent aufeinander abgestimmt werden. |

### Muster zur Entkopplung von Komponenten

| **Brücke (Bridge)** | |
|---|---|
| *Problem* | Fachliche Anwendungsklassen und deren Implementierungsvarianten sollen einfach kombiniert und weiterentwickelt werden können. |
| *Beispiel* | Es sollen Mengen-Klassen implementiert werden (z. B. einfache Menge, Multimenge). Zur Verwaltung der Elemente stehen verschiedene Container-Klassen zur Verfügung (Listen, Bäume etc.). Die Mengen-Klassen bilden die fachlichen Anwendungsklassen; die Container-Klassen sind die Varianten der Implementierung. |
| **Fassade (Facade)** | |
| *Problem* | Eine Komponente soll nicht den gesamten, sondern nur einen eingeschränkten Funktionsumfang anderer Komponenten kennen. |
| *Beispiel* | Eine Komponente zur statistischen Auswertung benötigt nur wenige Funktionen eines Kundeninformationssystems. |
| **Beobachter (Observer)** | |
| *Problem* | Mehrere Objekte müssen ihren Zustand anpassen, wenn sich ein bestimmtes Objekt ändert. |
| *Beispiel* | Alle GUI-Objekte, die ein Dateisystem darstellen, müssen ihre Darstellung anpassen, wenn Dateien eingefügt oder gelöscht werden. |

**Muster zur Trennung der unterschiedlichen von den gemeinsamen Merkmalen**

| | |
|---|---|
| **Abstrakte Fabrik (Abstract Factory)** | |
| *Problem* | In einer Anwendung müssen kontextabhängig Objekte unterschiedlicher Klassen erzeugt werden. |
| *Beispiel* | Ein grafischer Editor muss abhängig vom eingesetzten Monitortyp unterschiedliche grafische Objekte erzeugen. |
| **Schablonenmethode (Template Method)** | |
| *Problem* | Ein Algorithmus, zu dem es mehrere Implementierungen gibt, soll so realisiert werden, dass man neue Varianten einfach hinzufügen kann. |
| *Beispiel* | Das Löschen eines Objekts aus einer Objektsammlung kann allgemein beschrieben werden. Je nach Objektsammlung ist dieser Algorithmus jedoch unterschiedlich zu implementieren. |
| **Strategie (Strategy)** | |
| *Problem* | Objekte, die sich nur dadurch unterscheiden, dass sie gleiche Aufgaben teilweise durch verschiedene Algorithmen lösen, sollen durch eine Klasse, nicht durch eine Klassenhierarchie implementiert werden. |
| *Beispiel* | Objekte, die Eingabemasken implementieren, unterscheiden sich darin, wie die Eingaben geprüft werden. |
| **Zustand (State)** | |
| *Problem* | Ein Objekt muss sein Verhalten abhängig vom Zustand ändern. |
| *Beispiel* | Die Berechnung der Mietkosten eines Leihwagens ist davon abhängig, welcher Fahrzeugkategorie der Wagen zugeordnet ist. |

### 16.5.3 Die praktische Anwendung der Entwurfsmuster

Im Alltag übernimmt eine Person viele Rollen, sie ist beispielsweise Mutter, Tochter, Angestellte, Wählerin, Vereinspräsidentin. Ähnlich verhält es sich mit den Klassen einer Architektur. Jedes einzelne Entwurfsmuster beschreibt ein Lösungsschema für ein bestimmtes Entwurfsproblem. Da es im konkreten Fall eine Reihe solcher Entwurfsprobleme gibt, kann eine Klasse gleichzeitig an mehreren Entwurfsmustern beteiligt sein, d. h., eine Klasse implementiert verschiedene Rollen der einzelnen Muster. Ein solcher musterbasierter Entwurf hilft, das häufig sehr komplexe Zusammenspiel von Klassen zu modellieren und zu verstehen.

Wir wollen dies am Beispiel des JHotDraw-Rahmenwerks zeigen[6]. Dieses objektorientierte Rahmenwerk definiert und implementiert die gemeinsame Architektur für interaktive grafische Editoren. Es erfüllt unter anderem die folgenden Anforderungen:

---

6 JHotDraw wurde von Erich Gamma entwickelt. Als Vorlage diente HotDraw, ein Rahmenwerk zur 2D-Visualisierung von Kent Beck und Ward Cunningham, implementiert in SMALLTALK. Wir beziehen uns hier auf JHotDraw in der Version 5. Mittlerweile ist JHotDraw ein Open-Source-Projekt unter Sourceforge (JHotDraw, o. J.).

- Die Reaktion auf eine Benutzerinteraktion (z. B. einem Mausklick in der Zeichenfläche) muss sich leicht ändern lassen.
- Eine Grafik soll auf beliebig vielen Zeichenflächen darstellbar sein.
- Der Algorithmus, der die Zeichenfläche nach einer grafischen Manipulation aktualisiert, soll einfach austauschbar sein.

Abbildung 16–29 zeigt den Kern des JHotDraw-Klassenentwurfs. Darin kommen die Entwurfsmuster *Strategy*, *Observer*, *Mediator* und *State* zum Einsatz. Die zentralen Klassen sind `DrawingEditor`, `DrawingView`, `Drawing` und `Tool`.

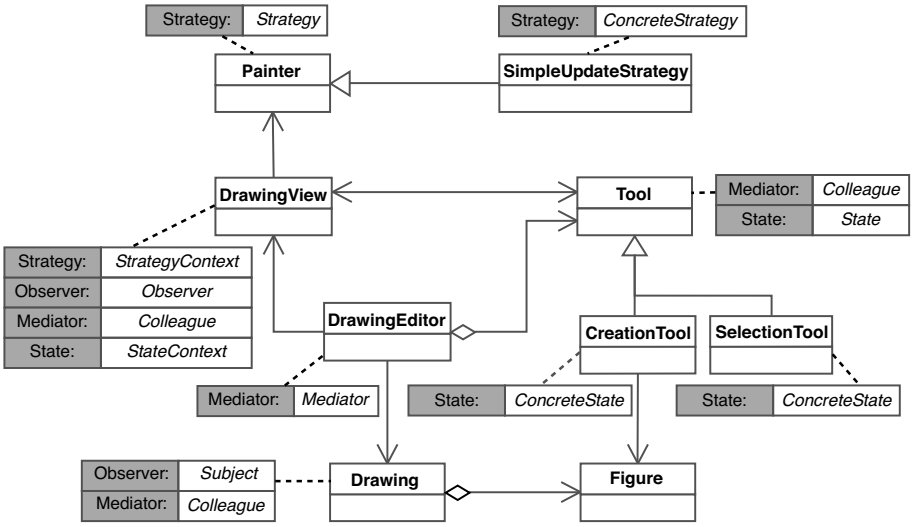



***Abb. 16–29*** *Überlagerung der Rollen im Rahmenwerk JHotDraw*

Um das komplexe Zusammenspiel dieser Klassen an einer Stelle zu koordinieren und die Klassen voneinander zu entkoppeln, wird das Muster *Mediator* verwendet. Mediator sieht vor, dass ein Vermittlerobjekt (*Mediator*) realisiert wird, das die Interaktion aller Objekte kapselt. Jedes an der Interaktion beteiligte Objekt (*Colleague*) braucht dann nur dieses Vermittlerobjekt zu kennen. Die Klasse `DrawingEditor` übernimmt die Rolle des Mediators. Die Klassen, deren Zusammenspiel vom Mediator koordiniert wird, also die Klassen `DrawingView`, `Drawing` und `Tool`, haben alle eine Rolle *Colleague*.

Damit mehrere Zeichenflächen dieselbe Grafik darstellen, werden die entsprechenden Klassen nach dem Muster *Observer* entworfen. Die Klasse `Drawing` liefert die beobachteten Objekte, die Zeichnungen. Sie hat im Observer-Muster die Rolle *Subject*. Die Klasse `DrawingView` realisiert die Zeichenflächen, die über Änderungen an den Zeichnungen informiert werden. Sie übernimmt damit die Rolle *Observer*.

Damit der Algorithmus zum Aktualisieren der Zeichenfläche leicht ausgetauscht werden kann, wird er nach dem Muster *Strategy* separat modelliert. In der Klasse `DrawingView` wird der Aktualisierungsalgorithmus benötigt, deshalb übernimmt diese Klasse die Rolle *StrategyContext*. Die Klasse `Painter` definiert die Schnittstelle des Algorithmus und spielt darum die Rolle *Strategy*. Ihre Unterklasse `SimpleUpdateStrategy` implementiert in der Rolle *ConcreteStrategy* einen einfachen Aktualisierungsalgorithmus.

Die Reaktion eines Editors auf gleiche Benutzerinteraktionen kann unterschiedlich sein. Muss beispielsweise in einem Fall bei einem Mausklick ein neues Symbol erzeugt werden, so muss in einem anderen Fall das ausgewählte Symbol markiert werden. Dieses unterschiedliche Verhalten wird durch das Muster *State* erzielt. Die Signaturen der Operationen, die je nach Zustand unterschiedliches Verhalten zeigen sollen, werden in eine separate Klasse ausgelagert (*State*), die die Wurzel einer Klassenhierarchie bildet. In Unterklassen wird das zustandsabhängige Verhalten implementiert (*ConcreteState*). In der Klasse, die sich zustandsabhängig verhalten soll (*StateContext*), wird eine Referenz auf die Wurzel dieser Klassenhierarchie angelegt. Je nach Zustand wird dieser Referenz ein konkretes Objekt der Zustandsklassenhierarchie zugewiesen, an das die Ausführung der zustandsabhängigen Operationen delegiert wird.

Im Beispiel werden alle Benutzerinteraktionen, die je nach Zustand des Editors unterschiedlich sein können, in einer eigenen Klasse `Tool` zusammengefasst, die die Rolle *State* übernimmt. Ihre Unterklassen, z. B. `CreationTool` und `SelectionTool`, realisieren das unterschiedliche Verhalten dadurch, dass sie die gemeinsame Schnittstelle entsprechend redefinieren (*ConcreteState*). Die Klasse `DrawingView` übernimmt die Rolle *StateContext*, indem sie die Tool-Objekte verwendet, an die die Ausführung der Benutzerinteraktionen delegiert wird.

Eine weiter gehende Betrachtung der Entwurfsmuster in JHotDraw und in anderen Rahmenwerken ist in Riehle (2000) enthalten.

### 16.5.4 Bewertung

Die Entwicklung von Entwurfsmustern gilt als eine der wichtigsten Innovationen des Software Engineerings in den letzten Jahren. Wir sehen bei ihrem Einsatz die folgenden Vorteile:

- Die Entwurfsmuster geben uns die Möglichkeit, die Erfahrungen anderer zu nutzen und erprobte Lösungen einzusetzen.
- Sie unterstützen uns, nichtfunktionale Anforderungen, wie Änderbarkeit oder Wiederverwendbarkeit, beim Architekturentwurf zu berücksichtigen.
- Sie schaffen ein Vokabular und erleichtern dadurch die Dokumentation von Architekturen und die Kommunikation über Architekturen.
- Sie können beim Reengineering vorhandener Software als Hilfsmittel zur Analyse dienen.

- Sie können flexibel miteinander kombiniert werden. So kann eine Klasse in verschiedenen Rollen an unterschiedlichen Entwurfsmustern beteiligt sein.
- Nicht zuletzt helfen Entwurfsmuster dabei, den Software-Entwurf zu lehren.

Natürlich sind Entwurfsmuster kein Allheilmittel. Manchmal werden sie sogar regelrecht missbraucht; so werden Fehlkonstruktionen wie der übermäßige Gebrauch globaler Datenstrukturen mit der Anwendung eines Entwurfsmusters (in diesem Fall »Singleton«) begründet.

Ein Entwurfsmuster zu verstehen ist einfach. Man braucht jedoch viel Entwurfserfahrung, um die Muster sinnvoll einzusetzen.

## 16.6 Weitere Arten der Wiederwendung von Architekturen

Beim Entwurf von Architekturen verwenden wir Wissen und erprobte Lösungen, wann immer das möglich ist. Neben Entwurfsprinzipien gibt es eine Vielzahl von Mustern auf der Ebene der Systemarchitektur und des Entwurfs, die dafür zur Verfügung stehen. Wir haben einige davon in diesem Kapitel vorgestellt.

Nachfolgend beschreiben wir weitere Ansätze zur Wiederwendung im Kontext des Entwurfs von Architekturen. Unsere Sammlung ist natürlich nicht vollständig.

### 16.6.1 Klassenbibliotheken

Da die Entwicklung von Software teuer ist, liegt es nahe, so viele Komponenten wie möglich mehrfach zu verwenden. Was bereits zuvor eingesetzt wurde, ist in der Regel geprüft, zuverlässig im Einsatz und sofort verfügbar. Die Entwicklungstiefe wird durch die Verwendung vorhandener Bausteine reduziert. Alle diese Aspekte tragen dazu bei, dass Software schneller und günstiger entwickelt werden kann.

Bereits in den Sechzigerjahren wurden Programmbibliotheken (z.B. für Matrixoperationen in FORTRAN) entwickelt und eingesetzt. Allerdings war die Flexibilität dieser Bausteine sehr begrenzt.

Als die Objektorientierung aufkam, war eine ihrer wichtigsten Verheißungen, dass damit die Wiederverwendung signifikant verbessert würde. Der Schlüssel dazu ist die Vererbung zwischen Klassen. Schon frühzeitig wurden Klassenbibliotheken entwickelt, deren Funktionalität bei der Anwendungsentwicklung oft benötigt wird. Vorreiter war hier die SMALLTALK-80-Entwicklungsumgebung, die dem Anwendungsentwickler eine Vielzahl universeller Klassen zur Verfügung stellte. Wir definieren:

Eine **Klassenbibliothek** besteht aus einer Menge von Klassen, die wiederverwendbar sind und allgemein – also unabhängig vom Anwendungskontext – nutzbare Funktionalität anbieten.

Aus Sicht der Anwendung werden die Klassen einer Bibliothek direkt benutzt, oder die Klassen der Anwendung erben von den Bibliotheksklassen. Klassenbibliotheken setzen somit das Offen-geschlossen-Prinzip um (Abschnitt 16.3.6); sie können von der Anwendung unverändert genutzt, aber auch flexibel erweitert werden.

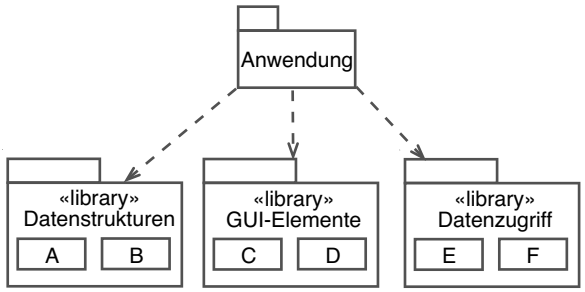



***Abb. 16–30*** *Nutzung von Klassenbibliotheken*

Abbildung 16–30 zeigt eine Situation, bei der die Anwendung Klassen verschiedener Bibliotheken nutzt. Die Bibliothek `Datenstrukturen` stellt allgemein wiederverwendbare Datenstrukturen wie Listen oder Bäume zur Verfügung; die Bibliothek `GUI-Elemente` enthält Klassen, die benutzt werden, um die Bedienoberfläche zu realisieren; die Bibliothek `Datenzugriff` bietet Klassen an, die Daten der Anwendung in einer Datenbank persistent, also über den einzelnen Programmlauf hinaus, speichern können.

Klassenbibliotheken sind heute feste Bestandteile der Entwicklungsumgebungen, die zu den objektorientierten Programmiersprachen gehören.

### 16.6.2 Rahmenwerke

Wenn immer wieder sehr ähnliche Anwendungen entwickelt werden, sollten dazu nicht nur einzelne Klassenbibliotheken genutzt werden, sondern die Anwendungen sollten auf der Basis einer generischen Lösung entwickelt werden. Eine solche generische Lösung wird auch als *Rahmenwerk* (*Framework*) bezeichnet. Es gibt in der Literatur verschiedene Definitionen, wir halten uns an Züllighoven:

> Ein **Rahmenwerk** (Framework) ist eine Architektur aus Klassenhierarchien, die eine allgemeine generische Lösung für ähnliche Probleme in einem bestimmten Kontext vorgibt.
>
> Züllighoven (2005)

Bei einem Rahmenwerk werden nicht nur einzelne Klassen, sondern die gesamte Architektur und die Konstruktion aus kooperierenden Klassen wiederverwendet. Im Gegensatz zu einer Klassenbibliothek gibt ein Rahmenwerk der Anwendung auch den Kontrollfluss vor.

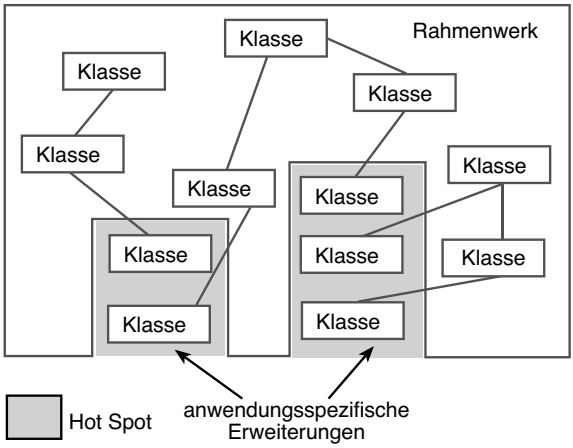



***Abb. 16–31*** *Schema einer rahmenwerkbasierten Anwendung*

Ein Rahmenwerk hat definierte Schnittstellen, an denen die generische Lösung durch anwendungsspezifischen Code erweitert werden kann (siehe Abb. 16–31). Diese Stellen werden als *Hot Spots*[7] (Pree, 1997) bezeichnet. Die Entwicklung einer Anwendung auf Basis eines Rahmenwerks unterscheidet sich drastisch von der herkömmlichen Anwendungsentwicklung, bei der lediglich Klassenbibliotheken genutzt werden. Während dort der gesamte Kontrollfluss durch den Anwendungsentwickler festgelegt wird, müssen bei der rahmenwerkbasierten Entwicklung die anwendungsspezifischen Teile in das Rahmenwerk und in den darin vorgegebenen Kontrollfluss integriert werden.

Ein Rahmenwerk kann so konstruiert sein, dass man es ohne Kenntnis seiner Interna verwenden, also zu einer Anwendung erweitern kann. Wir sprechen in diesem Fall von einem *geschlossenen Rahmenwerk*. Die Alternative ist eine Konstruktion, die vom Entwickler der Anwendung verlangt, dass er die Interna kennt, insbesondere den Code der Hot Spots, da er neuen anwendungsspezifischen Code entwickeln und integrieren muss. In diesem Fall handelt es sich um ein *offenes Rahmenwerk*. In der Literatur werden dafür ähnlich wie beim Test die Attribute »black box« und »white box« verwendet; während die Metapher der Black Box anschaulich ist, hat die Charakterisierung als White Box keinen Sinn, denn ein weißer Kasten ist weder durchsichtig noch offen. Darum bleiben wir bei »geschlossen« und »offen«.

#### Offene Rahmenwerke

Um ein offenes Rahmenwerk zu einer Anwendung zu erweitern, muss der Entwickler die Architektur, den vorgegebenen Kontrollfluss und die im Rahmenwerk

7 Dieser Begriff stammt ursprünglich aus der Geologie und bezeichnet dort heiße Magmakammern im Erdinneren.

realisierten Mechanismen sehr genau kennen. Technisch gesehen wird ein offenes Rahmenwerk dadurch erweitert, dass die – häufig abstrakten – Klassen der Hot Spots spezialisiert werden. Das erfordert einiges Wissen darüber, welche Klassen zu welchem Zweck spezialisiert und welche Methoden redefiniert werden müssen. In der Regel kennt der Entwickler das Rahmenwerk und die darin realisierten Mechanismen und Abläufe aber nicht genau. Eine Dokumentation, die alle Erweiterungsmöglichkeiten vollständig beschreibt, wäre umfangreich und aufwendig. Darum gehören zu offenen Rahmenwerken neben einer Dokumentation der zentralen Entwurfsentscheidungen und Komponenten der Architektur auch Beispielanwendungen, die der Entwickler analysieren und als Vorlagen nutzen kann, um seine eigene Anwendung zu schaffen.

#### Geschlossene Rahmenwerke

In der Praxis entsteht ein geschlossenes Rahmenwerk aus einem offenen, nachdem damit ausreichende Erfahrungen gesammelt wurden. Dann können Bereiche identifiziert werden, die in geschlossener Form zur Anwendungsentwicklung verwendbar sind.

Ein geschlossenes Rahmenwerk erweitert der Entwickler, indem er Objekte spezieller Rahmenwerksklassen erzeugt und konfiguriert, auf die dann das Rahmenwerk zugreift. Dazu können auch Werkzeuge angeboten werden, mit denen das Rahmenwerk erweitert und konfiguriert wird. Wenn die Umstellung vom offenen zum geschlossenen Rahmenwerk nur teilweise vollzogen ist, wenn also bestimmte Hot Spots durch Unterklassenbildung, andere durch eine werkzeuggestützte Konfiguration erweitert werden (siehe beispielsweise Lichter, Schneider, 1993), spricht man auch von einer »Grey Box«.

#### Bewertung

Um ein Rahmenwerk zu entwickeln, muss man sein Anwendungsgebiet sehr gut kennen. Technisch ist anzustreben, dass sich die anwendungsspezifischen Erweiterungen möglichst sauber in das Rahmenwerk integrieren lassen. Beim Architekturentwurf von Rahmenwerken werden deshalb an den Integrationsstellen häufiger als sonst üblich Entwurfsmuster eingesetzt, insbesondere solche, die es erlauben, universelle Merkmale von variablen, in diesem Sinne anwendungsspezifischen Merkmalen sauber zu trennen.

Bis ein Rahmenwerk wirklich brauchbar ist, wird viel Aufwand investiert. Darum muss sein Nutzen entsprechend groß sein, sonst lohnt sich die Entwicklung nicht. Wir betrachten diesen Aspekt und andere Aspekte der Wiederverwendung in Kapitel 23.

Es sei an dieser Stelle darauf hingewiesen, dass Rahmenwerke eine naheliegende Implementierungstechnik für Software-Produktlinien sind.

### 16.6.3 Produktlinienarchitektur

Die Entwicklung von Produktlinien ist ein vielversprechender Ansatz, eine Reihe ähnlicher Produkte, die auf einer einzigen Plattform basieren, kostengünstig und schnell zu entwickeln, indem man den gemeinsamen Kern nur einmal realisiert. Dazu ist es notwendig, die Unterschiede und die Gemeinsamkeiten der Produkte zu identifizieren. Dann wird die Plattform so entwickelt, dass sie die Gemeinsamkeiten aller Produkte enthält und es erlaubt, produktspezifische Ergänzungen systematisch hinzuzufügen. Offensichtlich spielt die Architektur der Plattform eine zentrale Rolle.

Northrop und Clements definieren eine Produktlinienarchitektur wie folgt:

**product line architecture** — A core asset that is the software architecture for all the products in a software product line. A product line architecture explicitly provides variation mechanisms that support the diversity among the products in the software product line.

Northrop, Clements (2007)

Eine *Produktlinienarchitektur* ist demnach die Architektur des gemeinsamen Kerns aller Produkte einer Familie. Die Architekturen der einzelnen Produkte werden auf Basis der Produktlinienarchitektur entworfen; an definierten Stellen sind sie individuell ausgeprägt.

Um eine Produktlinienarchitektur produktspezifisch zu erweitern, können verschiedene Mechanismen verwendet werden.

- In Unterklassen (also mithilfe der Vererbung) werden notwendige Erweiterungen implementiert oder Standardlösungen des Kerns überschrieben.
- Es werden Erweiterungspunkte definiert, wo produktspezifische Teile an die Plattform angedockt werden können.
- Aus einer Reihe vorgefertigter Komponenten werden diejenigen ausgewählt, die zu einem speziellen Produkt gehören. Dieser Schritt wird als (Produkt-) Konfiguration bezeichnet.

Technisch bietet es sich an, Produktlinienarchitekturen durch Rahmenwerke oder durch Kombinationen von Komponenten und Rahmenwerken zu realisieren.

Informationen über erfolgreiche Produktlinienentwicklungen finden sich in der sogenannten »Software Product Line Hall of Fame« (SPLC, o. J.). Die Entwicklung von Produktlinien wird aus organisatorischer und technischer Sicht detailliert z. B. in Pohl, Böckle, van der Linden (2005) vorgestellt.

### 16.6.4 Referenzarchitektur

Eine *Referenzarchitektur* (auch Standard- oder Modellarchitektur genannt) ist gegenüber der Produktlinienarchitektur, die für eine Menge ähnlicher Produkte gilt, weiter abstrahiert. Eine Referenzarchitektur definiert für einen ganzen Anwendungsbereich, d. h. für alle Software-Systeme dieses Bereichs, eine

erprobte und wiederverwendbare Architektur. Es ist offensichtlich, dass eine Referenzarchitektur sehr abstrakt formuliert sein muss und nicht im Kopf eines Forschers entsteht, sondern als Essenz aus den Erfahrungen, die in der Praxis bei einer Reihe ähnlicher Anwendungsentwicklungen gesammelt wurden. In Anlehnung an Beneken (2008) definieren wir:

Eine **Referenzarchitektur** definiert Software-Bausteine für einen Anwendungsbereich durch ihre Strukturen und Typen, ihre Zuständigkeiten und ihre Interaktionen.

Eine bereits seit Langem erprobte Referenzarchitektur existiert für den Compilerbau (siehe Abb. 16–19). Ein wesentlich umfangreicheres Beispiel für eine Referenzarchitektur ist die Versicherungsanwendungsarchitektur der Deutschen Versicherungswirtschaft (GDV, 2001). Sie spezifiziert versicherungsfachliche Prozesse und Funktionsgruppen, z. B. die Partner- und Vertragsverwaltung, sowie fachliche Datentypen in Form von Geschäftsobjekten.

Nach dem Schwerpunkt unterscheidet Beneken (2008) zwischen funktionaler, logischer und technischer Referenzarchitektur.

Eine *funktionale Referenzarchitektur* (auch fachliche Referenzarchitektur genannt) modelliert für einen bestimmten Bereich den Funktionsumfang der Anwendungssysteme durch Funktionsgruppen und deren Datenflussbeziehungen.

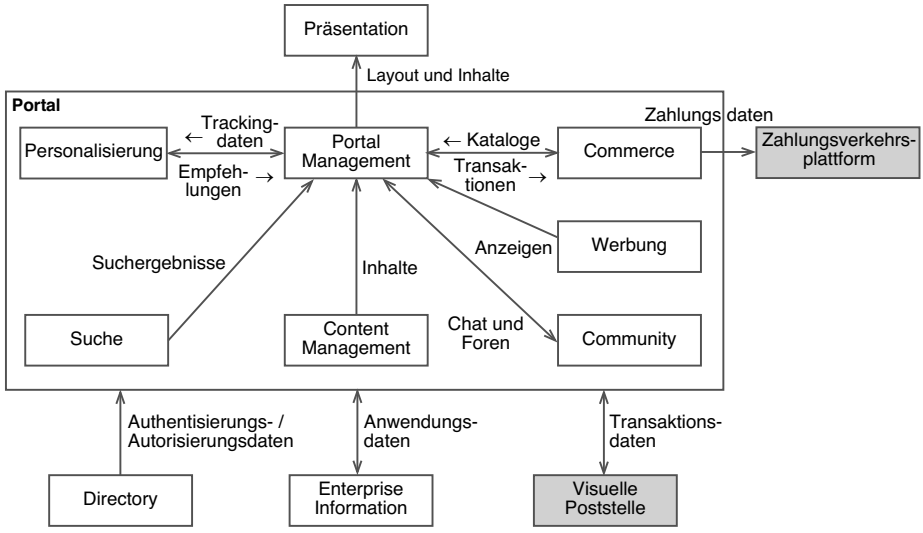



***Abb. 16–32*** *Funktionale Referenzarchitektur für Portal-Software*

Für den Bereich der sogenannten Portal-Software wurde die in Abbildung 16–32 gezeigte Referenzarchitektur vorgeschlagen (Taubner, 2003). Da funktionale Referenzarchitekturen nur die Funktionen modellieren, die allen Anwendungen gemeinsam sind, müssen sie erweiterbar sein. In der Abbildung 16–32 sind bei-

spielhaft behördenspezifische Erweiterungen an dieser Referenzarchitektur grau unterlegt. Wird eine funktionale Referenzarchitektur für eine konkrete Anwendung ausgeprägt, dann kann eine Funktionsgruppe auf mehrere Komponenten aufgeteilt werden. Ebenso kann aber auch eine Komponente mehrere Funktionsgruppen implementieren.

Eine *logische Referenzarchitektur* liegt zwischen der funktionalen und der technischen Referenzarchitektur. Sie definiert für die Anwendungen des betrachteten Bereichs eine Architektur in Form von Software-Bausteinen (beispielsweise Schichten und Komponenten) und deren Beziehungen. Die Schnittstellen der Software-Bausteine sind zwar technisch motiviert, werden aber typischerweise nur informal beschrieben.

Ein Beispiel für eine logische Referenzarchitektur ist die in Abbildung 16–33 dargestellte Quasar-Referenzarchitektur für betriebliche Informationssysteme (Haft, Humm, Siedersleben, 2005).

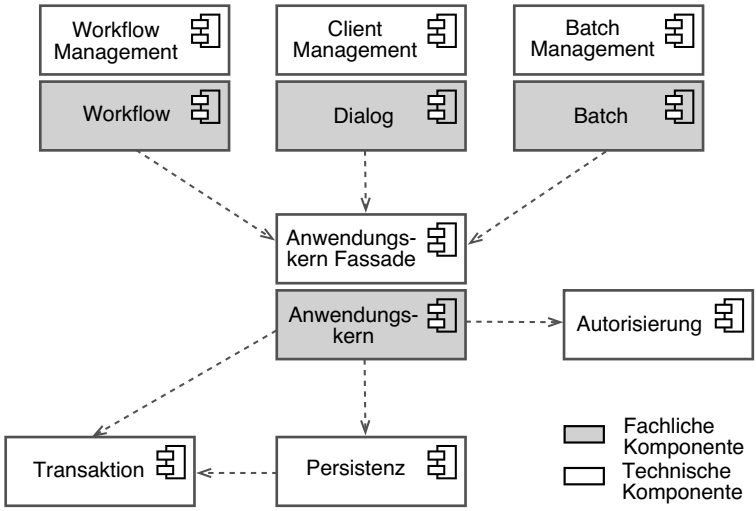



***Abb. 16–33*** *Quasar-Referenzarchitektur für Informationssysteme*

Die zentrale Komponente ist der Anwendungskern, der die fachliche Logik des Informationssystems realisiert. Dieser nutzt über Schnittstellen Komponenten zur Autorisierung und zum Lesen und Speichern der Daten. Der Anwendungskern selbst wird über eine Fassade benutzt (siehe S. 431), typischerweise von Dialog-, Workflow- und Batch-Komponenten. Diese Referenzarchitektur unterscheidet weiterhin zwischen fachlichen und technischen Komponenten und setzt damit das Prinzip »Trennung von Zuständigkeiten« um. Ein weiteres bekanntes Beispiel einer logischen Referenzarchitektur ist das ISO/OSI-Referenzmodell für Kommunikationssysteme (ISO/IEC 7498-1, 1996).

Eine *technische Referenzarchitektur* legt die Implementierungstechnologien (z. B. Sprachen, Bibliotheken, Rahmenwerke, Komponenten, Kommunikationsmechanismen) fest, um logische Referenzarchitekturen zu realisieren. Bekannte Beispiele für technische Referenzarchitekturen sind J2EE und .NET:

- J2EE ist eine von Sun entwickelte Architektur für komponentenbasierte, mehrschichtige Anwendungen.
- .NET fasst eine Reihe von Technologien der Firma Microsoft zusammen, um Webanwendungen und Arbeitsplatzanwendungen für das Windows-Betriebssystem auf einer einheitlichen Plattform zu erstellen.

Bei Produktlinien- und Referenzarchitekturen handelt es sich um generische Software-Architekturen, die zwar konkrete Strukturen vorgeben, aber die Produktarchitektur nicht bis ins Detail festlegen, sodass verschiedene Ausprägungen möglich bleiben. Produktlinien- und Referenzarchitekturen können nur auf Basis großer Erfahrungen in der Software-Entwicklung entstehen. Und selbst wenn solche Erfahrungen verfügbar sind, ist es schwierig, tragfähige und anpassbare Referenzarchitekturen zu entwerfen. Verfügt man jedoch über erprobte Referenzarchitekturen, dann ist es fahrlässig, sie zu ignorieren und eigene, neue Architekturen zu entwickeln.

## 16.7 Der Entwurf von Anwendungssoftware

Ein Großteil der entwickelten Software ist Anwendungssoftware. Züllighoven definiert diese Klasse von Software folgendermaßen:

**Anwendungssoftware** – Software, um fachliche Aufgaben in einem Anwendungsbereich oder mehreren Anwendungsbereichen zu erledigen. Dabei werden vorhandene oder neue Problemlösungsstrategien durch Software realisiert.

Züllighoven (2005)

Anwendungssoftware modelliert immer einen Ausschnitt der realen Welt und orientiert sich an einem Einsatzkontext. Anwendungssoftware ist damit eng an den Anwendungsbereich gebunden.

**Anwendungsbereich** – kann eine Organisation, ein Bereich innerhalb einer Organisation oder ein Arbeitsplatz sein.
Ein Anwendungsbereich ist weit genug gefasst, um die für die Konstruktion von Modellen relevanten fachlichen Zusammenhänge während der Ist-Analyse zu verstehen.
Durch einen Anwendungsbereich ist typischerweise auch eine entsprechende Anwendungsfachsprache festgelegt.

Züllighoven (2005)

Wenn sich der Anwendungsbereich verändert, muss die entsprechende Software ebenfalls angepasst werden, ansonsten wird sie nutzlos. Es kommt auch vor, dass sich ein Anwendungsbereich durch die Einführung von Software verändert. Bei

Anwendungssoftware handelt es sich also immer um E-Programme (siehe Abschnitt 9.3).

Bei der Entwicklung von Anwendungssoftware ist es besonders wichtig, dass die Begriffe der Fachsprache eines Anwendungsbereichs auch in der Architektur und in der Implementierung der Anwendungssoftware verwendet werden, damit die Auswirkungen von Veränderungen im Anwendungsbereich möglichst einfach und genau in der Software lokalisiert werden können.

Im Anwendungsbereich wird die Ist-Analyse durchgeführt und das fachliche Modell entwickelt. Es besteht im Kern aus den fachlichen Abläufen (den Funktionen oder Geschäftsprozessen) und den relevanten Begriffen. Wir haben bereits in Abschnitt 15.3 beschrieben, wie wichtig klar definierte Begriffe und die darauf basierenden Begriffsmodelle sind.

Nun stellt sich die Frage, wie wir diese Abläufe und Begriffe finden, damit wir sie modellieren können. Viele Arbeiten dazu basieren auf dem in Abbott (1983) beschriebenen Verfahren der sprachlichen Analyse, das auch als »Noun-Verb Analysis« bezeichnet wird. Dabei werden die im Anwendungsbereich vorhandenen Dokumente sprachlich untersucht. Substantive können Kandidaten für Begriffe, Adjektive Kandidaten für Eigenschaften der Begriffe und Verben Kandidaten für Abläufe und Prozesse sein. Natürlich reicht es nicht aus, ausschließlich Dokumente sprachlich zu analysieren. Es wird immer auch Fachwissen benötigt, um die Elemente des fachlichen Modells und deren Beziehungen zu definieren.

Nachfolgend stellen wir zwei Ansätze vor, die für den fachlichen Entwurf von Anwendungssoftware genutzt werden können.

### 16.7.1 Der objektorientierte Entwurf

Die zentralen Elemente der objektorientierten Programmierung, also Objekte und Klassen, die mit der ersten objektorientierten Sprache SIMULA 67 (Dahl, 2002) eingeführt wurden, sollten es dem Programmierer erlauben, die Objekte der realen Welt eins zu eins in Software zu repräsentieren.

Das Ziel des objektorientierten Entwurfs besteht darin, Klassen als fachliche Programmbausteine so zu entwerfen, dass sie die relevanten Begriffe des betrachteten Anwendungsbereichs repräsentieren. Technisch gesehen werden die Klassen konsequent nach dem Prinzip des Information Hiding (Abschnitt 16.3.3) konstruiert, d. h., sie verstecken Entwurfsentscheidungen und Interna und bieten an ihrer öffentlichen Schnittstelle einen Satz fachlich motivierter Dienstleistungen an. Publikationen wie beispielsweise das Buch von Booch et al. (2007) unterteilen den objektorientierten Entwurf in vier zentrale Aktivitäten:

a) Identifizieren von Objekten und Klassen
b) Festlegen des Verhaltens der Objekte und Klassen
c) Identifizieren von Beziehungen zwischen den Klassen
d) Festlegen der Schnittstellen zwischen den Klassen

Natürlich genügt es nicht, diese Tätigkeiten einmal sequenziell zu durchlaufen; sie werden so oft wiederholt, bis die Klassenstruktur ausreichend detailliert ist, damit sie als Basis für die Codierung herangezogen werden kann.

Die richtige Wahl der fachlichen Klassen ist der wichtigste Schritt zu einem guten objektorientierten Entwurf. Wir wollen diesen Schritt nachfolgend am Beispiel einer Anwendungssoftware zeigen.

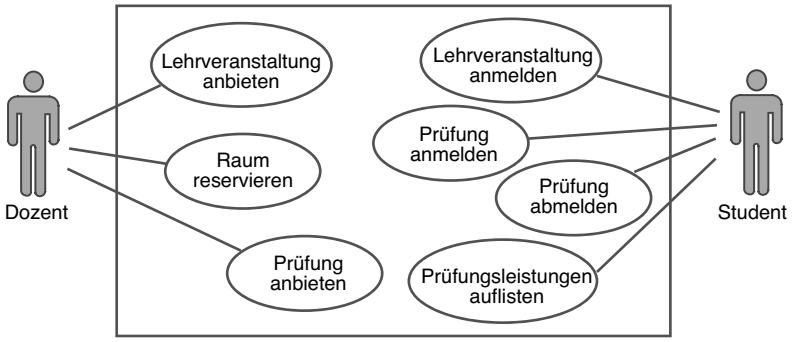



***Abb. 16–34*** *Fachliche Funktionen, modelliert mit Use Cases*

Betrachten wir dazu eine Software zur Verwaltung von Lehrveranstaltungen und Prüfungen an einer fiktiven Hochschule. Ein Ausschnitt der fachlichen Funktionen ist in Abbildung 16–34 in Form eines Use-Case-Diagramms dargestellt. Dozierende können Lehrveranstaltungen und Prüfungen anbieten, sie können Räume für Lehrveranstaltungen oder Prüfungen reservieren. Studierende können sich für Lehrveranstaltungen anmelden, sie können sich für Prüfungen an- oder abmelden und alle ihre Prüfungsleistungen auflisten. Abbildung 16–35 zeigt einen Ausschnitt der relevanten Begriffe und ihrer Beziehungen in Form eines fachlichen Begriffsmodells.

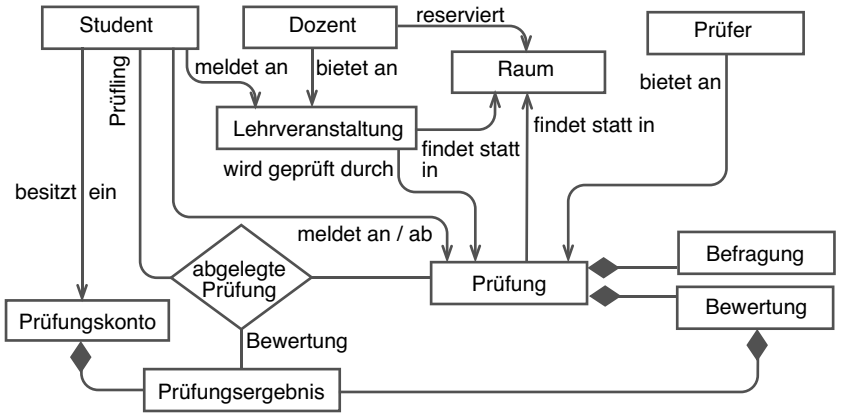



***Abb. 16–35*** *Fachliches Begriffsmodell*

Aus dem fachlichen Begriffsmodell des Anwendungsbereichs können zentrale fachliche Klassen der Architektur abgeleitet werden, beispielsweise die Klassen `Student`, `Prüfung` und `Dozent`.

### 16.7.2 Domain-Driven Design von Anwendungsbereichen

Häufig ist der betrachtete Anwendungsbereich umfangreich und komplex; das fachliche Begriffsmodell ist dann zwangsläufig ebenso umfangreich und komplex. In einem solchen Fall muss der Anwendungsbereich in abgegrenzte Teilbereiche gegliedert werden. Der von Evans (2004) vorgestellte Ansatz »Domain-Driven Design« (DDD) gibt eine Anleitung, wie ein komplexer Anwendungsbereich modularisiert werden kann. Im Folgenden verwenden wir, wie in der Literatur zu DDD üblich, den Begriff *Domäne* als Synonym für Anwendungsbereich und den Begriff *Subdomäne* als Synonym für Anwendungsteilbereich.

DDD unterstützt die Entwickler, Anwendungssoftware fachlich modular zu entwerfen, d. h., für jede fachliche Subdomäne wird eine eigene Anwendung entwickelt, die diese bestmöglich widerspiegelt. Damit das gelingt, müssen Analytiker eng mit den Experten des Anwendungsbereichs zusammenarbeiten, insbesondere bei der Modellierung der Domänen und bei der Festlegung der Subdomänen.

Die Aufteilung einer Domäne in Subdomänen bezeichnet DDD als *strategischen Entwurf*, weil dabei die zentralen Strukturen basierend auf der fachlichen Modularisierung der Domäne festgelegt werden.

Auf der Ebene der Software-Architektur und der Implementierung wird jede identifizierte Subdomäne im besten Fall durch einen sogenannten *begrenzten Kontext* (Bounded Context) realisiert. Es können aber auch mehrere sein, das entscheiden die Architekten. Den detaillierten Entwurf eines Kontextes bezeichnet DDD als *taktischen Entwurf*. Auch dazu bietet das Verfahren Techniken an.

Die nachfolgende Beschreibung der zentralen DDD-Techniken basiert auf den Büchern von Evans (2004) und Vernon (2016), die diesen Ansatz vorstellen.

Eine Subdomäne ist dadurch charakterisiert, dass die darin vorhandenen Begriffe semantisch zusammengehören und dass es Geschäftsprozesse gibt, die darin vollständig durchgeführt werden. Die Begriffe und die Bezeichner der Geschäftsprozesse bilden den Kern der Fachsprache in einer Subdomäne. Diese Sprache wird als *allgegenwärtige Sprache* (Ubiquitous Language) bezeichnet, da sie nicht nur von den Fachexperten, sondern auch auf Ebene der Software-Architektur und der Implementierung verwendet wird. Wichtig ist, dass die Begriffe einer Fachsprache klar und eindeutig definiert sind, um Fehlinterpretationen zu vermeiden. In einer komplexen Domäne gibt es viele Begriffe, die sich eindeutig einer Subdomäne zuordnen lassen. Es gibt aber oft auch Begriffe, die in verschiedenen Subdomänen unterschiedliche Bedeutungen haben; sie bezeichnen nicht dasselbe Konzept. Dementsprechend müssen die Software-Einheiten, die einen solchen Begriff technisch umsetzen, in diesen Subdomänen unterschiedlich implementiert werden.

Um die Subdomänen und ihre Grenzen zu identifizieren, können das fachliche Begriffsmodell und Modelle der Geschäftsprozesse genutzt werden. Das Begriffsmodell entsteht iterativ in Zusammenarbeit zwischen den Fachexperten und Entwicklern. DDD bezeichnet diesen Prozess als Collaborative Modeling. Das sogenannte »Domain Storytelling« bietet Methoden, Sprachen und Werkzeuge, um die Aussagen – die Geschichten (stories) – der Fachexperten über die Domäne aufzunehmen und in einer einfachen Bildsprache aufzubereiten, damit sofort erkannt werden kann, ob die Entwickler die Aussagen richtig verstanden haben. Domain Storytelling wird in Hofer und Schwentner (2021) ausführlich vorgestellt.

Nachfolgend wollen wir am Beispiel einer fiktiven Hochschule zeigen, wie sich ein fachliches Begriffsmodell entwickeln kann.

---

Um das Begriffsmodell zu erstellen, bildet der Software-Hersteller gemeinsam mit der Hochschule ein Modellierungsteam. Vertreten sind darin Fachexperten aus verschiedenen Abteilungen der Hochschulverwaltung, aus den Fachbereichen sowie Analytiker und Architekten des Herstellers.

Das Team trifft sich zu einem ersten Workshop und startet die Modellierung im Bereich der Lehre. Folgende zentrale Begriffe werden identifiziert: *Student*, *Dozent*, *Lehrveranstaltung*, *Raum*, *Prüfung*, *Prüfungskonto* und *Prüfer*. Zusätzlich wird modelliert, dass ein Raum ein Teil eines *Gebäudes* ist. Abbildung 16–36 zeigt diese Begriffe als Version 1 des Begriffsmodells.

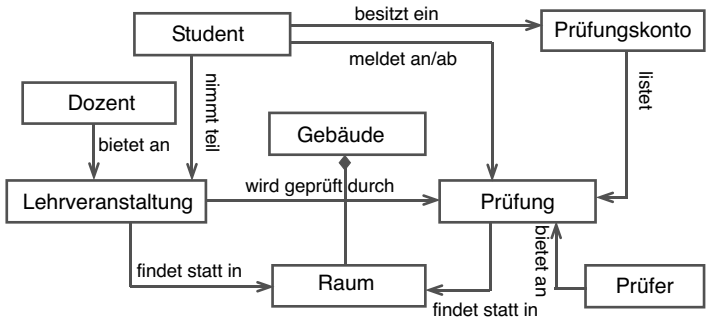



***Abb. 16–36*** *Fachliches Begriffsmodell, Version 1*

Im nächsten Workshop wird das Modell erweitert. Der Begriff *Studiengang* wird eingeführt. Ein Studiengang definiert eine *Studienordnung* und eine *Prüfungsordnung*. Die Lehrveranstaltungen zum Studiengang werden in der Studienordnung beschrieben, alles Relevante zu den Prüfungen steht in der Prüfungsordnung. Zudem wird modelliert, dass Studierende in einem Studiengang *immatrikuliert* sein müssen. Die Version 2 des Modells zeigt Abbildung 16–37.

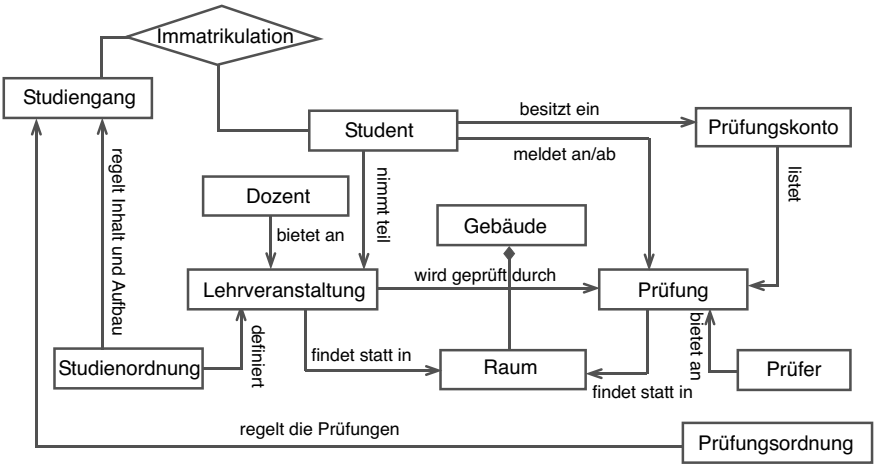



***Abb. 16–37*** *Fachliches Begriffsmodell, Version 2*

Im darauffolgenden Workshop wird modelliert, dass die studierendenbezogenen Informationen vom *Studiensekretariat*, die Prüfungskonten vom *Prüfungsamt* und die Gebäude vom *Gebäudemanagement* verwaltet werden. Um seine Aufgaben zu erledigen, benötigt das Prüfungsamt auch Informationen aus den Studienordnungen und Prüfungsordnungen.

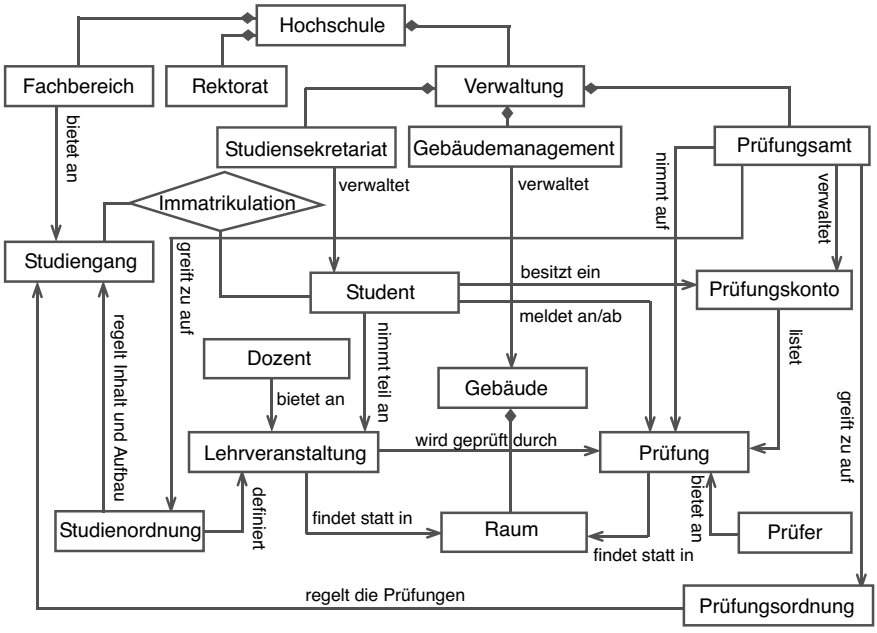



***Abb. 16–38*** *Fachliches Begriffsmodell, Version 3*

Das Prüfungsamt, das Gebäudemanagement und das Studiensekretariat sind Teile der *Verwaltung*. Eine *Hochschule* kennt daneben noch *Fachbereiche*, die Studiengänge anbieten, sowie ein *Rektorat*. Die Version 3 des Begriffsmodells hat die in Abbildung 16–38 gezeigte Struktur.

Natürlich ist dieses Modell noch unvollständig. Es fehlen noch viele Aspekte, die für eine Hochschule relevant sind, wie beispielsweise die Bereiche Finanzen oder Forschungsmanagement. Auch die bereits betrachteten Bereiche sind bei Weitem nicht vollständig modelliert.

Wenn das Begriffsmodell auf diese Weise Stück für Stück komplettiert wird, entsteht letztlich ein sehr großes und komplexes Modell. Nimmt man dieses Modell als Basis für die Entwicklung eines Hochschulinformationssystems, dann entsteht mit hoher Wahrscheinlichkeit ein intern stark gekoppeltes monolithisches System.

Um das zu verhindern, wird eine komplexe Domäne in Subdomänen unterteilt. Dazu folgen wir dem Prinzip der Trennung von Zuständigkeit (siehe Abschnitt 16.3.4). Wir fassen also die Begriffe des Modells in einer Subdomäne zusammen, die zusammen mit den entsprechenden Geschäftsprozessen eine abgeschlossene Zuständigkeit ergeben. Diese Begriffe müssen in der Fachsprache der Subdomäne wichtig und bekannt sein.

In unserem Beispiel legt das Modellierungsteam die folgenden Subdomänen fest: *Studierende*, *Personal*, *Lehrveranstaltungen*, *Prüfungen*, *Studiengänge* und *Gebäude*. Wenn ein Begriffsmodell, wie in Abbildung 16–39 dargestellt, in Subdomänen aufgeteilt wird, kann es Überschneidungen zwischen den Subdomänen geben. So ist der Begriff »Student« in den Subdomänen »Lehrveranstaltungen«, »Prüfungen« und natürlich auch in der Subdomäne »Studierende« relevant. Wenn ein Begriff in mehreren Subdomänen und damit auch in deren Fachsprachen benutzt wird, muss geklärt werden, ob die Konzepte, die mit diesem Begriff bezeichnet werden, auch tatsächlich dieselben sind. Oft haben einzelne Subdomänen eine unterschiedliche Perspektive auf ein scheinbar gemeinsames Konzept, nutzen in der Fachsprache aber die gleichen Begriffe. In unserem Beispiel könnte der Begriff »Student« in den verschiedenen Subdomänen durch passendere Begriffe ersetzt werden, die die unterschiedlichen Konzepte besser repräsentieren. So könnte beispielsweise in der Subdomäne »Lehrveranstaltungen« der Begriff »Teilnehmer«, in der Subdomäne »Prüfungen« der Begriff »Kandidat« gewählt werden. Gleiches gilt für den Begriff »Raum«, der in der Subdomäne »Gebäude« ein anderes Konzept bezeichnet als in den Subdomänen »Prüfungen« und »Lehrveranstaltungen«.

Unser Beispiel zeigt aber auch, dass ein Modell Begriffe enthalten kann, die keiner Subdomäne zugeordnet sind. Entweder sind diese Begriffe nicht relevant für die Entwicklung der Anwendungssoftware, oder es fehlen noch Subdomänen. In unserem Beispiel sind die Begriffe der modellierten Hochschulorganisation

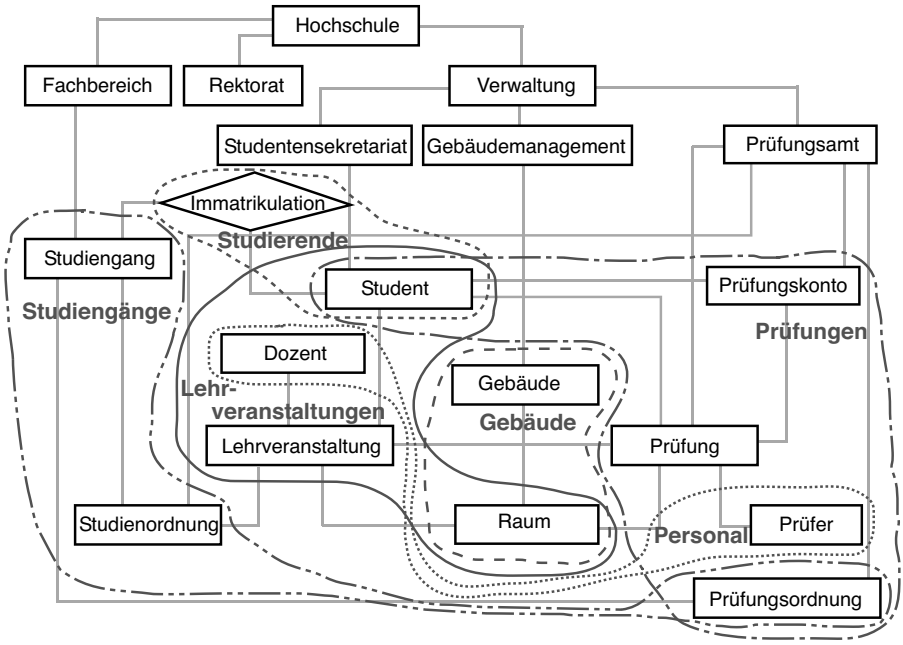



***Abb. 16–39*** *Fachliches Begriffsmodell mit Subdomänen*

nicht relevant. Möglicherweise sind die identifizierten Subdomänen auch noch zu umfangreich und müssten weiter unterteilt werden.

Eine ausführliche Diskussion über die fachlich angemessene Modellierung einer Domäne und des darauf basierenden Zuschnitts der Subdomänen ist enorm wichtig, da sich die getroffenen Entscheidungen direkt auf die Architektur der Anwendungssoftware auswirken.

Natürlich startet die Modellierung der Domäne nicht im luftleeren Raum. In vielen Organisationen, so auch an Hochschulen, gibt es Aufgabenbereiche, die sich in der Struktur der Organisation wiederfinden. Das können beispielsweise Abteilungen, Geschäftsbereiche oder Business Units sein. Diese Strukturen dienen als Ausgangsbasis für die Modellierung der Domäne und Subdomänen.

Ein Beispiel für die Modellierung einer komplexen Domäne, der Domäne »Öffentlicher Verkehr«, findet man in Scholz (2012).

### 16.7.3 Die Gestaltung der Schnittstellen

Technisch werden die identifizierten Subdomänen durch Kontexte realisiert. Da diese Daten austauschen und Dienste nutzen, muss entschieden werden, wie die Schnittstellen zwischen den Kontexten zu gestalten sind. Abbildung 16–40 zeigt die Kontexte unserer Beispieldomäne, die in diesem Fall den Subdomänen entsprechen, und ihre fachlichen Beziehungen.

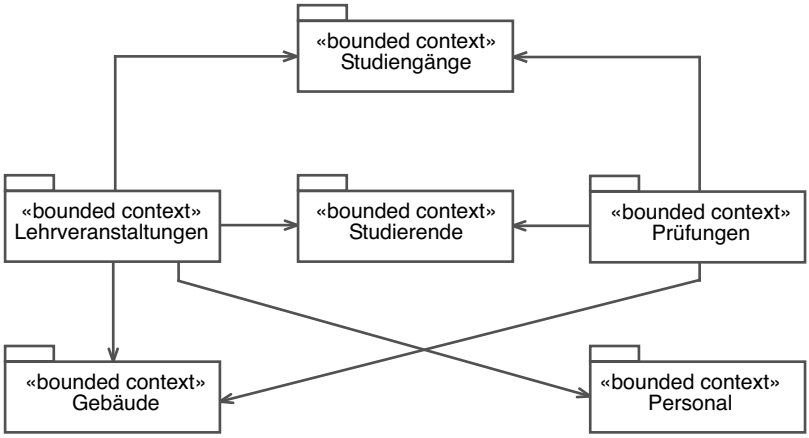



***Abb. 16–40*** *Kontexte und ihre Beziehungen (UML-Paket-Diagramm)*

DDD unterstützt den Entwurf der Schnittstellen, indem sieben Muster für die Kommunikation zwischen Kontexten angegeben werden. Da es sich bei den meisten dieser Schnittstellenmuster um eine Kunden-Anbieter-Beziehung handelt, bezeichnen wir im Folgenden einen Kontext, der Dienste an der Schnittstelle zur Verfügung stellt, als »Anbieter«, einen Kontext, der Dienste verwendet, als »Kunde«. Die folgenden Schnittstellenmuster schlägt DDD vor:

- *Geteilter Kern (Shared Kernel)*: Wenn sich zwei oder mehrere Kontexte Begriffe teilen, dann kann dieser überlappende Teil des Modells gemeinsam entwickelt und genutzt werden. Das kann den Aufwand für die Implementierung reduzieren. Der Preis dafür ist, dass die Kontexte dadurch extrem stark gekoppelt werden. Wenn jeder Kontext von einem eigenen Team entwickelt wird, dann müssen sich diese Teams entsprechend intensiv abstimmen. Das kann problematisch und manchmal auch organisatorisch nur schwer möglich sein. Dieses Muster sollte also nur in sehr gut begründeten Fällen gewählt werden.
- *Kunde/Anbieter (Customer/Supplier)*: Der Anbieter entwickelt das eigene Modell und stellt seinen Kunden die benötigten Dienste und Informationen zur Verfügung. Die Kunden sind vom Anbieter abhängig, sie geben ihre Anforderungen an die Services der Schnittstelle dem Anbieter bekannt.
- *Veröffentlichte Sprache (Published Language)*: Eine solche Sprache ist in vielen Fällen ein wohl dokumentiertes und standardisiertes Format, um Daten auszutauschen. Dieses Format muss allen, die entsprechende Daten erzeugen bzw. konsumieren, bekannt sein. Beispiele für dieses Muster sind das iCal-Format für Kalenderdaten und das EDIFACT-Format zum Austausch von Daten im Geschäftsverkehr.
- *Antikorruptionsschicht (Anticorruption Layer)*: Wie der Name nahelegt, schützt dieses Muster den Kunden davor, dass sein fachliches Modell mit Tei-

len eines Anbietermodells vermischt wird. Dazu entwickelt der Kunde an seiner Schnittstelle eine Art Isolationskomponente, die bei Änderungen der vom Anbieter zur Verfügung gestellten Schnittstelle angepasst werden muss. Dieses Muster kann insbesondere verwendet werden, um vorhandene Fremd- oder Altsysteme lose zu integrieren.

- *Offen angebotener Dienst (Open-Host-Service)*: Dieses Muster sieht vor, dass der Anbieter die Dienste an seiner Schnittstelle auf Basis der Kundenwünsche in einer eigenen Komponente zur Verfügung stellt. Diese Komponente wandelt die eigene interne Modellrepräsentation in für die Kunden einfach zu nutzende Dienste um. Für die Kunden wird oft zusätzlich eine angepasste veröffentlichte Sprache für den Datenaustausch angeboten.
- *Mitläufer (Conformist)*: Bei diesem Muster übernimmt der Kunde die vom Anbieter angebotene Schnittstelle ohne Änderungen und integriert sie tief in die Anwendung. Das kann dann notwendig sein, wenn eine sehr umfangreiche Schnittstelle verwendet werden soll oder muss, beispielsweise die Amazon-Partner-Schnittstelle.
- *Getrennte Wege (Separate Ways):* Wenn zwei Kontexte keine Informationen austauschen oder Dienste voneinander nutzen müssen, ist auch keine Schnittstelle zwischen ihnen erforderlich. Beide können unabhängig voneinander entwickelt werden.

Die Entscheidungen für die konkrete Gestaltung der Schnittstellen sollten explizit getroffen werden; die finanziellen und organisatorischen Randbedingungen und Konsequenzen müssen in die Entscheidung einfließen. Wenn beispielsweise zwei Kontexte überschneidende Begriffe enthalten, aber nicht zusammen entwickelt werden sollen, wird in Kauf genommen, dass derselbe Begriff aus verschiedenen Perspektiven zweimal entwickelt werden muss.

Wir wollen einige dieser Schnittstellenmuster auf unser Beispiel anwenden. Dazu nehmen wir an, dass die Hochschule bereits ein Gebäudemanagementsystem und eine Software zur Verwaltung der Mitarbeiter einsetzt oder diese als Standard-Software einkaufen will. Da diese Fremdsysteme integriert werden müssen, entscheiden wir, dass die Kunden ausschließlich über Schnittstellen der Art »Antikorruptionsschicht« (*anti corruption layer,* ACL) Informationen und Dienste mit diesen Systemen austauschen.

Da der Kontext »Studierende« zentral ist, sollen dessen Dienste ausschließlich über Schnittstellen des Typs »Offen angebotener Dienst« (OHS) den Kunden zur Verfügung gestellt werden. Damit studierendenbezogene Daten in einem standardisierten Format ausgetauscht werden können, wird zusätzlich eine entsprechende »Veröffentlichte Sprache« (PL) von der Schnittstelle unterstützt. Gleiches gilt für die Dienste und Informationen des Kontexts »Studiengänge«. Abbildung 16–41 zeigt die Kontexte und die gewählten Schnittstellenmuster.

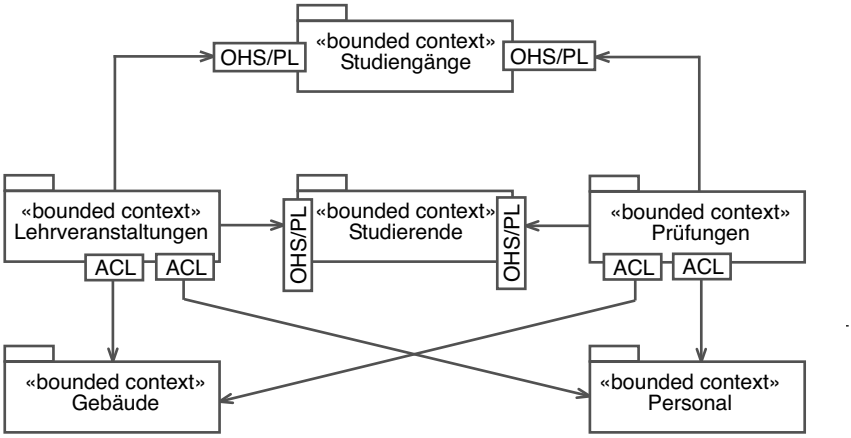



***Abb. 16–41*** *Kontexte mit Angabe der Schnittstellenmuster*

### 16.7.4 Der technische Entwurf

Als Ergebnis des strategischen Entwurfs liegen die Kontexte und ihre Schnittstellen vor, wobei jeder Kontext sein eigenes Begriffsmodell realisiert. DDD unterstützt den Entwurf der Kontexte (den sogenannten taktischen Entwurf), indem eine standardisierte technische Schichtenarchitektur (siehe Abschnitt 16.4.1), bestehend aus Präsentations-, Anwendungs-, Domänen- und Infrastrukturschicht, vorgegeben wird. Zusätzlich werden die folgenden Muster für den Entwurf der Software-Einheiten der Anwendungs- und Domänenschicht angeboten:

- *Wert-Objekt (Value Object)*: Wert-Objekte werden verwendet, um den Zustand anderer Objekte zu beschreiben; sie haben keine eigene Identität. Die Bewertungen von Prüfungen (z.B. 1,0 oder 3,7) sind Beispiele solcher Wert-Objekte. Dieses Muster ist nicht neu, es wurde von Bäumer et al. (1998) vorgestellt.
- *Entität (Entity)*: Entitäten sind die grundlegenden Bausteine der Domänenschicht. Sie haben einen Zustand, der u.a. durch Wert-Objekte beschrieben wird, und einen definierten Lebenszyklus. Da Entitäten die Begriffe des Modells repräsentieren, müssen sie in der Regel gespeichert und wiederhergestellt werden können. Eine Prüfung ist ein Beispiel für eine Entität, die unter anderem ein Wert-Objekt für die erzielte Bewertung besitzt.
- *Aggregat (Aggregate)*: Ein Aggregat gruppiert zusammengehörende Entitäten und Wert-Objekte. Es gibt immer eine sogenannte Wurzel-Entität, über die alle Bestandteile des Aggregats erreicht werden können. Aggregate werden als Einheit konsistent verändert. Ein Prüfungskonto kann als Aggregat realisiert werden; es enthält alle Prüfungen einer Studentin oder eines Studenten.

- *Fabrik (Factory):* Fabriken erzeugen die benötigten Entitäten, Aggregate und auch Wert-Objekte. Fabriken können als eigene Bausteine entworfen werden oder auch durch Fabrik-Methoden realisiert werden.
- *Domänendienst (Domain Service)*: Ein solcher Dienst repräsentiert eine fachliche Funktionalität, die über die Funktionen einzelner Entitäten hinausgehen. Domänendienste sind zustandslos, sie verändern aber Aggregate und deren Entitäten. Ein Beispiel für einen Domänendienst ist die Konversion von Bewertungen in das angloamerikanische System.
- *Domänenereignis* (*Domain Event*): Domainenereignisse unterstützen die Realisierung fachlicher Prozesse. Für jedes fachliche Ereignis, das zu Änderungen von einem Aggregat oder mehreren Aggregaten führen muss, wird ein entsprechendes Domainenereignis definiert. Wenn das Ereignis eintritt, können darauf beispielsweise Domainendienste entsprechend reagieren. Ereignisse werden nicht nur innerhalb eines Kontextes, sondern auch anderen Kontexten gemeldet. Diese werden dadurch lose gekoppelt. Ein Beispiel für ein Domänenereignis ist eine Exmatrikulation. Als Reaktion darauf muss u. a. das Prüfungskonto des entsprechenden Studierenden gesperrt werden.
- *Repository*: Ein Repository kapselt technische Details von Bausteinen der Infrastrukturschicht in der Domänenschicht, damit diese Schichten möglichst lose gekoppelt sind. Oft kapselt ein Repository eine Datenbank, die die Aggregate und Entitäten der Domänenschicht speichert.
- *Anwendungsdienst (Application Service)*: Ein Anwendungsdienst entspricht in etwa einem fachlichen Use Case. Ein solcher Dienst kapselt Elemente der Domänenschicht (z. B. Domänendienste und Repositorys) und bietet eine Schnittstelle an, über die die fachliche Funktion aufgerufen werden kann. Diese Aufrufe befinden sich typischerweise in der Präsentationsschicht. Beispiele für Anwendungsdienste sind das Eintragen einer Bewertung für eine Prüfung oder die Übernahme von Prüfungsbewertungen, die an einer anderen Hochschule erzielt wurden.

Abbildung 16–42 zeigt beispielhaft die DDD-Schichtenarchitektur und die Anwendung der beschriebenen Muster für den Kontext »Prüfungen«.

Da ein Aggregat nicht explizit als Klasse modelliert und implementiert wird, übernimmt die Klasse, die die Wurzel-Entität realisiert, diese Rolle. In der Abbildung ist das die Klasse »Prüfungsordner«, die Bestandteile des Aggregats sind farblich unterlegt.

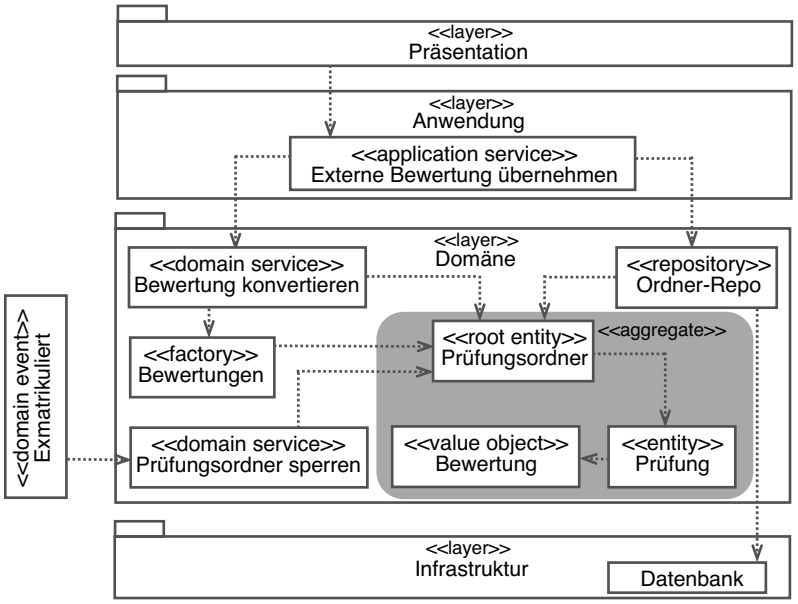



***Abb. 16–42*** *Beispiel einer DDD -Schichtenarchitektur*

Carola Lilienthal (2020) beschreibt einen Ansatz, bei dem die Kontexte durch Microservices (siehe Abschnitt 16.4.6) realisiert werden. Die einzelnen Microservices können nach der DDD-Schichtenarchitektur aufgebaut sein oder auch Komponenten enthalten, die dem Port-Adapter-Architekturmuster (siehe Abschnitt 16.4.2) entsprechen.

### 16.7.5 Bewertung

Der fachliche Entwurf von Anwendungssoftware ist eine kreative und nicht triviale Tätigkeit. Wenn wir objektorientiert entwerfen, können wir fachliche Klassen identifizieren, die die Begriffe des Anwendungsbereichs auf der Ebene der Architektur repräsentieren.

Domain-Driven Design geht darüber hinaus. DDD unterstützt die Modellierung von Domänen, Subdomänen und abgegrenzten Kontexten und gibt Anleitung für die Gestaltung der Schnittstellen auf der Ebene der Systemarchitektur durch die Techniken des strategischen Entwurfs. Mithilfe der Techniken und Muster des taktischen Entwurfs wird jeder Kontext nach einem einheitlichen Konstruktionsschema entworfen, mit einer klaren Verteilung der Verantwortlichkeiten auf die Elemente der Architektur. Das erleichtert das Verständnis und damit auch die Wartung und Weiterentwicklung.

DDD ist jedoch kein Allheilmittel und kann auch nicht immer und überall eingesetzt werden. Die publizierten Erfahrungen bei der Entwicklung komplexer

Anwendungssysteme zeigen, dass DDD eine gute Unterstützung bietet, Anwendungssoftware auf Basis der Fachlichkeit angemessen zu modularisieren.

DDD ist sehr gut dokumentiert, die Bücher von Evans (2004) und Vernon (2016) stellen DDD detailliert vor.

## 16.8 Die Qualität der Architektur

Bei Architekturen unterscheiden wir zwischen der *Soll-Architektur* und der *Ist-Architektur* eines Systems. Die Soll-Architektur ist der Bauplan, der für die Konstruktion verwendet wird. Die am Ende der Entwicklung implementierte Architektur eines Systems wird als Ist-Architektur bezeichnet. Im besten Fall entspricht die Ist-Architektur der Soll-Architektur. In der Praxis ist dies aber selten der Fall, da bei der Konstruktion frühere Architekturentscheidungen revidiert werden. Beispielsweise werden Architekturelemente noch einmal unterteilt oder Beziehungen zwischen Elementen etabliert, die nicht geplant waren. Sowohl die Soll- als auch die Ist-Architektur müssen geprüft und bewertet werden.

### 16.8.1 Qualitätskriterien für Architekturen

Für den Architekturentwurf sind die nichtfunktionalen Anforderungen besonders wichtig (Abschnitt 15.4.4), da sie die Freiheit der Konstruktion einschränken. Das gilt vor allem für die Anforderungen an die folgenden Qualitäten:

- *Testbarkeit*
  Jede Software muss getestet werden. Wird dies beim Architekturentwurf nicht berücksichtigt, kann man nicht erwarten, dass sich das implementierte System leicht testen lässt. In jedem Fall wird der Test durch eine Architektur, die hohe Lokalität schafft, wesentlich erleichtert, also durch eine Gliederung in abgeschlossene, einzeln testbare Einheiten. Auch spezielle Testschnittstellen tragen zur höheren Testbarkeit bei.
- *Wartbarkeit, Erweiterbarkeit*
  Systeme, die eingesetzt werden, müssen dann und wann korrigiert, vor allem aber immer wieder den sich ändernden Anforderungen angepasst werden. Auch wenn man beim Architekturentwurf die Zukunft nicht genau kennt, kann man doch Bereiche des Systems, die sehr wahrscheinlich angepasst oder erweitert werden müssen, so entwerfen, dass der Aufwand dafür akzeptabel ist.
- *Portierbarkeit*
  Systeme, die viele Jahre eingesetzt werden, müssen immer wieder auf andere Plattformen (Betriebssysteme) übertragen werden oder Änderungen der umgebenden Software (z. B. Datenbanksysteme, Middleware) angepasst werden. Solch eine Anpassung wird als Portierung bezeichnet. Die Portierung wird wesentlich erleichtert, wenn der Architekt dafür sorgt, dass die Abhängigkeiten von vorgegebener Software an wenigen Stellen konzentriert sind.

### 16.8.2 Prüfung einer Soll-Architektur

Die Qualität der Soll-Architektur bestimmt maßgeblich die Qualität des entwickelten Systems, und zwar dauerhaft. Darum ist es wichtig, die Qualität der Soll-Architektur zu prüfen. Da die Qualität nicht absolut definiert ist, sondern nur relativ zu den Anforderungen, müssen wir wissen, welche der Anforderungen durch die Architektur umgesetzt werden müssen.

Die Soll-Architektur wird typischerweise auf Basis der von den Architekten erstellten Architekturbeschreibung und der darin enthaltenen Architekturmodelle geprüft. Die Soll-Architektur ist gut, wenn sie dazu beiträgt, dass das realisierte System die daran gestellten Anforderungen erfüllt. Jede Bewertung oder Prüfung muss sich an diesem Ziel orientieren. Meist gibt es mehrere mögliche Architekturen; dann müssen diese gegenübergestellt und verglichen werden.

Ist die Soll-Architektur (hier insbesondere die statische Struktur) erst einmal gewählt und in Code gegossen, dann lässt sie sich nur noch mit sehr hohem Aufwand verändern. Darum sollte die Entscheidung für eine bestimmte Architektur gründlich überprüft und dann erst umgesetzt werden.

Die dynamische Sicht, die Funktion, liefert die Anforderungen an die Statik. Trotzdem kann die Funktion später meist relativ leicht verändert werden. In Amerika sieht man oft Kirchen, die als Restaurants oder Schuhgeschäfte dienen, bei uns werden viele alte Industriebauten als Museen oder Theatersäle genutzt, und manches Schloss beherbergt eine Universität. Es ist das Kennzeichen einer guten Soll-Architektur, dass sie solche Änderungen nicht grundsätzlich ausschließt.

Die systematische Prüfung einer Soll-Architektur erfolgt am besten durch ein Review (siehe Abschnitt 13.5). Dazu müssen natürlich die Anforderungen und Bewertungskriterien vorher festgelegt sein. Als Gutachter eines Architektur-Reviews kommen nur Fachleute infrage, die über das notwendige Wissen und über ausreichende Entwurfserfahrung verfügen.

Der Architekturentwurf und seine Bewertung kann durch Prototypen unterstützt werden, die in diesem Fall nicht dazu dienen, die Bedienschnittstelle festzulegen. Vielmehr wird man sich auf unklare und riskante Bereiche konzentrieren. Mit Prototypen können insbesondere Anforderungen an Leistungsaspekte, Skalierbarkeit und Lastverteilung untersucht und bewertet werden.

### 16.8.3 Prüfung einer Ist-Architektur

Um die Ist-Architektur bewerten zu können, muss sie zuerst aus der Implementierung des Systems extrahiert werden; es handelt sich dabei also um eine typische Reverse-Engineering-Aktivität. Dazu werden zuerst aus dem Code die vorhandenen Architekturelemente wie Pakete oder Klassen und ihre Beziehungen extrahiert. Dies kann durch statische Analyse der Code-Basis (siehe z. B. Mendonça, Kramer, 2001), aber auch durch Analyse von Informationen, die zur Laufzeit

protokolliert werden, geschehen (siehe z. B. Nicolaescu, Lichter, Hoffmann, 2017). Natürlich benötigt man dafür entsprechende Werkzeuge.

Bei der Prüfung der Ist-Architektur können auf Basis der rekonstruierten Architekturinformationen zum einen wichtige allgemeine Architekturqualitäten bewertet werden (siehe Abschnitt 16.8.1). Zum anderen, kann geprüft werden, ob die Ist-Architektur noch der Soll-Architektur und dem eventuell gewählten Architekturmuster entspricht.

Beim Übergang von der Soll- zur Ist-Architektur gehen leider häufig wichtige strukturbildende Informationen verloren. So kann beispielsweise nicht mehr aus den Bezeichnern von Code-Einheiten abgeleitet werden, in welches Subsystem oder in welche Schicht die Code-Einheiten gehören. Deshalb müssen die aus dem Code extrahierten Elemente und Beziehungen auf Basis der Soll-Architekturbeschreibung (sofern es eine solche gibt) und insbesondere mit dem Wissen der Architekten zu einem Modell der Ist-Architektur zusammengesetzt werden. Dies ist eine notwendige, aber leider nicht automatisierbare Tätigkeit, um die Konformität zwischen der Ist- und der Soll-Architektur zu prüfen.

Carola Lilienthal (2020) schlägt einen Ansatz für die Prüfung von Software-Architekturen vor, der beide Prüfaspekte umfasst. Folgende Qualitäten stehen dabei im Mittelpunkt:

- *Modularität*: Die Architektur soll so modularisiert sein, dass die einzelnen Einheiten in sich geschlossen sind und eine eigenständige Aufgabe übernehmen (also einen hohen Zusammenhalt haben und nach dem Prinzip »Trennung von Zuständigkeiten« entworfen wurden). Die Modularisierung soll zudem dazu führen, dass alle Architekturelemente ähnlich umfangreich sind und dass die Elemente nicht zu stark gekoppelt sind. Letztlich sollen die Schnittstellen der Einheiten explizit und angemessen definiert sein.
- *Musterkonsistenz*: Wurde die Soll-Architektur nach einem Architekturmuster entworfen, dann muss die Ist-Architektur die dafür bekannten Prinzipien und Regeln einhalten.
- *Zyklenfreiheit*: Eine Architektur ist zyklenfrei, wenn die Beziehungen zwischen den Architekturelementen auf allen Ebenen ohne Rückbeziehungen auskommen.

Dass zyklische Beziehungen zwischen Architekturelementen viele wichtige Qualitätseigenschaften negativ beeinflussen, haben schon Dijkstra (1968a) und Parnas (1979) erkannt. Wenn Architekturelemente in einer zyklischen Beziehung stehen, dann müssen diese im Rahmen der Wartung und Weiterentwicklung häufiger geändert werden und sind dementsprechend fehleranfälliger. Umgekehrt sind Systeme, deren Architekturelemente zyklenfrei miteinander verbunden sind, leichter verständlich, einfacher erweiterbar und testbar. Es ist darum sinnvoll, Zyklen in Architekturen zu verbieten. Bei der Bewertung einer Architektur sollte ausdrücklich auf Zyklenfreiheit geprüft werden.

Um die Modularität zu bewerten, können auch Kennzahlen ermittelt und analysiert werden. Besonders wichtig sind Kennzahlen, die den Zusammenhalt der Architekturelemente und die Kopplung zwischen den Elementen zählen oder bewerten. Dazu gibt es in der Literatur viele Metriken. Koziolek (2011) enthält eine gute Übersicht über Metriken, die für die Bewertung von Ist-Architekturen vorgeschlagen wurden. Aber auch einfache Größenmaße wie die Anzahl der Klassen in einem Paket oder die Anzahl der angebotenen Schnittstellenoperationen unterstützen diese Bewertung.

Wie schon erwähnt, braucht man spezielle Werkzeuge, um Ist-Architekturen zu prüfen. Zum einen müssen diese aus der Code-Basis die entsprechenden Architekturelemente und deren Beziehungen extrahieren. Zum anderen müssen sie Metriken berechnen, die für die Bewertung der Ist-Architektur relevant sind. Will man die Zyklenfreiheit untersuchen, dann muss das Werkzeug Zyklen im Abhängigkeitsgraph erkennen und dokumentieren. Um die Konformität zwischen der Ist- und der Soll-Architektur zu prüfen und um ggf. Verletzungen feststellen zu können, muss dem Werkzeug ein Modell der Soll-Architektur vorliegen, um es mit der rekonstruierten Ist-Architektur vergleichen zu können.

Die Vorgehensweise bei der Prüfung von Ist-Architekturen kann folgenderweise visualisiert werden:

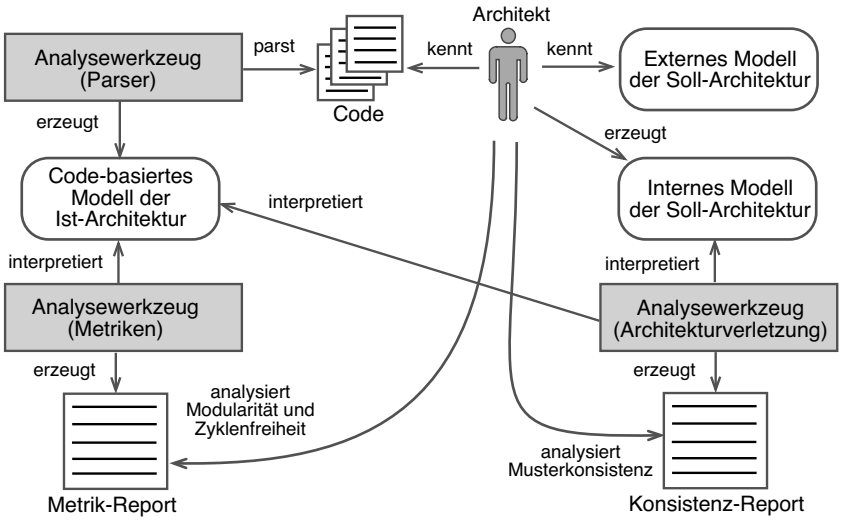



***Abb. 16–43*** *Vorgehensweise beim Prüfen einer Ist-Architektur*

Verwendet man ein Werkzeug, das kein Modell der Soll-Architektur nutzen kann, dann können nur die Modularität und die Zyklenfreiheit untersucht werden. Um die Musterkonsistenz zu prüfen, muss der Code auf die Architekturelemente der Soll-Architektur abgebildet werden. Das besorgen Architekt und Entwickler.

### 16.8.4 Ein Beispiel für die Bewertung von Ist-Architekturen

Als Beispiel verwenden wir eine einfache strenge Schichtenarchitektur mit den Schichten A, B, C und D, die den üblichen Regeln für strenge Schichtenarchitekturen entspricht. Die Ist-Architektur von Release 1 des Systems wird konform zur Soll-Architektur implementiert. Um die Ist-Architektur zu rekonstruieren und deren Qualität zu bewerten, verwenden wir das Werkzeug SONARGRAPH-ARCHITECT[8]. Abbildung 16–44 zeigt links das Modell der Soll-Architektur und rechts das vom Werkzeug erzeugte Diagramm der rekonstruierten Ist-Architektur.

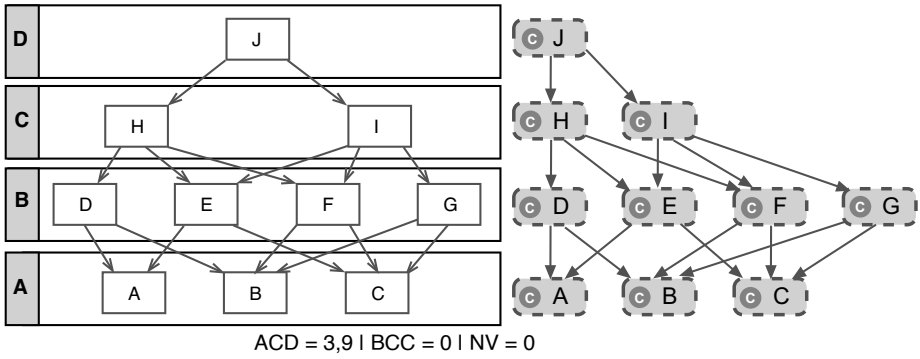



***Abb. 16–44*** *Soll-Architektur und rekonstruierte Ist- Architektur (Release 1)*

Aus der Vielzahl der von SONARGRAPH-ARCHITECT angebotenen Metriken wählen wir die folgenden aus:

- *Average Component Dependency* (ACD; Lakos, 1996): Die durchschnittliche Anzahl der Komponenten, von denen eine Komponente direkt und indirekt abhängig ist.
- *Biggest Component Cycle Group* (BCC): Anzahl der Komponenten im größten Zyklus.
- *Number of Violations* (NV): Anzahl der Abhängigkeiten zwischen Komponenten, die gegen die Soll-Architektur verstoßen.

Abbildung 16–44 zeigt ebenfalls die gemessenen Metrikwerte für Release 1. Um den ACD-Wert einordnen zu können, bestimmen wir für diese Architektur zwei Referenzwerte:

a) Den ACD-Wert, wenn alle Komponenten einer Schicht alle Komponenten der darunterliegenden Schicht verwenden; dieser Wert ist 5,5.
b) Den ACD-Wert, wenn alle Komponenten alle anderen Komponenten verwenden; dieser Wert ist 10.

8 SONARGRAPH-ARCHITECT ist ein eingetragenes Warenzeichen der Firma hello2morrow.

Der gemessene ACD-Wert = 3,9 ist kleiner als der Referenzwert (a) und demnach akzeptabel; Zyklen werden keine erkannt.

Um Architekturverletzungen erkennen zu können, muss dem Werkzeug ein explizites Modell der Soll-Architektur vorliegen. Dazu stellt SONARGAPH-ARCHITECT eine ADL zur Verfügung, mit der spezifiziert werden kann, welche Code-Einheiten zu einem Architekturelement gehören und welche Abhängigkeiten zwischen Architekturelementen erlaubt sind. Da in unserem Beispiel alle Code-Einheiten einer Schicht in einem eigenen Verzeichnis liegen, kann man dies sehr einfach in der ADL formulieren. Wir definieren weiterhin, dass jede Schicht nur von der direkt darunterliegenden abhängig sein darf.

Dies sieht am Beispiel der Schicht D folgendermaßen aus.

```
artifact SchichtD {
    include "*/levelD/**" // enthält alle Komponenten dieses Verzeichnisses
    connect to SchichtC   // darf nur Komponenten dieser Schicht benutzen
}
```

Im Zuge von Wartungsarbeiten und der Integration neuer Anforderungen für das Release 2 werden zwei zusätzliche Abhängigkeiten implementiert (J benutzt D, I benutzt C). Die rekonstruierte Ist-Architektur von Release 2 und die gemessenen Metrikwerte zeigt Abbildung 16–45.

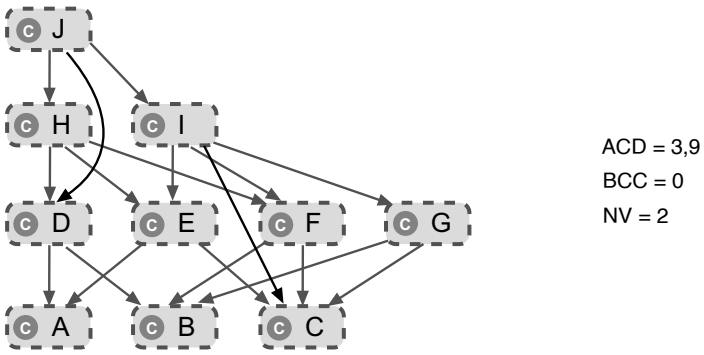



***Abb. 16–45*** *Ist-Architektur (Release 2) mit Architekturverletzungen*

Der ACD-Wert bleibt unverändert, weil die neuen Abhängigkeiten den Kopplungswert der Komponenten I und J nicht verändern, da J schon indirekt von D, und I von C abhängig ist. Wie man erkennt, ist die Ist-Architektur von Release 2 jedoch nicht mehr konform zur Soll-Architektur. Das Werkzeug erkennt zwei Architekturverletzungen (J aus Schicht D benutzt D aus Schicht B, I aus Schicht C benutzt C aus Schicht A) und visualisiert diese entsprechend. Das ist aber nur möglich, weil das Werkzeug ein explizites Modell der Soll-Architektur kennt.

Die rekonstruierte Ist-Architektur von Release 3 ist in Abbildung 16–46 dargestellt. Es wurden weitere Abhängigkeiten zwischen den Komponenten implementiert (C benutzt I, C benutzt D, D benutzt C), die zu Zyklen führen und auch die Soll-Architektur verletzen. Die neuen Abhängigkeiten und die dadurch entstandenen Zyklen haben eine massive Auswirkung auf die durchschnittliche Kopplung (ACD), die jetzt über dem Maximalwert liegt (unser Referenzwert a), wenn keine Zyklen existieren. Der größte Zyklus enthält sechs Komponenten; das sind 60 % der Komponenten dieser Architektur! Zudem wurden zwei weitere Architekturverletzungen entdeckt (C aus Schicht A benutzt D aus Schicht B und auch I aus Schicht C).

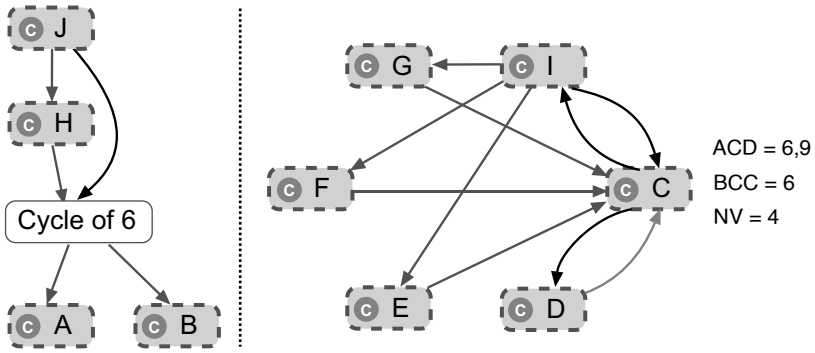



***Abb. 16–46*** *Ist-Architektur (Release 3) und Zyklus-Diagramm*

Fazit: Fast immer entfernt sich die Ist-Architektur während der Wartung und Weiterentwicklung von der definierten Soll-Architektur. In unserem Beispiel kann bei Release 3 von einer Schichtenarchitektur keine Rede mehr sein. Nur mithilfe eines expliziten Soll-Modells ist ein Werkzeug in der Lage, Abweichungen der Ist- von der Soll-Architektur zu entdecken. Ansonsten müssen Architekten auf Basis der erhobenen Metrikwerte und eines Code-Reviews Architekturverletzungen erkennen. Rekonstruierte Architektur-Diagramme wie in unserem einfachen Beispiel stehen ihnen in der Regel nicht zur Verfügung.

Natürlich kann es immer notwendig werden, die Soll-Architektur während der Lebenszeit eines Systems zu verändern. In diesem Fall wird diese, nicht die ursprüngliche Soll-Architektur, für die Bewertung der Ist-Architektur herangezogen.

Nur wenn Architekturverletzungen frühzeitig erkannt werden, kann durch entsprechende Refactoring-Maßnahmen sichergestellt werden, dass die Ist-Architektur auch über viele Releases der Soll-Architektur entspricht. Und nur so können Technische Schulden (siehe Abschnitt 22.5), die die Architektur betreffen, verhindert werden. Deshalb sollten Workshops, in denen die Ist-Architektur durch Architekten und Entwickler analysiert und bewertet wird, regelmäßig durchgeführt werden. Analysewerkzeuge liefern dazu wichtige und objektive

Informationen. Wenn ein Software-Hersteller einen automatisierten Bereitstellungs- oder Auslieferungsprozess nutzt, dann sollte ein solches Analysewerkzeug Teil der dazu erstellten Pipeline sein (siehe Abschnitt 20.5). Wenn dieses nicht akzeptable Ergebnisse liefert, wird die Pipeline abgebrochen. Die Architekten und Entwickler können dann die Ursachen für die schlechten Prüfungsergebnisse identifizieren und die Architektur verbessern.

Eine ausführliche Beschreibung der Prüfung von Ist-Architekturen ist in Lilienthal (2020) zu finden.