FACADE

OBSERVER

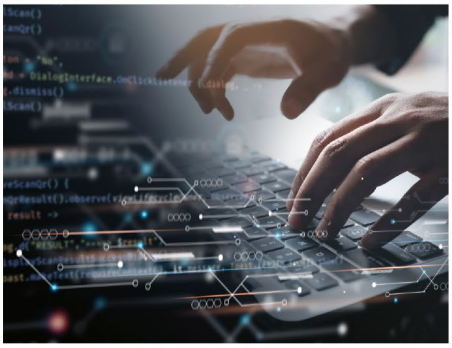
FACTORY

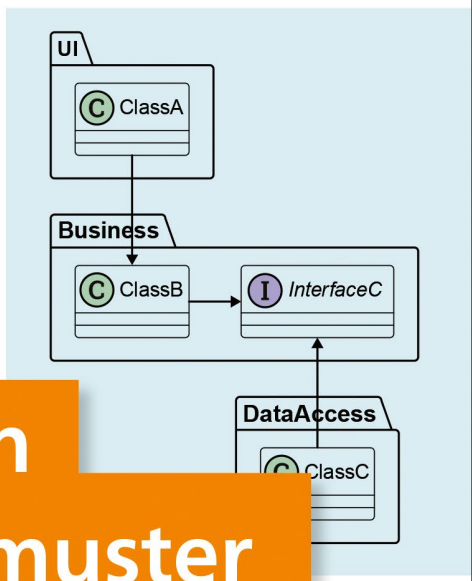
OPEN-CLOSED-PRINCIPLE

EVENT-SOURCING

CLOUD

Kristian Köhler





Softwaredesign und Entwurfsmuster

Das umfassende Handbuch

- Bewährte Entwurfsmuster für modernes Softwaredesign
- Über 30 wichtige Patterns und Designprinzipien
- Mit Diagrammen, Praxistipps und Codebeispielen

Alle Codebeispiele zum Download und auf GitHub

Rheinwerk Computing

Auf einen Blick

Inhaltsverzeichnis

Trennung der Funktionalität und Nutzung einer Facade

Alternativ zu einer kompletten Trennung und Bekanntgabe der Details der Klasse `Product` kann eine sogenannte *Facade*, wie in Abbildung 2.9 skizziert, aufgenommen werden. Sie enthält Code für die Instanziierung der Daten- und Akteurlogikklassen sowie eine Weiterleitung der Aufrufe zu den entsprechenden Methoden. Bei diesem Ansatz werden die Daten weiterhin in der separaten Klasse `ProductData` gehalten, die allerdings beim Aufrufenden nicht mehr bekannt sein muss.

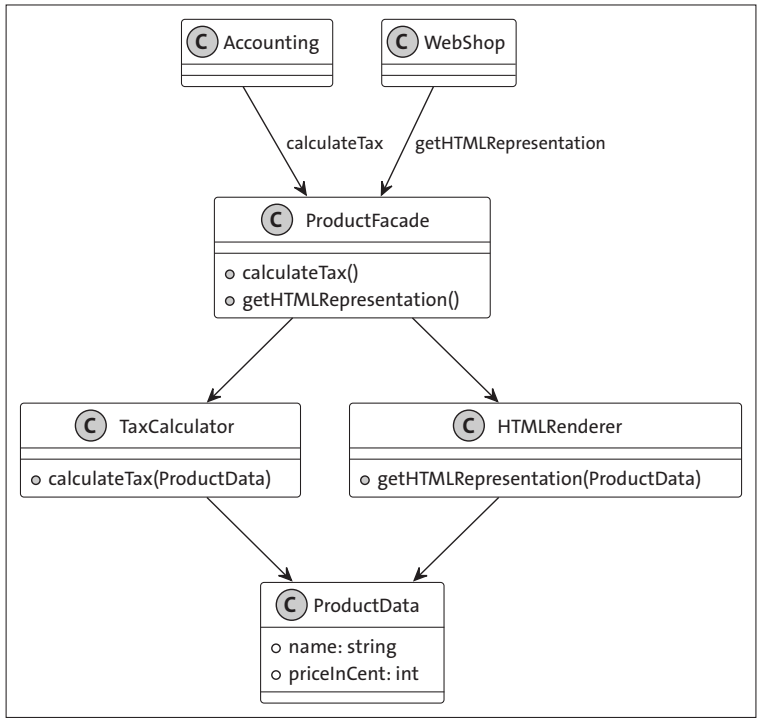


Abbildung 2.9 SRP mit Facade

Der Vorteil dieses Ansatzes besteht in der einfacheren Schnittstelle für die Klassen `WebShop` und `Accounting`, deren verantwortliche Entwickler und Entwicklerinnen sich nicht mit den Details zur Instanziierung oder Verwendung der Daten- und Logikklassen auseinandersetzen müssen. Für sie bleibt die Schnittstelle einfach zu verwenden.

Datenobjekt als Facade zur Abtrennung der Funktionalität

Eine dritte Umsetzungsvariante besteht darin, das Datenobjekt selbst als `Facade` zu verwenden und damit auch die Möglichkeit zu schaffen, wichtige Funktionalität innerhalb dieser Klasse zu implementieren. Abbildung 2.10 zeigt ein entsprechendes Klassendiagramm.

Bei diesem Ansatz können die Klassen `WebShop` und `Accounting` weiter mit einer einfachen Schnittstelle arbeiten, und Änderungen an den ausgelagerten Logiken der einzelnen Akteure beeinflussen sich nicht gegenseitig.

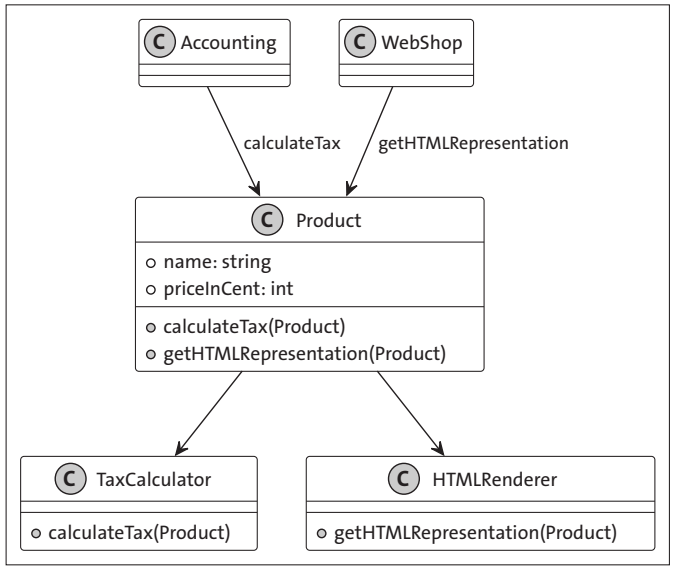


Abbildung 2.10 SRP mit Datenobjekt als Facade

Vorteile durch das Single-Responsibility-Prinzip

Für jede Umsetzungsvariante gilt, dass die Logik in einem separaten Modul bzw. einer Klasse implementiert wird und somit eine entsprechende *Kapselung* der Funktionalität stattfindet. Das fördert die Lesbarkeit und die Wartbarkeit des entstandenen Codes. Fehler innerhalb der Logikklassen können beispielsweise leichter identifiziert und dementsprechend behoben werden.

Wird Funktionalität klar abgegrenzt und eine klare Verantwortlichkeit definiert, wird eine Widerverwendung einer Logikklasse in anderen Anwendungsbereichen zudem ermöglicht.

2.3.2 Das Open-Closed-Prinzip (OCP)

Mit dem *Open-Closed-Prinzip*, das bereits 1988 von Bertrand Meyer in seinem Buch *»Object-Oriented Software Construction«* veröffentlicht wurde, wird ausgedrückt, dass Software so geschrieben werden soll, dass sie erweiterbar ist, ohne bestehende Funktionalität zu beeinflussen.

> *»A software artifact should be open for extension but closed for modification.«*
> *(Bertrand Meyer)*

Die meiste Software wird nicht in einem Schritt entwickelt, fertiggestellt und anschließend nicht mehr angepasst. Die an die Software gestellten Anforderungen ändern sich ja mit der Zeit und Anpassungen werden nötig.

Das Ziel einer guten Softwarearchitektur besteht darin, dass notwendige Anpassungen am System nicht immer wieder zu tiefgreifenden Änderungen führen, sondern dass die Änderungen sich auf einen kleinen Teilbereich beziehen und möglichst keine Auswirkungen auf andere Bereiche besitzen. Im Idealfall sind z. B. bei Erweiterungen der Funktionalität keinerlei Anpassungen an bestehendem Code notwendig.

Das von Bertrand Meyer beschriebene Open-Closed-Prinzip basierte ursprünglich auf dem Konzept der Vererbung von Klassen. Neue Funktionalität oder Attribute werden (wie in Abbildung 2.11 dargestellt) in neue, von der Basisklasse abgeleitete Klassen aufgenommen, um den bestehenden Code nicht anpassen zu müssen. Im Beispiel wird die Klasse `OriginalClass` um das Attribut `attributeB` und die Methode `doExtendedSomething` erweitert.

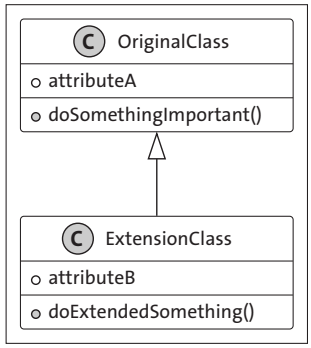


Abbildung 2.11 Das Open-Closed-Prinzip mittels Vererbung

Im Laufe der Zeit hat sich die Umsetzung des Open-Closed-Prinzips von einer primären Betonung auf Vererbung hin zur Nutzung von Polymorphie verschoben. Der Grund dafür waren unter anderen Robert C. Martin und Joshua Bloch, der Autor des Buchs *»Effective Java«*, die in ihren Publikationen immer wieder auf die enge Kopplung bei der Vererbung von Klassen und deren negative Auswirkungen auf die Abhängigkeiten hinwiesen.

Der polymorphe Ansatz nutzt, wie in Abbildung 2.12 visualisiert ist, Interfaces, für die mehrere Implementierungen vorhanden sein können. Diese Interfaces werden nicht mehr verändert, und Funktionalität kann über zusätzliche konkrete Klassen erweitert werden.

Die Einführung der Interfaces führt zu einer neuen Abstraktionsebene in der Anwendung und ermöglicht dadurch eine lose Kopplung der einzelnen unabhängigen Komponenten, die die entsprechende Logik umsetzen.

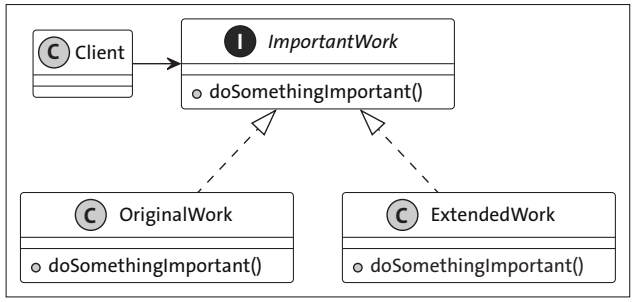


Abbildung 2.12 Das Open-Closed-Prinzip durch Polymorphie

Das Open-Closed-Prinzip auf Architekturebene

Das Beispiel einer Software zur Schulungsverwaltung soll den möglichen Einsatz des Open-Closed-Prinzips für eine Softwarearchitektur aufgezeigen: Mithilfe der Software soll eine Internet-Präsenz mit Schulungsangeboten aufgebaut werden. Die entsprechend benötigten Schulungsdaten liegen in einer Datenbank vor und sollen von dort ausgelesen werden. Neben einer HTML-Repräsentation im Browser soll für jede Schulung ein PDF-Dokument erstellt werden können, dessen Layout sich von dem der Internetpräsenz unterscheidet.

Unter Beachtung des Single-Responsibility-Prinzips des vorherigen Abschnitts und der damit verbundenen Trennung der Verantwortlichkeiten entsteht für das Beispiel ein grober Entwurf wie in Abbildung 2.13. Auch wenn die Darstellung nicht ganz UML-konform ist, soll verdeutlicht werden, dass die Verantwortlichkeit für das Laden aus der Datenbank bei der Komponente `DBTraining` liegt und die Verantwortlichkeit für die Aufbereitung der Daten für die Darstellungen in der Komponente `PresentationView`. Zwei weitere Komponenten übernehmen die Ausgabe in HTML bzw. PDF (`HTMLPresenter` und `PDFPresenter`).

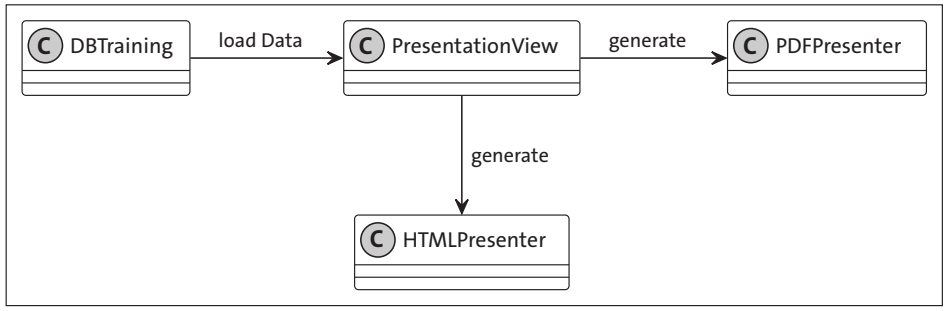


Abbildung 2.13 Grobe Darstellung der Schulungsanwendung

Nachdem die grobe Einteilung der Verantwortlichkeiten getroffen ist, lassen sich erste Klassen und deren Abhängigkeiten definieren.

Abbildung 2.14 zeigt das Package `Controller`, in dem die Klasse `GeneratorController` sowie die Interfaces `PresentationGenerator` und `TrainingRepository` definiert sind. Der `GeneratorController` übernimmt die Steuerung des kompletten Ablaufs der Generierung und nutzt dazu eine `TrainingRepository`- sowie eine `PresentationGenerator`-Implementierung.

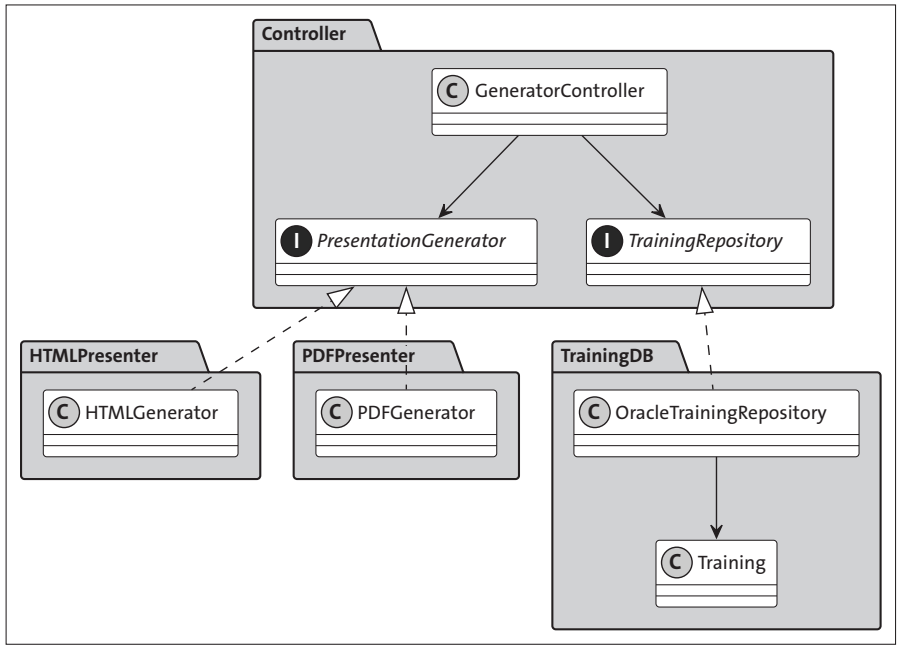


Abbildung 2.14 Die Trainingsanwendung in Klassendarstellung

In diesem Klassendiagramm lassen sich die Abhängigkeiten zwischen den Komponenten und die Einteilung dieser in sogenannte *höhere und niedrigere Anwendungsschichten* erkennen. Eine niedrigere Anwendungsschicht besitzt Abhängigkeiten zu einer höheren Schicht, aber höhere Schichten sind niemals von einer der niedrigeren Schichten abhängig. Jede Abhängigkeit im Diagramm wird als gerichteter Pfeil visualisiert und zeigt ausschließlich in die Richtung der verwendeten Abhängigkeit. Das hier verwendete Prinzip der *Dependency Inversion* wird in Abschnitt 2.3.5 ausführlicher betrachtet.

Abhängigkeiten zwischen Komponenten wirken sich potenziell auf deren Änderungsbedarf aus. Werden z. B. Änderungen an einer referenzierten Komponente vorgenommen, kann dies zu Änderungen an der referenzierenden Implementierung führen. Werden hingegen referenzierende Komponenten angepasst, ist die referenzierte Komponente dagegen geschützt und muss nicht verändert werden.

> *»If a component A should be protected from changes in component B, then component B should depend on component A.« (Robert C. Martin)*

Für das Beispiel bedeutet das, dass sich das Package `Controller` in einer höheren Anwendungsschicht befinden, da es keine weiteren Abhängigkeiten zu anderen Packages besitzt. Die Packages `HTMLPresenter`, `PDFPresenter` und `TrainingDB` befinden sich in einer niedrigeren Schicht, da sie jeweils Abhängigkeiten in Form von Interfaces in die höhere Schicht des `Controller`-Packages aufweisen. Ändert sich z. B. die Implementierung der HTML-Ausgabe, hat dies keine Auswirkung auf das `Controller`-Package. Dasselbe gilt für die Implementierung der PDF-Ausgabe. Änderungen innerhalb der Steuerungslogik des `GeneratorController` werden im Beispiel als unwahrscheinlich angesehen.

Die Einteilung eines Systems und seiner Komponenten in Abhängigkeitsschichten setzt das Open-Closed-Prinzip auf Architekturebene um. Höher gelegene Schichten werden vor Änderungen in niedrigeren Schichten geschützt und können durch sie nicht geändert werden. Erweiterungen können über die Nutzung von Polymorphie bzw. durch Implementierung von Interfaces einer höher gelegenen Schicht erfolgen. Das Beispiel könnte zusätzlich mit einem `WordPresenter` erweitert werden, ohne dass Logik innerhalb des `Controller`-Packages angepasst werden muss.

Das Options-Pattern in Go als Beispiel für das Open-Closed-Prinzip im Code

Das *Options-Pattern* ist in Go eine Designpraxis, die es ermöglicht, Funktionen mit einer variablen Anzahl von Parametern bzw. mit Standardwerten zu erstellen, ohne die Lesbarkeit und die Erweiterbarkeit des Codes zu beeinträchtigen. In Go wird diese Technik oft verwendet, um *Konstruktoren* bzw. *Factory-Methoden* von Structs mit optionalen Parametern zu implementieren.

> **Das Options-Pattern in anderen Programmiersprachen**
>
> Das *Options-Pattern* findet man auch in anderen Programmiersprachen analog umgesetzt. Das *Builder-Pattern*, das in Abschnitt 4.2 vorgestellt wird, ist im Übrigen ebenfalls eine Umsetzung des Open-Closed-Prinzips.

Folgendes Beispiel zeigt das Pattern: In Listing 2.58 können mit der Funktion `NewServerConfig` neue Instanzen vom Typ `Config` erzeugt werden. Als Parameter für die Erzeugung wird eine Liste an `string`-Werten übergeben.

```
func NewServerConfig(param1, param2, param3, param3 string) Config {
    //doSth
}
```

Listing 2.58 Factory-Methode zum Erstellen von Objekten ohne OCP (Go)

Ohne passende Parameternamen ist es, wie ich in Abschnitt »Passende Parameter« (siehe Abschnitt 2.2.2) gezeigt habe, zum einen schwer, die Semantik der einzelnen

Parameter zu erkennen. Zum anderen können keine weiteren Parameter ohne einen sogenannten *Breaking-Change* (also eine inkompatible Änderung) der Schnittstelle aufgenommen werden. Aufrufender Code muss bei Änderungen angepasst werden und das Open-Closed-Prinzip ist verletzt.

Vorsicht vor zu allgemeinen und generischen Datentypen als Parameter

In einer Schnittstelle sollten möglichst explizite Datentypen verwendet werden. Wenn Sie aus Bequemlichkeit nur sehr allgemeine und generische Datentypen (z. B. `string`, `int` oder `float`) als Parametertypen verwenden und darin die unterschiedlichsten Daten übergeben, wird die Schnittstelle zusätzlich schwer verständlich: Ein `string`-Wert kann z. B. Dezimalzahlen, XML-Strings oder IP-Adressen beinhalten. Das macht eine Schnittstelle schwer lesbar und wartbar.

Beispiel:

```
//Schwer verständlich
func NewServerConfig(param1 string, param2 string) {…}
//Besser lesbar durch die Datentypen IpAddress und Port
Func NewServerConfig(addr IpAddress, port Port) {…}
```

Mithilfe des Option-Patterns sieht die angepasste Schnittstelle der Factory-Methode so wie in Listing 2.59 aus:

```
func NewServerConfig (opts ...Option) ServerConfig {
    //doSth
}
```

Listing 2.59 Factory-Methode mit Options-Pattern (Go)

Im Unterschied zur ersten Variante mit den vielen Parametern wird hier der abstrakte Typ `Option` als Parameter erwartet, und da es sich bei der Definition des Parameters um einen *Variadic-Parameter* handelt, kann dieser bei einem Aufruf beliebig oft angegeben werden.

Die Anpassung der Funktion führt dazu, dass beliebige Option-Implementierungen übergeben werden können und die Liste der möglichen Optionen zu einem späteren Zeitpunkt durch Neuimplementierungen erweitert werden kann. Abbildung 2.15 zeigt ein entsprechendes UML-Diagramm.

`Option`-Typen sind (wie in Listing 2.60) in Go meist als *Function-Typen* implementiert, wobei die Signatur den zu konfigurierenden Typ als Parameter besitzt. Im Beispiel aus Listing 2.60 soll dementsprechend ein `ServerConfig`-Objekt mit Werten gefüllt werden:

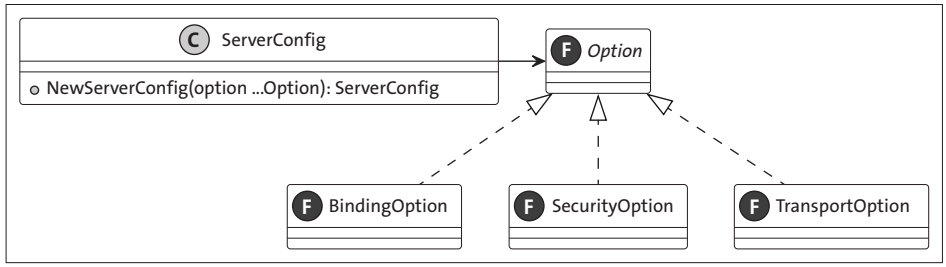


Abbildung 2.15 Das Go-Option-Pattern

```
type Option func(config *ServerConfig)
```

Listing 2.60 Function-Typ für das Option-Pattern in Go

Eine konkrete `Option`-Implementierung kann, wie in Listing 2.61, als *Closure* umgesetzt werden, die von einer Funktion als Rückgabewert geliefert wird:

```
func WithBinding(binding string) Option {
   return func(config *ServerConfig) {
      config.Addr = binding
   }
}
```

Listing 2.61 Konkrete »Option«-Implementierung (Go)

Innerhalb der Factory-Methode `NewServerConfig` kann, unabhängig von der konkreten Umsetzung des `Option`-Typs, über sämtliche übergebenen Instanzen iteriert und damit eine neue Konfiguration erstellt werden. Werden Werte nicht übergeben, können diese bei der Initialisierung eines Objekts mit Standardwerten gefüllt werden:

```
func NewServerConfig(opts ...Option) ServerConfig {
    config := &ServerConfig{}
    for _, opt := range opts {
        opt(config)
    }
    return *config
}
```

Listing 2.62 Auswerten der »Option«-Instanzen (Go)

Der Aufruf der Funktion ist anschließend leicht verständlich und erweiterbar:

```
config := NewServerConfig (WithBinding (":8080 "))
```

Listing 2.63 Nutzung des Option-Patterns (Go)

2.3.3 Das Liskovsche Substitutionsprinzip (LSP)

Barbara Liskov – eine US-amerikanische Informatikerin, die 2008 unter anderem als zweite Frau für ihre Arbeit in der Informatik mit dem Turing Award ausgezeichnet wurde – stellte 1987 auf der OOPSLA-Konferenz unter dem Titel *»Data Abstraction and Hierarchy«* zum ersten Mal ihr sogenanntes Substitutionsprinzip vor. Sie beschrieb, dass eine Unterklasse in Bezug zu ihrer Basisklasse bestimmte Bedingungen erfüllen muss, damit die Unterklasse die Basisklasse fehlerfrei ersetzen kann.

Anders formuliert besagt das Prinzip, dass eine Anwendung, die mit Objekten einer Basisklasse korrekt arbeitet, auch mit davon abgeleiteten Klassen weiterhin, also ohne Anpassung am Code, korrekt arbeiten muss.

Die Informatikerin Barbara Liskov beschreibt ihre Vorstellung einer korrekten Klassenhierarchie formal so:

> *»What is wanted here is something like the following* **substitution property***: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for oz, then S is a subtype of T.« (Barbara Liskov, OOPSLA 1987)*

Eine solche korrekte Klassenhierarchie wird durch die Einhaltung folgender Bedingungen erreicht:

- Abgeleitete Klassen dürfen keine Vorbedingungen der Basisklasse einschränken bzw. verschärfen.
- Nachbedingungen eines Einsatzes der abgeleiteten Klasse dürfen nur verstärkt werden.
- Bedingungen, die im Basistyp wahr sind, müssen auch im abgeleiteten Typ wahr bleiben.

Die Begriffe *Vorbedingung* und *Nachbedingung* geben an, welche Bedingungen vor bzw. nach einem Aufruf des Typs bzw. einer Methode erfüllt sein müssen, damit der Aufruf als fehlerfrei gilt. Eventuell muss die Schnittstelle des Objekts in einer bestimmten Reihenfolge aufgerufen werden oder müssen bestimmte, korrekte Parameter übergeben werden.

Man spricht auch von einem sogenannten *Contract* oder *Vertrag*, der zwischen dem Aufrufenden und dem Aufgerufenen geschlossen wird und eventuell nicht direkt in der Programmierschnittstelle ersichtlich ist. Weicht eine abgeleitete Klasse von diesem Vertrag ab, widerspricht das dem *Liskovschen Substitutionsprinzip* (LSP) und der Client kann nicht ohne Anpassungen fehlerfrei eingesetzt werden.

Überprüfen können Sie das korrekte Verhalten verschiedener Klassen z. B. mit einem Unit-Test, der für die Basisklasse dasselbe Ergebnis wie für eine abgeleitete Klasse liefern muss.

Mögliche Probleme bei der Vererbung von Klassen

Das Beispiel in Abbildung 2.16 soll die Problematik eines nicht eingehaltenen Liskovschen Substitutionsprinzips noch einmal verdeutlichen.

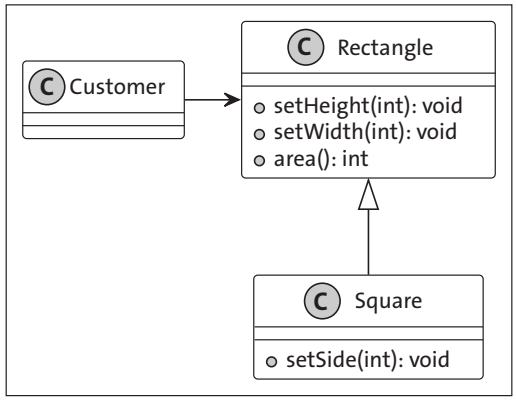


Abbildung 2.16 Beispiel für die Verletzung des Liskovschen Substitutionsprinzips

Die Klasse `Rectangle` repräsentiert ein Rechteck, bei dem die Höhe und die Breite über Setter-Methoden gesetzt werden können. Mit dem Aufruf der Methode `area` kann daraufhin die Fläche des Rechtecks ermittelt werden. In Listing 2.64 ist die Klasse in Java implementiert:

```
public class Rectangle {
    protected int height;
    protected int width;

    public void setHeight(int height) {
        this.height = height;
    }
    public void setWidth(int width) {
        this.width = width;
    }
    public int getArea() {
        return height * width;
    }
}
```

Listing 2.64 »Rectangle«-Klasse in Java für die Berechnung einer Fläche

Ein Aufrufender kann die Klasse instanziieren und, wie in Listing 2.65 gezeigt, die Fläche berechnen lassen:

```
Rectangle rectangle = new Rectangle();
rectangle.setWidth(10);
rectangle.setHeight(2);
System.out.println(rectangle.getArea()); // Ausgabe: 20
```

Listing 2.65 Verwendung der »Rectangle«-Klasse

Da es sich bei einem Quadrat um ein spezielles Rechteck handelt, wird nun (in Listing 2.66) die Klasse `Square` von der Klasse `Rectangle` abgeleitet und entsprechend angepasst. Beim Setzen der Höhe bzw. der Länge werden die Seitenlängen über die neue Methode `setSide` immer auf denselben Wert gesetzt:

```
public class Square extends Rectangle {
    @Override
    public void setHeight(int height) {
        setSide(height);
    }
    @Override
    public void setWidth(int height) {
        setSide(height);
    }
    public void setSide(int side) {
        this.width = side;
        this.height = side;
    }
}
```

Listing 2.66 Implementierung einer abgeleiteten Klasse

Das Ergebnis im Client ist nun allerdings etwas verwirrend:

```
Rectangle rectangle = new Rectangle();
rectangle.setWidth(10);
rectangle.setHeight(2);
System.out.println(rectangle.getArea()); //Ausgabe: 20

Rectangle square = new Square();
square.setWidth(10);
square.setHeight(2);
System.out.println(square.getArea()); //Ausgabe: 4
```

Listing 2.67 Verwirrendes Ergebnis, da das LSP nicht eingehalten wurde

Obwohl es sich um zwei Variablen vom Typ `Rectangle` handelt und diese gleich aufgerufen werden, unterscheidet sich in Listing 2.67 unerwarteterweise das Ergebnis. Ein Unit-Test, der die Funktionalität der Basisklasse testet, könnte nicht fehlerfrei mit einer Instanz der abgeleiteten Klasse ausgeführt werden. Das Ergebnis würde abweichen, und das Liskovsche Substitutionsprinzip wäre damit verletzt.

Muss man das Liskovsche Substitutionsprinzip immer einhalten?

Der strikte Einsatz des LSPs wird manchmal in der Praxis als problematisch betrachtet. Nicht immer kann sich eine abgeleitete Klasse genau gleich wie die Basisklasse verhalten.

Folgendes kurzes Beispiel in Java zeigt den Widerspruch:

```
System.out.println(rectangle.toString().equals(square.toString()));
```

Hier wird jeweils für beide Objekte die `toString`-Methode aufgerufen, die eine Textrepräsentation des Objekts liefert. Die Ausgabe des Vergleichs lautet `false`, da die beiden Objekte nicht identisch sind und das potenziell vorhandene polymorphe Verhalten gewünscht ist. Das LSP ist verletzt.

Trotz dieses Beispiels ist in den meisten Anwendungsfällen die Einhaltung des LSP dennoch sinnvoll. Für den Praxiseinsatz sollte definiert werden, in welchem Rahmen bzw. welchem Kontext das Prinzip für die Klassen gelten soll.

Das Liskovsche Substitutionsprinzip in der Architektur

Das Prinzip der fehlerfreien Substitution lässt sich nicht nur, wie von Barbara Liskov ursprünglich beschrieben, auf der Ebene des Codes einsetzen, sondern auch auf eine Softwarearchitektur erweitern.

Einige Open-Source-Projekte bieten beispielsweise schnittstellenkompatible Server-Implementierungen für bestimmte HTTP/Rest-Schnittstellen kommerzieller Produkte. Clients können sich daraufhin entscheiden, ob sie die Verbindung mit einer kommerziellen oder einer auf der Open-Source-Lösung basierenden Server-Instanz aufbauen möchten. Das Ziel ist es, die Migration zwischen den Produkten zu erleichtern und Anwendern die Möglichkeit eines Wechsels zu geben.

Der *MinIO*-Server, ein Open-Source-veröffentlichter *Object Store*, bietet beispielsweise eine Amazon-S3-kompatible Schnittstelle zum Verwalten von Dateien an. Er kann dadurch entweder als Ersatz für S3 oder zum lokalen Testen einer eigenen Implementierung verwendet werden.

Würden sich die Implementierungen unterschiedlich verhalten, müsste der Client-Code angepasst und eine entsprechende Codeweiche eingebaut werden. Auch hier würde das Liskovsche Substitutionsprinzip verletzt werden.

Vorsicht mit Codeweichen!

Ein Hinweis und *Code Smell*, der zeigt, ob das Liskovsche Substitutionsprinzip verletzt ist, sind im Code vorhandene Weichen, die zwischen unterschiedlichen Implementierungen unterscheiden und entsprechende Anpassungen vornehmen.

2.3.4 Das Interface-Segregation-Prinzip (ISP)

Ein Ziel einer guten Softwarearchitektur besteht – wie ich bereits mehrfach erwähnt habe – darin, Software wartbar und erweiterbar zu machen. Eine zu hohe Kopplung durch Abhängigkeiten zwischen den Komponenten der Software kann dies erschweren oder im schlimmsten Fall komplett verhindern.

Das *Interface-Segregation-Prinzip (ISP)* konzentriert sich aus diesem Grund auf die Definition von Schnittstellen in Bezug auf die Clients, die sie einsetzen, und minimiert die daraus resultierende Kopplung der Komponenten. Das Prinzip besagt, dass kein Code von anderem Code abhängen sollte, den er nicht benötigt:

> *»Clients should not be forced to depend upon interfaces that they do not use.« (Robert C. Martin)*

Eine umfangreiche Schnittstelle mit vielen definierten Methoden führt oftmals bei neuen konkreten Umsetzungen der Schnittstelle dazu, dass nicht alle Methoden implementiert werden oder dass auf der anderen Seite im Clientcode nicht alle Methoden verwendet werden. Dadurch entstehen unnötige Abhängigkeiten, die bei Erweiterungen mit gepflegt werden müssen.

Beispiel für das Interface-Segregation-Prinzip auf Codeebene

Ich möchte die abstrakte Beschreibung des Prinzips anhand eines Beispiels erläutern: Ein Druckerhersteller entwickelt für seine Produkte eine Software, mit der die Geräte verwaltet werden können. Da bisher nur Laserdrucker hergestellt wurden, wurde die Klasse `LaserPrinter` mit der entsprechenden Logik umgesetzt. Die Klasse enthält, wie in Abbildung 2.17 zu sehen ist, die drei Methoden `print`, `suppliesSummary` und `supportsColor`.

Mit der Methode `suppliesSummary` kann der Status der Verbrauchsmaterialien abgefragt werden, die Methode `supportsColor` liefert die Information, ob es sich um einen Farbdrucker handelt, und mit der Methode `print` kann ein Druck ausgelöst werden.

Da der Hersteller plant, weitere Produkte auf den Markt zu bringen, entschließen sich die Entwickler dazu, eine abstrakte Beschreibung für Drucker in Form eines Interfaces zu erstellen und die drei Methoden dort zu definieren. Die bisherige Klasse `LaserPrinter` implementiert nun das neue Interface `Printer`, was Sie ebenfalls in Abbildung 2.17 sehen.

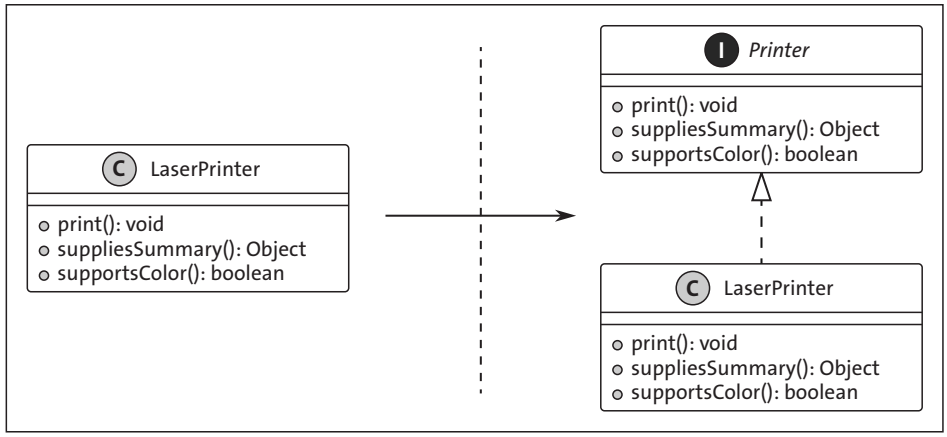


Abbildung 2.17 Herauslösen des Printer-Interfaces

Parallel zu den Druckern wird an der Herstellung reiner Kopierer gearbeitet, und in der Software für die Kopierer soll eine ähnliche Funktionalität angeboten werden wie bei den bestehenden Produkten. Aus diesem Grund wird, wie in Abbildung 2.18 gezeigt, eine neue Klasse `Copier` erstellt, die ebenfalls das Interface `Printer` implementiert. Somit können auch die neuen Kopierer von der bisherigen Client-Software, die das Interface `Printer` nutzt, polymorph verwendet werden.

Damit sämtliche Funktionalität der Klasse `Copier` in der Client-Software angeboten werden kann, wird das Interface `Printer` zusätzlich um die Methode `startCopy` erweitert, da die Klasse `Copier` eine entsprechende Methode besitzt.

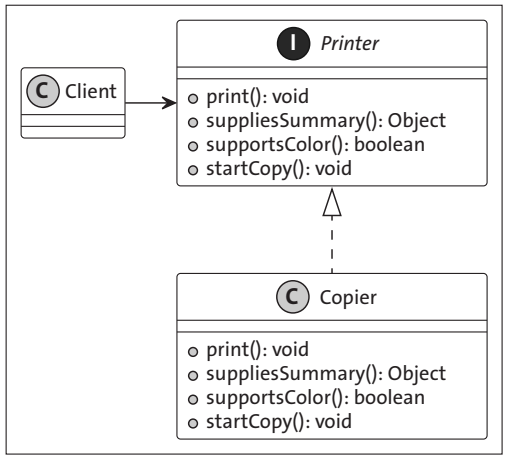


Abbildung 2.18 Die Klasse »Copier« (Kopierer), die das Interface »Printer« umsetzt (schlecht)

Da der Copier keine Druck-Funktionalität besitzt, wird die Methode print leer implementiert bzw. liefert, wie Sie in Listing 2.68 sehen, beim Aufruf direkt einen Fehler:

```
//Unnötige leere Implementierung wird aufgezwungen!
public class Copier implements Printer {
    @Override
    public void print() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    ...
}
```

Listing 2.68 Implementierung der »print«-Methode in der Klasse »Copier« (Java)

Auch die Klasse LaserPrinter muss entsprechend angepasst werden. Auch sie liefert eine Exception, wenn die nicht benötigte Methode startCopy aufgerufen wird:

```
//Unnötige leere Implementierung wird aufgezwungen!
public class LaserPrinter implements Printer {
    @Override
    public void startCopy() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    …
}
```

Listing 2.69 Leere Implementierung der »startCopy«-Methode in »LaserPrinter« (Java)

Durch die Beispielimplementierung wird klar, dass das Interface Printer nicht für beide Klassen geeignet ist. Es musste jeweils eine leere Methode in die entsprechende konkrete Umsetzung eingefügt werden.

Schlimmer als die leeren Implementierungen in den Klassen ist die eingeführte Abhängigkeit zwischen den Klassen LaserPrinter und Copier. Ändert sich z. B. die Funktionalität des Kopierers und muss die Signatur der startCopy-Methode im Interface Printer angepasst werden, hat dies ebenfalls Auswirkungen auf die Klasse LaserPrinter, obwohl diese von der Anpassung eigentlich nicht betroffen ist.

Durch eine Auftrennung des Interfaces Printer in einzelne, spezielle und voneinander unabhängige Interfaces für Drucker und Kopierer kann dieses Problem gelöst werden.

Jeder Client erhält eine speziell an seine Anforderungen angepasste Schnittstelle. Gemeinsame Funktionalität, wie die Methoden suppliesSummary und supprtsColor, werden weiterhin in einem gemeinsamen Interface PageDuplicator definiert. Abbildung 2.19 zeigt ein entsprechendes Klassendiagramm.

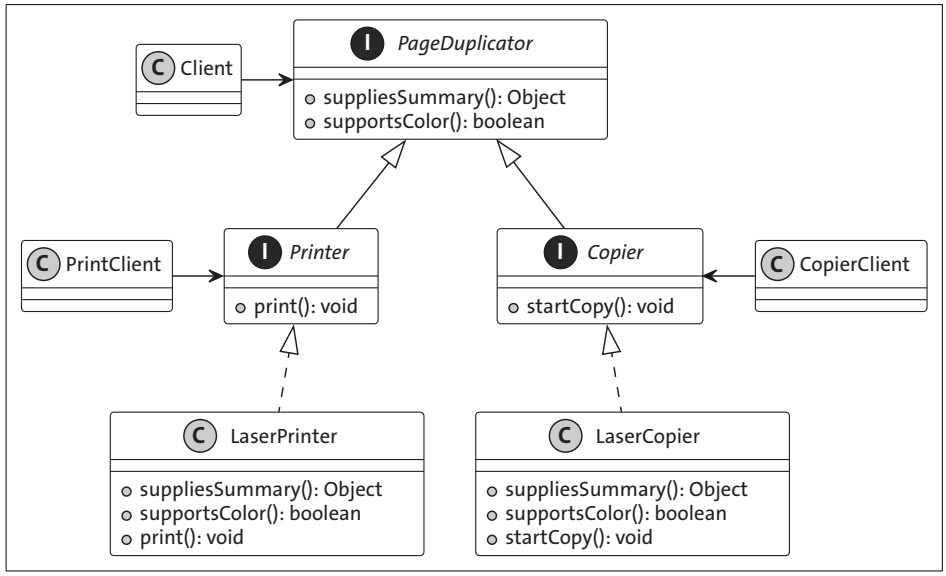


Abbildung 2.19 Auftrennung der Interfaces zur Umsetzung des ISP

Die Einführung der zusätzlichen Abstraktionsschicht in Form des Interfaces `PageDuplicator` ermöglicht es allgemeinen Client-Anwendung weiterhin, mit beiden Klassen über dieses Interface zu arbeiten und gemeinsames Verhalten klar abzugrenzen. Abhängigkeiten zwischen den beiden Interfaces `Printer` und `Copier` sind abgebaut, und eine potenzielle Änderung an der Funktionalität eines Interfaces hat keine Auswirkungen auf die Implementierungen des anderen Interfaces. Die enge Kopplung zwischen ihnen wurde aufgehoben.

Eine weitere positive Nebenwirkung der Änderung ist folgende: Falls der Druckerhersteller eines Tages neue Multifunktionsdrucker, die drucken und kopieren können, anbieten möchte, kann z. B. eine Klasse `MultifunctionPrinter` erstellt werden, die beide Interfaces implementiert (siehe Abbildung 2.20).

Durch eine Abstraktion von Schnittstellen lassen sich oftmals Kopplungen und Abhängigkeiten minimieren oder ganz auflösen. Rob Pike, einer der Initiatoren von Go, sagt beispielsweise zusätzlich, dass die Abstraktion einer Schnittstelle umso geringer ist, je mehr Methoden sie besitzt:

»The bigger the interface, the weaker the abstraction.« (Rob Pike)

Abstraktionen und die Schaffung von clientbezogenen, schlanken Schnittstellen, wie sie das Interface-Segregation-Prinzip vorsieht, helfen dabei, Code wartbarer und erweiterbarer zu machen.

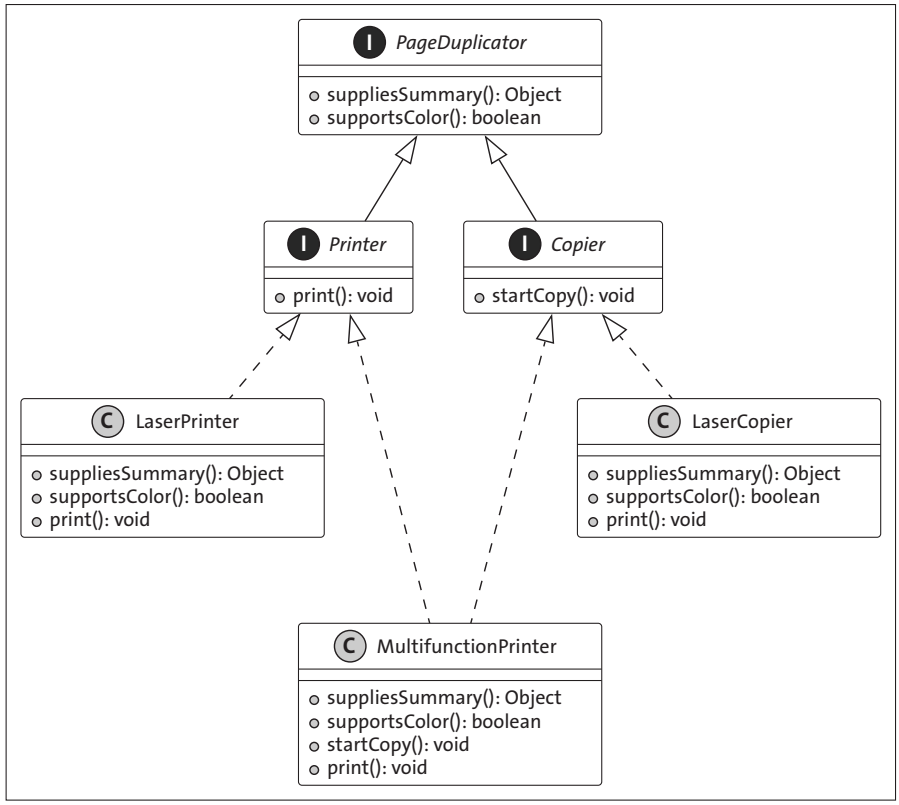


Abbildung 2.20 Multifunktionsdrucker implementieren zwei Interfaces.

Das Interface-Segregation-Prinzip auf Architekturebene

Die Schaffung und Nutzung von clientbezogenen, schlanken Schnittstellen kann ebenfalls auf Architekturebene hilfreich sein und zum Beispiel unnötige Abhängigkeiten reduzieren.

Wenn beispielsweise eine Bibliothek vor ihrer Nutzung immer über einen zwangsläufig benötigten Lademechanismus Konfigurationsdaten aus einer Datenbank lädt, aber eine aufrufende Anwendung keine Funktion verwendet, die eine entsprechende Konfiguration benötigt, ist auch hier das ISP verletzt:

```
Bibliothek bib = new Bibliothek(oracle); // Parameter muss angegeben werden!
                                         // Angabe von null nicht möglich.
                                         // Konfiguration wird geladen.
Result r = bib.addInteger(1,2); // Hier wird keine Konfiguration benötigt!
```

Listing 2.70 Eine Bibliothek verletzt das ISP wegen unnötiger Datenbank. (Java)

Die aufrufende Anwendung enthält dadurch eine Abhängigkeit von einer Datenbank, die gar nicht verwendet wird (siehe Listing 2.70).

Wie auf Codeebene wird die Anwendung schwerer wartbar und der unnötige Ballast bzw. die Abhängigkeiten können später Probleme oder zumindest Schwierigkeiten verursachen.

2.3.5 Das Dependency-Inversion-Prinzip (DIP) und die Inversion of Control

In Abschnitt 2.3.2, »Das Open-Closed-Prinzip (OCP)«, bin ich bei der Beschreibung von dessen Architektur bereits auf die Einteilung der Komponenten einer Software in *höhere und niedrigere Anwendungsschichten* eingegangen.

Der Hintergrund für diese Einteilung ist das sogenannte *Dependency-Inversion-Prinzip (DIP)*. Es hat zum Ziel, dass höhere Anwendungsschichten mit den in ihnen enthaltenen komplexen Logiken nicht von niedrigeren Anwendungsschichten mit Hilfsfunktionen und technischen Details abhängig sind. Die Absicht dahinter ist, wieder einmal, eine hohe Wiederverwendbarkeit und Wartbarkeit der Komponenten der höheren Anwendungsschichten zu gewährleisten.

Robert C. Martin definiert das Ziel einer Einführung des *Dependency-Inversion-Prinzips* wie folgt:

- Höhere Anwendungsschichten dürfen nicht von niedrigeren Schichten abhängig sein. Beide Schichten sollten ausschließlich von Abstraktionen abhängen.
- Abstraktionen sollten nicht von Details abhängig sein. Details sollten von Abstraktionen abhängen.

Für die Anwendungsentwicklung bedeutet das, dass alle Abhängigkeiten von einer zu einer anderen Schicht immer über Abstraktionen und niemals über konkrete Details definiert sein sollten. Ganz konkret: Abhängigkeiten zwischen Schichten sollten Sie immer über Interfaces implementieren und nicht über Klassen.

Der Grund besteht (wie ich schon beim Open-Closed-Prinzip erklärt habe) in dem durch die Abhängigkeiten potenziell entstehenden Zwang zu Änderung, der eingegrenzt werden soll. Änderungen an einer Komponente können Anpassungen an der referenzierenden Komponente nach sich ziehen. In der Regel unterliegen konkrete Implementierungen häufiger Änderungen als deren abstrakte Schnittstelle.

Gutes Software- und Schnittstellendesign zeichnet sich dementsprechend unter anderem dadurch aus, dass Sie Schnittstellen stabil halten und Änderungen möglichst nur in konkreten Umsetzungen dieser Interfaces durchführen bzw. nötig machen. Referenzieren Sie ausschließlich stabile Schnittstellen, schützt das vor Änderungen.

Mit folgenden Regeln kann ein solches Softwaredesign, bei dem Änderungen sich nicht auf weitere Komponenten ausbreiten, aufgebaut werden:

- Keine konkreten Klassen referenzieren, die sich häufig ändern.
- Keine Vererbungshierarchien auf Klassen aufbauen, die sich häufig ändern.
- Keine konkreten Implementierungen überschreiben.

Abhängigkeiten von stabilen Schnittstellen

In der Praxis wird es nicht möglich sein, sämtliche Abhängigkeiten von Komponenten mittels abstrakten Schnittstellen zu definieren. Verwendet man z. B. konkrete Klassen oder Typen aus der Standardbibliothek einer Programmiersprache, ist es nicht sinnvoll, diese hinter einer abstrakten Schnittstelle zu verbergen.

Ausschlaggebend für die Einführung einer abstrakten Schnittstelle ist die Änderungswahrscheinlichkeit bzw. die Frequenz, mit der Änderungen vorgenommen werden.

Standardbibliotheken der verschiedenen Programmiersprachen und die in ihnen enthaltenen Klassen oder Typen können Sie in der Regel als stabil betrachten und konkret verwenden. Sogenannte *Breaking-Changes*, also inkompatible Anpassungen der Schnittstellen dieser Bibliotheken, werden so gut wie möglich vermieden, da ansonsten alle bestehenden Programme in dieser Programmiersprache nach einer Aktualisierung auf die neue, angepasste Version modifiziert werden müssten. Das wollen sicherlich alle Programmiersprachen-Teams vermeiden!

Die Nutzung abstrakter Schnittstellen zwischen Komponenten führt zur namensgebenden Umkehrung der Abhängigkeitsrichtung (*Inversion of Control*) zwischen diesen und damit auch zu einem Schutz der referenzierenden Komponente vor Änderungen.

Folgendes Beispiel soll das DIP verdeutlichen: Abbildung 2.21 zeigt in einem Klassendiagramm einen Ausschnitt einer Anwendung, in der über ein User-Interface Daten aus einer Datenbank dargestellt werden. Der Ausschnitt enthält drei Klassen in drei unterschiedlichen Anwendungsschichten, deren Abhängigkeiten als Pfeile dargestellt sind.

Verfolgt man die Abhängigkeitspfeile der Anwendung, stellt man fest, dass im Beispiel die Anzeigekomponente von der Datenzugriffskomponente abhängt. Änderungen an konkreten Klassen in dieser Komponente könnten dementsprechend zu Anpassungen in der Darstellungsschicht führen. Das ist für Datenobjekte, die um neue, zur Darstellung gedachte Felder erweitert werden, eventuell noch sinnvoll, aber sicher nicht, wenn z. B. das Datenbankprodukt gewechselt wird und die entsprechenden Klassen angepasst werden müssen.

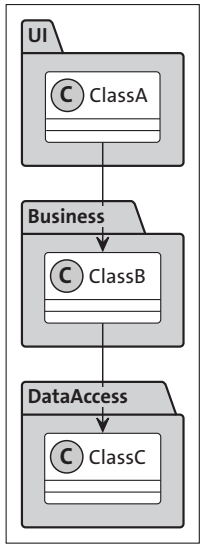


Abbildung 2.21 Abhängigkeiten zwischen Komponenten

Nach dem Dependency-Inversion-Prinzip sollten die Abhängigkeiten zwischen den Komponenten immer über Abstraktionen stattfinden. Im Beispiel führt das dementsprechend, wie in Abbildung 2.22 dargestellt, zur Einführung des Interfaces `InterfaceC`, das eine abstrakte Schnittstelle zur Ablage und zum Laden der Daten definiert.

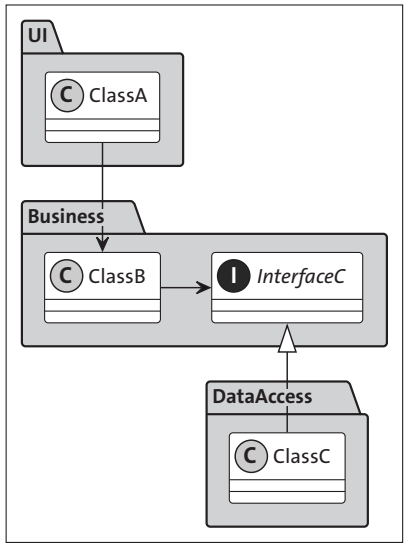


Abbildung 2.22 DIP bei Komponentenabhängigkeiten

Mit der Einführung der abstrakten Schnittstelle innerhalb der Business-Komponente hat sich die Abhängigkeitsrichtung gedreht. Der Abhängigkeitspfeil zeigt nicht mehr von der Business-Komponente zur DataAccess-Komponente bzw. von `ClassB` zu `ClassC`, sondern genau in die entgegengesetzte Richtung. Änderungen an der technischen Umsetzung von `ClassC` wirken sich nicht mehr auf die Darstellungskomponente aus.

Um es kurz zusammenzufassen: Bei Beachtung des Dependency-Inversion-Prinzips entstehen lose gekoppelte Anwendungen bzw. Komponenten, die einfacher und klarer aufgebaut sind. Diese Flexibilität führt insgesamt zu einer besseren Testbarkeit und Erweiterbarkeit der Komponenten. Während der Ausführung von Unit-Tests kann z. B. zusätzlich eine konkrete Implementierung durch eine Mock-Implementierung ersetzt werden.

2.4 Information Hiding

Das Prinzip *Information Hiding* ist ein Grundpfeiler für ein gutes Softwaredesign, und ich habe es bereits an einigen Stellen indirekt beschrieben, ohne es explizit zu nennen. Es bedeutet, dass Daten und Funktionen innerhalb einer Komponente möglichst nach außen verborgen werden und nur die wichtigen bzw. relevanten Informationen außerhalb der Komponente sichtbar sind.

Die Schlüsselprinzipen des *Information Hiding* sind *Kapselung* und *Abstraktion*. Beide sollen dafür sorgen, die Abhängigkeiten der Komponenten so zu minimieren, dass ein flexibleres, robusteres und besser wartbares Softwaredesign entsteht.

Je geringer die Abhängigkeiten zwischen den Komponenten sind, desto besser sind die Komponenten vor Änderungen an einer referenzierten Komponente geschützt, die von außen vorgenommen wurden. Insgesamt sind die Vorteile des Information Hiding:

- verringerte Abhängigkeiten zwischen Komponenten
- geringere Komplexität der Anwendung
- robustere Software
- bessere Wartbarkeit und Erweiterbarkeit
- bessere Wiederverwendbarkeit der Komponenten

Viele der Prinzipien, die ich in den vorigen Abschnitten vorgestellt habe, implementieren das *Information Hiding* bzw. unterstützen Sie bei dessen Umsetzung:

- **Kapselung** – Durch die Kapselung werden interne Details versteckt. Der Client erhält ausschließlich Zugriff eine exportierte Schnittstelle.

- **Abstraktion** – Über Abstraktion werden Interfaces geschaffen, die Details einer konkreten Implementierung verbergen.
- **klare Grenzen zwischen Komponenten** – Klare Grenzen zwischen Komponenten entstehen durch eine klare Definition der Schnittstelle und die Abschottung der Details bzw. der Informationen.
- **das Interface-Segregation-Prinzip** – Der Einsatz dieses Prinzips ermöglicht es, schlanke Schnittstellen für Clients zu definieren und somit Informationen zu verstecken und Abhängigkeiten zu minimieren.
- **das Dependency-Inversion-Prinzip** – Mit dem DIP wird die Richtung einer Abhängigkeit umgedreht, indem Abstraktionen verwendet werden. Die Abstraktion sorgt für eine Minimierung der Abhängigkeiten.

Das Prinzip des *Information Hiding* kann und sollte in vielen Ebenen der Softwareentwicklung eingesetzt werden, und Sie sollten immer – unabhängig von der aktuellen Abstraktionsebene – darauf achten, möglichst wenige Informationen zu exportieren. Achten Sie immer auf die übergebenen Parameter bzw. Rückgabewerte einer Schnittstelle! Sie bestimmen die übergebene Informationsmenge und beeinflussen damit direkt das Information Hiding.

Anwenden können Sie das Prinzip unter anderem bei der Definition von:

- Klassen
- Interfaces
- Packages
- Modulen
- Bibliotheken
- Anwendungen
- externen Schnittstellen (z. B. der REST-Schnittstelle)
- ...

»Weniger ist mehr, aber nicht weniger als nötig.« (Weisheit des Information Hiding)

2.5 Inversion of Control und Dependency Injection

Jedes Programm besitzt einen Programmablauf bzw. *Kontrollfluss* (engl. *Flow of Control*), der die Reihenfolge der auszuführenden Befehle definiert. Eine Komponente ruft beispielsweise eine wiederverwendbare Bibliothek auf, um allgemeine Aufgaben zu erledigen. In diesem Fall liegt die Kontrolle über den Programmablauf in der Komponente selbst, da sie den Code für den Aufruf enthält.

Mit dem *Inversion of Control-(IoC-)*Prinzip hingegen wird der Kontrollfluss einer Komponente nicht mehr in ihr selbst definiert, sondern von einer externen Komponente bzw. einem Framework.

Gerne wird dieser Ansatz humoristisch mit der Filmindustrie in Hollywood verglichen: Nicht die Schauspieler rufen die ganze Zeit bei der Produktionsfirma an und erkundigen sich nach dem Stand der Ausschreibung, sondern sie geben die Verantwortung an einen Agenten ab, der die Vermittlung für sie übernimmt und sie anruft, wenn es etwas Neues gibt.

»Don't call us, we'll call you« (das Hollywood-Prinzip)

Das IoC-Prinzip ermöglicht in der Softwareentwicklung zum einen die Implementierung generischen Codes innerhalb von Bibliotheken bzw. Frameworks, der flexibel an verschiedene Anforderungen angepasst werden kann. Zum anderen werden Abhängigkeiten innerhalb der aufgerufenen Komponente minimiert.

In vielen Bibliotheken und Frameworks (z. B. im Spring Framework in Java) kommen IoC für das Abhängigkeitsmanagement und das damit verbundene *Dependency-Injection-(DI-)*Prinzip zum Einsatz. Abhängigkeiten werden hierbei nicht mehr von der Komponente selbst aufgelöst, sondern von einem sogenannten *Container* bzw. einer *Laufzeitumgebung*, der bzw. die die Abhängigkeit per Methodenaufruf von außen an bzw. für die Komponente setzt.

Wird z. B. in einer Komponente eine Datenbankverbindung benötigt, muss die Komponente nicht selbst eine Verbindung aufbauen, sondern definiert eine Schnittstelle, über die sie eine Verbindung übergeben bekommen kann. Listing 2.71 zeigt ein Beispiel ohne *Dependency Injection*:

```
public class ShipService {
    public void storeShip(Ship ship) throws NamingException, SQLException {
        Context context = new InitialContext();
        DataSource dataSource =
            (DataSource)context.lookup("database_connection");

        Connection connection = dataSource.getConnection();
        //...
    }
}
```

Listing 2.71 »ShipService« ohne Dependency Injection (Java)

Durch den Einsatz der *Dependency Injection* ändert sich das Beispiel so, wie in Listing 2.72 gezeigt. Die Abhängigkeit einer `DataSource`-Instanz wird nicht mehr innerhalb der Komponente mittels *JNDI* (dem *Java Naming and Directory Interface*) aufgelöst, sondern von einer externen Komponente (dem Container) übernommen. Dieser nutzt

die Methode `setDataSource`, um eine entsprechende `DataSource`-Implementierung von außen zu setzen:

```
public class ShipService {
    private DataSource dataSource;
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public void storeShip(Ship ship) {
        Connection connection = dataSource.getConnection();
        //...
    }
}
```

Listing 2.72 »ShipService« mit DataSource-Abhängigkeit als DI (Java)

Die resultierende Komponente enthält durch die Anpassung weniger Abhängigkeiten und wird wartbarer und besser testbar. Die Variante ohne DI war im Gegensatz zu der angepassten Version z. B. auf eine vorhandene JNDI-Umgebung angewiesen. Wie und woher der Container in der angepassten Variante die Referenz zur `DataSource` bezieht, spielt im Code der Komponente keine Rolle.

Zusammenfassend lässt sich sagen, dass mit den Prinzipien *Inversion of Control* und *Dependency Inversion* sauberere, modularere und flexiblere Software erstellt werden kann und dass die meisten Frameworks diesen Ansatz forcieren.

2.6 Separation of Concerns und Aspektorientierung

Das Designprinzip *Separation of Concerns* (SoC) ist ein weiteres grundlegendes Entwurfsprinzip, bei dem Software in verschiedene Teilbereiche untergliedert wird, die jeweils eine spezifische Aufgabe erledigen. Anwendungen sollen nicht als komplexer, undurchsichtiger Block entwickelt werden, sondern in kleineren, wartbareren Blöcken. Das Prinzip bildet unter anderem die Basis für die beiden bereits vorgestellten SOLID-Prinzipien *Interface-Segregation-Prinzip* und *Single-Responsibility-Prinzip*.

Eine Trennung der Anliegen, die das Prinzip beschreibt, findet man in verschiedenen Bereichen und Ebenen von Software. In der Webentwicklung wird beispielsweise der textuelle Inhalt einer Seite innerhalb einer HTML-Datei abgelegt, während das Aussehen über CSS in separaten Dateien erstellt wird und das dynamische Verhalten ebenfalls separat in JavaScript-Dateien.

Die Vorteile einer Aufteilung in kleinere Blöcke liegen – wie bei den entsprechenden SOLID-Prinzipien beschrieben – auf der Hand:

- bessere Lesbarkeit
- bessere Wartbarkeit
- erhöhte Wiederverwendbarkeit

Eine starre und unreflektierte Auftrennung der Anwendung in verschiedene funktionale Blöcke ist in der Softwareentwicklung allerdings problematisch, da sich Themen oder Anforderungen oftmals über mehrere Anwendungsteile bzw. Blöcke hinweg erstrecken und es mit der Auftrennung zu Codeduplizierungen kommt (wie in Abbildung 2.23 innerhalb der Komponenten A, B und C). Kent Beck prägte im Jahr 1997 für solche übergreifenden Themen bzw. Anforderungen den Begriff *Cross Cutting Concerns*. Hierzu zählen beispielsweise:

- **Sicherheit** – In so gut wie allen Anwendungen müssen Authentifizierung, Autorisierung und Verschlüsselung berücksichtigt werden.
- **Logging** – Jede relevante Funktion sollte Informationen protokollieren.
- **Transaktionen** – Geschäftslogik, die sich über mehrere Aufrufe erstreckt und mehrere Datenbankoperationen beinhaltet, muss meist zuverlässig als Einheit behandelt werden.

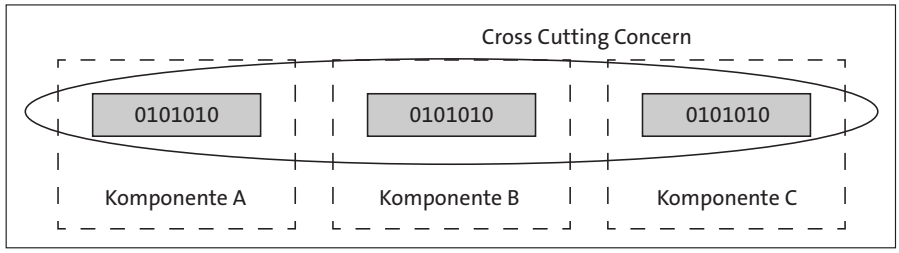


Abbildung 2.23 Cross Cutting Concern

2.6.1 Aspektorientierte Programmierung

Ein Lösungsansatz, um diese Codeduplizierungen der *Cross Cutting Concerns* und die damit verbundenen Probleme der Lesbarkeit und Wartung anzugehen, ist die *aspektorientierte Programmierung* (AOP). Bei ihr wird generische Funktionalität aus den einzeln aufgeteilten Blöcken in unabhängigen Komponenten zusammengefasst und über eine Konfiguration an angegebenen Stellen zur Ausführung gebracht.

Gregor Kiczales und seine Kollegen bei Xeroc PARC haben ihre Ideen zur aspektorientierten Programmierung 1997 auf der »European Conference on Object-Oriented Programming« vorgestellt und daraufhin mit *AspectJ* eine konkrete Implementierung ihrer Ideen vorgestellt. In der Java-Welt hat sich das AOP-Framework mittlerweile als De-facto-Standard etabliert.

AOP im Java-Enterprise-Umfeld

Die aspektorientierte Programmierung ist mittlerweile ein fester und grundlegender Teil der Java-Enterprise-Entwicklung geworden. AOP-Funktionalitäten finden sich sowohl in *JavaEE* als auch z. B. im verbreiteten *Spring Framework*. In beiden Fällen wird gemeinsam genutzte Funktionalität in zentrale Komponenten ausgelagert und muss nicht immer neu implementiert werden. Dieser Ansatz macht die beiden Umgebungen sehr mächtig.

In jedem AOP-Framework wird ein sogenanntes *Join Point Model* definiert, das eine dynamische Definition von Querschnittsbelangen durch präzise Bestimmung ihrer Einbindungspunkte im Programmcode ermöglicht. AspectJ verwendet hierzu folgende Begriffe:

- **Join point** – Ein *Join point* ist ein Ausführungspunkt, an dem die ausgelagerte Funktionalität eingebracht werden kann. Das ist beispielsweise ein Methodenaufruf, eine Objektinitialisierung oder ein Feldzugriff.
- **Pointcut** – Ein *Pointcut* markiert einen bestimmten Joint point in der Anwendung. Er stellt z. B. einen Aufruf einer spezifischen Methode einer konkreten Klasse dar.
- **Advice** – Ein *Advice* enthält die eigentliche ausgelagerte Funktionalität sowie die Zuordnung zu einem speziellen Pointcut.
- **Weaving** – Advices werden in den bestehenden Code über das sogenannte *Weben* bzw. *Weaving* eingebracht. Dies kann bei AspectJ entweder zum Compile-Zeitpunkt mit dem sogenannten *Compile-Time-Weaving* oder erst später zur Laufzeit mittels *Load-Time-Weaving* erfolgen.

Die Nutzung von AspectJ kann auf verschiedene Arten erfolgen. Die wahrscheinlich gängigste Variante für die Definition eines *Join Point Model* für die eigene Software besteht in der Angabe von Annotations im Quellcode.

Listing 2.73 zeigt ein Beispiel für einen mit Annotations versehenen Advice: Mit der `@Pointcut`-Annotation der Methode `updatePositionPointcut` wird ein Pointcut für alle Aufrufe der Methode `updatePosition` der Klasse `com.example.Ship` definiert. Dieser Pointcut kann in der Folge über seinen Methodennamen `updatePositionPointcut` referenziert werden.

Advice-Arten in AspectJ

In AspectJ kann man über drei verschiedene Arten den Ausführungszeitpunkt eines Advice angeben. Für jeden Join point besteht die Möglichkeit, die eingewobene Funktionalität entweder davor, danach oder umschließend aufzunehmen. Bei der umschließenden Variante kann im Code, wie in Listing 2.73 zu sehen ist, mit dem Aufruf der Methode `proceed` entschieden werden, wann der umschlossene Code aufgerufen werden soll.

Folgende Advice-Arten bzw. zugehörige Annotations können Sie nutzen:

- `@Before` – vor dem aufgerufenen Join point
- `@Around` – um den aufgerufenen Join point
- `@After` – nach dem aufgerufenen Join point

Der Advice, also das Zusammenbringen von Pointcut und Funktionalität, wird im Beispiel mit der Methode `around` definiert. Sie besitzt die Annotation `@Around` mit der Angabe des entsprechenden Pointcuts `updatePositionPointcut` und enthält die einzuwebende Funktionalität:

```
@Aspect
public class ShipAdvice {

    @Pointcut("call(void com.example.Ship.updatePosition(*))")
    public void updatePositionPointcut() {
    }

    @Around("updatePositionPointcut()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("Calling " + pjp.getSignature().getName());
        Object methodResult = pjp.proceed();
        System.out.println("After " + pjp.getSignature().getName());    }
}
```

Listing 2.73 AspectJ-Advice für die »Ship«-Klasse (Java)

Wird an einem Objekt der Klasse `com.example.Ship` die Methode `updatePosition` aufgerufen, wird die eingewebte Funktionalität entsprechend aufgerufen:

```
public class Ship {
    ...
    public void updatePosition(Position position) {
        System.out.println("Updating position: " + position);
        this.position = position;
    }
...
```

Listing 2.74 Ausschnitt aus der Klasse »Ship« und »updatePosition« (Java)

```
public static void main(String[] args) {
    System.out.println("Starting");
    Position pos = new Position();
```

```
    Ship ship = new Ship("Titanic", pos);
    ship.updatePosition(pos);
}
```

Listing 2.75 Beispielanwendung mit AspectJ (Java)

Das Beispiel in Listing 2.75 gibt dementsprechend folgende Zeilen aus:

```
Starting
Calling updatePosition
Updating position: com.example.Position@5d41d522
After updatePosition
```

Listing 2.76 Ausgabe der Beispielanwendung

2.7 Mit Unit-Tests die Qualität sicherstellen

Software muss – unabhängig vom Softwaredesign oder der eingesetzten Programmierprinzipien – korrekt arbeiten und die definierten Anforderungen z. B. in Bezug auf Funktionalität, Leistungsfähigkeit oder Sicherheit umsetzen. Um das sicherzustellen, muss Software getestet werden.

Das geschieht mithilfe von Testfällen, in denen ein erwartetes Verhalten für zuvor definierte Parameter festgehalten wird. Eine erfolgreiche Überprüfung des erwarteten Verhaltens führt zu einem Testerfolg. Liefert die Software ein abweichendes Ergebnis bzw. verhält sie sich abweichend, ist der Test fehlgeschlagen.

Für jede moderne Programmiersprache steht mittlerweile mindestens eine Bibliothek oder ein Framework zum Erstellen von Tests zur Verfügung, und eine Ausführung ist in den meisten Entwicklungsumgebungen direkt möglich. Für Python können Sie beispielsweise das Framework *unittest*, für Java *JUnit* und für Go das im Standardumfang enthaltene Testframework verwenden. Neben den funktionalen Anforderungen können oftmals in den Umgebungen auch weitere Anforderungen getestet werden, z. B. Anforderungen an die Leistungsfähigkeit mithilfe von Benchmarks.

Die Definition von Testfällen ähnelt sich in allen Varianten. Listing 2.77 zeigt einen Python-Test, der zwei Testfälle definiert, in denen String-Methoden überprüft werden. Der erste Testfall nutzt die `assertEquals`-Methode, um Werte auf Gleichheit zu überprüfen. Beim ersten an die Methode übergebenen Wert handelt es sich um den tatsächlichen, berechneten Wert; der zweite Wert stellt das erwartete Ergebnis dar. Weichen die beiden Parameter voneinander ab, ist der Test fehlerhaft.

Der zweite Testfall arbeitet analog, allerdings mit `assertTrue` bzw. `assertFalse`, bei denen tatsächliche Werte auf einen booleschen Wert überprüft werden.

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

if __name__ == '__main__':
    unittest.main()
```

Listing 2.77 Unit-Test mit »unittest« für Python

Ändern sich die Implementierung der getesteten Methoden `isupper` oder `upper`, kann durch eine erneute Ausführung der Tests geprüft werden, ob die Testfälle weiterhin erfüllt werden und die Software weiterhin fehlerfrei funktioniert.

Eine Durchführung der Tests kann entweder durch manuelle Schritte oder automatisiert erfolgen, z. B. als Teil einer Build-Pipeline. Die zweite Variante einer automatisierten Ausführung ist in der aktuellen Softwareentwicklung verbreitet und kann als Standard angesehen werden.

Mit einer automatisierten und regelmäßigen Ausführung der Tests wird kontinuierlich sichergestellt, dass alle Anforderungen an die Software auch nach vorgenommenen Änderungen noch erfüllt werden. Das Ziel dieser sogenannten *Regressionstests* ist es, nicht bekannte und problematische Zusammenhänge zwischen Komponenten aufzudecken und sicherzustellen, dass auch nicht direkt von den Änderungen betroffene Komponenten weiterhin fehlerfrei arbeiten.

Oftmals werden Tests nach ihren Abhängigkeiten bzw. ihrer Isolation unterschieden. Betreffen Tests beispielsweise nur eine einzelne Einheit bzw. Komponente, spricht man von *Unit-Tests* oder *Komponententests*. Werden in einem Test mehrere Einheiten bzw. Komponenten zusammen getestet, spricht man beispielsweise von *Integrationstests*. Der Übergang zwischen den Begriffen ist meist fließend, und oft werden Sie zusätzliche Begriffe wie *Service-Tests* oder *UI-Tests* lesen. Wichtig ist, unabhängig von der Begrifflichkeit, bei der Erstellung eines Testfalls eine klare Grenze zwischen den Tests zu definieren und auch hier z. B. das *Single-Responsibility-Prinzip* für Tests anzuwenden.

Mike Cohn, einer der Mitbegründer der *Agile Alliance* und der *Scrum Alliance*, hat mit der sogenannten *Testpyramide* ein Konzept entwickelt, das Tests nach ihrer Ausführungsgeschwindigkeit und den durch sie verursachten Kosten hierarchisch in eine

Pyramide einordnet. Das Konzept besagt, dass Unit-Tests die breite Basis eines Testkonzepts darstellen sollen, da mit ihnen Software schnell und isoliert von anderen Komponenten getestet werden kann und sie somit eine rasche Rückmeldung zum Zustand der Software liefern. Tests mit einer höheren Integration zu anderen Komponenten werden langsamer ausgeführt und sollen demnach auch nicht so häufig durchgeführt werden, da sie kostspieliger sind. Abbildung 2.24 zeigt die Pyramide mit ihrer entsprechenden Einteilung.

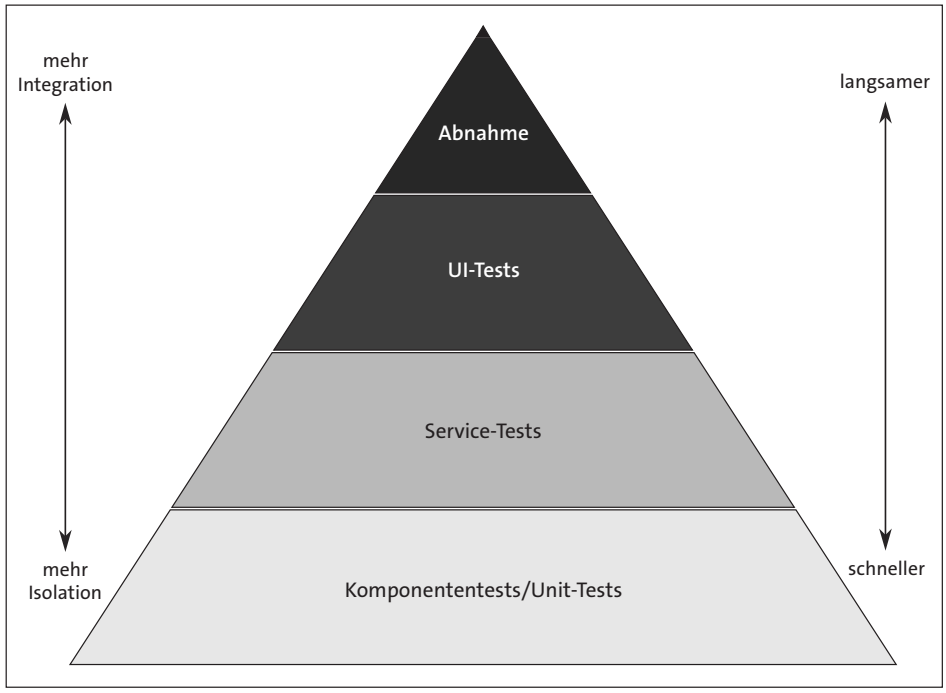


Abbildung 2.24 Testpyramide

Für effektives Testen müssen Softwarekomponenten dementsprechend so gestaltet sein, dass sie möglichst einzeln und isoliert getestet werden können. Dies erfordert ein Softwaredesign, das Komponenten modularisiert und ihre Abhängigkeiten klar definiert. Hierbei spielen die SOLID-Prinzipien und ihr Einsatz eine wichtige Rolle, da sie ein solches Design unterstützen. Gut aufgebaute Komponenten lassen sich deutlich einfacher testen als unstrukturierte und mit vielen Abhängigkeiten versehene Komponenten.

Insbesondere mit dem Konzept der *Dependency Injection* in Komponenten lässt sich der Testprozess vereinfachen, da die benötigten Abhängigkeiten zum Testzeitpunkt durch alternative Implementierungen ersetzt werden können. Auf diese Weise können spezifische Szenarien abgebildet und unerwünschte Nebenwirkungen ausgeschlossen werden.

Sogenannte *Mock-Frameworks*, die für die verschiedenen Programmiersprachen verfügbar sind, bieten die Möglichkeit, zum Testzeitpunkt Stellvertreter- bzw. *Mock-Objekte* für bestehende Interfaces oder Klassen zu erzeugen. Sie besitzen dieselbe Schnittstelle, können analog verwendet werden und bieten den Vorteil, dass ihr Verhalten während des Tests bzw. vor dem Test für den jeweiligen Testfall konfiguriert werden kann.

Listing 2.78 und Abbildung 2.25 zeigen die Klasse `ShipService`, deren Methode `getShipWithActualPosition` im Folgenden getestet wird. Der Service besitzt eine Abhängigkeit zum Interface `ShipTracker`, das zum Bestimmen der Position eines Schiffs die Methode `getPositionForShipName` anbietet. Getestet werden soll, ob während des Aufrufs der `ShipService`-Methode `getShipWithActualPosition` eine Implementierung des `ShipTracker`-Interfaces mit den richtigen Parametern aufgerufen wird und ob am Ende des Methodenaufrufs ein korrektes `Ship`-Objekt zurückgegeben wird.

Das Beispiel ist in Java mit *JUnit* implementiert und verwendet das *EasyMock*-Framework für die Generierung des Mock-Objekts.

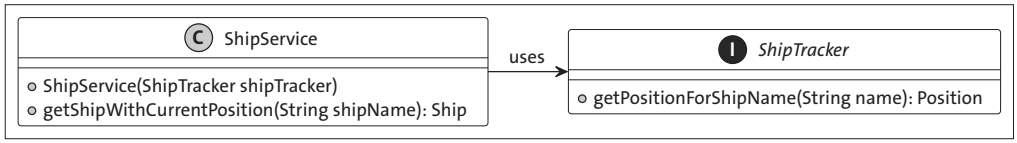


Abbildung 2.25 Aufbau der Klasse »ShipService«

```
public class ShipService {

    private ShipTracker shipTracker;

    public ShipService(ShipTracker tracker) {
        this.shipTracker = tracker;
    }

    public Ship getShipWithCurrentPosition(String shipName) {

        Position shipPosition =
            this.shipTracker.getPositionForShipName(shipName);
        return new Ship(shipName, shipPosition);

    }

}
```

Listing 2.78 Beispiel, das mit EasyMock getestet werden soll (Java)

Das Interface `ShipTracker` enthält nur eine Methode, die Sie in Listing 2.79 sehen:

```
public interface ShipTracker {
    public Position getPositionForShipName(String name);
}
```

Listing 2.79 Das »ShipTracker«-Interface (Java)

Innerhalb der Testmethode `testGetShipWithCurrentPosition` in Listing 2.80 wird keine »echte« `ShipTracker`-Implementierung verwendet, sondern ein von EasyMock erzeugtes *Mock-Objekt*. Erzeugt wird die Instanz über die JUnit-5-Extension von EasyMock und die Annotation `@Mock`, die für die Attributdefinition angegeben wird.

Vor jeder Testdurchführung wird durch die Methode `setUp` eine neue `ShipService`-Instanz erzeugt, die per Dependency Injection eine Referenz zum `ShipTracker`-Mock-Objekt übergeben bekommt.

Zu Beginn des eigentlichen Tests wird das Mock-Objekt so konfiguriert, dass er beim Aufruf der Methode `getPositionForShipName` ein `Position`-Objekt zurückgeliefert:

```
expect(shipTrackerMock.getPositionForShipName(shipName))
    .andReturn(new Position());
```

Am Ende des Tests wird mit der Methode `assertEquals` überprüft, ob das erzeugte `Ship`-Objekt den passenden Namen besitzt und ob die Methoden des Mock-Objekts tatsächlich aufgerufen wurden. Das übernimmt die `verify`-Methode aus der EasyMock-Bibliothek.

```
@ExtendWith(EasyMockExtension.class)
public class ShipServiceTest {

    ShipService shipService;

    @Mock
    ShipTracker shipTrackerMock;

    @BeforeEach
    public void setUp() {
        shipService = new ShipService(this.shipTrackerMock);
    }

    @Test
    public void testGetShipWithCurrentPosition() {

        //given
        String shipName = "Titanic";
```

```
        expect(shipTrackerMock.getPositionForShipName(shipName))
              .andReturn(new Position());
        replay(shipTrackerMock);

        //when
        Ship titanic = this.shipService.getShipWithCurrentPosition(shipName);

        //then
        assertEquals(shipName, titanic.getName());

        verify(shipTrackerMock);
    }

}
```

Listing 2.80 Unit-Test für »ShipService« mit EasyMock (Java)

Unit-Tests sind ein wichtiger Bestandteil einer professionellen Softwareentwicklung. Sie helfen, Fehler frühzeitig zu erkennen und die Qualität der Software zu verbessern, und legen potenzielle Seiteneffekte offen, die ohne die Tests eventuell erst auf Produktionssystemen auftreten würden.

Auch bei der Entwicklung von Softwarearchitekturen spielen Patterns eine entscheidende Rolle. Sie tragen – wie bereits beim Softwaredesign auf Sourcecode-Ebene – maßgeblich dazu bei, zentrale Eigenschaften und das grundlegende Verhalten von Anwendungen zu definieren, allerdings auf höherer Abstraktionsebene.

Manche dieser sogenannten *Softwarearchitektur-Patterns* eignen sich beispielsweise für hochskalierbare Anwendungen, während andere eher für agil entwickelte Anwendungen geeignet sind.

Um eine passende Architektur für die eigene Anwendung und deren fachlichen Anforderungen zu wählen, ist es wichtig, die Eigenschaften, Stärken oder Schwächen verschiedener Architekturansätze zu kennen.

Eine Klassifizierung von Architektur-Patterns findet mit sogenannten *Architekturstilen* statt, die anhand der eingesetzten Struktur oder der Organisation des Codes sowie der Interaktionen zwischen den verschiedenen Softwarekomponenten eine Einteilung vornehmen.

In Projekten ist meist eine Person in der Rolle des *Softwarearchitekten*. Sie trifft die Entscheidungen zu *Softwarearchitekturen* und stellt sicher, dass die Softwarearchitektur auch umgesetzt wird.

Innerhalb einer Anwendung können verschiedene Architekturen koexistieren. Ihr Einsatz ist nicht statisch, und man sollte sie an veränderte Anforderungen anpassen können. Auch wenn Martin Fowler betont, dass Architekturänderungen schwierig sind, müssen Architekturen flexibel genug sein, um sich an neue Herausforderungen oder veränderte Anforderungen anpassen zu lassen.

Innerhalb dieses Kapitels lernen Sie die vielfältigen Aufgaben des Softwarearchitekten sowie verschiedene Architekturstile, Architektur-Patterns und Patterns zur Unterstützung einer Implementierung dieser Architekturen kennen.

5.1 Die Rolle des Softwarearchitekten

Softwarearchitekten tragen die Verantwortung für die Definition und die erfolgreiche Umsetzung einer *Softwarearchitektur*. Sie stellen sicher, dass die Architektur den Anforderungen des Projekts entspricht und langfristig erfolgreich bleibt.

Simon Brown, Autor des Buchs *»Software Architecture for Developers«*, definiert praxisnah und pragmatisch sechs Aufgabengebiete der Softwarearchitekten-Rolle:

- Verständnis des fachlichen und technischen Kontexts
- Definition einer tragfähigen Softwarearchitektur
- Umgang mit technischen Herausforderungen und Risiken

- Weiterentwicklung der Architektur
- aktive Beteiligung an der Entwicklung
- Qualitätssicherung

Er betont, dass Softwarearchitekten nicht nur Entscheidungen treffen, sondern diese auch (mit) umsetzen müssen. Aus seiner Sicht dürfen keine *Elfenbeinturm-Architekten* entstehen, die nicht zusammen mit dem Team an Lösungen und der Architektur arbeiten.

Sehen wir uns diese Punkte nun im Einzelnen an.

Verständnis des fachlichen und technischen Kontexts

In einem Projekt ist es die Aufgabe eines Softwarearchitekten, die fachlichen und technischen Anforderungen an die Software zu kennen und zu überblicken. Diese bilden die Grundlage für alle seine weiteren Entscheidungen im Architekturprozess.

Obwohl manche technischen Rahmenbedingungen oder nicht-fachliche Anforderungen großen Einfluss auf eine Softwarearchitektur haben können, werden sie oftmals nur vage oder nicht konkret formuliert. Daher muss der Softwarearchitekt diese Restriktionen identifizieren, aufnehmen und in Bezug auf die Softwarearchitektur bewerten sowie deren Auswirkungen kommunizieren.

Definition einer tragfähigen Softwarearchitektur

Der Softwarearchitekt entwirft die Struktur einer Anwendung, indem er das Kontextwissen der Anwendung nutzt, um die zugrunde liegenden fachlichen Herausforderungen effektiv zu lösen. Dabei muss er stets die technischen Einschränkungen sowie das Potenzial für eine Weiterentwicklung der Architektur berücksichtigen.

Zu den technischen Einschränkungen zählen im Unternehmensumfeld oftmals Vorgaben zur eingesetzten Programmiersprache, zu den Frameworks, die verwendet werden sollen, oder Vorgaben zur Ausführungsumgebung.

Manche serverbasierten Softwaresysteme werden in einem eigenen Rechenzentrum ausgeführt, andere wiederum werden bei einem der großen Hyperscalern betrieben. So müssen viele verschiedene Aspekte berücksichtigt werden und in die Entscheidung für eine bestimmte Architektur einfließen.

Jede Entscheidung und die Architektur selbst müssen in adäquater Form dokumentiert werden, damit sie jederzeit diskutiert und kommuniziert werden können. Die entsprechenden Themen habe ich in Kapitel 3, »Sourcecode und Dokumentation der Softwareentwicklung«, vorgestellt.

Umgang mit technischen Herausforderungen und Risiken

Jede Entscheidung in einer Softwarearchitektur kann sich möglicherweise als ungünstig oder falsch erweisen und im Nachhinein zu Problemen führen. Das können

Entscheidungen zur gewählten technologischen Basis, aber auch zur gewählten Softwarearchitektur sein. Nicht jede neue Technologie, jeder Ansatz oder jedes gekaufte Produkt erfüllt alle Versprechungen der Werbeslogans.

Deshalb erfordert die Auswahl einer Technologie oder eines Architekturansatzes sorgfältige Überlegungen und sollte viele verschiedene Einflussfaktoren berücksichtigen. Unabhängig von der Entscheidungsfindung bleibt ein Restrisiko, mit dem konstruktiv umgegangen werden muss.

Es ist Aufgabe des Architekten, diese Entscheidungen zu treffen, sie zu hinterfragen, zu validieren und gegebenenfalls anzupassen. Viele Probleme treten erst bei einem Einsatz der jeweiligen Technologie oder des jeweiligen Architekturansatzes auf, obwohl sie in der Theorie vielversprechend klangen.

Entscheidungen in einer Softwarearchitektur sind nicht statisch, und Softwarearchitekten sollten proaktiv potenzielle Probleme aufdecken und diese möglichst früh im Lebenszyklus einer Anwendung entschärfen.

Weiterentwicklung der Architektur

Wie ich bereits beschrieben habe, sollte eine Architektur nicht als statisch und unflexibel wahrgenommen werden. Anforderungen können sich während der Laufzeit eines Projekts ändern oder getroffene Annahmen können sich als wenig praktikabel oder nicht umsetzbar erweisen.

In manchen Fällen ist es sinnvoll, getroffene Entscheidungen noch anzupassen. Architekten sollten deshalb nicht nur anfänglich in Projekten mitwirken und Architekturen vorgeben, sondern kontinuierlich an deren Entwicklung beteiligt sein und gegebenenfalls die Weiterentwicklung der Architektur vorantreiben und begleiten.

Manchmal braucht es auch neue, kreative Lösungsideen, denn für jede Anwendung dieselbe Lösungsschablone zu verwenden, ist oftmals nicht zielführend.

Aktive Beteiligung an der Entwicklung

Softwarearchitekten sollten sich aktiv an der Softwareentwicklung beteiligen, um genauestens zu verstehen, wie die gewählte Architektur in der Praxis von den Entwicklern wahrgenommen und umgesetzt wird.

Das bedeutet nicht, dass Architekten vollständig in den Entwickleralltag integriert werden müssen, aber eine Beteiligung am Entwicklungsprozess und eine aktive Einbindung in ihn ist ratsam.

Qualitätssicherung

Keine Technologie und kein Architekturansatz garantiert fehlerfreie Software. Qualität ist ein explizites Ziel, das stetig überprüft und verbessert werden muss.

Die Verantwortung für die Qualität von Software liegt nicht bei einer einzigen Person oder Rolle, sondern ist eine gemeinschaftliche Aufgabe, die von mehreren Beteiligten

erledigt werden muss. Die Aufgaben eines Softwarearchitekten beinhalten dabei unter anderem:

- die Dokumentation und Kommunikation der Softwarearchitektur und der ihr zugrunde liegenden Entscheidungen
- regelmäßige Code-Reviews und technische Unterstützung des Entwicklungsteams
- das Vorgeben technischer Standards und Richtlinien sowie die Überprüfung, ob diese eingehalten werden
- die kontinuierliche Überprüfung und Validierung der gewählten Softwarearchitektur
- das Treffen von Designentscheidungen, die z. B. die Performance, Sicherheit und Skalierbarkeit der Anwendung gewährleisten

5.2 Softwarearchitekturstile

Ein *Architekturstil* legt auf höherer Abstraktionsebene eine grundlegende Struktur für die Organisation von Softwarekomponenten und deren Beziehungen untereinander fest. Jeder Stil basiert dabei auf spezifischen Prinzipien und Vorgehen, um bestimmte Anforderungen zu erfüllen, wie beispielsweise Skalierbarkeit, Modularität oder Wartbarkeit.

Architekturstile können auf verschiedene Weisen gruppiert werden. Die folgenden Softwarearchitekturen repräsentieren verschiedene Stile, die sich auf die grundlegende Struktur und Interaktion zwischen den einzelnen Komponenten konzentrieren.

5.2.1 Client-Server-Architektur

Eines der grundlegenden Konzepte bei verteilten Anwendungen und Diensten ist das sogenannte *Client-Server-Modell*. Dabei handelt es sich um ein Verteilungsmodell, in dem Anwendungen oder Dienste von einer sogenannten *Server*-Komponente für eine *Client*-Komponente zur Verfügung gestellt werden. Zwischen den Komponenten findet die Kommunikation über ein Netzwerk statt. Die Kommunikation basiert auf einem Request-Response-Mechanismus, bei dem ein Client eine Anfrage an den Server sendet, die der Server anschließend beantwortet.

Bei den Clients kann man zwischen verschiedenen Typen unterscheiden. Zu den wichtigsten zählen:

- **Thin Clients** – Bei einem Thin Client handelt es sich um einen Client, der nur minimale Ressourcen verbraucht und maßgeblich die Ressourcen des Servers nutzt. Ein Beispiel für Thin-Clients sind Webbrowser, wie z. B. Mozilla Firefox oder Google Chrome, die klassische serverseitige Webapps ohne viel JavaScript anzeigen.

- **Thick oder Fat Clients** – Als Thick bzw. Fat Client wird ein leistungsfähiger Client bezeichnet, der weniger Ressourcen des Servers benötigt, da Verarbeitungs- und Speicheraufgaben lokal auf dem Endgerät erledigt werden. Beispiele für Fat Clients sind das lokal ausgeführte Microsoft Office oder Videospiele, die lokal sehr viel Rechenleistung erfordern.

Sind Single Page Applications Thin Clients oder Fat Clients?

Eine sogenannte *Single Page Application* (SPA) ist eine Webanwendung, bei der alle Inhalte und Funktionen innerhalb einer einzigen HTML-Seite angezeigt werden. Anstatt wiederholt die gesamte Seite zu laden, werden benötigte Daten und Oberflächenelemente dynamisch über JavaScript nachgeladen und angezeigt.

Da meist ein großer Teil der Anwendungslogik vom Server auf den Client ausgelagert wird, werden SPAs oftmals als Fat Clients betrachtet. Eine klare Abgrenzung zwischen Fat oder Thin Client ist jedoch schwer und hängt stark damit zusammen, welche Komponente den Großteil der Verarbeitung übernimmt.

Im Client-Server-Modell wartet ein Server aktiv auf Anfragen der Clients, wobei die vom Server hierfür angebotene Dienstleistung auch als *Dienst* bzw. *Service* bezeichnet wird.

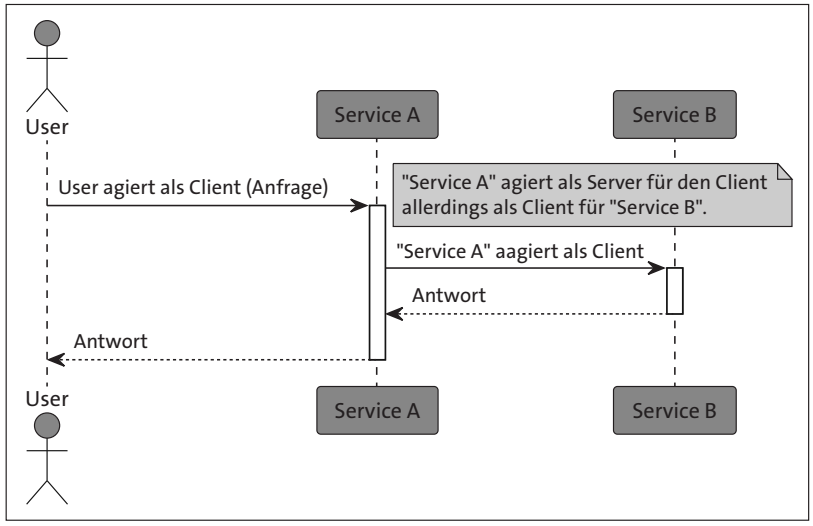


Abbildung 5.1 Beispiel für ein Client-Server-Modell mit mehreren Clients

Für die Kommunikation zwischen den Partnern wird ein dienstspezifisches Protokoll verwendet.

Die meisten modernen Softwarearchitekturen basieren auf dem Client-Server-Modell, und oftmals können Komponenten sowohl als Server wie auch als Client auftreten. Wie Sie in Abbildung 5.1 sehen, wird `Service A` vom User als *Service* aufgerufen.

`Service A` ruft allerdings wiederum `Service B` auf und tritt in dieser Kommunikation als *Client* auf.

5.2.2 Schichtenarchitektur und Service-Layer

Die *Schichtenarchitektur*, auch als *n-Tier-Architektur* bekannt, ist eines der verbreitetsten Architekturmuster in der Informationstechnologie, insbesondere in Enterprise-Anwendungen.

Durch die Nähe zu traditionellen Kommunikations- und Organisationsstrukturen von Unternehmen ist der Ansatz für die meisten Entwickler und Architekten ein vertrauter Standard. Oftmals findet man in größeren Unternehmen z. B. Datenbankteams, Geschäftslogik- oder Serviceteams sowie Frontend-Teams, die sich jeweils um die Belange eines speziellen Bereichs bzw. einer Schicht kümmern.

Die Einteilung einer Software in einzelne Schichten, die diesen Strukturen entsprechen, ist dementsprechend eine recht einfache Technik, um ein komplexes System in kleinere, gut überschaubare Einheiten zu untergliedern. Jede Schicht baut dabei, wie in Abbildung 5.2 gezeigt, auf einer darunterliegenden Schicht auf und nutzt die zur Verfügung gestellten Services bzw. Schnittstellen. Für die aufgerufene Schicht ist der Aufruf transparent. Sie kennt die auf ihr aufbauende bzw. aufbauenden Schichten allerdings nicht. Die Abhängigkeit ist immer nur in eine Richtung definiert: von einer höheren zu einer niedrigeren Schicht.

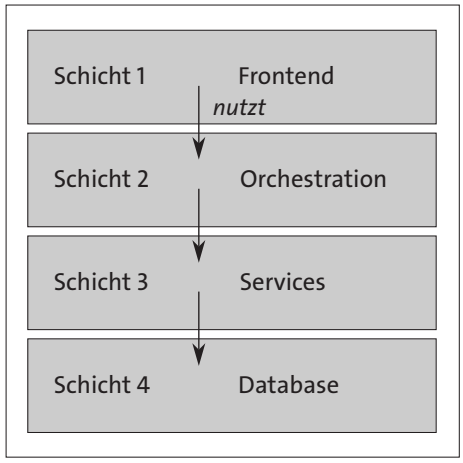


Abbildung 5.2 Schichtenarchitektur

Jede Schicht stellt eine eigene separate Schnittstelle zur Verfügung und kapselt darüber die unter ihr liegenden Schichten. Im obigen Beispiel nutzt Schicht 2 die Services von Schicht 3, die ihrerseits auf die Datenbank in Schicht 4 zugreift. Für Schicht 2 ist der Zugriff auf Schicht 4 transparent und die Schicht ist sich dessen nicht bewusst.

Das Konzept zur Isolation und Trennung der einzelnen Schichten wird mit dem Begriff *Isolationsschichten* bzw. *Layers of Isolation* beschrieben und fördert die Wartbarkeit und Erweiterbarkeit der Software. Änderungen in einer Schicht wirken sich durch die strikte Trennung und klaren Schnittstellen nicht direkt auf weitere Schichten aus. Die Vorteile dieses Konzepts bestehen zusätzlich unter anderem aus:

- der klaren Definition der Verantwortlichkeiten für jede Schicht und damit der Umsetzung des Prinzips *Separation of Concern*. Die einzelnen Schichten können separat besser getestet und verwaltet werden.
- Einzelne Schichten können separat verstanden werden, ohne das »große Ganze« zu kennen. Beim Speichern einer Datei über ein Betriebssystem, das auch in verschiedene Schichten untergliedert ist, muss man sich beispielsweise nicht mit den Details des Dateisystems oder der Verwaltung der Daten auf einer SSD-Festplatte befassen. Die Speicher-API kann einzeln verstanden und ohne Detailwissen über die restliche Speicherung angewendet werden.
- Schichten können durch adäquate Implementierungen mit den gleichen Schnittstellen ausgetauscht werden. Eine Anwendung, die beispielsweise Daten über die Betriebssystem-API speichert, kann ohne Anpassungen mit SSD-Festplatten oder mechanischen HDD-Festplatten arbeiten, solange die Schnittstelle gleich bleibt.
- Abhängigkeiten zwischen den Schichten werden minimiert. Mit der Abstraktion und Kapselung zwischen den Schichten werden die einzelnen Bestandteile unabhängiger, und komplexe Abhängigkeiten über mehrere Schichten hinweg werden unterbunden.
- Die Definition von Schichten mit expliziten Schnittstellen bietet Möglichkeiten zur Standardisierung. So wird beispielsweise mit dem *Open Systems Interconnection Model* (abgekürzt: OSI-Schichtenmodell) eine Schichtenarchitektur für Netzwerkprotokolle definiert, die z. B. für das Verständnis von Standards wie TCP oder IP nützlich ist.
- Auf einer Schicht können mehrere Schichten aufbauen. Im Beispiel aus Abbildung 5.2 baute auf einer Schicht immer genau eine weitere Schicht auf. In einer Schichtenarchitektur können aber auch mehrere Schichten auf einer Schicht basieren. Soll z. B. eine Anwendung mehrere Benutzerschnittstellen besitzen, können mehrere parallele Frontend-Schichten erstellt werden, z. B. eine für Webbrowser-basierte Oberflächen und eine für Handy-Apps.

Die Nachteile einer Schichtenarchitektur sind jedoch:

- Oftmals kapseln einzelne Schichten die darunterliegenden Schichten nicht ausreichend und Änderungen müssen über mehrere Schichten hinweg nachgezogen werden. In manchen Projekten werden beispielsweise Objekte, die in der Benutzer-

schnittstelle verwendet werden, aus einem Datenbank-Schema generiert und brechen damit das Schichtenmodell auf.

- Viele Schichten können die Performance eines Systems negativ beeinflussen.

3-Schichten-Architektur

Schon in den Anfängen der Softwareentwicklung wurde die Aufteilung von Anwendungen in hierarchisch angeordnete modulare Schichten eingesetzt, das wachsende Maß an Komplexität zu bewältigen. In den 1990er-Jahren erlebte diese Entwicklung einen Meilenstein mit der Verbreitung der *Client-Server-Architektur*. Diese trennte Aufgaben zwischen Client und Server und nutzte häufig eine Datenbank als zentrale Datenquelle auf der Serverseite.

Bei den Client-Anwendungen handelte es sich meist um Desktop-Anwendungen, die als sogenannte *Rich Clients* ausgelegt waren und Daten lokal auf dem Clientgerät verarbeiteten oder auch größere Datenmengen zwischenspeicherten.

Viele Hersteller boten für die Implementierung dieser Architekturen Komponentenbibliotheken oder auch komplette Entwicklungsumgebungen für die unterschiedlichsten Betriebssysteme an. In ihnen konnte man über grafische Werkzeuge per Drag-and-drop eine Anwendung erstellen. Die Komponentenbibliotheken umfassten unter anderem UI-Komponenten, die es ermöglichten, direkte SQL-Datenbankanbindungen zu konfigurieren und Anwendungen schnell zu erstellen. Beispiele hierfür sind Entwicklungsumgebungen wie *Borland Delphi*, *Microsoft Visual Basic* und *PowerBuilder* von Sybase.

Der beschriebene Ansatz führte allerdings zu Problemen bei der Wiederverwendung von Code der Geschäftslogik. Denn oftmals wurde diese Logik direkt – und damit voneinander isoliert – in die einzelnen Frontend-Seiten bzw. Formulare einer Anwendung implementiert. Das erschwert die Wiederverwendung von bereits existierenden Komponenten und erhöht den Aufwand für Wartungsarbeiten und Erweiterungen, da Änderungen an mehreren Stellen vorgenommen werden müssen. Je komplexer die Anwendung wird, desto deutlicher zeigt sich diese Problematik.

Teilweise wurden daraufhin Logikblöcke direkt in der Datenbank als *Stored Procedures* implementiert, während andere Blöcke in spezielle Komponenten bzw. Bibliotheken ausgegliedert und gemeinsam genutzt wurden. Stored Procedures sind immer datenbankabhängig, und durch ihre Nutzung verliert man die Möglichkeit einer einfachen Portierung zu einer alternativen Persistenz-Technologie. Die Verwaltung der zentralen Bibliotheken und die Unterstützung mehrerer Versionen gleichzeitig wurde ebenfalls zu einer Herausforderung.

In der Folge wurden die sogenannten *3-Schichten-Architekturen* immer beliebter. Dies wurde vor allem durch die Verbreitung der objektorientierten Programmiersprachen und Technologien begünstigt, wie z. B. *Java* und *JavaEE*. Außerdem mussten mit der

Verbreitung des Internets Anwendungen plötzlich neben einer Desktop-Anwendung parallel auch Webinterfaces besitzen. Logik, die nur innerhalb einer Desktop-Anwendung implementiert war, war dazu hinderlich, da sie nicht zusätzlich in einem weiteren Client wiederverwendet werden kann.

Klassischerweise wurden und werden 3-Schichten-Architekturen, deren einzelne Schichten teilweise auch physikalisch getrennt sind, wie folgt untergliedert:

- **Präsentationsschicht** – Über diese Schicht werden Dienste oder Informationen dem Benutzer zur Verfügung gestellt. Dies kann über Rich Clients, HTML-basierte Oberflächen oder auch über Programmierschnittstellen erfolgen.
- Die **Geschäftslogikschicht** bzw. **Domänenschicht** ist der Kern der Anwendung und enthält die eigentliche Logik sowie alle Prozesse und Regelwerke, die für den Geschäftsbetrieb essenziell sind. Hier werden Daten aus der Datenbank verarbeitet, Berechnungen durchgeführt, Prüfungen auf Richtigkeit oder Gültigkeit vorgenommen und Entscheidungen auf Basis der implementierten Regeln getroffen.
- **Datenhaltungsschicht** bzw. **Datenquelle** – Diese Schicht übernimmt die Kommunikation mit externen Systemen, die verschiedene Aufgaben für die Anwendung erfüllen oder Daten speichern. Häufig werden Datenbanken als externe Systeme genutzt.

Die Betrachtung der Abhängigkeiten zwischen den Schichten ist genauso wichtig wie die Festlegung ihrer Verantwortlichkeiten. In einer Schichtarchitektur sollten Abhängigkeiten immer nur in eine Richtung verlaufen und niemals bidirektional. In der 3-Schichten-Architektur bedeutet das beispielsweise: Die Präsentationsschicht kann auf die Geschäftslogikschicht zugreifen, aber nicht umgekehrt. Mit dieser Regel kann potenziell eine Präsentationsschicht oder eine Datenhaltungsschicht leicht durch eine alternative Implementierung ersetzt werden, ohne Geschäftslogik nachimplementieren zu müssen.

n-Schichten-Architekturen

Schichten-Architekturen müssen nicht aus genau drei Schichten bestehen. Mark Richards fügt in seinem Buch *»Software Architecture Patterns«* beispielsweise immer eine vierte Schicht zur klassischen 3-Schichten-Architektur hinzu:

- Präsentation
- Geschäftslogik
- Speicherung/Persistenz
- Datenbank

Die Anzahl der Schichten ist nicht relevant, solange die Schichtentrennung und deren hierarchischer Aufbau eingehalten werden. Man spricht oftmals einfach nur von *n-Schichten-Architekturen*.

Das Prinzip der offenen und geschlossenen Schichten

Ein Schlüsselprinzip der Schichtenarchitektur ist nach Mark Richards, dem Autor des Buchs *»Software Architecture Patterns«*, die Einteilung der einzelnen Schichten in offene bzw. geschlossene Einheiten. Er bezieht sich nicht auf das obligatorische *Open-Closed-Prinzip*, dass eine Schicht bzw. ihre Schnittstelle offen für Erweiterungen und geschlossen gegenüber Änderungen sein soll. Richards beschreibt mit den Begriffen, welche Schichten an der Bearbeitung einer Anfrage beteiligt werden oder ob beispielsweise eine Schicht übersprungen werden kann.

In Abbildung 5.3 besteht die Schichtenarchitektur aus vier *geschlossenen Schichten*. Das heißt, dass eine Anfrage, die von der Schicht 1 entgegengenommen wird, durch alle darunterliegenden Schichten verarbeitet bzw. durchgereicht wird, bis sie Schicht 4, die Datenbankschicht, erreicht. Jede Schicht ist an der Abarbeitung der Anfrage beteiligt und stellt die Trennung der einzelnen Schichten sicher.

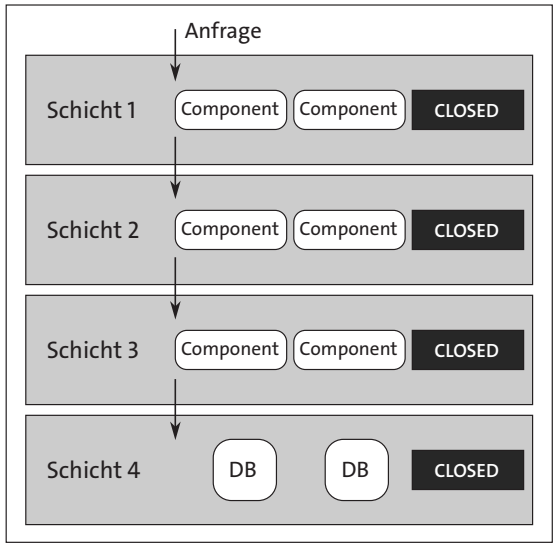


Abbildung 5.3 Geschlossene Schichten

Die strikte Trennung und Isolation der Schichten ist essenziell für eine gute Wartbarkeit der Anwendung und sorgt dafür, dass sich Änderungen an einer Schicht sich nicht im kompletten System ausbreiten und nachgezogen werden müssen.

In manchen Situationen ist es allerdings sinnvoll, sogenannte *offene Schichten* zu definieren, bei denen Anfragen nicht durch diese hindurch, sondern an ihnen vorbeigeleitet werden dürfen. In Abbildung 5.4 ist beispielsweise die Schicht 2 als offen gekennzeichnet und eine Anfrage von Schicht 1 könnte in diesem Fall direkt an Schicht 3 weitergegeben werden, ohne in Schicht 2 verarbeitet zu werden.

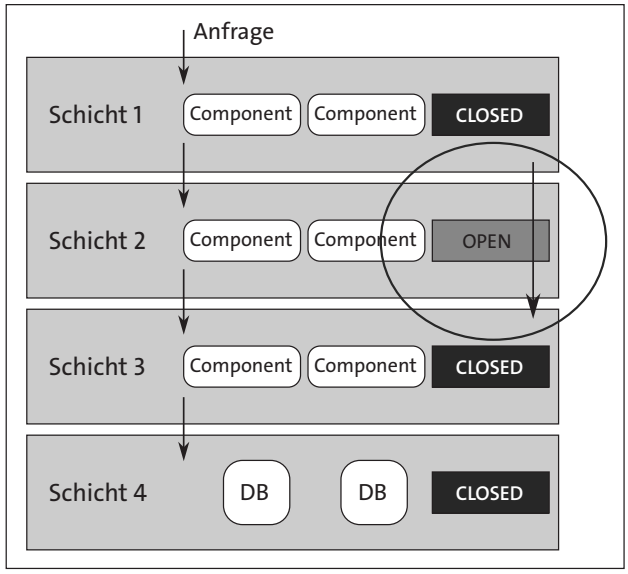


Abbildung 5.4 Geschlossene und offene Schichten

Wenn beispielsweise gemeinsam genutzte Geschäftskomponenten in die Architektur integriert werden sollen, die sowohl von der bestehenden Geschäftslogikschicht als auch von der bisherigen Präsentationslogik genutzt werden sollen, dann kann eine zusätzliche *Serviceschicht* bzw. ein sogenannter *Service-Layer* definiert werden. Diese Schicht befindet sich architektonisch unterhalb der Geschäftslogikschicht und ist somit für die Präsentationslogik nicht direkt erreichbar. Durch die Definition der neuen Schicht wird bereits über die Schichtenarchitektur eine Zugriffsbeschränkung für die neue Schicht erreicht.

Allerdings müsste mit der Einführung der neuen Schicht, die ja unterhalb der Geschäftslogikschicht liegt, nun jeder Aufruf durch diese Schicht verarbeitet werden. In diesem Fall ist es sinnvoll, die Schicht als offen zu kennzeichnen und Aufrufe direkt an die nächsttiefere Schicht an ihr vorbeizuleiten.

Der Einsatz von offenen Schichten muss wohlüberlegt erfolgen. Zu viele offene Schichten und direkte Zugriffe auf andere Schichten können ein System und dessen Komponenten unübersichtlich und stark gekoppelt werden lassen. Die Schichtenarchitektur verliert dadurch ihre Vorteile, und Systeme werden anfälliger für ausufernde Änderungen und somit auch schwerer zu testen und zu warten.

Open-Closed-Schichten und Patterns

Offene und geschlossene Schichten werden manchmal im Rahmen von Patterns definiert. Für JavaEE wurden im Jahr 2001 z. B. die *»Design Patterns for Optimizing the Performance of J2EE Applications«* veröffentlicht. Diese Auflistung von einigen Patterns

zur Optimierung von Java-Enterprise-Anwendungen enthält beispielsweise das sogenannte *Fast-Lane-Reader Pattern*, das eine Art Überholspur und die Umgehung von mehreren Schichten für schnellere Datenbankzugriffe beschreibt. In diesem Fall werden für Lesezugriffe alle Schichten als offen betrachtet.

Mittlerweile gilt dieses Pattern jedoch eher als Anti-Pattern: Eine vollständige Umgehung aller Schichten ist nicht zu empfehlen, da sie die gesamte Schichtenarchitektur untergräbt.

5.2.3 Event-Driven Architecture

Eine *Event-Driven Architecture* (EDA) ist ein Architekturstil, der auf der Verarbeitung von Ereignissen, sogenannten *Events*, basiert. Verarbeitung bedeutet hier die Art und Weise, wie auf bestimmte Ereignisse reagiert wird. Jedes Event stellt dabei eine Zustandsänderung innerhalb eines Systems dar, wie z. B. die Änderungen einer Kundenadresse.

Die Architektur besteht aus unabhängigen Komponenten, die über Ereignisse miteinander kommunizieren und jeweils Daten empfangen und verarbeiten. Dadurch, dass die Kommunikation asynchron stattfindet, können die Komponenten flexibel und unabhängig voneinander verwendet und skaliert werden.

Dieser Ansatz eignet sich sowohl für große als auch für kleine Anwendungen und bietet die folgenden Vorteile:

- **Entkopplung der einzelnen Komponenten** – Die einzelnen Komponenten empfangen und verarbeiten einzelne Ereignisse. Die Kommunikation zwischen ihnen erfolgt über eine Abstraktionsschicht, die sogenannten *Channels* (siehe auch im Kapitel zur Kommunikation zwischen Services). Solange das Datenformat der ausgetauschten Ereignisse stabil bleibt, erfordern Änderungen an einer Komponente keine zusätzlichen Anpassungen an weiteren Komponenten.
- **Erweiterungen der Funktionalität** – Neue Features können durch das Hinzufügen neuer Komponenten aufgenommen werden, ohne bestehende Komponenten anpassen zu müssen.
- **Skalierbarkeit** – In einer EDA werden die Ereignisse asynchron verarbeitet. Dadurch können sie parallel oder bei Lastspitzen zeitverzögert verarbeitet werden.
- **Flexibilität** – Komponenten können unabhängig voneinander entwickelt, erweitert und zusammengestellt werden. Dadurch lässt sich Funktionalität flexibel anpassen, aufnehmen oder entfernen.
- **Echtzeitverarbeitung** – Ereignisse können direkt verarbeitet werden, was die Erstellung sogenannter *reaktiver Anwendungen* ermöglicht.

Die Kommunikation zwischen den Komponenten kann in einer Event-Driven Architecture mit zwei Topologien bzw. Patterns umgesetzt werden:

- *Message Processor-* bzw. *Mediator-Topologie*
- *Message Broker-* bzw. *Broker-Topologie*

Bei der *Mediator-Topologie* wird ein zentraler Vermittler, der sogenannte *Mediator* oder *Message Processor* eingesetzt. Dieser steuert die Koordination und Verarbeitung von Ereignissen und orchestriert so die Nachrichtenverarbeitung innerhalb der Komponenten. Dieser Ansatz ist besonders für komplexere Workflows innerhalb einer Anwendung geeignet.

Die *Broker-Topologie* hingegen verteilt die Ereignisse über einen zentralen *Event-Broker*, der ausschließlich für die Weiterleitung und Verteilung der Ereignisse zuständig ist.

Oftmals wird eine Event-Driven Architecture mit Messaging-Technologien umgesetzt, die wir in Kapitel 6, »Kommunikation zwischen Services«, noch ausführlicher betrachten.

Message Processor

Die Orchestrierung eines Prozesses durch einen zentralen Mediator bzw. *Process Manager* ist immer dann sinnvoll, wenn ein Ereignis von mehreren Komponenten in einem komplexeren Workflow verarbeitet werden muss.

Löst z. B. ein Kunde eine Bestellung in einem Shop-System aus, kann ein zentraler Process Manager die weiteren Schritte orchestrieren:

- Validierung der Bestellung, indem die Verfügbarkeit des Artikels geprüft wird und dieser für den Kunden reserviert wird
- Kommunikation mit dem Zahlungsdienstleister für die Zahlungsabwicklung
- Versandvorbereitung
- Rechnungserstellung und Versand an Kunden
- Mailversand an Kunden mit Einladung zum Feedback-Formular

Im beschriebenen Fall sind die Reihenfolge der Einzelschritte und die Abhängigkeit zwischen ihnen entscheidend, da der Versand erst nach erfolgreicher Zahlung erfolgen darf. Der Process Manager stellt sicher, dass die einzelnen Schritte koordiniert und korrekt ausgeführt werden, bevor das Event an weitere Komponenten weitergeleitet wird.

Der Einsatz eines zentralen Process Managers bzw. Mediators führt zum sogenannten *Hub-and-Spoke*-Nachrichtenfluss-Pattern, das in Abbildung 5.5 visualisiert ist.

Bei diesem Pattern startet eine sogenannte *Trigger Message* einen neuen Prozess innerhalb des Process Managers. Der Process Manager analysiert anschließend die Nachricht und bestimmt anhand der enthaltenen Logik und Regeln, welche Kompo-

nente die Verarbeitung übernehmen soll. Daraufhin leitet er die Nachricht an die entsprechende Komponente weiter. Sobald die Komponente ihre Aufgabe erledigt hat, sendet sie ihre Antwort an den Prozessmanager zurück. Dieser entscheidet erneut anhand der Nachricht, welche Komponente den nächsten Verarbeitungsschritt ausführt, und leitet die Nachricht entsprechend weiter.

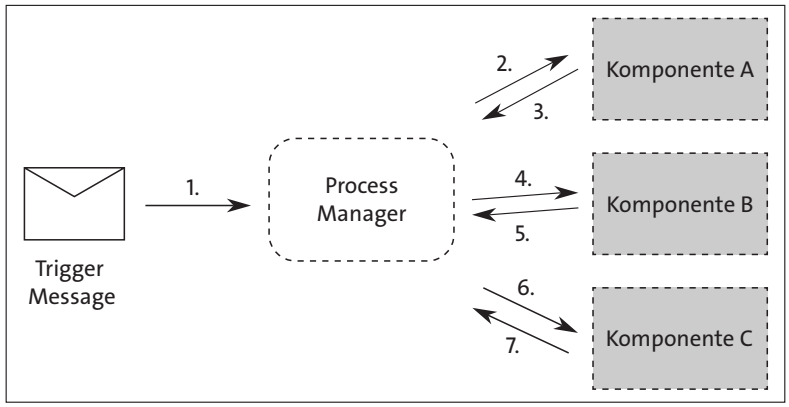


Abbildung 5.5 Event-Driven Architecture mit Process Manager

Der Process Manager führt keine eigene Geschäftslogik mit oder an den Nachrichten durch. Seine Aufgabe besteht ausschließlich in der korrekten Weiterleitung der Nachrichten an die lose gekoppelten, unabhängigen Komponenten, die sehr spezifische Aufgaben erfüllen.

Umgesetzt werden kann ein zentraler Process Manager bzw. Mediator entweder mit einer Eigenimplementierung – oder Sie greifen auf bestehende Softwareprodukte und Bibliotheken zurück. Verschiedene kommerzielle und Open-Source-Produkte können hierzu eingesetzt werden. Unter anderem sind das:

- *Spring Integration*
- *Apache Camel*
- *IBM App Connect Enterprise*
- Redux-Middleware wie *Redux Thunk*

Vorteile

- Unabhängige, lose gekoppelte Komponenten können integriert werden.
- flexible und vielfältige Einsatzmöglichkeiten des Managers, z. B. die Ausführung von parallelisierten Prozessen, wodurch die meisten Interaktionsabläufe realisiert werden können
- zentrale Verwaltung der einzelnen Geschäftsprozesse und bessere Überwachungsmöglichkeiten
- Verringerung der Direktverbindungen zwischen verschiedenen Systemen

Nachteile

- Die Einführung eines zentralen Process Managers birgt die Gefahr eines *Single-Point-of-Failure*.
- Der Process Managers enthält eine umfangreiche Logik und zahlreiche Regeln für die Nachrichtenverteilung, die dort auch verwaltet werden müssen.
- Die Architektur wird im Einsatz zunehmend komplexer, da zusätzliche Komponenten verwaltet und überwacht werden müssen.

Message Broker

Bei der *Message-Broker*-Topologie werden die Nachrichten ebenfalls über eine zentrale Stelle, den sogenannten *Message Broker*, ausgetauscht. Dieser enthält allerdings keine Logik oder Regeln zum Abbilden eines Prozesses, sondern ist nur für die Nachrichtenvermittlung der einzelnen Komponenten zuständig. Die entsprechende Logik für den Nachrichtenfluss ist über die beteiligten Komponenten verteilt.

In der Message-Broker-Topologie verarbeiten die einzelnen Komponenten Ereignisse und erzeugen ihrerseits Ereignisse, die das Resultat der ausgeführten Aktion widerspiegeln (siehe Abbildung 5.6).

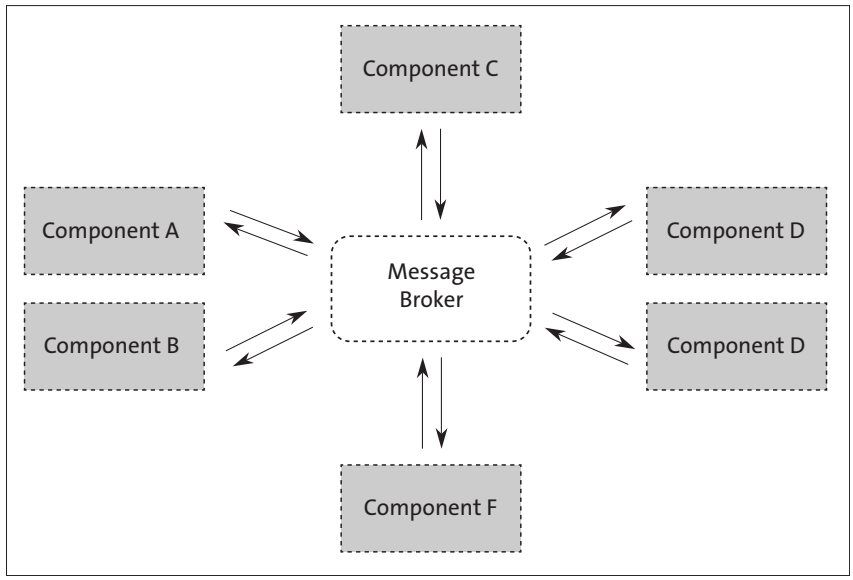


Abbildung 5.6 Aufbau einer EDA mit Message Broker

Wenn z. B. ein Kunde eine Bestellung in einem Shop-System auslöst, erzeugt die Warenkorb-Komponente das Ereignis »Neue Bestellung« und übermittelt es an den Message Broker. Alle Komponenten, die an diesem Ereignis interessiert sind, erhalten das Ereignis und können es verarbeiten. Als Ergebnis liefert die Validierungskompo-

nente das Ereignis »Bestellung erfolgreich validiert« und ermöglicht so der nächsten interessierten Komponente den weiteren Arbeitsfortschritt. Dieser Vorgang wiederholt sich fortlaufend, bis der Bestellprozess abgeschlossen ist oder keine interessierte Komponente mehr vorhanden ist.

Die Arbeitsweise dieser Topologie lässt sich gut mit einem Staffellauf vergleichen: Jede Komponente übergibt das Ereignis, nachdem sie es bearbeitet hat, an die nächste Komponente in der Kette.

Diese Topologie ist für Anwendungsfälle geeignet, in denen nur eine begrenzte Logik für die Nachrichtenverteilung erforderlich ist und eine zentrale Orchestrierung der Prozesse nicht erwünscht ist.

Für die Umsetzung einer Message-Broker-Topologie bieten sich verschiedene bewährte Softwareprodukte und Bibliotheken an. Unter anderem sind das:

- *Apache Kafka*
- *RabbitMQ*
- Googles *Cloud Pub/Sub*
- *AWS SNS und SQS*

Vorteile

- lose Kopplung der einzelnen Komponenten, da der Message Broker die Vermittlung der Nachrichten übernimmt
- Änderungen und das Hinzufügen neuer Komponenten sind ohne Anpassungen an bestehenden Komponenten möglich.
- asynchrone und flexible Verarbeitung der Nachrichten durch die Komponenten
- gute Möglichkeiten einer Skalierung der Komponenten, da diese auch parallel ausgeführt werden können
- Durch den zentralen Message Broker wird die Komplexität von vielen Einzelverbindungen zwischen Komponenten reduziert.

Nachteile

- In einigen Fällen ist lediglich die Nachrichtenübermittlung von der verwendeten Technologie unabhängig, während die Komponenten den nächsten Verarbeitungspartner in der Kette kennen, an den sie die Nachricht senden.
- Der Message Broker kann zu einem Engpass werden, da alle Nachrichten über ihn laufen müssen.
- steigende Komplexität der Architektur beim Einsatz, da zusätzliche Komponenten verwaltet und überwacht werden müssen

5.2.4 Microkernel Architecture bzw. Plugin Architecture

Die *Microkernel-Architektur* ist ein Architekturstil, bei dem ein Kernsystem über eine Erweiterungsschnittstelle um neue Funktionen ergänzt werden kann. Oftmals wird der Ansatz auch als *Plugin Architecture Pattern* bezeichnet.

Eine Anwendung wird dabei so strukturiert, dass einzelne Funktionsmodule, die sogenannten *Plugins*, nahtlos zu einer Kernanwendung hinzugefügt bzw. aus ihr entfernt werden können. Diese Plugins sind eigenständige Komponenten, die über definierte Schnittstellen mit der Hauptanwendung kommunizieren. Die sogenannten *Extension Points* definieren diese Schnittstellen. Abbildung 5.7 stellt das Modell dar.

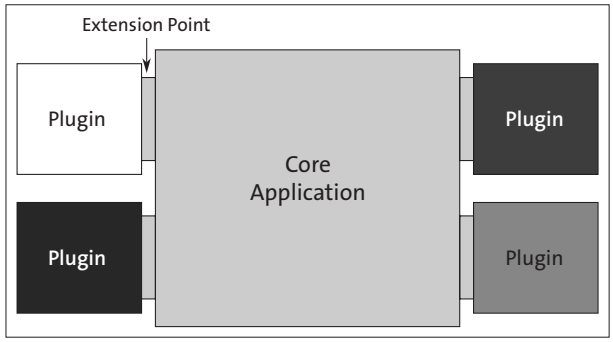


Abbildung 5.7 Microkernel- bzw. Plugin-Architektur

Mit diesem Ansatz kann neue Funktionalität hinzugefügt werden, ohne die Hauptanwendung anpassen zu müssen.

Das Pattern wird oftmals für die Entwicklung von Anwendungen eingesetzt, die als vollständig autonome Software ausgeliefert werden sollen, wobei einzelne Funktionen dennoch unabhängig entwickelt und in eine Hauptanwendung integriert werden können.

Der Einsatz einer Plugin-Architektur kann auch bei der Strukturierung von Serveranwendungen vorteilhaft sein, insbesondere in Kombination mit dem Konzept von *Vertical Slices* (siehe Abbildung 5.8).

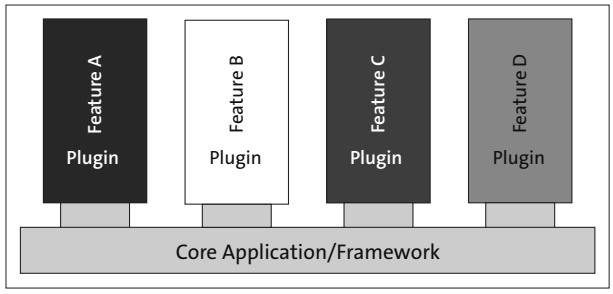


Abbildung 5.8 Plugin-Architektur mit Vertical Slices

Dabei werden besser isolierte und unabhängige Module geschaffen, die jeweils einen Geschäftsbereich abdecken. Diese Module, die als eigenständige Vertical Slices fungieren, kommunizieren miteinander über einen Mechanismus, der von der Kernanwendung oder dem zugrunde liegenden Framework bereitgestellt wird.

Folgende Open-Source-Projekte verwenden eine Microkernel-Architektur und nutzen einen Erweiterungsmechanismus:

- *Eclipse*-Plugins
- *Visual Studio Code*-Extensions
- *WordPress*-Plugins
- *Blender*-Plugins

Die Kernprinzipien des Patterns können wie folgt zusammengefasst werden:

- **Modularität und Wartbarkeit** – Durch die Aufteilung in unabhängige Module, die Plugins, die jeweils für eine spezifische Funktionalität zuständig sind, lässt sich die Anwendung einfacher warten und die Module beeinflussen sich nicht gegenseitig.
- **Erweiterbarkeit** – Ein Grundpfeiler dieser Architektur ist die Erweiterbarkeit. Entwickler können neue Funktionalität erstellen, ohne die Kernanwendung anpassen zu müssen. Ein solches Design gewährleistet die Anpassungsfähigkeit der Software an veränderte Anforderungen oder neue Technologien, ohne dass eine grundlegende Überarbeitung erforderlich ist.
- **Trennung der Verantwortlichkeiten** – Die Aufteilung der Anwendung in mehrere Module, die jeweils eine spezifische Funktionalität übernehmen, trennt dessen Verantwortlichkeiten (siehe Abschnitt 2.6, »Separation of Concerns und Aspektorientierung«).

Beispiel

Das folgende Beispiel stellt die Funktionsweise einer Microkernel-Architektur dar. Eine Anwendung liest dabei eine XML-Konfigurationsdatei mit Plugin-Konfigurationen ein und führt diese entsprechend der Plugin-Schnittstelle aus.

Das `Plugin`-Interface, das von allen Plugins implementiert werden muss, sehen Sie in Listing 5.1:

```
package plugins;

public interface Plugin {
    void execute();
}
```

Listing 5.1 Plugin-Interface (Java)

Innerhalb einer XML-Konfigurationsdatei müssen alle `Plugin`-Implementierungen aufgelistet werden, damit sie von der Kernanwendung genutzt werden können. Listing 5.2 zeigt eine solche Konfigurationsdatei:

```
<plugins>
    <plugin>
        <name>GreetingPlugin</name>
        <class>myplugins.GreetingPlugin</class>
    </plugin>
    <plugin>
        <name>AnotherPlugin</name>
        <class>myplugins.AnotherPlugin</class>
    </plugin>
</plugins>
```

Listing 5.2 Plugin-Konfigurationsdatei

Die Kernanwendung in Listing 5.3 lädt über einen `PluginLoader` die einzelnen konfigurierten Plugins und führt deren Funktionalität in einer Schleife aus:

```
package app;

import plugins.Plugin;
import java.util.List;

public class MainApplication {
    public static void main(String[] args) {
        String xmlPath = "plugins.xml";
        List<Plugin> plugins = PluginLoader.loadPluginsFromXML(xmlPath);

        System.out.println("Loaded Plugins:");
        for (Plugin plugin : plugins) {
            plugin.execute();
        }
    }
}
```

Listing 5.3 Microkernel-Anwendung, die Plugins nutzt (Java)

Vorteile

- Flexibilität und Erweiterbarkeit der Anwendung, da Änderungen und Erweiterungen über Plugins vorgenommen werden. Die Kernanwendung ist davon unberührt.
- Schon innerhalb der Architektur werden Verantwortlichkeiten getrennt.

- Dass die Plugins isoliert voneinander entwickelt werden können, kann die Entwicklung der Anwendung effektiver machen.

Nachteile

- Anpassungen an der Kernanwendung sind nahezu ausgeschlossen. Änderungen an der Schnittstelle zu den Plugins würden eine Änderung bei allen Plugins nach sich ziehen.
- Die Anwendung kann von Plugins abhängig werden, wenn wichtige Funktionalität nicht von der Kernanwendung selbst, sondern auch durch Plugins bereitgestellt wird.

5.2.5 Microservices

Über lange Zeit wurden Enterprise-Anwendungen in Form von großen zusammenhängenden Einheiten entworfen und umgesetzt. Dieser Ansatz wird als *monolithischer Architekturstil* bezeichnet. Sämtliche Geschäftsfunktionalitäten werden hierbei in einer Einheit ausgeliefert und zur Laufzeit innerhalb eines (Betriebssystem-)Prozesses ausgeführt.

Die interne Strukturierung und Aufteilung in Komponenten solcher Anwendungen wird mithilfe der jeweiligen Programmiersprache durchgeführt, beispielsweise durch die Verwendung von Klassen, Funktionen oder Namensräumen.

Mit diesem Ansatz wurden und werden viele erfolgreiche Produkte bzw. Projekte umgesetzt. Allerdings nahmen auch mit zunehmender Größe und Projektlaufzeit der Anwendung oftmals die Probleme und die Frustration über die Anwendung bei Anwendern, Fachabteilungen und Entwicklern zu.

Einige Projektteams und Firmen begannen als Reaktion darauf, ihre monolithischen Systeme aufzulösen und durch kleinere unabhängigere Anwendungen zu ersetzen bzw. neue Bestandteile als separate Applikationen zu entwickeln.

James Lewis und Martin Fowler trugen daraufhin im Jahr 2012 die Gemeinsamkeiten der verschiedenen Ansätze der unterschiedlichen Teams zusammen und beschrieben das entstandene Modularisierungskonzept. In der Folge entstand der Begriff *Microservices*. Eine feste Definition existiert jedoch nicht. Daher werden häufig die von Lewis und Fowler beschriebenen Eigenschaften als Orientierung für die Umsetzung dieses Architekturstils herangezogen.

Adrian Cockroft, der bei Netflix maßgeblich an der Umstellung auf eine Microservice-Architektur beteiligt war, beschreibt diesen Architekturansatz beispielsweise so:

> *»Service-oriented architecture composed of loosely coupled elements that have bounded contexts« (Adrian Cockcroft)*

Anwendungen, die dem Architekturstil der Microservices folgen, weisen dementsprechend folgende Eigenschaften auf:

- **Komponenten als separate Services** – Anwendungen sind meist aus mehreren einzelnen Komponenten zusammengesetzt. Bei monolithischen Anwendungen werden alle Komponenten zu einer Einheit zusammengefügt und ausgeführt. Bei den Microservices hingegen wird jede Komponente als separater Service bzw. separate Instanz ausgeführt.
- **Lose Kopplung zwischen den Services** – Die Abhängigkeiten zwischen den einzelnen Komponenten sind minimal und die Kommunikation zwischen den Komponenten ist *technologieagnostisch*, das heißt, sie ist von der Technologie der einzelnen Services unabhängig. Fachliche Änderungen an einer Komponente ziehen nicht zwangsläufig Änderungen an weiteren Komponenten nach sich.
- **Bounded Context** – Jeder Service ist um eine eigenständige, klar abgegrenzte Business-Funktionalität herum gebaut. Es bestehen minimale fachliche Abhängigkeiten zu anderen Komponenten. In diesem Kontext wird gerne die UNIX-Philosophie erwähnt: »Do one thing and do it well.« Die Aufteilung von Komponenten bzw. Bounded Contexts ist in einer Microservice-basierten Anwendung oftmals die größte Herausforderung.

Smart endpoints and dumb pipes

In der Microservice-Welt hat sich der Ansatz von *smart endpoints and dumb pipes* verbreitet. Die Idee dahinter ist, die einzelnen Endpunkte, also die Services, mit allem auszustatten, was sie benötigen, und sie gleichzeitig so weit wie möglich von anderen Services zu entkoppeln. Die Kommunikation zwischen den Services soll möglichst leichtgewichtig sein und nicht auf komplexe Protokolle oder Datenformate aufgebaut werden.

Ein Zitat, das in diesem Zusammenhang oft verwendet wird, stammt von Ian Robinson: *»Be of the web, not behind the web«*. Das bedeutet, es sollen möglichst offene, weitverbreitete Standards aus dem Internet verwendet werden. HTTP, Rest, JSON und XML sind Beispiele hierfür.

- **Passende Technologien** – Jeder Service wird in einer für die entsprechende Anforderung passenden Technologie umgesetzt. Ein zentraler Zwang zu einem Technologie-Stack entfällt. Durch die klaren Schnittstellen über standardisierte Protokolle bleiben die Services interoperabel und können weiterhin zusammenarbeiten.
- **Dezentrale Datenhaltung** – Jeder Service ist für seine Daten verantwortlich und speichert diese in einer eigenen Persistenz-Schicht, auf die nur über die Schnittstellen des Service zugegriffen werden kann.

- **Dezentrale Verwaltung** – In Microservice-Umgebungen gibt es keine zentrale Instanz, die für alle Services zuständig ist und diese steuert oder verwaltet. In *serviceorientierten Architekturen* (SOA) ist das z. B. ein sogenannter *Enterprise Service Bus* (ESB).

Service-Oriented Architecture (SOA) und Enterprise Service Bus (ESB)

Mit dem Begriff *serviceorientierte Architektur* (SOA) beschreibt man einen Architekturstil, bei dem Anwendungen in kleine wiederverwendbare Services aufgeteilt und zentral durch einen *Enterprise Service Bus* (ESB) gesteuert werden. Alle Services innerhalb der SOA nutzen ein gemeinsames Daten-Repository. Dadurch wird es schwieriger, die Services unabhängig voneinander zu verwalten.

Geschäftsprozesse werden zentral innerhalb des ESB abgebildet bzw. gesteuert, und der Ausfall eines Service kann zu einem Ausfall des gesamten Geschäftsablaufs führen.

- **Ein Team** – Die Verantwortung für einen Service liegt bei einem Team. Im Gegensatz zu den monolithischen Anwendungen, bei denen sich mehrere Teams die Verantwortung teilen, ist in einer Microservice-basierten Umgebung jedes Team für »seinen« Service verantwortlich. Absprachen zwischen den Teams werden minimiert, da z. B. Releases unabhängig voneinander geplant werden können.

Die ideale Teamgröße für Microservices

Über die ideale Teamgröße gibt es ebenfalls die verschiedensten Aussagen. Laut Martin Fowler gibt es bei Amazon z. B. die »Zwei-Pizza-Team«-Größe. Das soll so viel bedeuten wie: Das Team soll so groß sein, dass ihm zwei Pizzen zum Sattwerden reichen. Über die Größen der Pizzen kann spekuliert werden.

In der Realität haben sich Teamgrößen von drei bis zwölf Personen pro Service als sinnvoll herauskristallisiert. Bei mehr als zwölf Personen nimmt der Kommunikations-Overhead wieder zu, und bei weniger als drei Personen kann nicht mehr von einem Team gesprochen werden.

- **Infrastruktur und Automatisierungen** – In Microservice-basierten Anwendungen wird die Konfiguration bzw. der Aufbau von Infrastruktur so weit wie möglich automatisiert. Technologien wie *Docker* oder *Terraform* sind mit der Verbreitung dieses Architekturstils entstanden und haben sich seitdem stark verbreitet.
- **Auf Fehlersituationen vorbereitet sein** – Die Aufteilung einer Anwendung in mehrere einzelne, eigenständige Komponenten erhöht die Wahrscheinlichkeit, dass es zu Fehlern kommt, insbesondere bei der Interaktion zwischen den Komponenten. Daher muss in einer Microservice-Architektur jeder Service zu jeder Zeit auf mögliche Fehlersituationen vorbereitet sein.

Kritik an Microservices

Organisationen können, wenn sie eine komplexe und stark skalierbare Anwendungslandschaft besitzen, von den Vorteilen des Microservice-Architekturstils profitieren. Allerdings darf man nicht die Herausforderungen vernachlässigen, die bei der Verwendung dieses Ansatzes entstehen:

- **steigende Komplexität** der Systemlandschaft durch die Aufteilung der Anwendung in viele separate Services, die verwaltet werden müssen
- **Overhead für Infrastruktur**, da oft zusätzliche Bestandteile wie zentrale Monitoring-Lösungen oder Service-Discovery-Systeme benötigt werden
- **Kommunikations-Overhead**, da zwischen den Services keine lokalen Methodenaufrufe mehr durchgeführt werden, sondern ausschließlich Remote-Aufrufe. Diese müssen z. B. über das Netzwerk übertragen und abgesichert werden.
- **Datenkonsistenz und Transaktionen** sind durch die Verteilung der Anwendung schwieriger umzusetzen, und eine Datenkonsistenz über mehrere Services kann eventuell erst zeitverzögert hergestellt werden.
- Das **Testen und Debuggen** des Gesamtprozesses ist schwer.
- **Organisatorische Herausforderungen** entstehen, da jedes Team unabhängig agiert und für seinen Service verantwortlich ist. Unter Umständen müssen neue Teams zusammengestellt werden und z. B. Betriebsteams auf die einzelnen Teams aufgeteilt werden. Microservices erfordern eine klare Kommunikation sowie eine Zusammenarbeit und Koordination zwischen den Teams.

Manche Organisationen haben ihre monolithische Anwendung auf viele kleinere Services aufgeteilt und danach festgestellt, dass der Aufwand für die Implementierung und den Betrieb der Microservices den Nutzen überstieg, den die Microservices mit sich brachten. Wie bei jeder Technologie oder jedem Pattern sollte ein Einsatz wohlüberlegt sein und sollten die Vor- und Nachteile gründlich abgewogen werden. Die genannten Herausforderungen würden bei einer Pattern-Beschreibung unter dem Punkt »Konsequenzen« aufgeführt.

5.3 Stile zur Anwendungsorganisation und Codestruktur

Ein Architekturstil legt, wie bereits beschrieben, eine grundlegende Struktur für die Organisation von Softwarekomponenten und deren Beziehungen untereinander fest. Jeder Stil basiert dabei auf spezifischen Prinzipien und Vorgehen, um bestimmte Anforderungen zu erfüllen. Beispiele hierfür sind Skalierbarkeit, Modularität oder Wartbarkeit.

Zudem benötigen alle Anwendungen, unabhängig von ihrer Größe, eine klare, eindeutige Domänenlogik, die die spezifischen Regeln und die entsprechenden Geschäftsprozesse implementiert.

Daher liegt der Schwerpunkt dieses Abschnitts auf einer effizienten Organisation der Anwendung und einer strukturierten Codebasis, um eine effektive und wartbare Implementierung der Geschäftslogik zu gewährleisten.

5.3.1 Domain-Driven Design

Eric Evans stellte bei seiner Arbeit fest, dass viele Softwareprojekte aufgrund unklarer Anforderungen, mangelnder Kommunikation zwischen den Softwareentwicklern und den Fachexperten oder durch unzureichende Modellierung der Geschäftsprozesse scheiterten bzw. ernst zu nehmende Probleme hatten. Gleichzeitig beobachtete er eine Entwicklung in der objektorientierten Community, deren Ziel es war, die Softwareentwicklung besser an die realen Geschäftsanforderungen anzupassen und die alltäglichen Herausforderungen besser zu bewältigen. Er bezeichnete diesen Ansatz als *Domain-Driven Design* (DDD).

Im Jahr 2003 veröffentliche Eric Evans das Buch *»Domain-Driven Design: Tackling Complexity in the Heart of Software«*, in dem er diesen Ansatz bzw. diese Philosophie ausführlich beschrieb und damit maßgeblich den Begriff des *Domain-Driven Designs* prägte.

Das dicke Blaue – Big Blue Book

Eric Evans' Buch von 2003 wird oftmals wegen seiner Farbe und seinem Umfang als *Big Blue Book* bezeichnet. Zehn Jahre später veröffentlichte Vaughn Veron das Buch *»Implementing Domain-Driven Design«*, das einen roten Einband besitzt und in der Community aus diesem Grund gerne als *Big Red Book* bezeichnet wird. Verons Buch ist etwas praxisorientierter und fokussiert sich darauf, wie DDD umgesetzt werden kann.

Folgende Prämissen stellt Evans für geschäftsbezogene und nicht technisch orientierte Software auf:

- Der Fokus von Softwareprojekten muss auf der Fachlichkeit liegen.
- Komplexe Fachlichkeit sollten auf einem Modell beruhen.

Das Domain-Driven Design beschreibt keine expliziten Programmierprinzipien, sondern beschäftigt sich mit der Modellierung von Geschäftsmodellen und deren Prozessen sowie mit der Definition eines Vokabulars zur Modellierung eines Fachgebiets, einer sogenannten *Domäne*. Es ist ein Ansatz zur Entwicklung komplexer Softwaresysteme, die sich mit der Zeit weiterentwickeln können.

Auch wenn eine Nähe zur objektorientierten Softwareentwicklung besteht und es in der Praxis dort oftmals für die Analyse und das Design eingesetzt wird, ist das Domain-Driven Design auch bei anderen Vorgehensweisen einsetzbar und nicht auf die objektorientierte Programmierung beschränkt.

Einsatzgebiete des Domain-Driven Designs

In der Vergangenheit haben sich das Domain-Driven Design und seien Ansätze immer weiter verbreitet. Sie gelten mittlerweile als eine der wichtigsten Vorgehensweisen, um fachliche Funktionalitäten in komplexen Softwaresystemen zu organisieren und abzubilden.

Das Domain-Driven Design kann z. B. mit dem Konzept des *Bounded Context* dazu beitragen, Modulgrenzen zu finden, und deshalb die Aufgliederung einer Anwendung in mehrere Bestandteile oder Services zu unterstützen. Beim Entwurf von Microservice-Architekturen hilft das Konzept dabei, klare Servicegrenzen zwischen den einzelnen Microservices zu definieren und eine strukturierte Kommunikation zwischen ihnen zu ermöglichen.

Einer der wichtigsten Punkte ist dabei die enge Zusammenarbeit und Kommunikation zwischen Entwicklern und Fachexperten, weshalb sich der Ansatz besonders in agilen, iteraktiven Umfeldern stark verbreitet hat.

Der Kern jedes Domain-Driven Designs besteht in der Modellierung einer klar abgegrenzten Fachdomäne. In einer objektorientierten Umsetzung stellt das die Basis zur Definition von Domänenklassen dar und ermöglicht deren strikte Abgrenzung bzw. Entkopplung von den anderen Funktionalitäten des Systems.

In Abbildung 5.9 sehen Sie ein schlichtes Beispiel eines Domänenmodells für die Verwaltung von Schulungen. Es enthält keine technischen Abhängigkeiten und beschreibt ausschließlich Fachlichkeit.

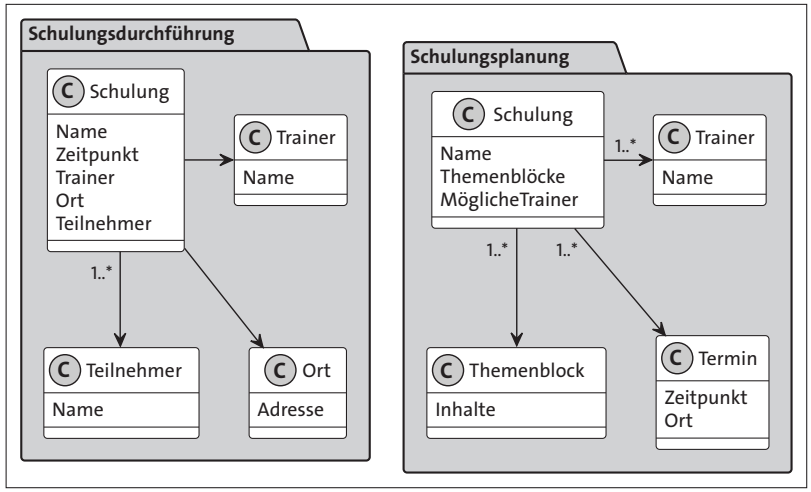


Abbildung 5.9 Beispiel für ein Klassendiagramm

Das Beispiel führt allerdings zu zwei weiteren Domain-Driven-Design-Konzepten, die innerhalb der sogenannten *Strategic Design Phase* definiert werden:

- eine einheitliche Sprache
- ein abgeschlossener Kontext bzw. eine abgeschlossene Einheit

Ubiquitous Language bzw. einheitliche Sprache

Zur Modellierung der Fachlichkeit gehört beim Domain-Driven Design (DDD) immer die Schaffung einer einheitlichen, allgegenwärtigen und für alle Beteiligten verbindlichen Sprache. Dieses Vokabular soll die Kommunikation zwischen Fachexperten und Softwareentwicklern fördern und damit sicherstellen, dass ein gemeinsames Verständnis der Fachlichkeit vorhanden ist. Eine solche *Ubiquitous Language* (dt. »allgegenwärtige Sprache«) ist nicht zwangsläufig allgemeinverständlich und erfordert oftmals eine tiefe Kenntnis des fachlichen Kontextes.

In Projekten ohne eine einheitliche Sprache müssen sehr häufig Begriffe zwischen Fachexperten, Entwicklern oder anderen Projektbeteiligten übersetzt oder erläutert werden. Das gilt nicht nur zwischen Fachexperten und Entwicklern, sondern manchmal auch zwischen den Entwicklern selbst. Verwirrung und Missverständnisse, die durch eine fehlende einheitliche Sprache entstehen, führen zwangsläufig zu erhöhtem Aufwand in der Entwicklung und letztlich zu fehleranfälliger und weniger stabiler Software.

Aus Sicht des Domain-Driven Designs bedeutet das, dass die fachlichen Begriffe der Ubiquitous Language sich unverändert im Code wiederfinden sollten und dass diese nicht übersetzt werden sollen.

Im Beispiel aus Abbildung 5.9 wurden z. B. folgende Begriffe verwendet:

- Schulung
- Trainer
- Teilnehmer
- Ort

Ein Fachexperte, der tief in der spezifischen Domäne verankert ist, könnte im dargestellten Modell dem Begriff »Ort« widersprechen. Aus seiner Sicht wäre »Durchführungsort« präziser und passender. Diese Unterschiede in der Wortwahl stellen jedoch kein Hindernis dar, sondern bieten vielmehr die Chance für einen konstruktiven Austausch zwischen Fachleuten und Entwicklern, sodass sich das gemeinsame Modell und eine einheitliche Sprache entwickeln können.

Die Ubiquitous Language sollte auf jeden Fall festgehalten werden und konsequent in Dokumenten oder Diagrammen zum Einsatz kommen. Im einfachsten Fall erstellt man ein Glossar mit allen Begriffen, die in der Sprache vorkommen.

Bounded Context bzw. abgeschlossener Kontext

Das Beispiel in Abbildung 5.10 zeigt allerdings noch eine zweite Herausforderung. Die Anwendung besteht aus zwei Bestandteilen: Schulungsplanung und Schulungsdurchführung. Jeder Teilbereich definiert den Begriff »Schulung« unterschiedlich. Er ist somit nicht eindeutig definiert.

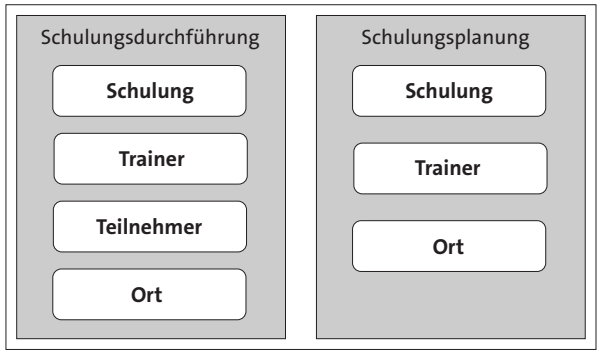


Abbildung 5.10 Beispiele für »Bounded Context«

Eine mögliche Lösung ist die Kombination der beiden Modelle in einem gemeinsamen Modell. In diesem Kontext können eindeutige Begriffe wie z. B. »Schulungstermin« und »Schulungsdefinition« eingeführt werden. Alternativ könnte auch ein allgemeiner Schulungsbegriff definiert werden, der in beiden Teilbereichen verwendet werden kann. Obwohl diese Vorgehensweise für dieses Beispiel hilfreich sein könnte, führt sie zu einem umfangreicheren, komplexeren und möglicherweise unübersichtlicheren Modell.

Erweitert man das Modell auf diese Weise sukzessive immer weiter, führt das höchstwahrscheinlich zu einem recht komplexen, kaum noch wartbaren Domänenmodell mit unklaren Verantwortlichkeiten.

Das Domain-Driven Design kennt als mögliche Lösung das Konzept eines sogenannten *Bounded Context*. Mit diesem abgeschlossenen und klar begrenzten Kontext wird ein Bereich definiert, in dem ein Fachmodell mit entsprechenden Regeln und einer einheitlichen Sprache gilt.

Ein Bounded Context zeichnet sich durch folgende Punkte aus:

- **Klare Grenze** – Der Kontext definiert, welche Teile eines Systems innerhalb oder außerhalb liegen. Diese Grenze kann funktional, organisatorisch oder technischer Natur sein.
- **Einheitliche Sprache** – Innerhalb des Kontextes gilt eine Ubiquitous Language, damit alle Begriffe und Konzepte für alle Teammitglieder eindeutig und verständlich sind.

- **Spezifische Modelle und Regeln** – Wie im Beispiel oben angesprochen, besitzt jeder Kontext ein spezifisches, fachliches Modell mit eigenen Regeln und Logiken. Dies ermöglicht eine exakte Trennung und eine Fokussierung auf die kontexteigene Fachlichkeit, um diese präzise abbilden zu können.
- **Klare Verantwortlichkeiten** – Ein Kontext beschreibt nicht nur Geschäftsmodelle, sondern auch die zugehörigen Verantwortlichkeiten der enthaltenen Modelle und Komponenten. Dadurch soll sichergestellt werden, dass es zu weniger Konflikten zwischen den verschiedenen Teilsystemen kommt. In der Softwareentwicklung können dies auch klar definierte Team-Verantwortlichkeiten sein.

Im obigen Schulungsbeispiel kann dieser Ansatz zu zwei Bounded-Context-Instanzen mit jeweils einem eigenen Geschäftsmodell führen. Jedes Modell hat genau die Verantwortlichkeit für einen entsprechenden Bereich und definiert exakt nur die Abhängigkeiten und Attribute, die für diesen Bereich notwendig sind. Am Beispiel mit der Klasse `Schulung` sieht man recht gut, dass die beiden Schulungen doch recht unterschiedlich sind und im Bereich der Schulungsdurchführung andere Attribute interessant sind als im Bereich der Schulungsplanung.

Evans stellt klar, dass ein Bounded Context nicht zwangsläufig eine Modulgrenze darstellt und dass in Modulen mehrere Kontexte parallel definiert sein können. Ein Bounded Context ist in erster Linie keine technische Grenze, auch wenn diese oftmals durch Technik umgesetzt wird.

Durch die Auftrennung der Modelle und die Definition zweier Kontexte wird das obige Beispiel übersichtlicher, klarer und wartbarer. Änderungen in einem Teilbereich haben keine direkten Auswirkungen auf den zweiten Bereich, und die Weiterentwicklung kann getrennt voneinander stattfinden, was in einem gemeinsamen Modell nicht möglich wäre. Teilt man das Beispiel in zwei Module, eventuell auch Services, könnten z. B. auch zwei Teams parallel daran arbeiten und ihre beiden Versionen unabhängig voneinander ausliefern.

5.3.2 Strategisches und taktisches Design

Das Domain-Driven Design definiert mehrere zeitliche Phasen, die der Entwicklungsprozess durchläuft. Vereinfacht kann man zwei wichtige Phasen hervorheben:

- *Strategisches Design* bzw. *Strategic Design*
- *Taktisches Design* bzw. *Tactical Design*

Innerhalb des strategischen Designs wird die Fachdomäne analysiert und modelliert. Es werden Bounded Contexts und deren Beziehung zueinander aufgestellt. Der Fokus liegt hier klar auf der umzusetzenden Fachlichkeit und einer groben Gliederung des Systems.

Während der Phase des taktischen Designs wird das Domänenmodell verfeinert und mithilfe von bestehenden Bausteinen näher beschrieben. Das Domain-Driven Design definiert folgende Bausteine:

- Services
- Aggregates
- Entitys
- Value Objects
- Repositorys
- Factorys

Im Schulungsbeispiel haben wir bisher ein strategisches Design erstellt und eine taktische Weiterentwicklung sollte folgen.

5.3.3 Hexagonale Architektur bzw. Ports and Adapters

Die *hexagonale Architektur*, die auch als *Ports-and-Adapters-Architektur* bezeichnet wird, wurde von Alistair Cockburn entwickelt und erstmals im Jahr 2005 in einem Blogbeitrag von ihm vorgestellt. Mit seinem Architekturansatz verfolgt er das Ziel, Softwaresysteme flexibler, testbarer und leichter wartbar zu machen.

Alistair Cockburn entwickelte diesen Ansatz, nachdem er in vielen Anwendungen, deren Quellcode er im Laufe der Jahre gelesen und damit gearbeitet hatte, die Geschäftslogik mit dem Code der Oberflächenlogik verschmolzen sah. Daraus entstanden unter anderem folgende Probleme:

- **Die Anwendungen ließen sich nur schlecht automatisiert testen** – Wenn die zu testende Logik direkt in der Oberflächenlogik enthalten ist, können Änderungen an der Darstellung, z. B. eine Größenänderung oder die Positionsänderung eines Buttons, zu Fehlern beim Testen der Logik führen.
- **Eine automatisierte Hintergrundverarbeitung ist unmöglich** – Ist die Logik mit der Oberflächenlogik verschmolzen, lässt sich diese auch nicht separat ausführen, z. B. innerhalb eines Batch-Prozesses. Ein Aufruf muss immer über die Oberfläche erfolgen.
- **Keine alternativen Aufrufwege möglich** – Aus denselben Gründen, also weil die Geschäftslogik über die Oberfläche aufgerufen werden muss, können keine weiteren Clients angeschlossen werden. Möchte z. B. eine weitere Anwendung die Logik aufrufen, müsste diese ebenfalls die Oberfläche verwenden. Eine externe Programmierschnittstelle steht nicht zur Verfügung.

Viele zu dieser Zeit entstandenen unternehmenskritischen Anwendungen verfolgten eine logische Schichtentrennung und unterteilten die Anwendung dementspre-

chend in mehrere logische Bereiche. Verbreitet war die 3-Schichten-Architektur mit User-Interface, Business- und DataAccess-Schicht, wie in Abbildung 5.11 dargestellt.

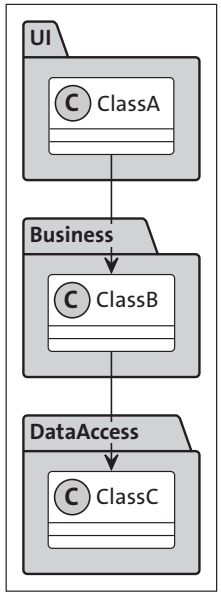


Abbildung 5.11 Klassische 3-Schichten-Architektur

Entdeckte man in solchen Anwendungen Verletzungen der Schichtentrennung – beispielsweise durch Code, der in einer nicht dafür vorgesehenen Schicht implementiert wurde –, versuchte man, diesen Code in die korrekte Schicht zu verschieben. Wenn die dafür benötigte Schicht fehlte, wurde sie oft kurzerhand eingeführt. So entstand ein iterativer Prozess, der ohne effektive Kontrollen schnell zu Chaos führen kann, da immer mehr neue Schichten mit nicht klar abgegrenzten Aufgaben entstehen. Ohne eine effektive Kontrolle, ob die Schichtenstruktur eingehalten wird, ist der Einsatz dieses Modells nicht sinnvoll umzusetzen.

Zusätzlich neigen derart aufgebaute Schichtenmodelle dazu, Abhängigkeiten innerhalb der Anwendung nicht so abzubilden, wie es die Geschäftslogik erfordern würde. In Abbildung 5.11 gibt es z. B. eine Abhängigkeit der Business-Schicht von der DataAccess-Schicht. Das heißt, dass die Geschäftslogik in extremen Fällen von der technischen Umsetzung einer Datenhaltung oder von Änderungen in der Infrastruktur abhängig ist und technische Änderungen die Geschäftslogik beeinflussen können.

Wenn z. B. ein datenbankabhängiger Datentyp für die Speicherung von Objekt-IDs in der Datenbank verwendet wird und diese Datenbankabhängigkeit in den Geschäftsobjekten als Attribut übernommen wird, ist ihre Nutzung an das jeweilige Datenbankprodukt gebunden. Ein Einsatz ohne die entsprechende Datenbankbibliothek oder den Zugriff auf die Datenbank wird deutlich erschwert oder sogar unmöglich.

Das beeinträchtigt die Testdurchführung, und Änderungen an der eingesetzten Datenbanktechnologie erfordern Anpassungen im Datenmodell der Geschäftslogik.

Jede Abhängigkeit schränkt die Weiterentwicklung ein und macht eine Architektur unflexibler. Änderungen in Abhängigkeiten erzwingen eventuell Anpassungen an den abhängigen Komponenten. In langlebigen Systemen wird es, je länger die Software eingesetzt wird, unter Umständen häufiger zu Infrastrukturänderungen kommen als zu grundlegenden Änderungen der Geschäftslogik.

Solche Abhängigkeitsprobleme wirken sich immer auch auf das Testen der Anwendung aus. Jede Abhängigkeit im Code muss zum Zeitpunkt des Tests erfüllt sein. Hängt die Geschäftslogik von einer Datenbank ab, kann nur mit dieser Datenbank getestet werden. Das macht die Tests komplexer und unflexibler.

In den klassischen Schichtenmodellen spricht man immer von Abhängigkeiten, die von »oben nach unten« oder »von rechts nach links« bestehen. Die Richtung hängt davon ab, ob das Modell als Grafik horizontal oder vertikal dargestellt wird.

Cockburns Ansatz ordnet diese Abhängigkeiten neu. Das zentrale Element ist bei ihm die Anwendungslogik, die sich im »Inneren« befindet. Um sie herum, im »Äußeren«, werden externe Ressourcen über Ports angebunden (siehe Abbildung 5.12). Zwischen einem Port und der Geschäftslogik vermitteln sogenannte *Adapter*. Abhängigkeiten bestehen nur von außen nach innen.

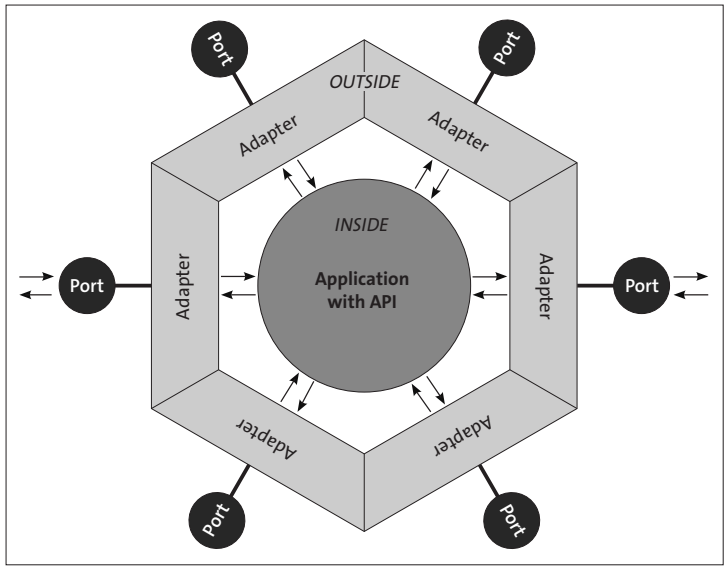


Abbildung 5.12 Hexagonale Architektur bzw. »Ports and Adapter«

Mit dem namensgebenden Sechseck in Abbildung 5.12 wollte Cockburn zeigen, dass es nicht nur zwei externe Abhängigkeiten, wie Datenhaltung und Darstellung, geben kann, sondern dass eine Vielzahl von externen Ressourcen angebunden werden können. Die Zahl sechs ist hier nur symbolisch zu verstehen, weshalb der Ansatz auch oftmals als *Ports and Adapter* bezeichnet wird.

Der Begriff *Port* steht für einen Punkt, an dem kompatible externe Komponenten angebunden werden können – analog zu einem Betriebssystem-Port, über den externe Komponenten kommunizieren können, die das entsprechende Protokoll unterstützen. Der *Adapter* hat die Aufgabe, die Kommunikation, die über einen Port geführt wird, in die Schnittstelle des »inneren Kerns« der Anwendung zu übersetzen und damit eine »Protokollumwandlung« durchführen.

Bei einem Rest-Call zu einer Anwendung würde der HTTP-Port die externen Anfragen entgegennehmen und ein entsprechender Adapter (bei Java z. B. ein Servlet) würde die Parameter auswerten und die Anfragen im passenden Format an die eigentliche Geschäftslogik weitergeben.

Für die Anbindung einer Datenbank stellt der Datenbanktreiber den Port dar und eine schlanke Zugriffsschichtenkomponente, die Adapter-Implementierung, würde die Übersetzung zwischen Geschäftslogik und Datenbanktreiber vornehmen.

Diese Form der Anwendungsstruktur erlaubt es, den Kern der Anwendung von äußeren Änderungen zu isolieren und auf diese Weise die Flexibilität und die Testbarkeit zu erhöhen.

Um die Abhängigkeiten der Anwendungsteile nach diesem Modell auszurichten, lässt sich auf Codeebene für eine Umsetzung das *Inversion of Control*-Prinzip einsetzen (siehe Abschnitt 2.5). Durch die Definition eines fachlich orientierten Interfaces innerhalb der Business-Schicht, wie in Abbildung 5.13 gezeigt, kann man die Abhängigkeit zwischen den beiden Schichten umdrehen. Wichtig ist hier, dass das eingeführte Interface ausschließlich fachlich orientiert ist und keine technischen Details enthält. Die *Ports and Adapters*-Architektur ist hierzu die logische Fortsetzung.

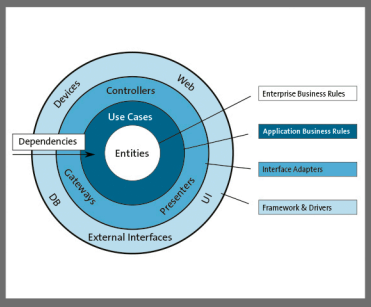
Durch die Einführung der fachlichen Interfaces in die Geschäftslogik-Komponente kann beim automatisierten Testen über ein Testframework recht einfach ein Stellvertreterobjekt, einen sogenanntes *Mock-Objekt*, eingesetzt werden (siehe Abbildung 5.14). Dessen Verhalten kann für jeden Testfall individuell angepasst werden. Im Beispiel einer Datenbankanbindung entfallen damit aufwendige Testdaten bzw. die Vorkonfigurationen der Datenbank.

Softwaredesign und Entwurfsmuster

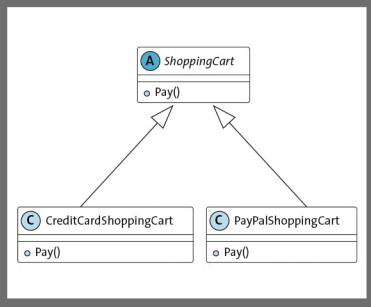
»So geht gute Software.«

Bewährte Entwurfsmuster und modernes Softwaredesign

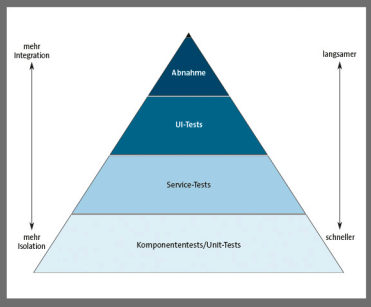
Was sind die Zutaten für gute Software? Entwurfsmuster und ein sauberes, nachhaltiges Design gehören ohne Frage dazu. Viele Patterns haben sich seit Jahrzehnten bewährt, manche sind neu. Im Detail wandelt sich, was gutes Design bedeutet, zum Beispiel für Cloud-native Anwendungen. Das nötige Rüstzeug bekommen Sie hier!



Grundkonzepte verstehen



Entwurfsmuster einsetzen



Verantwortungsvoll entwickeln

Designprinzipien verstehen

Lernen Sie das Fundament kennen, auf dem langlebige Software steht. Jedes Designprinzip hat seine Stärken, die Sie zu nutzen lernen. Aber auch für Nachteile und Fallstricke entwickeln Sie hier ein gutes Gespür.

Entwurfsmuster richtig einsetzen

Im Detail mag der Teufel stecken, aber oft auch das Erfolgsgeheimnis. Sehen Sie an Beispielen, worauf es bei den einzelnen Entwurfsmustern ankommt und wie Sie sie in einer modernen Umgebung einsetzen.

Mit Best Practices ins Ziel

Profitieren Sie von der Erfahrung des Autors. Realistische Projektbeispiele, lehrreicher Code sowie Tipps zu Unit-Tests, Dokumentation und mehr – so werden Sie zum Experten für professionelles Design!

Mit allen Codebeispielen zum Herunterladen

Kristian Köhler ist Softwarearchitekt, Entwickler und Geschäftsführer der Source Fellows GmbH mit Sitz in Reutlingen. Ihn begeistert es besonders, fachliche und technische Komplexität zu analysieren und ein passendes Design zu finden – für nachhaltige, effiziente Softwarelösungen.

Aus dem Inhalt

- Designprinzipien
- Clean Code & Clean Architecture
- Domain Driven Design
- Microservice Patterns
- Patterns der »Gang of Four«
- SOLID-Prinzipien
- Messaging Patterns
- Resilience Patterns
- Verteilte Anwendungen
- Architekturmuster
- Cloud-native Patterns
- Anwendungsorganisation
- Dokumentation und Tests

Rheinwerk Computing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

€ 39,90 [D] € 41,10 [A]

Softwareentwicklung
ISBN 978-3-367-10259-4