Joachim Goll

Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik

Strategien für schwach gekoppelte, korrekte und stabile Software

IT DESIGNERS GRUPPE

Springer Vieweg

Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik

Joachim Goll

Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik

Strategien für schwach gekoppelte, korrekte und stabile Software

Springer Vieweg

Joachim Goll
Esslingen, Deutschland

ISBN 978-3-658-20054-1 ISBN 978-3-658-20055-8 (eBook)
https://doi.org/10.1007/978-3-658-20055-8

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

Springer Vieweg

Einbandabbildung: designed by eStudioCalamar © Fotolia / shutterstock

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

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

Vorwort

Dieses Buch beschreibt die wichtigsten Entwurfsprinzipien für Software. Entwurfsprinzipien sind bewährte, einfache und klare Denkkonzepte des Software Engineering, die dem Entwickler helfen, gute Softwaresysteme zu konstruieren. Entwurfsprinzipien greifen in die Konstruktion eines Systems ein und betreffen die für den Entwickler sichtbare Qualität des Codes eines Programms. Der Schwerpunkt der meisten Entwurfsprinzipien liegt eindeutig auf der Entkopplung der Softwareteile bzw. der Abschwächung von Abhängigkeiten. Im Zuge der Clean-Code-Bewegung wird den Entwurfsprinzipien eine gesteigerte Aufmerksamkeit zuteil.

Entwurfsprinzipien wurden aus dem Erfahrungsschatz der Softwareentwickler abgeleitet. Entwurfsprinzipien sind Handlungsanweisungen für den erfolgreichen Bau von Systemen. Es sind abstrakte Grundsätze, die dem Handeln zugrunde gelegt und deren Konsequenzen überprüft werden können. Ein gutes Verständnis der Entwurfsprinzipien ist für den Bau einer hochwertigen Software unverzichtbar. Entwurfsprinzipien lenken den Blick des Entwicklers in die richtige Richtung. Sie überzeugen durch ihre zwingende Logik und durch grundsätzlich positive Erfahrungen über Jahrzehnte hinweg.

Die wesentlichen Entwurfsprinzipien zu verstehen und vor seinen Augen zu haben, ist Voraussetzung für das Schreiben einer guten Software durch den Entwickler. Inwieweit einzelne Entwurfsprinzipien dann jeweils berücksichtigt werden, hängt von der jeweiligen Projektsituation ab. So kann es sein, dass der Performance des Systems in einem Projekt eine höhere Priorität eingeräumt wird als bestimmten Entwurfsprinzipien. Verstöße gegen Entwurfsprinzipien müssen aber generell sehr sorgsam abgewogen werden, da die Nichtbeachtung von Entwurfsprinzipien zu erheblichen Mängeln der zu schreibenden Software führen kann und damit auch Qualitätsziele verfehlt werden können.

Qualitätsziele für den Entwurf sind beispielsweise

- die Entkopplung von Softwareteilen,
- Einfachheit und Verständlichkeit,
- Testbarkeit oder
- Stabilität.

Die genannten Qualitätsmerkmale betreffen die Güte der Konstruktion eines Softwaresystems und damit natürlich auch die Verantwortung der Softwareentwickler. Solche Merkmale werden innere[1] Qualitätsmerkmale genannt.

Es gibt keine allgemein anerkannten Kataloge von Entwurfsprinzipien. Daher enthält dieses Buch nur eine subjektive Auswahl, die dem Autor in der Praxis häufig begegnet ist und auch in der Clean-Code-Bewegung eine große Rolle spielt.

[1] Neben inneren gibt es auch äußere Qualitätsmerkmale wie Performance oder Bedienbarkeit, die für den Kunden direkt sichtbar und wichtig sind.

Kapitel 1 analysiert die historische Entwicklung des Konzepts von Software-Modulen. Kapitel 2 untersucht die Frage, wie Abhängigkeiten bei der Konstruktion von Systemen entstehen können. Kapitel 3 bis 10 diskutieren wichtige Entwurfsprinzipien und Konzepte für die Konstruktion schwach gekoppelter, korrekter und stabiler Software. Kapitel 3 befasst sich dabei mit Entwurfsprinzipien zur Vermeidung von Überflüssigem, letztendlich um unnütze Abhängigkeiten zu vermeiden, Kapitel 4 mit Entwurfsprinzipien für die Konstruktion schwach gekoppelter Teilsysteme und Kapitel 5 mit Entwurfsprinzipien für die Konstruktion korrekter Programme. Kapitel 6 untersucht Entwurfsprinzipien für die Stabilität bei Programmerweiterungen, Kapitel 7 erörtert das Konzept "Inversion of Control", Kapitel 8 enthält Entwurfsprinzipien nahe am Code, Kapitel 9 betrachtet Entwurfsprinzipien auf Systemebene und Kapitel 10 gibt eine Übersicht über verschiedene Kategorien der in diesem Buch vorgestellten Entwurfsprinzipien und Konzepte.

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

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

Alle Beispielprogramme des Buches sind auf dem begleitenden Webauftritt unter www.springer.com zu finden, damit Sie diese bequem selbst ausführen und nachvollziehen können.

Schreibweisen

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

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

Danksagung

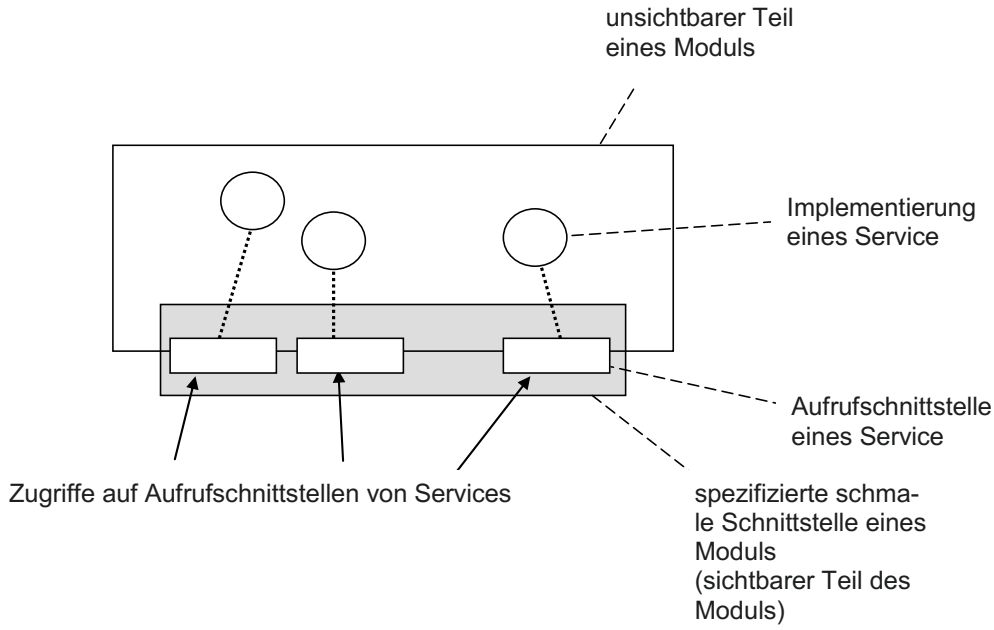
Frau Patricia Maier, Herrn Prof. Dr. Manfred Dausmann, Herrn Timo Acquistapace, Herrn Christoph Gschrey, Herrn Lukas Jaeckle, Herrn Benjamin Jester, Herrn Markus Just, Herrn Marcel Kilian, Herrn Maximilian Schall und Herrn Michael Watzko danken wir für zahlreiche wertvolle Diskussionen sowie Herrn Jordanis Andreanidis für die professionelle Durchführung des Konfigurationsmanagements.

Esslingen, im Dezember 2017 Das Architektenteam der IT-Designers Gruppe

Inhaltsverzeichnis

Kapitel 1

Software-Module



1 Software-Module

Kapitel 1.1 befasst sich mit der Zerlegung eines physisch vorhandenen Systems in seine Teile, die als Module bezeichnet werden. Kapitel 1.2 erörtert die verschiedenen Konzepte zur Konstruktion von Modulen. Die Vorteile der Aufteilung eines Systems in Module werden in Kapitel 1.3 betrachtet.

1.1 Modularisierung von Systemen

Bereits beim Bau der ersten Softwaresysteme wurde erkannt, dass es sehr wichtig ist, sie aus Teilsystemen – also **modular** – aufzubauen. Die folgende Abbildung zeigt beispielhaft ein modular aufgebautes System aus drei Modulen:

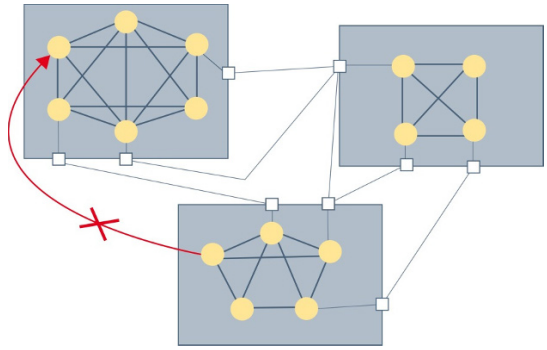


Abbildung 1-1 Module mit starkem inneren Zusammenhang ihrer Elemente und schwachen Wechselwirkungen nach außen[2]

Module stellen Klassen bzw. Funktionen und Daten zur Lösung von bestimmten Problemen zur Verfügung. Was genau ein Modul enthält, ist abhängig von der Programmiersprache. Für das Konzept der Module spielt es jedoch keine Rolle, ob beispielsweise prozedural oder objektorientiert gearbeitet wird.

Modularisierung ist die Strukturierung eines physischen Systems in schwach wechselwirkende Systemteile, die Module. Diese kann man benutzen, ohne eine Kenntnis über die innere Struktur der Module zu haben.

Ein **Modul** ist eine physische Einheit eines Systems im Rahmen der technischen Lösung (Lösungsbereich), also in der Welt der Konstruktion.

Die folgenden Unterkapitel dienen einer genaueren Betrachtung.

[2] Ein direkter Zugriff auf die Elemente eines Moduls ist verboten (durchgekreuzt im Bild). Die kleinen eckigen Kästchen auf dem Rand der Module symbolisieren die Schnittstellen der Module.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8_1

1.1.1 Strukturierung eines Systems in Teile

Eine Aufteilung eines Systems führt zu Teilsystemen einer geringeren Komplexität und damit zu einer besseren Verständlichkeit.

Eine Aufteilung eines Systems in Module kann generell nach dem Prinzip "Teile und Herrsche"[3] unter Finden geeigneter **Abstraktionen**[4] für die Zuständigkeiten der verschiedenen Teile des Systems erfolgen.

Wird ein System in Module strukturiert, so bedeutet das, dass ein Modul nur eine festgelegte, sogenannte **schmale Schnittstelle**[5] nach außen zur Verfügung stellt, die so wenig wie möglich über das Innere eines Moduls offenlegt. Über diese Schnittstelle eines Moduls können die anderen Module zugreifen, um die vom Modul angebotenen Dienste (Services) zu verwenden. Man kann sagen, dass eine Schnittstelle das Bindeglied zwischen den Nutzern des Moduls und der Implementierung des Moduls darstellt.

Nur die Schnittstelle eines Moduls ist nach außen sichtbar. Alle Informationen eines Moduls sollen über die Schnittstelle des betreffenden Moduls ausgetauscht werden. Damit beeinflusst ein Modul andere Module nur wenig.

Die innere Struktur eines Moduls, d. h. seine Implementierung, ist gekapselt und bleibt nach dem Prinzip **"Information Hiding"**[6] von David L. Parnas verborgen.

In der deutschen Literatur wird das Prinzip "**Information Hiding**" auch als **Geheimnisprinzip** bezeichnet.

Ein Modul darf nur die Schnittstelle eines anderen Moduls kennen. Die Verwendung einer Schnittstelle, ohne das Innere des Moduls zu kennen, wird **Benutzungsabstraktion** genannt.

Modularisieren ist also mehr, als nur zu separieren.

Nach der Festlegung der Schnittstellen der Module können die verschiedenen Module von verschiedenen Entwicklern parallel bearbeitet und beispielsweise getrennt voneinander programmiert und getestet werden.

Da Module stets schmale Schnittstellen als Abstraktion ihrer Leistung aufweisen müssen, können die einzelnen Module auch leichter ausgetauscht werden.

---

[3] siehe Kapitel 9.1

[4] Auf den Begriff der "Abstraktion" wird in Kapitel 1.2.2 eingegangen.

[5] siehe "Schnittstelle" im Begriffsverzeichnis

[6] siehe Kapitel 1.2.3 und Kapitel 4.2

1.1.2 Kapselung änderbarer Designentscheidungen in Modulen

Bekannt wurde der Begriff eines **"Moduls"** durch die Veröffentlichung "On the criteria to be used in decomposing systems into modules" von David L. Parnas im August 1971 an der Carnegie Mellon University in Pittsburgh, Pennsylvania [Par71]. In dieser Arbeit zitierte David L. Parnas wiederum Richard Gauthier und Stephan Pont, die den Begriff des "Moduls" bereits in ihrem Buch "Designing Systems Programs" [Gau70] im Jahre 1970 verwendeten.

Ein Modul nach David L. Parnas sollte zur Erzeugung von **Stabilität** schwierige Designentscheidungen oder Designentscheidungen, die sich vermutlich noch ändern werden, in seinem Inneren kapseln.

Die **Kapselung von Designentscheidungen** durch Parnas beruhte bereits auf dem Begriff der **"Änderungswahrscheinlichkeit"** eines Moduls, einer Sichtweise, die aber erst durch Robert C. Martin im Jahre 2002 endgültig mit dessen Begriff der **"Verantwortlichkeit"** als Grund für Veränderungen eines Moduls allgemein akzeptiert wurde [Mar02, p. 97].

Vor Robert C. Martin (2002) war ein modulares System trotz des fortgeschrittenen Ansatzes von David L. Parnas, der seiner Zeit voraus war, in der Praxis meist nur der Gegensatz zu einem monolithischen System: Ein Modul eines modularen Systems beschrieb ein abgekapseltes Teilsystem, das über eine schmale Schnittstelle zugänglich war.

Dennoch ist es heutzutage unumstritten, dass ein Modul nur eine **einzige Verantwortlichkeit** als Grund für Änderungen besitzen soll.

1.1.3 Wann spricht man heutzutage von einem Modul?

Ein Modul basiert heutzutage (siehe Kapitel 1.2) auf

- einer einzigen Verantwortlichkeit,
- dem Verbergen der Implementierung ("Information Hiding") und
- der Benutzungsabstraktion.

1.1.4 Was steckt in einem Modul?

Was ein Modul enthält, ist vollkommen variabel. Ein Modul als Einheit des Lösungsbereichs kann folgende Funktionalitäten umfassen:

- eine Funktionalität des reinen Problembereichs,
- eine Funktionalität des reinen Lösungsbereichs, aber auch
- eine Funktionalität des Problembereichs gemischt mit einer Funktionalität des Lösungsbereichs.

1.2 Konzepte zur Konstruktion von Modulen

Module werden durch Abstraktion und unter Verwendung einer einzigen Verantwortlichkeit pro Modul gefunden. Auf diese Konzepte soll in den folgenden Unterkapiteln eingegangen werden.

1.2.1 Eine einzige Verantwortlichkeit pro Modul

Nach dem **"Single Responsibility Principle"** von Robert C. Martin (2002) darf es nur eine **einzige Verantwortlichkeit** pro Modul geben.

Mit dem "Single Responsibility Principle" soll verhindert werden, dass bei der Veränderung einer Verantwortlichkeit eine andere Verantwortlichkeit eines Systems in unbeabsichtigter Weise beschädigt wird.

Das "Single Responsibility Principle" wird detailliert in Kapitel 4.7 analysiert.

1.2.2 Abstraktionen

Eine Abstraktion arbeitet generell das Wesentliche heraus und lässt das Unwesentliche weg. Es geht dabei um die Fragen: "Was ist wichtig?" bzw. "Was ist unwichtig?"

Abstraktionen werden bei der Modularisierung eines Systems in benachbarte Teilsysteme in zwei verschiedenen Bereichen benötigt. Diese sind:

- **Identifikation der verschiedenen Teile eines Systems**, damit das System in benachbarte Teile geschnitten werden kann (Entwurfsprinzip "Teile und Herrsche" bzw. "divide et impera").
- **Finden der schmalen Schnittstellen der Module**, welche Services als Abstraktion ihrer Implementierung anbieten (Benutzungsabstraktion) und dabei die Implementierung verbergen ("Information Hiding").

Nur durch den Einsatz von Abstraktionen wird die Komplexität eines Systems beherrschbar.

1.2.3 Kapselung von Modulen unter Abstraktion und "Information Hiding"

Die Implementierung eines Moduls darf nicht bei den Implementierungsentscheidungen anderer Module herangezogen werden. Darum wird die Implementierung gekapselt.

Die Konzepte Abstraktion und "Information Hiding" sind bei Kapselung eng miteinander verwoben.

Der Begriff der "**Kapselung**" umfasst

- **Benutzungsabstraktion** und
- **"Information Hiding"** der Implementierung.

Im Folgenden werden diese Konzepte und ihre Zusammenhänge erläutert.

Mit **Abstraktion** wird das Verhalten einer Kapsel durch eine **schmale Schnittstelle** nach außen sichtbar gemacht. Diese schmale Schnittstelle enthält die Aufrufschnittstellen der vom Modul bereitgestellten Services, wie in der folgenden Abbildung angedeutet:

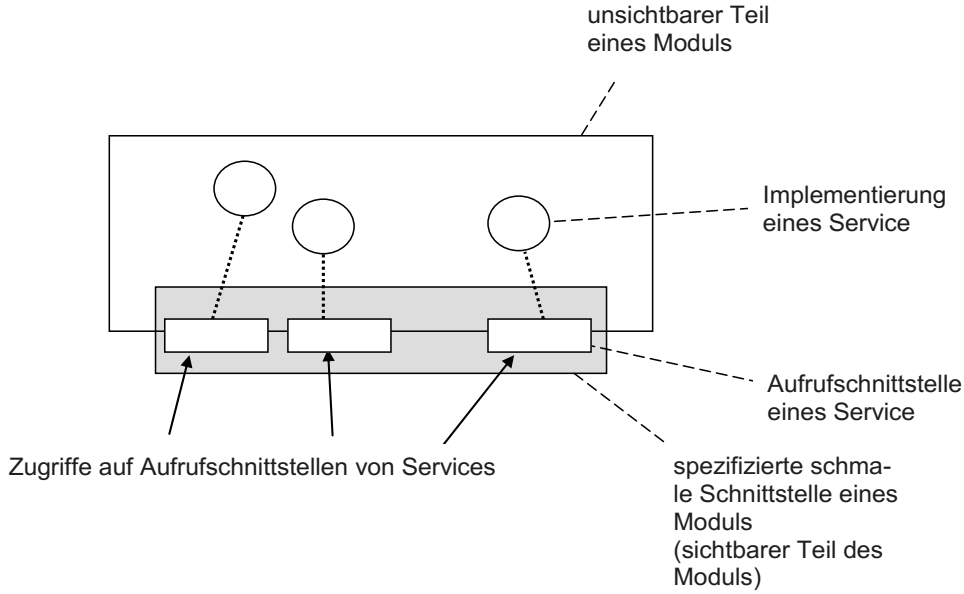


Abbildung 1-2 Gekapselter Zugriff auf ein Modul

Die inneren Details der Kapsel werden verborgen (**"Information Hiding"** der Implementierung). Hierbei wird der direkte Zugriff auf das Innere dieses Moduls unterbunden. Es ist stattdessen nur ein Zugriff über definierte Aufrufschnittstellen der einzelnen Services möglich.

1.2.4 Abstraktion und "Information Hiding" bei Klassen

Die Wichtigkeit der Kapselung zeigt sich auch auf der Ebene von Klassen. Klassen bestehen aus Methoden und Daten. Hierbei sind die Daten und die Implementierungen der Methodenrümpfe verborgen. Im Allgemeinen darf ein Zugang zu einem Objekt nur über die nach außen freigegebenen, sichtbaren Methodenköpfe der **Schnittstellenmethoden**[7] erfolgen.

[7] auch öffentliche Methoden genannt

Der Zugriff über die Methodenköpfe der Schnittstellenmethoden verhindert, dass Invarianten eines Programms[8] – also Bedingungen über Daten, die alle Schnittstellenmethoden einer Klasse einhalten müssen – durch beliebige Zugriffe von außen umgangen werden können.

Ein Nutzer einer Klasse soll von deren Innenleben so wenig wie möglich erfahren.

Abstraktion und "Information Hiding" für Klassen sind in folgender Abbildung dargestellt:

Klasse aus Methoden und Daten

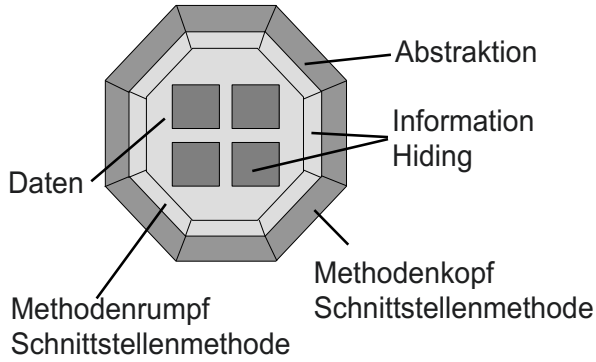


Abbildung 1-3 Daten und Methoden – die Bestandteile von Objekten

Die Operationen bzw. Methoden, die auf den Daten eines Objekts arbeiten, werden in der Regel als eine "innere Fähigkeit" eines Objekts betrachtet. Wie die Operationen im Detail ablaufen, ist Sache des betreffenden Objekts. Von außen sieht man dann nur, wie die Schnittstellenmethoden aufgerufen werden können.

Ein Objekt sollte nur über definierte Schnittstellenmethoden mit seiner Umgebung in Kontakt treten. Die Methodenköpfe der Schnittstellenmethoden bilden die Schnittstellen eines Objekts nach außen und stellen eine Abstraktion einer Leistung des Objekts dar. Diese Abstraktionen werden vom Objekt beim Aufruf von Schnittstellenmethoden zur Verfügung gestellt.

Verborgen wird in der Regel die Implementierung von Klassen, d. h.

- die **Daten**,
- die Implementierung des nach außen sichtbaren Verhaltens, also die Implementierung von **Rümpfen der Schnittstellenmethoden**, sowie
- **Service-Methoden**[9] (**Hilfsmethoden** für das eigene Objekt), die nur modulintern benutzt werden und nach außen nicht sichtbar sind.[10]

---

[8] siehe Kapitel 5.1.3.3

[9] Service-Methoden zur Verwendung im eigenen Objekt sind in Abbildung 1-3 nicht dargestellt. Solche Service-Methoden können nur durch Methoden desselben Objekts aufgerufen werden.

Die Implementierung einer Klasse soll vor der Außenwelt verborgen sein ("Information Hiding"). Dadurch wird vermieden, dass eine Klasse von der Implementierung einer anderen Klasse abhängt.

Nur die **Aufrufschnittstellen der Schnittstellenmethoden** dürfen in der Regel nach außen sichtbar sein. Über sie bietet eine Klasse nach außen ihre Leistungen an.

1.3 Vorteile der Modularisierung

Eine **Modularisierung** hat für ganze Systeme die folgenden **Vorteile**:

- **Beherrschbarkeit komplexer Systeme**

  Modulare Systeme bestehen aus schwach wechselwirkenden Modulen als Teile des Systems. Solche Systeme sind aufgrund der abgeschwächten Abhängigkeiten leichter zu entwickeln, zu testen und zu warten als monolithische Systeme ohne Modulstruktur.

- **Erhöhung der Flexibilität von Systemen**

  Bei Änderungen und Anpassungen sind nur wenige Module betroffen – im Idealfall nur ein einziges – und nicht das ganze System.

Die einzelnen **Module** haben die im Folgenden genannten **vorteilhaften Eigenschaften**[11]:

- **Benutzungsabstraktion**

  Das Testen und die Verwendung eines Moduls kann erfolgen, ohne dass man dessen Aufbau kennt.

- **Gegenseitige Nichtbeeinflussung**

  Durch Verbergen der Implementierung werden andere Module nicht von der Implementierung eines Moduls abhängig. Solange die Schnittstelle eines Moduls nicht verändert wird, kann man im Inneren eines Moduls beliebige Änderungen durchführen, da dessen Implementierung durch die Schnittstelle gekapselt und damit nach außen nicht sichtbar ist.

---

[10] Java erfüllt dieses Ziel nur zum Teil. In Java ist der Zugriffsschutz nur klassenbezogen realisiert. Dies bedeutet, dass es nur Objekten fremder Klassen verwehrt werden kann, auf ein anderes Objekt zuzugreifen. Ein Objekt einer bestimmten Klasse kann in Java stets auf alle Methoden und Datenfelder – sowohl öffentliche als auch private – eines Objekts derselben Klasse zugreifen, wenn es dieses kennt.

[11] Bitte beachten Sie, dass diese Eigenschaften nicht "orthogonal" zueinander und damit unabhängig voneinander sind. So sind beispielsweise die getrennte Entwickelbarkeit und Testbarkeit der Module mit der gegenseitigen Nichtbeeinflussung und der Kontextunabhängigkeit verknüpft.

- **Single Responsibility**

  Damit wird vermieden, dass bei der Bearbeitung einer Verantwortlichkeit in einem bestimmten Modul eine andere Verantwortlichkeit unabsichtlich beschädigt wird.

- **Getrennte Entwickelbarkeit, Testbarkeit und Änderbarkeit**

  Wenn die Schnittstellen der Module eines Systems feststehen, können diese Module unabhängig voneinander entwickelt, getestet und geändert werden.

- **Einhaltung des Lokalitätsprinzips**

  Alle Informationen eines Moduls befinden sich an einer einzigen Stelle.

- **Kontextunabhängigkeit**

  Module sind weitgehend kontextunabhängig[12]. Sie sind fast unabhängig von ihrer Umgebung entwickel- und verwendbar.

- **Potenzielle Wiederverwendung**

  Module können unter Umständen mit wenig Aufwand in anderen Systemen wiederverwendet werden.

- **Wartbarkeit**

  Module können auch leichter an sich ändernde Systemumgebungen angepasst werden und sind damit leichter wartbar.

1.4 Zusammenfassung

Kapitel **1.1** behandelt die Aufteilung eines physischen Systems in Module. Ein Modul ist eine physische Zerlegungseinheit eines Systems im Lösungsbereich. Ein Modul ist vom Konzept her unabhängig davon, ob beispielsweise prozedural oder objektorientiert gearbeitet wird.

Kapitel **1.2** diskutiert die Konzepte zur Konstruktion von Modulen:

- eine einzige Verantwortlichkeit pro Modul,
- die Abstraktion zu einer schmalen Schnittstelle (Benutzungsabstraktion) und
- "Information Hiding" der Implementierung.

Kapitel **1.3** befasst sich mit den Vorteilen der Modularisierung.

Die Vorteile der Modularisierung sind, wenn man ganze Systeme betrachtet:

- die Beherrschbarkeit komplexer Systeme und
- die Erhöhung der Flexibilität von Systemen.

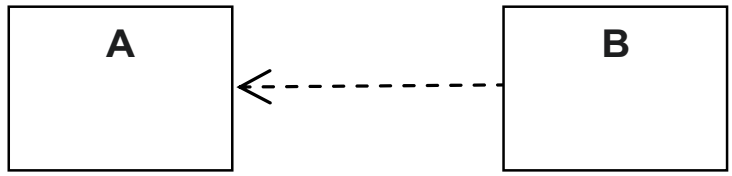
[12] Module kommunizieren nur über schmale Schnittstellen. Das ist ihr einziger Kontext, von dem sie abhängig sind.

Die Vorteile der einzelnen Module sind:

- Benutzungsabstraktion,
- gegenseitige Nichtbeeinflussung,
- Single Responsibility,
- getrennte Entwickelbarkeit, Testbarkeit und Änderbarkeit,
- Einhaltung des Lokalitätsprinzips,
- Kontextunabhängigkeit,
- potenzielle Wiederverwendung und
- Wartbarkeit.

Kapitel 2

Abhängigkeiten



2 Abhängigkeiten

Dieses Kapitel stellt keine Entwurfsprinzipien vor, sondern es betrachtet die Entstehung und Abschwächung von Abhängigkeiten. Die Abschwächung von Abhängigkeiten ist das Ziel der meisten Entwurfsprinzipien, damit eine Programmänderung weniger Folgeänderungen nach sich zieht.

Ralf Westphal[13] schreibt in seinem Blog [Wes10]:

"Abhängigkeiten sind eine der größten Geißeln der Softwareentwicklung. Wenn es ein allgemein anerkanntes Prinzip gibt, dann ist es, Abhängigkeiten zu minimieren. Jede Hilfe ist da willkommen."

Doch wie lassen sich Abhängigkeiten überhaupt feststellen?

Abhängigkeiten zwischen Modulen sind immer dann vorhanden, wenn ein Modul ein anderes Modul benötigt, um seine Aufgabe durchführen zu können.

So kann beispielsweise eine Komponente nur dann kompiliert werden, wenn eine andere Komponente vorhanden ist, oder eine Komponente kann eine bestimmte Datei an einem bestimmten Ort benötigen, damit sie ihre Aufgabe erfüllen kann.

Abhängigkeiten gibt es aber nicht nur

- zwischen selbst geschriebenen Komponenten, sondern auch
- zu Fremdsoftware wie verwendeten Bibliotheken, Frameworks, APIs, Datenbanken oder Betriebssystemen.

Oftmals enthält eine Software mehr und stärkere Abhängigkeiten als notwendig. Hierdurch erhöht sich der Aufwand

- für die Weiterentwicklung und die Wiederverwendung von Komponenten sowie
- für die Wartung einer Software

erheblich.

Das folgende Kapitel 2.1 befasst sich mit der Entstehung von Abhängigkeiten und Kapitel 2.2 mit den technischen Möglichkeiten, diese abzuschwächen.

[13] Ralf Westphal ist passionierter Blogger und gibt regelmäßig verschiedene grundlegende Überlegungen in seinem Blog „One Man Think Tank“ (Ein-Mann-Denkfabrik) wieder. In seinem Artikel „Abhängigkeiten bewusster wahrnehmen“ vom 07.09.2010 beschreibt er die Problematik von Abhängigkeiten und das Streben nach Hilfe, diese Abhängigkeiten abzuschwächen.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8_2

2.1 Entstehung von Abhängigkeiten

Kapitel 2.1.1 diskutiert statische Beziehungen und damit statische Abhängigkeiten zwischen Elementen. Kapitel 2.1.2 behandelt Abhängigkeiten, die zur Laufzeit auftreten, und Kapitel 2.1.3 befasst sich mit logischen Abhängigkeiten.

2.1.1 Statische Beziehungen und statische Abhängigkeiten zwischen Elementen

Von einer **statischen Abhängigkeit** spricht man dann, wenn eine Komponente schon zur **Kompilierzeit** von

- einer anderen Komponente beziehungsweise
- ihrer Umgebung wie etwa einem Framework

abhängig ist.

Statische Abhängigkeiten sind bei Programmiersprachen, welche alle Programmteile zur Kompilierzeit prüfen, leicht zu erkennen. Der Compiler weist beim Kompilieren darauf hin, wenn ihm etwas fehlt, um erfolgreich kompilieren zu können.

Als **statische Beziehungen** zwischen den Klassen gibt es bei UML:

- Assoziation,
- Generalisierung,
- Realisierung und
- Abhängigkeit.

Dynamische Beziehungen nach UML werden nicht im Detail betrachtet. Lediglich auf Abhängigkeiten zur Laufzeit wird in Kapitel 2.1.2 eingegangen.

Alle statischen Beziehungen sind streng genommen statische Abhängigkeiten, da in diesen Beziehungen Elemente andere Elemente benötigen.

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

Assoziationen, Generalisierungen und Realisierungen haben eine spezielle Semantik. Sie werden als eine besondere Form der statischen Abhängigkeit eingestuft.

Im Folgenden werden die verschiedenen statischen Beziehungen nach UML dargestellt:

- Eine **Assoziationsbeziehung** beschreibt die Verknüpfung zweier Klassen. Bei einer Assoziation brauchen sich beide Partner gegenseitig. Durch Navigationspfeile kann die Zugriffsrichtung eingeschränkt werden. Im Folgenden ein Beispiel für eine Assoziation:

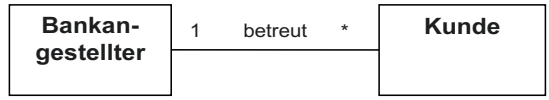


Abbildung 2-1 Ein Beispiel für eine Assoziation

Diese Abbildung zeigt, dass ein Bankangestellter viele Kunden betreut. Ferner kennt ein Kunde dabei den für ihn zuständigen Bankangestellten. Diese Beziehung ist bidirektional.

Bei einer Assoziation spielen deren Spezialfälle "**Komposition**" und "**Aggregation**" ebenfalls eine wichtige Rolle. Klassen können dabei andere Klassen enthalten bzw. diese referenzieren. Dies entspricht, dass Objekte andere Objekte enthalten bzw. diese referenzieren.

Eine **Komposition** ist ein Spezialfall einer Assoziation. Bei der Komposition gehört ein Teil genau zu einem zusammengesetzten Ganzen. Ein Teil lebt genauso lange wie das Ganze.

So enthält beispielsweise ein erstelltes Haus, wenn es nicht umgebaut wird, fest eine bestimmte Anzahl von Räumen. Im Folgenden das Klassendiagramm dieses Beispiels:

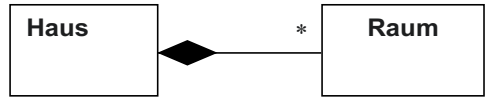


Abbildung 2-2 Ein Beispiel für eine Komposition

Eine **Aggregation** ist ein Spezialfall einer Assoziation.

Bei einer **Aggregation** können mehrere aggregierende Komponenten ein und dieselbe aggregierte Komponente referenzieren. Die Lebensdauer einer aggregierenden und einer aggregierten Komponente kann verschieden lang sein.

Im Folgenden ein Beispiel für eine Aggregation:

Abbildung 2-3 Ein Beispiel für eine Aggregation

Hier hat ein Auto vier Räder, die ausgetauscht werden können, beispielsweise wenn sie abgefahren sind. Genauso ist der Tausch von Winterrädern gegen Sommerräder und umgekehrt möglich.

- Die **Generalisierung** bzw. die **Vererbung** führt zu einer dauerhaften Beziehung zweier Klassen, die sich in einer Hierarchie befinden. Eine Vererbung ist eine **"is a"-Beziehung**: Eine abgeleitete Klasse ist auch vom Typ ihrer Basisklasse.

  Durch die Vererbung zur Kompilierzeit schafft man eine dauerhafte, statische Korrelation zwischen Basisklasse und abgeleiteter Klasse. Die abgeleitete Klasse ist in diesem Falle abhängig von der Basisklasse, da sie deren Struktur (die Datenfelder) und deren Verhalten (die Methoden) erbt.

  Im Folgenden ein Beispiel für eine "is a"-Beziehung:

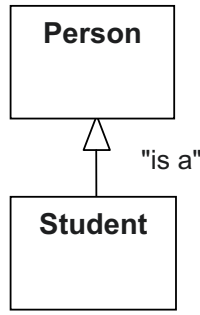


Abbildung 2-4 Ein Beispiel für eine Generalisierung

Abbildung 2-4 zeigt, dass ein Student auch eine Person ist. Je nach Situation kann ein Student in der Rolle Student oder Person auftreten.

- Bei einer **Realisierung** wird der vorgegebene Vertrag[14] des zu realisierenden Classifier[15] realisiert. Der realisierende Classifier muss diesen Vertrag erfüllen und ist damit vom Vertrag des zu realisierenden Classifier abhängig.

  Ein einfaches Beispiel für eine Realisierung ist eine Schnittstelle, die den Vertrag vorgibt, sowie eine Klasse, welche den von der Schnittstelle vorgegebenen Vertrag realisiert. Dies zeigt die folgende Abbildung:

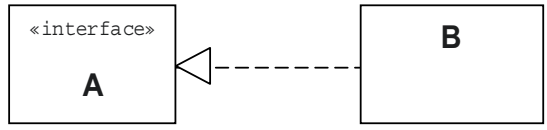


Abbildung 2-5 Ein Beispiel für eine Realisierung

Die Klasse `B` realisiert die Schnittstelle `A`.

Generell gibt bei einer Realisierung der zu realisierende Classifier den Vertrag vor.

---

[14] siehe Kapitel 5.1
[15] zum Begriff eines "Classifier" siehe Begriffsverzeichnis

Die Implementierung einer Schnittstelle ist stark von der Definition der Schnittstelle abhängig. Bei Änderungen an einer Schnittstelle müssen sogleich Anpassungen an allen Implementierungen vorgenommen werden.

- Bei der **Abhängigkeit** eines Elements von einem anderen kann die Änderung des unabhängigen Elements zu einer Änderung des abhängigen Elements führen. Das abhängige Element ist damit vom unabhängigen Element abhängig.

  Hier die Darstellung einer Abhängigkeit:

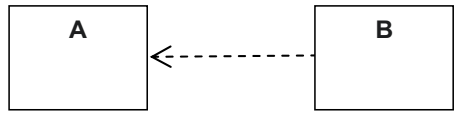


Abbildung 2-6 Ein Beispiel für eine Abhängigkeit

Diese Abbildung zeigt, dass die Klasse `B` von der Klasse `A` abhängig ist.

2.1.1.1 Programmierbeispiel für eine Assoziation

Skizze der Aufgabe

Eine Assoziation lässt sich anhand des folgenden Beispiels über eine Stadt und ihre Bürger erklären. Hierbei lebt jeder Bürger in einer Stadt. In einer Stadt können mehrere Bürger wohnen. Jeder Bürger muss wissen, in welcher Stadt er wohnt und jede Stadt muss ihre Einwohner kennen. Es handelt sich also um eine **bidirektionale Assoziation**.

Aufgabe im Detail

Ein Bürger kann sich in einer Stadt als Einwohner anmelden. Die neue Stadt übernimmt dabei die Aufgabe, den Bürger von der vorherigen Stadt abzumelden und ihn bei sich zu registrieren. Dazu hat die Stadt eine Methode `einwohnerAbmelden()`[16] der Sichtbarkeit `private` und eine Methode `einwohnerAnmelden()` der Sichtbarkeit `public`. Die neue Stadt ruft dann beim Anmelden die Methode `setBewohnteStadt()` des Bürgers auf. Diese Methode ist `protected` und damit nur innerhalb des Package `stadtverwaltung`, in welchem sich die Klassen `Buerger` und `Stadt` befinden, zugänglich. Diese Methode kann also nicht von außerhalb des Pakets aus aufgerufen werden. Wäre dies der Fall, so könnten leicht Zustände auftreten, die inkonsistent sind. Beispielsweise könnte ein Bürger eine Referenz auf eine Stadt halten, in der er gar nicht registriert ist.

Programm

Jetzt das Programm:

```
// Datei: Buerger.java

package stadtverwaltung;
```

[16] Nur die neue Stadt darf den Bürger abmelden, der Bürger selber darf das nicht.

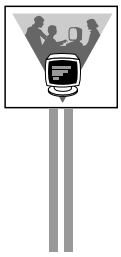
```
public class Buerger
{
   private String name;
   private Stadt bewohnteStadt;

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

   public String getName()
   {
      return name;
   }

   public Stadt getBewohnteStadt()
   {
      return bewohnteStadt;
   }

   protected void setBewohnteStadt (Stadt bewohnteStadt)
   {
      this.bewohnteStadt = bewohnteStadt;
   }
}

// Datei: Stadt.java

package stadtverwaltung;

import java.util.ArrayList;
import java.util.List;

public class Stadt
{
   private String name;
   private List<Buerger> einwohner = new ArrayList<>();

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

   public String getName()
   {
      return name;
   }

   public List<Buerger> getEinwohner()
   {
      return einwohner;
   }

   public void einwohnerAnmelden (Buerger neuerEinwohner)
   {
      if (neuerEinwohner.getBewohnteStadt() != null)
      {
```

```
            neuerEinwohner
                  .getBewohnteStadt()
                  .einwohnerAbmelden (neuerEinwohner);
         }

         neuerEinwohner.setBewohnteStadt (this);
         einwohner.add (neuerEinwohner);
      }

      private void einwohnerAbmelden (Buerger abzumeldendenEinwohner)
      {
         if (einwohner.remove (abzumeldendenEinwohner))
         {
            abzumeldendenEinwohner.setBewohnteStadt (null);
         }
      }

      @Override
      public String toString()
      {
         StringBuilder stringBuilder = new StringBuilder();

         for (Buerger buerger : einwohner)
         {
            if (stringBuilder.length() != 0)
            {
               stringBuilder.append (", ");
            }
            stringBuilder.append (buerger.getName());
         }

         return "Bewohner von " + name + ": " + stringBuilder;
      }
}
```

// Datei: AssoziationsBeispiel.java

```
import stadtverwaltung.Buerger;
import stadtverwaltung.Stadt;

public class AssoziationsBeispiel
{
   public static void main (String[] args)
   {
      Buerger buerger1 = new Buerger ("Max Mueller");
      Buerger buerger2 = new Buerger ("Bernd Walter");
      Buerger buerger3 = new Buerger ("Sarah Maier");
      Buerger buerger4 = new Buerger ("Kim Bauer");

      Stadt stadt1 = new Stadt ("Stuttgart");
      Stadt stadt2 = new Stadt ("Esslingen");

      System.out.println ("Die Buerger melden sich an...");
      stadt1.einwohnerAnmelden (buerger1);
      stadt1.einwohnerAnmelden (buerger2);
      stadt2.einwohnerAnmelden (buerger3);
      stadt2.einwohnerAnmelden (buerger4);
```

```
        System.out.println (stadt1);
        System.out.println (stadt2);

        System.out.println ("\nEinige Bewohner ziehen um...");
        stadt2.einwohnerAnmelden (buerger1);
        stadt1.einwohnerAnmelden (buerger3);

        System.out.println (stadt1);
        System.out.println (stadt2);
    }
}
```



Die Ausgabe des Programms ist:

```
Die Buerger melden sich an...
Bewohner von Stuttgart: Max Mueller, Bernd Walter
Bewohner von Esslingen: Sarah Maier, Kim Bauer

Einige Bewohner ziehen um...
Bewohner von Stuttgart: Bernd Walter, Sarah Maier
Bewohner von Esslingen: Kim Bauer, Max Mueller
```

2.1.1.2 Programmierbeispiel für eine Komposition

Skizze der Aufgabe

Am Beispiel einer Waschmaschine soll nachfolgend die Komposition erklärt werden. Eine Waschmaschine ist aus verschiedenen Komponenten zusammengeschraubt. Im folgenden Beispiel sollen nur die Komponenten

- Waschtrommel und
- Wasserpumpe

betrachtet werden, die in Java als Elementklassen programmiert werden sollen[17].

Die Waschtrommel und die Wasserpumpe sollen die gleiche Lebensdauer wie die Waschmaschine haben. Damit wird hier eine Komposition betrachtet.

Aufgabe im Detail

Die Waschtrommel kann

- starten und
- stoppen.

Die Wasserpumpe kann Wasser

- einlassen bzw.
- abpumpen.

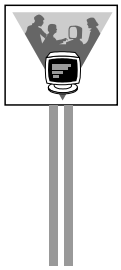
[17] In der Realität kann bei Reparaturen doch das eine oder andere ausgetauscht werden. Die modellierten Klassen lassen einen solchen Austausch aber nicht zu.

Programm

Hier das Programm:

```
// Datei: Waschmaschine.java

public class Waschmaschine
{
   private Waeschetrommel waeschetrommel = new Waeschetrommel();
   private Wasserpumpe wasserpumpe = new Wasserpumpe();

   private class Waeschetrommel
   {
      public void starten()
      {
         System.out.println ("Die Waeschetrommel beginnt zu" +
                             " schleudern.");
      }

      public void stoppen()
      {
         System.out.println ("Die Waeschetrommel stoppt.");
      }
   }

   private class Wasserpumpe
   {
      public void wasserEinlassen()
      {
         System.out.println ("Wasser wird in die" +
                            " Trommel eingelassen.");
      }

      public void wasserAbpumpen()
      {
         System.out.println ("Wasser wird aus der" +
                            " Trommel abgepumpt.");
      }
   }

   public void waschgangDurchfuehren()
   {
      System.out.println ("Waschgang wird gestartet.");
      wasserpumpe.wasserEinlassen();
      waeschetrommel.starten();
      waeschetrommel.stoppen();
      wasserpumpe.wasserAbpumpen();
      waeschetrommel.starten();
      waeschetrommel.stoppen();
      System.out.println ("Waschgang wurde durchgeführt.");
   }
}

// Datei: KompositionsBeispiel.java

public class KompositionsBeispiel
{
```

```
    public static void main (String[] args)
    {
       Waschmaschine waschmaschine = new Waschmaschine();
       waschmaschine.waschgangDurchfuehren();
    }
}
```



Die Ausgabe des Programms ist:

```
Waschgang wird gestartet.
Wasser wird in die Trommel eingelassen.
Die Waeschetrommel beginnt zu schleudern.
Die Waeschetrommel stoppt.
Wasser wird aus der Trommel abgepumpt.
Die Waeschetrommel beginnt zu schleudern.
Die Waeschetrommel stoppt.
Waschgang wurde durchgeführt.
```

2.1.1.3 Programmierbeispiel für eine Aggregation

Aufgabe

Zwei Kinder spielen. Das erste Kind hat 3 Murmeln in seinem Murmelbeutel, aggregiert also 3 Murmeln. Das andere Kind hat keine Murmel. Da bekommt es vom ersten Kind eine Murmel geschenkt.

Die referenzierten Objekte (Murmeln) und das referenzierende Objekt (Murmelbeutel) sollen eine verschieden lange Lebensdauer haben. Damit soll in diesem Beispiel eine **Aggregation** betrachtet werden.

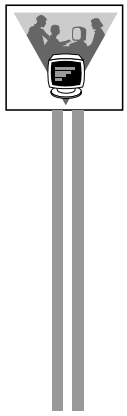
Programm

Jetzt das entsprechende Programm:

```
// Datei: Kind.java

import java.util.ArrayList;
import java.util.List;

public class Kind
{
   private List<Murmel> murmelBeutel = new ArrayList<Murmel>();

   public void erhalteMurmel (Murmel murmel)
   {
      murmelBeutel.add (murmel);
   }

   public void schenkeMurmel (Kind beschenktesKind)
   {
      beschenktesKind.erhalteMurmel (murmelBeutel.remove(0));
   }

   public void murmelnZaehlen()
   {
```

```
      System.out.println ("Ich habe " + murmelBeutel.size() +
                          " Murmeln!");
   }

   public void mitMurmelnSpielen()
   {
      for(Murmel murmel : murmelBeutel)
      {
         murmel.mitMurmelSpielen();
      }
   }
}

// Datei: Murmel.java

public class Murmel
{
   private String farbe;

   public Murmel (String farbe)
   {
      this.farbe = farbe;
   }

   public void mitMurmelSpielen()
   {
      System.out.println ("Die " + farbe + "e Murmel rollt.");
   }
}

// Datei: AggregationsBeispiel.java

public class AggregationsBeispiel
{
   public static void main (String[] args)
   {
      Kind erstesKind = new Kind();
      erstesKind.erhalteMurmel (new Murmel ("rot"));
      erstesKind.erhalteMurmel (new Murmel ("blau"));
      erstesKind.erhalteMurmel (new Murmel ("gruen"));

      Kind zweitesKind = new Kind();

      System.out.println ("Das erste Kind zaehlt " +
                          "seine Murmeln:");
      erstesKind.murmelnZaehlen();

      System.out.println ("Das zweite Kind zaehlt " +
                          "seine Murmeln:");
      zweitesKind.murmelnZaehlen();

      System.out.println ("Das erste Kind schenkt dem " +
                          "zweiten Kind eine Murmel.");
      erstesKind.schenkeMurmel (zweitesKind);
```

```
        System.out.println ();

        System.out.println ("Das erste Kind spielt:");
        erstesKind.mitMurmelnSpielen();

        System.out.println ();

        System.out.println ("Das zweite Kind spielt:");
        zweitesKind.mitMurmelnSpielen();
    }
}
```



Die Ausgabe des Programms ist:

```
Das erste Kind zaehlt seine Murmeln:
Ich habe 3 Murmeln!
Das zweite Kind zaehlt seine Murmeln:
Ich habe 0 Murmeln!
Das erste Kind schenkt dem zweiten Kind eine Murmel.

Das erste Kind spielt:
Die blaue Murmel rollt.
Die gruene Murmel rollt.

Das zweite Kind spielt:
Die rote Murmel rollt.
```

2.1.1.4 Programmierbeispiel für eine Vererbung

Aufgabe

Als Beispiel für eine **Vererbung** sollen eine Person und ein Student betrachtet werden. So beschreibt eine Klasse `Student`, die von der Klasse `Person` abgeleitet wird, eine spezielle Person, die als Student an einer Hochschule eingeschrieben ist. Dies bedeutet, dass eine sogenannte **"is a"-Beziehung** vorliegt. Eine abgeleitete Klasse stellt einen eigenen Typ dar, der oft als Subtyp bezeichnet wird. Objekte der abgeleiteten Klasse sind von diesem Subtyp, sind aber gleichzeitig auch vom Typ der Basisklasse. So tritt ein Student nur an der Hochschule als Student auf oder, wenn er als Student etwas billiger bekommt wie beispielsweise eine Fahrkarte. Ansonsten tritt der Student im Alltag als normale Person auf.

Zwischen Basisklasse und abgeleiteter Klasse herrscht wegen dieser "is a"-Beziehung eine starke Abhängigkeit, wenn die Daten und Methoden der Basisklasse in der abgeleiteten Klasse direkt sichtbar sind. Dies gilt für alle jene Elemente, welche nicht als privat in der Basisklasse vereinbart sind bzw. nicht überschrieben werden.

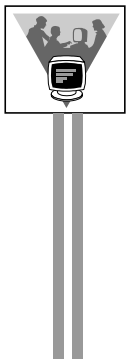
Programm

Das folgende Codestück zeigt die Klassen `Person`, `Student` und `VererbungsBeispiel`:

```
// Datei: Person.java

public class Person
{
   private String name;
   private String vorname;

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

   public void gebeDatenAus()
   {
      System.out.print ("\nNachname: " + name);
      System.out.print ("\nVorname: " + vorname);
   }
}
```

```
// Datei: Student.java

public class Student extends Person
{
   private String matrikelnummer;

   public Student (String vorname, String nachname,
                   String matrikelNummer)
   {
      super (vorname, nachname);
      this.matrikelnummer = matrikelNummer;
   }

   public void gebeMatrikelnummerAus()
   {
      System.out.print ("\nMatrikelnummer : " + matrikelnummer);
   }
}
```

```
// Datei: VererbungsBeispiel.java

public class VererbungsBeispiel
{
   public static void main (String[] args)
   {
      System.out.print ("Erstelle Person");
      Person person = new Person ("Max", "Mustermann");
      System.out.print ("\nErstelle Student");
      Student student = new Student ("Peter", "Vorzeigestudent",
                                     "1893");

      System.out.print ("\n\nAusgabe Person");
      person.gebeDatenAus();

      System.out.print ("\n\nAusgabe Student");
      student.gebeDatenAus();
      student.gebeMatrikelnummerAus();
   }
}
```



Die Ausgabe des Programms ist:

```
Erstelle Person
Erstelle Student

Ausgabe Person
Nachname: Mustermann
Vorname: Max

Ausgabe Student
Nachname: Vorzeigestudent
Vorname: Peter
Matrikelnummer : 1893
```

2.1.1.5 Programmierbeispiel für eine Realisierung

Aufgabe

Eine Klasse `Punkt` soll die Schnittstelle `IPunkt` **implementieren** bzw. **realisieren**.

Programm

```
// Datei: IPunkt.java

public interface IPunkt
{
   float getX();
   void setX (float x);
   float getY();
   void setY (float y);
}
```

```
// Datei: Punkt.java

public class Punkt implements IPunkt
{
   private float x, y;

   @Override
   public float getX()
   {
      return x;
   }

   @Override
   public void setX (float x)
   {
      this.x = x;
   }

   @Override
   public float getY()
   {
      return y;
   }
```

```
   @Override
   public void setY (float y)
   {
      this.y = y;
   }
}

// Datei: RealisierungsBeispiel.java

public class RealisierungsBeispiel
{
   public static void main (String[] args)
   {
      Punkt p = new Punkt();
      p.setX (3.0f);
      p.setY (4.0f);

      System.out.println ("Die Koordinaten des Punktes p sind:");
      System.out.println ("( " + p.getX() + " / " + p.getY() + " )");
   }
}
```

Die Ausgabe des Programms ist:

```
Die Koordinaten des Punktes p sind:
( 3.0 / 4.0 )
```

2.1.2 Abhängigkeiten zur Laufzeit

Bei Abhängigkeiten zur Laufzeit wird **zur Kompilierzeit eine Abstraktion referenziert** (**Aggregation**). **Während der Laufzeit** tritt ein die Abstraktion implementierendes Konstrukt an die Stelle der Abstraktion.

Durch den Ersatz der Abstraktion durch ein konkretes Konstrukt entsteht dynamisch eine sogenannte Abhängigkeit zur Laufzeit. Dabei hat das referenzierende Objekt zu dem referenzierten Objekt eine sogenannte use-Beziehung oder Benutzungsbeziehung.

Die Klasse eines konkreten Objekts[18], das zur Laufzeit an die Stelle der Abstraktion tritt, muss die **Abstraktion implementieren**.

Ein häufiges Beispiel ist, dass eine Klasse zur Kompilierzeit eine Schnittstelle nur aus Methodenköpfen als Abstraktion aggregiert. An die Stelle der Schnittstelle kann zur Laufzeit jedes Objekt treten, welches den Vertrag der Schnittstelle erfüllt.

[18] Ein anderes Beispiel sind Delegates in C#, welche ebenfalls eine Aggregation darstellen. Diese erlauben es, eine Referenz auf eine Methode inklusive dem dazugehörigen Objekt zu speichern. Es tritt hierbei kein Objekt an die Stelle der Abstraktion, sondern eine Referenz auf eine konkrete Methode.

Ein vorhandenes Programm mit Schnittstellen kann kompiliert werden. Dennoch kann es unter Umständen nicht lauffähig sein, wenn zur Laufzeit keine Instanzen an die Stelle der Schnittstellen treten.

Das Entwurfsmuster "Dependency Injection" (siehe Kapitel 4.8) verwendet ebenso wie beispielsweise das Strategiemuster (siehe Kapitel 6.3.4) oder das Dekorierermuster (siehe Kapitel 6.3.4) Abhängigkeiten zur Laufzeit.

Auf die Programmierbeispiele zur "Dependency Injection" in Kapitel 4.8.4.1 bis 4.8.4.3 wird hingewiesen.

2.1.3 Logische Abhängigkeiten

Logische Abhängigkeiten im Code sind Annahmen im Code über bestimmte Voraussetzungen.

Sie sind schwer zu finden und werden meist erst entdeckt, wenn ein Fehler bzw. eine Ausnahme auftritt.

Während statische Abhängigkeiten bereits zur Kompilierzeit bemerkt werden, werden sogenannte **logische Abhängigkeiten erst zur Laufzeit** sichtbar[19], nämlich gerade dann, wenn sie nicht erfüllt sind.

Logische Abhängigkeiten treten immer dann auf, wenn innerhalb von Komponenten Annahmen über ihre Funktionsweise getroffen werden. Beim Erstellen einer Komponente kann es nämlich leicht vorkommen, dass man zu dem Punkt kommt, an dem eine Annahme getroffen oder eine Bedingung erfüllt werden muss. Das kann im Sinne der Abhängigkeit gefährlich sein und zwar insofern, dass wenn eine andere Komponente die Komponente mit der getroffenen Annahme verwendet, diese ebenfalls von der getroffenen Annahme abhängig ist.

2.1.3.1 Vermeidung einer logischen Abhängigkeit im Code

Um unabhängig zu bleiben, sollten Annahmen wie beispielsweise über die Darstellung der Daten und dabei deren Formatierung oder Reihenfolge möglichst unterlassen oder in Verträgen formuliert werden.

Verträge werden in Kapitel 5.1 behandelt.

Natürlich lässt es sich nicht immer vermeiden, Annahmen treffen zu müssen. Die Funktionsweise einer Komponente muss deshalb gut durchdacht werden, damit im Laufe der Entwicklung einer Komponente keine falschen Annahmen erfolgen.

[19] Dies gilt auch für Abhängigkeiten zur Laufzeit.

Annahmen in Komponenten möglichst zu unterlassen, hört sich leichter an, als getan, denn gerade das Finden einer versteckten Abhängigkeit ist hier die Kunst.

Es kann auch Fälle geben, bei denen die Nichterfüllung der getroffenen Annahmen nicht zwingend zu Fehlern im Programm führen muss, sodass diese versteckten Abhängigkeiten nicht offensichtlich sind.

Um versteckte Abhängigkeiten zu finden, ist das Aufstellen von Verträgen[20] ein möglicher Ansatz. Dadurch wird offensichtlich, wenn Vor- und Nachbedingungen benachbarter Methodenaufrufe nicht zusammenpassen, wie im folgenden Beispiel zu sehen ist.

2.1.3.2 Konzeptionelles Beispiel für eine logische Abhängigkeit

Ein offensichtliches Beispiel für diese Art der Abhängigkeit ist die Formatierung eines Kalenderdatums. Im deutschsprachigen Raum wird das Datum in der Reihenfolge Tag, Monat und Jahr formatiert mit einem Punkt als Trennzeichen. Im angloamerikanischen Raum wird das Datum in der Reihenfolge Monat, Tag und Jahr formatiert mit einem Schrägstrich als Trennzeichen.[21] Die folgende Methode soll nun Bestandteil einer Komponente sein:

```
public int getTagDesMonats (Date datum)
{
   SimpleDateFormat formatierer = new SimpleDateFormat();
   String formatiertesDatum = formatierer.format (datum);
   String tagDesMonats = formatiertesDatum.split ("\\.")[0];
   return Integer.parseInt (tagDesMonats);
}
```

Diese Methode funktioniert wie gewünscht auf einem System, dessen Standorteinstellung (engl. locale) auf "deutsch" gesetzt ist.[22] Wird diese Komponente aber auf einem System eingesetzt, bei dem beispielsweise eine amerikanische Standorteinstellung gewählt wurde, so wird diese Methode nicht mehr funktionieren. Sie wird eine Exception werfen und das Programm wird abstürzen, falls diese Exception nicht behandelt wird.

Betrachtet man die Vor- und Nachbedingungen der Methodenaufrufe `format()` und `split()`, dann stellt man fest, dass beim Aufruf der Methode `split()` ein String erwartet wird, der einen Punkt enthält. Die Nachbedingung der Methode `format()` der Klasse `SimpleDateFormat` spezifiziert jedoch, dass das Ergebnis abhängig von der Standorteinstellung ist. Wie bereits erwähnt, enthält ein Datum im angloamerikanischen Raum keinen Punkt. Folglich ist bei einer solchen Standorteinstellung die Vorbedingung der anschließend aufgerufenen Methode `split()` nicht erfüllt.

---

[20] siehe Kapitel 5.1

[21] In den USA wird in der klassischen Schreibweise, die noch sehr verbreitet ist, der Monat dem Tag vorangestellt nach der Notation MM/TT/JJ(JJ).

[22] Die Standorteinstellungen können beispielweise in Java mit Hilfe der Klasse `java.util.Locale` geändert werden.

2.2 Abschwächung von Abhängigkeiten

In einer Software sind deren Komponenten fast immer voneinander abhängig. Die Abhängigkeit zwischen diesen Komponenten wird auch als deren Kopplung bezeichnet. Kopplungen erschweren generell eine Änderung oder Erweiterung eines Systems, da, wenn eine Komponente geändert wird, sofort infolge der Kopplungen Folgeänderungen anstehen. Ebenso sind Komponenten in anderen Systemen schwer wiederverwendbar.

Wären die Komponenten eines Systems fast unabhängig voneinander, könnten Systeme weitaus effizienter entworfen werden, da man nicht auf so vieles Rücksicht nehmen müsste. Leider sind die Teile einer entworfenen Software heutzutage jedoch oft enger gekoppelt, als man es eigentlich beabsichtigt hat.

Einen ersten Eindruck der wechselseitigen Kopplungen der Komponenten eines Systems kann man sich dadurch beschaffen, indem man einzelne Klassen isoliert betrachtet, das heißt, man versucht, sie aus ihrer natürlichen Umgebung "herauszuschälen". In der Regel erkennt man sofort, dass die "herauszuschälende" Klasse ohne ihre Umgebung gar nicht mehr kompiliert werden kann, da es Abhängigkeiten zu anderen Klassen und Funktionen gibt. Kopiert man dann alle für ein Kompilieren der herausgelösten Klasse erforderlichen anderen Klassen zu der betrachteten Klasse, so kann es sein, dass man bereits einen großen Teil des Quellcodes importiert hat. Das bedeutet wiederum, dass eine starke Kopplung herrscht und dass Änderungen einer Klasse Auswirkungen auf das Gesamtsystem haben können.

Eine starke Kopplung von Komponenten ist ein schlechtes Verhalten und führt zu einer Inflexibilität, sodass der Aufwand für die erforderlichen Änderungen der Software oft nicht kalkulierbar ist und somit Änderungen gar nicht praktikabel sind. Gerade in der Softwaretechnik ist es sehr von Nachteil, auf notwendige Änderungen nicht unmittelbar reagieren zu können, da sich die Anforderungen an eine Software oftmals schnell ändern können. Deshalb versucht man, Abhängigkeiten abzuschwächen, um die Software flexibler und damit leichter **wandelbar** (**evolvierbar**) zu machen.

Leider gibt es keine einzige Lösung, mit deren Hilfe Abhängigkeiten vollständig eliminiert werden könnten.[23]

Es muss je nach der vorliegenden Situation beurteilt werden, welche Lösung oder welcher Ansatz nützlich sein könnte, um Abhängigkeiten abzuschwächen. Im Folgenden sollen verschiedene Ansätze zur Abschwächung von Abhängigkeiten diskutiert werden. Hierzu gehört die Verwendung

- von Entwurfsprinzipien,
- von Mustern wie dem Schichtenmodell (siehe Kapitel 2.2.2) oder
- der Konzepte "Dependency Lookup" (siehe Kapitel 2.2.3) bzw. "Dependency Injection" (siehe Kapitel 2.2.4) für die Abschwächung von Abhängigkeiten beim Erzeugen von Objekten.

[23] Wenn die Kopplung zwischen Teilsystemen vollkommen entfallen würde, könnten diese Teilsysteme natürlich nicht mehr zusammenarbeiten.

2.2.1 Entwurfsprinzipien

Das Ziel der meisten Entwurfsprinzipien ist es, starke wechselseitige Abhängigkeiten der Komponenten eines Systems von vornherein zu vermeiden. Entwurfsprinzipien sind der Inhalt dieses Buchs.

2.2.2 Verwendung von Entwurfs- und Architekturmustern

Entwurfs- und Architekturmuster zielen oft darauf ab, Abhängigkeiten zu vermeiden.[24] Ein Beispiel hierfür ist das Architekturmuster "Layers". Das Architekturmuster "Layers" führt zur Bildung von Schichten unter Beseitigung von Abhängigkeiten.

Die Bildung von Schichten gemäß eines Schichtenmodells dient zur Strukturierung eines Systems in Teilsysteme in der Weise, dass eine tiefere Schicht von den höheren Schichten unabhängig ist.

Die folgende Abbildung zeigt die prinzipielle Struktur:

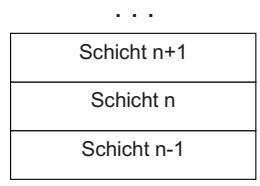


Abbildung 2-7 Schichtenmodell

Dabei bietet eine tiefere Schicht der höheren Schicht Dienste über Schnittstellen an. Damit ist die höhere Schicht nur von den Schnittstellen der tieferen Schicht, aber nicht von deren Realisierung abhängig.

2.2.3 Dependency Lookup

Bei dem Konzept "Dependency Lookup" sucht ein Objekt, das ein anderes Objekt braucht, nach diesem anderen Objekt z. B. in einem Register, um die Verknüpfung mit dem benötigten Objekt herzustellen.

Bei "Dependency Lookup" muss ein Objekt, das ein weiteres Objekt benötigt, die **Abstraktion der Klasse des benötigten Objekts** kennen und den durch die Abstraktion vorgegebenen Vertrag[25] einhalten, um die Methoden des gesuchten Objekts aufrufen zu können.

Ein suchendes Objekt muss aber nicht mehr die konkrete Klasse des von ihm benötigten Objekts kennen. Zur Suche braucht das suchende Objekt einen **Schlüssel** wie den Namen des gesuchten Objekts. Beim Suchen ist damit das suchende Objekt von dem anderen Objekt weitgehend entkoppelt.

---

[24] Für die Entwurfsmuster von Gamma et al. [Gam94] ist dies ein Ziel.

[25] siehe Kapitel 5.1

Das suchende Objekt ist aber ferner auch vom Register abhängig, da es von diesem seine benötigten Objekte bezieht.

Das Konzept "Dependency Lookup" wird detailliert in Kapitel 4.8.3 besprochen.

2.2.4 Dependency Injection

Die Erzeugung von Objekten und die Zuordnung von Abhängigkeiten zwischen Objekten wird an eine dafür vorgesehene Instanz delegiert. Diese Instanz, die auch **Injektor** genannt wird, erzeugt zur Laufzeit die Verknüpfungen zwischen den Objekten. Damit wird die Abhängigkeit zwischen nutzendem und benötigtem Objekt stark abgeschwächt, aber nicht vollständig aufgelöst.

Der Injektor dagegen muss alle beteiligten Objekte und deren Verknüpfungen kennen und ist damit von diesen abhängig, während die beteiligten Objekte vom Injektor unabhängig sind. Der Injektor kann hierbei die Informationen aus einer Konfigurationsdatei lesen und Fabriken benutzen, sodass die genannten Abhängigkeiten noch abgeschwächt werden können.

Zur Kompilierzeit kennt ein Objekt einer nutzenden Klasse statt der konkreten Klasse nur eine **Abstraktion der Klasse des von ihm benötigten Objekts**. Der Vertrag dieser Abstraktion muss vom Objekt der nutzenden Klasse eingehalten werden, damit das Objekt, welches der Injektor zur Laufzeit an die Stelle der Abstraktion setzt, vom nutzenden Objekt verwendet werden kann.

Das Konzept "Dependency Injection" wird detailliert in Kapitel 4.8.4 besprochen.

2.3 Zusammenfassung

Nach der Diskussion von statischen Abhängigkeiten, von Abhängigkeiten zur Laufzeit und von logischen Abhängigkeiten in Kapitel **2.1** befasst sich Kapitel **2.2** mit der Abschwächung von Abhängigkeiten durch den Einsatz von Entwurfsprinzipien, von Architektur- und Entwurfsmustern sowie durch die Konzepte "Dependency Lookup" und "Dependency Injection".

Kapitel 3

Entwurfsprinzipien zur Vermeidung von Überflüssigem

3 Entwurfsprinzipien zur Vermeidung von Überflüssigem

Die Prinzipien "Keep it simple, stupid" (KISS), "You aren't gonna need it" (YAGNI[26]) und "Don't Repeat Yourself" (DRY) einzuhalten, bedeutet, gewisse Dinge nicht zu tun.

Die genannten Prinzipien beeinflussen die Qualität der zu schreibenden Software erheblich, aber nicht die Konstruktion der Architektur des Systems. Zur Konstruktion der Architektur wird beim Entwerfen[27] zuerst eine Strategie[28] ausgewählt, die festlegt, wie ein System konstruiert werden soll. Die tatsächliche Architektur eines Systems ist dann das Ergebnis des Entwurfs.

Die Beschreibung einer Architektur umfasst:

- die Struktur (Statik) des Systems, nämlich die Zerlegung des Systems in Komponenten,
- das Zusammenwirken der Komponenten (Abläufe, Dynamik) sowie
- die Strategie für die Architektur.

Weder zur statischen Struktur, noch zu dynamischen Abläufen oder der Strategie für die Architektur tragen die in diesem Kapitel behandelten Prinzipien KISS, YAGNI und DRY bei. Sie erhöhen jedoch die Qualität einer Software und speziell auch die Qualität einer Architektur wie etwa deren Einfachheit.

Kapitel 3.1 erörtert das Prinzip KISS, das sich dafür ausspricht, Systeme so einfach wie möglich zu bauen und Komplexitäten zu vermeiden.

Komplexe Systeme sind weniger verständlich. Auch ihre Wartung wird erschwert.

Das "Agilistenprinzip" YAGNI (siehe Kapitel 3.2) empfiehlt, nur das zu bauen, was der Kunde explizit fordert.[29]

YAGNI schlägt vor, dass die Entwickler "spekulative" Funktionen weglassen. Eine notwendige Infrastruktur darf dennoch nicht außer Acht gelassen werden.

Das Prinzip DRY (siehe Kapitel 3.3) spricht sich wegen der Fehleranfälligkeit von Aktualisierungen generell gegen Replikate aus.

---

[26] Man findet in der Literatur auch die Schreibweise "Yagni".

[27] Der Entwurf sollte ständig iteriert werden, auch beim Programmieren. Man spricht beim Programmieren seit dem agilen "Extreme Programming" von Kent Beck [Bec99] vom sogenannten "Refactoring", dem Finden einer besseren Architektur für die Programme.

[28] Die Strategie für die Architektur braucht man, damit kein Teammitglied unbewusst gegen die Architektur arbeitet. Die Strategie für die Architektur beschreibt, wie man zerlegt und wie die Komponenten zusammenarbeiten. Man braucht im Team ein "shared understanding" für die Architektur.

[29] Das wird auch als "No Gold Plating" bezeichnet.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8_3

Fehleranfällige Stellen, nämlich Replikate, die man bei einer Aktualisierung leicht vergessen könnte, sind zu vermeiden.

3.1 Keep it simple, stupid

KISS ist eine Abkürzung für "Keep it simple, stupid".

3.1.1 Historie

Der Name für dieses Prinzip wird Kelly Johnson, einem Ingenieur bei Lockheed, im Jahre 1960 zugeschrieben. In der Ursprungsform von Johnson wurde auf das Komma verzichtet: "Keep it simple stupid".

3.1.2 Ziel

Das Prinzip KISS bedeutet, dass Systeme möglichst einfach und nicht komplex sein sollen.

Dieses Prinzip hört sich einfach an, ist aber wirkungsvoll. Generell gilt, dass es in einem komplexen System sehr schwierig sein kann, nachträglich Änderungen und Erweiterungen durchzuführen, ohne die Stabilität des Systems zu gefährden.

3.1.3 Bewertung

KISS fordert **Einfachheit.** Dieses Prinzip sollte beherzigt werden, auch wenn seine Umsetzung nicht immer einfach ist.

Prinzipiell ist es leichter, beim Bau eines Systems eine komplexe Systemarchitektur zu finden als eine besonders einfache.

Vorteile von KISS sind:

- **Einfachere Konstruktion des Systems**

  Einfache Systeme sind leichter zu verstehen, zu bauen, zu testen, zu ändern und zu warten als komplexe Systeme.[30]

[30] Was Einfachheit bedeutet, ist nicht absolut, sondern ist stets relativ zu den jeweiligen Umständen zu sehen. Bei einer Software für eine Mars-Mission ist "Einfachheit" sicher etwas ganz anderes als bei

- **Weniger Fehler, da weniger Komponenten**

  Je weniger Teile ein System besitzt, umso weniger Teile können Fehler aufweisen.

- **Weniger Abhängigkeiten**

  Je weniger Komponenten ein System hat, umso weniger Abhängigkeiten kann es zwischen dessen Komponenten geben.

- **Konzentration von Funktionen auf weniger Komponenten**

  Es besteht weniger Gelegenheit, einzelne Funktionen über mehrere Komponenten zu "verschmieren", da es weniger Komponenten gibt.

Es sollen einfache Systeme gebaut werden, aber keinesfalls monolithische Systeme.

Der **Nachteil** von KISS ist:

- Einfache Systeme sind meist schwieriger zu entwerfen.

3.2 You aren't gonna need it

YAGNI ist eine Abkürzung, welche für **"You aren't gonna need it"**[31], – auf Deutsch: **"Du wirst es nicht brauchen"** – steht.

3.2.1 Historie

YAGNI ist eine Regel des "Extreme Programming". YAGNI wird im Umfeld agiler Ansätze gerne verwendet und kann auf Ron Jeffries et al. aus dem Jahre 2000 [Jef00, p. 190] zurückgeführt werden.

3.2.2 Ziel

- Ein **Over-Engineering**[32] und
- **spekulative**, vom Kunden **nicht geforderte Funktionen** – dabei insbesondere eine **spekulative Generalisierung** –

sollten durch die Entwickler vermieden werden.

einem "Hello World"-Programm. Dennoch kann man versuchen, für beide Programme jeweils die einfachste Lösung zu finden.

[31] auch "You ain't gonna need it" oder "You aren't going to need it" genannt

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

Generalisierung bedeutet, Abstraktionen einzuführen, die auch in der Zukunft für weitere konkrete Fälle genutzt werden können.

Abstraktionen können oftmals durch das Typsystem der jeweiligen Sprache wie im Falle abstrakter Klassen oder durch Templates in C++ bzw. Generics in Java umgesetzt werden. Eine Generalisierung kann beispielsweise aber auch die Realisierung weiterer Methoden bedeuten, um zu erwartende, weitere Fälle abzudecken. Oftmals sind Generalisierungen durch die Entwickler aber zu voreilig, da sie später überhaupt nicht benötigt werden. Dies wird als **"Premature Generalization"** bezeichnet. Solche unnötigen Generalisierungen verzögern das ursprüngliche Projekt, erschweren die Ausbaubarkeit und kosten Geld. Auch Generalisierungen müssen vom Kunden gefordert werden. Es ist sinnvoll, vom Kunden nicht explizit geforderte Funktionen und Generalisierungen wegzulassen, falls diese nicht die benötigte Infrastruktur oder anerkannte Qualitätsziele wie Erweiter- und Änderbarkeit betreffen.

Martin Fowler [Fow15] sagt Folgendes:

"Yagni is not a justification for neglecting the health of your code base. Yagni requires (and enables) malleable[33] code."

Auch bei Anwendung von YAGNI braucht der Code eine gesunde Basis. YAGNI fordert und fördert einen flexibel änderbaren und erweiterbaren Code.

Diese Vorgehensweise steht im Gegensatz dazu, generalisierte Änderungen in vorauseilendem Gehorsam von Anfang im Hinblick darauf einzuplanen, dass eine spätere Generalisierung mehr Aufwand erfordern würde. Bei einem solchen Vorgehen stellt sich oftmals heraus, dass eine solche Generalisierung vom Kunden später gar nicht gewünscht wird und dass sich der Kunde im Laufe der Zeit ganz andere Forderungen ausdenkt. Üblicherweise bezahlt ein Kunde nur das, was er sich wünscht. Dies bedeutet aber nicht, dass man mit YAGNI keine änderungsfreundlichen Lösungen im Auge hat. Ganz im Gegenteil! Es ist bekannt, dass sich die Anforderungen an ein Softwaresystem rasch ändern können. Diese geänderten Anforderungen lassen sich in der Regel in einer einfachen Codestruktur leichter umsetzen als in einer komplexen, generalisierten Codestruktur.

Generalisierter Code wird besonders gerne dann geschrieben, wenn die Anforderungen des Kunden nicht genau genug sind, der Entwickler diese Ungenauigkeit fälschlicherweise akzeptiert und durch eine generalisierte Lösung zu kompensieren versucht.

Wenn generalisierte Features[34] durch die Entwickler in die Software eingebaut werden sollen, so muss der Kunde diese Features explizit wünschen und sie auch bezahlen.

[33] dt. "formbar" im Sinne einer Veränderung
[34] Ein Feature ist eine kleine, nützliche Funktion für den Kunden.

Bezahlt der Kunde eine Leistung, dann ist sie ihm auch etwas wert! Entwickler sollen nicht stillschweigend in ihrem eigenen Sinn handeln und die Kundenwünsche neu interpretieren. Unklare Anforderungen müssen geklärt oder abgelehnt werden.

Ein Entwickler sollte nur **klare Forderungen** implementieren. Er darf ein bestimmtes Programmstück erst dann entwickeln, wenn es auch tatsächlich vom Kunden gefordert wird.

Das Prinzip YAGNI klingt einfach, wird in der Praxis jedoch oft verletzt. Nicht geforderte Funktionalität wird nicht nur vom Kunden nicht entlohnt, sondern sie erhöht oftmals auch die Komplexität der Software. Dies gilt insbesondere dann, wenn unnötige Generalisierungen eingeführt werden.

3.2.3 Bewertung

Vorteile von YAGNI sind:

- **Weiterentwickelbarkeit**

  Der Code wird im Idealfall so schlank wie möglich und weniger kompliziert. Die Weiterentwicklung des Codes wird so nicht durch überflüssige Funktionalität behindert. Damit ist der Code leichter zu ändern und ist besser wartbar.

- **Kein Schneeballeffekt nicht geforderter Funktionen**

  Es kommt nicht zu einem Schneeballeffekt der Entwicklung nicht geforderter Features, bei dem nicht geforderte Features weitere nicht geforderte Features nach sich ziehen können.

- **Vermeiden unnötiger Einschränkungen**

  Unnötige Einschränkungen durch einen spekulativen, generalisierten Code treten nicht auf.

- **Mehr Zeit**

  Es steht mehr Zeit für die Umsetzung der geforderten Funktionalitäten zur Verfügung. Da jede zusätzliche Entwicklung Zeit für die Analyse, den Entwurf, die Programmierung, die Dokumentation, das Testen und den Support erfordert, spart man sich diesen unnötigen Aufwand, den der Kunde letztendlich nicht sieht und auch nicht bezahlt.

- **Vermeiden schwieriger Tests**

  Nicht geforderte Funktionalitäten sind schwer zu testen, da für diese auch keine Spezifikation vorliegt. Die verringerte Testbarkeit schlägt sich oft in einer erhöhten Fehlerzahl in diesen Funktionen nieder.

Der **Nachteil** einer **verkehrten Anwendung** von YAGNI ist[35]:

- **Fokussierung nur auf Anwendungsfunktionen und damit fehlende Weiterentwickelbarkeit des Systems**

  Für den Kunden ist die Infrastruktur des Systems hinter den Anwendungsfunktionen nicht sichtbar. Aber auch sie muss wegen der Weiterentwickelbarkeit gebührend – aber nicht übertrieben – berücksichtigt werden.

3.3 Don’t repeat yourself

Die Abkürzung **DRY** bedeutet "Don’t repeat yourself".

3.3.1 Historie

Das Prinzip DRY wurde von Dave Thomas und Andy Hunt in ihrem Buch "Der pragmatische Programmierer" aus dem Jahre 2003 formuliert [Tho03]. Es ist kein neues Prinzip, aber das Wortspiel "DRY" als Abkürzung für "Don't Repeat Yourself" ist ein sehr "griffiges" Akronym. Dieses Prinzip wurde 1997 durch Kent Beck als **"Once and only once"** [Bec97] bezeichnet. Früher war es als **Single-Source-Prinzip** bekannt.

3.3.2 Ziel

Das Prinzip DRY fordert, dass nichts gedoppelt werden darf wegen der Gefahr, eine Kopie bei einer Änderung zu vergessen.

Es geht darum, Fehler bei Änderungen zu vermeiden. Es sollte nichts mehrfach abgelegt werden, da man eines der **Replikate** beim Aktualisieren vergessen könnte. Alle diese Stellen sollen identisch sein, ansonsten entstehen inkonsistente Informationen und damit Fehler.

Kommentar des Autors:

Die Korrektheit einer Implementierung erfordert, dass alle redundanten Stellen identisch sind, und damit hängt eine redundante Stelle von allen anderen redundanten Stellen ab.[36]

Das Prinzip DRY betrifft **jegliche Art von Informationen** wie etwa Dokumentationen, Datenbankschemata, Build-Skripte oder Quellcode.

Das DRY-Prinzip gilt nicht nur für die Programmierung. Das DRY-Prinzip verlangt, dass jegliche Art an Information in einem Projekt genau ein einziges Mal (engl. single source) vorkommt.

[35] siehe [Mey14, p. 69]

[36] Typisch für eine Abhängigkeit ist, dass ein Abhängiger auch geändert werden muss, wenn derjenige geändert wird, von dem er abhängt.

Die Sicht von DRY, jegliche Information nur ein einziges Mal zu führen, gilt aber nicht für fehlertolerante Systeme. Fehlertolerante Systeme basieren prinzipiell auf Redundanzen.

Redundanzen können auch zur Steigerung der Performance in Echtzeitsystemen eingesetzt werden.

Bei Einhaltung des DRY-Prinzips können Änderungen nur an der einzigen Stelle des Vorkommens einer Information erfolgen, was das Pflegen dieser Information deutlich vereinfacht. Andere Stellen sind dabei nicht von der Änderung betroffen. Damit werden Abhängigkeiten reduziert und **Aktualisierungsprobleme** vermieden. Eine Replikation von Code kann beispielsweise durch das Auslagern in eine mehrfach verwendbare Methode vermieden werden. Redundanzen in den Datensätzen können durch die Normalisierung einer relationalen Datenbank beseitigt werden. Benötigt man eine bestimmte Information tatsächlich mehrfach, so sollte die Replikation unbedingt in automatischer Weise erfolgen.

3.3.3 Ursachen von Redundanzen

Ein Programmierer kann Redundanzen

- voll bewusst wie beispielsweise bei fehlertoleranten Systemen oder bei Performance-Problemen,
- aus Unkenntnis des DRY-Prinzips,
- aus Unachtsamkeit oder
- aus Zeitdruck

generieren.

Wenn sich die verschiedenen Teammitglieder nicht perfekt abstimmen, können ebenfalls leicht Redundanzen entstehen.

3.3.4 Behandlung von Redundanzen

Nichts ist einfacher, als Textstücke wie Quellcode durch Kopieren und Einfügen zu wiederholen. Es ist auch nicht in allen Fällen möglich, Replikationen zu vermeiden. So steht beispielsweise im Prototyp eines C-Programms zur Schnittstellenprüfung eine redundante Information zur eigentlichen Funktion (Funktionsname, Rückgabetyp, Typen der Übergabeparameter).

Braucht man tatsächlich redundante Informationen, so sollten diese am besten automatisch aus einer einzigen Quelle erzeugt werden.

Schreibt man Prototyp und Funktion von Hand, so ist in diesem Falle tröstlich, dass eventuelle Abweichungen beim Kompilieren oder Linken durch eine Fehlermeldung angezeigt werden. Aber auch die Lösung von Performance-Problemen oder eine fehlertolerante Lösung kann zu einer beabsichtigten Redundanz führen.

3.3.5 Bewertung

Das Prinzip DRY sollte nur in begründeten Ausnahmefällen verletzt werden.

Vorteile durch die Anwendung von DRY sind:

- **Vermeiden von Update-Fehlern**

  Die Entstehung von Fehlern infolge von Update-Problemen wird vermieden. Die Wahrscheinlichkeit des Auftretens von Fehlern durch Inkonsistenzen wird verringert. Die Informationen sind einfacher zu pflegen.

- **Abschwächung von Abhängigkeiten**

  Abhängigkeiten der Replikate untereinander werden verhindert.

- **Bessere Lesbarkeit**

  Da weniger Code geschrieben wird, trägt DRY zur besseren Lesbarkeit von Programmen bei.

- **Vermeiden mehrfacher Tests derselben Logik**

  Im Falle von Code muss ein und dieselbe Logik nicht mehrfach getestet werden.

Ein Nachteil bei der Anwendung von DRY kann entstehen durch

- **erforderliche, aber nicht vorhandene Redundanzen**

  Das DRY-Prinzip kann missbräuchlich als Argument angeführt werden, keine Redundanzen in einem System einzuführen, obwohl diese beispielsweise aus Performance-Gründen oder aus Gründen der Fehlertoleranz ihre Berechtigung hätten.

3.4 Zusammenfassung

Es werden die Entwurfsprinzipien KISS, YAGNI und DRY analysiert. Diese haben gemeinsam, dass sie Überflüssiges weglassen, aber nicht die Architektur des Systems entscheidend beeinflussen. Sie verringern jedoch Abhängigkeiten.

KISS (siehe Kapitel **3.1**) bedeutet, dass Systeme möglichst einfach und nicht komplex sein sollen. Änderungen und Erweiterungen sollen in einfacher Weise durchgeführt werden können, ohne die Stabilität des Systems zu gefährden.

YAGNI (siehe Kapitel **3.2**) fordert, dass ein Over-Engineering, spekulative Funktionen sowie vom Kunden nicht benötigte Generalisierungen bei der Entwicklung eines Systems vermieden werden sollen. Es soll nur das entwickelt werden, was der Kunde tatsächlich wünscht. Dadurch verringert sich die Komplexität des Systems. Ein Entwickler sollte nur klare Forderungen implementieren. Ein bestimmtes Programmteil

sollte erst dann entwickelt werden, wenn es auch tatsächlich gefordert wird. Das Weglassen spekulativer Funktionalität

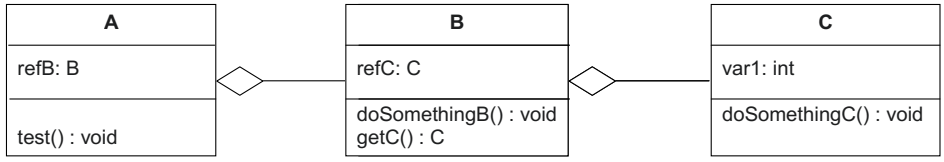
- erhöht die Änderbarkeit der Programme erheblich und
- führt zu einer schnelleren Realisierung der tatsächlich vom Kunden geforderten Funktionen.

Man muss aber trotz dieser Empfehlungen an die Erweiterbarkeit eines Systems und an den Bau einer vernünftigen Infrastruktur des Systems denken. Man darf nicht nur in solchen Features denken, die der Kunde direkt sieht.

Das DRY-Prinzip (siehe Kapitel **3.3**) verlangt, dass jegliche Art an Information in einem Projekt genau ein einziges Mal vorkommt. Durch das DRY-Prinzip sollen Aktualisierungsprobleme vermieden und dadurch die Komplexität verringert werden. Würden Replikate im System existieren, so würde es verschiedene Stellen geben, die voneinander abhängig sind, da sie stets identisch sein sollten. Muss eine dieser Stellen geändert werden, müssen alle anderen Stellen nachgezogen werden. Aus Konsistenzgründen darf dabei jedoch keine einzige Stelle vergessen werden. Beispielsweise Performance-Gründe oder fehlertolerante Architekturen können jedoch dazu führen, dass das DRY-Prinzip bewusst verletzt werden muss.

Kapitel 4

Entwurfsprinzipien für die Konstruktion schwach gekoppelter Teilsysteme



4 Entwurfsprinzipien für die Konstruktion schwach gekoppelter Teilsysteme

Abhängigkeiten stellen das größte Problem in der Softwareentwicklung dar. Abhängigkeiten zu vermeiden, ist daher für die Entwickler eine große Herausforderung.

Ein **wandelbares** oder **evolvierbares** System muss die Abhängigkeiten seiner Teile untereinander und mit der Umgebung so weit wie möglich abschwächen. Dadurch wird verhindert, dass infolge solcher Abhängigkeiten bei Änderungen oder Erweiterungen große Teile des Systems umgeschrieben werden müssen.

Erwünscht ist hingegen:

- Änderungen sollen etwa gleich viel Aufwand erfordern, egal ob sie zu Beginn einer Entwicklung oder erst später durchgeführt werden.
- Es sollte möglich sein, eine Systemkomponente relativ problemlos isoliert aus einem System "herauszuschälen", um sie isoliert zu testen oder ggf. in einem anderen System wiederverwenden zu können.

Bei heutigen Systemen erfordern spätere Änderungen in der Regel einen deutlich höheren Aufwand als zu Beginn einer Entwicklung.

Die beiden Prinzipien "Loose Coupling" und "Strong Cohesion" aus dem Jahre 1968 – oft zusammengefasst zu "Loose Coupling and Strong Cohesion" (siehe Kapitel 4.1) – sind die ältesten und nach Meinung des Autors die wichtigsten Prinzipien des Software Engineerings. Sie stammen noch aus der rein prozeduralen Welt. Dennoch können sie auch objektorientiert verwendet werden, da sie allgemeingültig formuliert sind. Diese Prinzipien verlangen eine Abschwächung der Abhängigkeiten zwischen verschiedenen Teilsystemen[37] und deren starke Kohäsion.

Kapitel 4.2 erklärt "Information Hiding" (dt. Geheimnisprinzip) für Module. Dieses Prinzip verbirgt modulinterne Daten und den Code eines Moduls – also die Implementierung – und erlaubt den Zugriff auf die Funktionalität dieses Moduls nur über eine schmale, minimale Schnittstelle.

Das Entwurfsprinzip "Separation of Concerns" (siehe Kapitel 4.3) spricht sich für eine saubere Trennung der verschiedenen Belange (Funktionalitäten) eines betrachteten Systems aus.

Das Entwurfsprinzip "Law of Demeter" (siehe Kapitel 4.4) stellt eine Interpretation von "Loose Coupling and Strong Cohesion" für objektorientierte Systeme dar.

[37] Teilsysteme, Komponenten und Module werden hier als Synonyme betrachtet.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8_4

Das "Dependency Inversion Principle" (siehe Kapitel 4.5) befasst sich speziell mit der Verwendung von Schnittstellen in Aufrufhierarchien, um Abhängigkeiten von Implementierungen der tieferen Ebene zu verhindern.

Das "Interface Segregation Principle" (siehe Kapitel 4.6) will, dass ein Client als Schnittstelle nur eine sogenannte Rollen-Schnittstelle verwendet, welche genau diejenigen Methoden enthält, die ein Client tatsächlich aufruft.

Das "Single Responsibility Principle" (siehe Kapitel 4.7) postuliert, dass ein Modul nur eine einzige Verantwortlichkeit – also eine einzige Funktionalität – als Grund für Änderungen hat. Andere Verantwortlichkeiten sollen jeweils in einem eigenen Modul liegen. Hiermit sollen bei Änderungen einer Verantwortlichkeit unbeabsichtigte Beschädigungen anderer Module vermieden werden. Dies erhöht die Stabilität.

"Dependency Lookup" und "Dependency Injection", welche in Kapitel 4.8 vorgestellt werden, sind keine Entwurfsprinzipien, aber zwei wichtige Konzepte, um Abhängigkeiten bei der Erzeugung von Objekten abzuschwächen.

4.1 Loose Coupling and Strong Cohesion

Das Entwurfsprinzip "Loose Coupling and Strong Cohesion" befasst sich mit der Modularisierung von Software. Es betrachtet die **Abschwächung von Abhängigkeiten** zwischen physischen Teilsystemen (Modulen) und den **Zusammenhalt der einzelnen Module**. Dass beide Prinzipien meist in einem Atemzug genannt werden, liegt daran, dass die beiden Größen "coupling" und "cohesion" keine unabhängigen Größen sind, sondern stark miteinander korreliert sind.

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

"Loose Coupling and Strong Cohesion" ist nach wie vor der beste Weg, um dafür zu sorgen, dass ein System testbar, stabil und änderbar wird. Auch der Code wird besser verständlich.

Wenn auch die Ziele von "Loose Coupling and Strong Cohesion" klar sind, so ist dennoch eine Quantifizierung der Erreichung dieser Ziele durch Metriken extrem schwierig.

Beim Identifizieren von in sich stark zusammenhängenden Modulen, die wechselseitig schwach gekoppelt sind, ist an erster Stelle der Problembereich zu analysieren.

So schreibt Edward Yourdon *[You79, p. 105]:*

"Adapting the system's design to the problem structure (or 'application structure') is an extremely important design philosophy."

Es kann aber nicht nur Module aus Funktionen des Problembereichs geben, sondern auch Module nur aus sogenannten "technischen" Funktionen, die im Lösungsbereich neu entstehen wie eine Datenbank-Schnittstelle, und ferner gemischte Module, welche Funktionen des Problembereichs plus neu hinzukommende technische Funktionen des Lösungsbereichs enthalten.

Klar ist der Problembereich sehr wichtig für die Modularisierung! Die Modularisierung erfolgt aber im Lösungsbereich und da darf man nicht vergessen, dass auch die technischen Funktionen, die im Lösungsbereich zu den Funktionen des Problembereichs hinzukommen, modularisiert werden.

4.1.1 Namensgebung

Statt von "Loose Coupling" spricht man auch von "Low" oder "Weak Coupling" und statt von einer "Strong Cohesion" auch von einer "High" oder "Tight Cohesion".

Leider gibt es für den Namen "Loose Coupling and Strong Cohesion" keine allgemein anerkannte Formulierung.

So nennt Alan M. Davis in seinem Buch "201 Principles of Software Development" aus dem Jahre 1995 [Dav95] dieses Prinzip "Use Coupling and Cohesion".

4.1.2 Historie

Der Vorschlag einer losen Kopplung und starken Kohäsion wurde zuerst von Larry Constantine im Jahre 1968 veröffentlicht (siehe [Bar68]). Er konzipierte die Qualitätsmerkmale "Coupling" und "Cohesion" in den sechziger Jahren als Teil des Structured Design für prozedurale Systeme.

4.1.3 Ziel

Es geht bei "Loose Coupling and Strong Cohesion" letztendlich um

- die **Stabilität** eines Systems gegen auftretende Fehler sowie
- eine leichte **Testbarkeit**, **Änderbarkeit/Erweiterbarkeit** und **Wartbarkeit**.

Je schwächer die Kopplung zwischen den Modulen und je stärker die Kohäsion innerhalb eines Moduls ist, umso besser ist die Modularisierung.

Bei schwacher Kopplung breiten sich Probleme weniger aus, da die wechselseitigen Abhängigkeiten der Module schwach sind. Dadurch werden die Systeme **stabiler**.

Schwach gekoppelte Module können leichter isoliert voneinander **getestet** werden.

Stark gekoppelte Systeme sind schwer zu **ändern** oder zu **erweitern**, da eine Änderung an einer bestimmten Stelle Änderungen an anderen Stellen nach sich zieht. Schwach gekoppelte Komponenten können überdies leichter in anderen Systemen **wiederverwendet** werden.

Die **Wartbarkeit** eines Systems wird durch die inneren Eigenschaften eines Systems bestimmt, insbesondere durch die **Testbarkeit** und **Änderbarkeit/Erweiterbarkeit**.

Durch einen starken logischen Zusammenhalt der Funktionen innerhalb der Module steigt die Wahrscheinlichkeit, dass im Falle von Änderungen eines Features nur ein einziges Modul betroffen ist. Im Idealfall sind alle Aspekte eines Features als eine **einzige Verantwortlichkeit** nach Robert C. Martin (siehe Kapitel 4.6) in einem eigenen Modul zusammengefasst.

Wären logisch zusammengehörige Methoden über verschiedene Module verstreut, so wären bei notwendigen Änderungen mit hoher Wahrscheinlichkeit mehrere Module betroffen.

4.1.4 "Coupling"

Stevens et al. [Ste74] definierten "Coupling" als ein Maß für die Stärke der Beziehung zwischen zwei Modulen des Entwurfs.

Hierzu schreibt Edward Yourdon [You79, p. 85]:

"The measure that we are seeking is known as coupling; it is a measure of the strength of interconnection. Thus, 'highly coupled' modules are joined by strong interactions; 'loosely coupled' modules are joined by weak interactions; 'uncoupled' or 'decoupled' modules have no interactions and are, thus, independent."

"If two modules are highly coupled, then there is a high probability that a programmer trying to modify one of them will have to make a change to the other."

"Loose Coupling" bedeutet, dass ein Modul nicht wesentlich von einem anderen Modul abhängt.

Würde es überhaupt keine Kopplung geben, so würden sich Änderungen stets nur lokal auf ein einziges Modul auswirken. Das würde andererseits aber bedeuten, dass die Module unabhängig voneinander arbeiten und keines der Module die Funktionen eines anderen Moduls benutzt.

Wenn Module zusammenarbeiten, liegt immer eine gewisse Kopplung vor.

Die beiden folgenden Abbildungen symbolisieren eine lose und eine starke Kopplung zwischen zwei Modulen. Dabei deuten die Pfeile an, welches Modul die Funktionen des anderen Moduls benutzt:

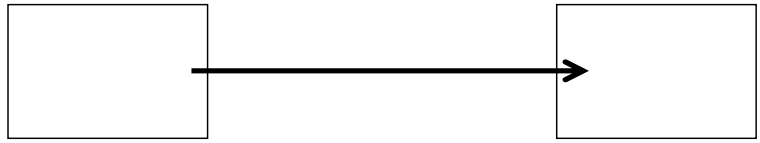


Abbildung 4-1 Lose Kopplung zwischen zwei Modulen (symbolisch)

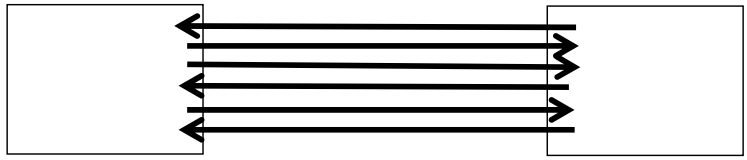


Abbildung 4-2 Starke Kopplung zwischen zwei Modulen (symbolisch)

4.1.5 "Cohesion"

Was "Cohesion" (dt. Zusammenhalt, Kohäsion) ist, beschreibt Edward Yourdon wie folgt [You79, p. 106]:

"What we are considering is the cohesion of each module in isolation – how tightly bound or related its internal elements are one to another. Other terms sometimes used to denote the same concept are 'modular strength', 'binding' and 'functionality'".

Kohäsion (engl. **cohesion**) als Maß für den Zusammenhalt der Funktionen eines Moduls gibt an, wie stark die Funktionen des betreffenden Moduls untereinander in Bezug stehen.

Ziel ist es, Module mit starker interner Kohäsion zu entwickeln. Solche Module sind einfacher zu warten.

Wenn die Kohäsion innerhalb der Module nur schwach ist, gibt es Schwierigkeiten, die Ursachen von Fehlern zu identifizieren oder diejenigen Stellen zu lokalisieren, die anzupassen sind, um mit neuen Requirements fertig zu werden.

"Strong Cohesion" wurde bereits von Tom DeMarco [DeM79] mit den Worten gefordert:

"The more valid a module's reason for existing as a module, the more cohesive it is."

4.1.6 Korrelation von "Coupling" und "Cohesion"

"Coupling" und **"Cohesion"** sind **Qualitätsmerkmale** eines Systems, die sich auf die Module eines Systems beziehen. Diese beiden Qualitätsmerkmale sind eng korreliert. Je stärker die Module in sich zusammenhängend sind ("high internal cohesion"), desto schwächer sind sie wechselseitig gekoppelt ("loose external coupling").

Edward Yourdon schreibt [You79, p. 106]:

"Clearly, cohesion and coupling are interrelated. The greater the cohesion of individual modules in the system, the lower the coupling between modules will be."

Änderungen an den Anforderungen eines Systems erfordern bei Vorliegen einer schwachen Kopplung zwischen den Modulen weniger Änderungen des Systems als bei starker Kopplung.

4.1.7 Schnittstellen[38]

Wenn Module schwach wechselwirken sollen, muss die Implementierung eines Moduls verborgen sein.

Bleiben die Schnittstellen der Module eines Systems stabil, so wird das Risiko einer parallelen Bearbeitung der einzelnen Module eines Systems durch jeweils verschiedene Projektmitarbeiter erheblich gesenkt. Dies liegt vor allem daran, dass kein Modul von den Implementierungsentscheidungen eines anderen Moduls betroffen ist.

Eine **lose Kopplung** (engl. **loose coupling**) bedeutet in der Softwarearchitektur, dass die Module einer Software nur über ihre **Schnittstellen** mit anderen Modulen kommunizieren dürfen.

Die generelle Verwendung von Schnittstellen für die Module eines Systems entspricht einer Abschwächung von Abhängigkeiten.

Das aufrufende Programm hängt bei Einhaltung der Verträge[39] einer Schnittstelle nur noch von der Schnittstelle selbst und nicht von deren Implementierung ab.

Wird die fehlerhafte Implementierung eines solchen Moduls korrigiert, so muss der Aufrufer nicht verändert werden.

[38] zum Begriff der "Schnittstelle" siehe Begriffsverzeichnis
[39] siehe hierzu Kapitel 5.1

4.1.8 Bewertung

Ein Entwurf gilt – wie schon erwähnt – nach "Loose Coupling and Strong Cohesion" als gut, wenn

- innerhalb eines Moduls eine möglichst hohe Bindungsstärke oder starke Kohäsion und
- zwischen den verschiedenen Modulen eine möglichst schwache Wechselwirkung

besteht.

Die **Vorteile** von schwach gekoppelten Modulen mit einer starken inneren Köhäsion sind[40]:

- **Schwächere Abhängigkeiten**

  Die Abhängigkeiten zwischen Modulen werden verringert.

- **Bessere Wiederverwendbarkeit**

  Die Wiederverwendbarkeit der Module wird erhöht.

- **Änderbarkeit und Austauschbarkeit**

  Die Änderbarkeit und Austauschbarkeit von Modulen wird erhöht, wenn die Schnittstellen stabil bleiben.

- **Stabilität des Systems**

  Die Stabilität des Systems wird erhöht, da die Ausbreitung von Fehlern verringert wird.[41]

- **Arbeitsteilige Entwicklung**

  Nach Festlegung der Schnittstellen können die Module jeweils getrennt für sich von verschiedenen Entwicklern parallel realisiert und nach ihrer Fertigstellung getrennt getestet werden.

4.1.9 Weitergehende Betrachtungen zu Schnittstellen

Die Betrachtung dieses Kapitels geht über die Aussagen von "Loose Coupling and Strong Cohesion" hinaus.

---

[40] Der Vorteil des "Single Responsibility Principle", nämlich dass bei der Änderung der einzigen Verantwortlichkeit eines Moduls eine andere Verantwortlichkeit nicht in unabsichtlicher Weise beschädigt wird, ist hierbei noch nicht berücksichtigt.

[41] Dies betrifft Fehlerfälle und Fehleingaben im Betrieb.

Neben Methodenköpfen als Aufrufschnittstellen von Funktionen umfasst eine Schnittstelle auch die Definition der **Semantik der Funktionen**[42], also die Spezifikation der Funktionen.

Der Ansatz, dass man gegen Schnittstellen programmiert, bedeutet, dass man nicht nur gegen Methodenköpfe der Schnittstellenmethoden, sondern gegen die **Verträge**[43] der Schnittstellenmethoden – letztendlich gegen die Spezifikation – programmiert.

Natürlich ist man dann aber von der gesamten Schnittstelle, also auch von deren Spezifikation abhängig. In der Praxis ist dieser Ansatz aber meist nur Theorie: Man beschränkt sich bei der Definition von Schnittstellen oft nur auf die Methodenköpfe und lässt die Semantik explizit weg, da diese nicht einfach zu beschreiben ist.

Zu einer Schnittstelle können nicht nur **funktionale Forderungen** gehören, sondern auch **nicht funktionale Eigenschaften** wie beispielsweise Performance[44] oder Security[45].

Dass man gegen Schnittstellen programmiert, kann man sowohl bei Vorliegen **gleichberechtigter Module auf gleicher Ebene** (engl. **peers**), als auch in einer **Aufrufhierarchie** betrachten:

- Auf derselben Ebene – also unter Gleichberechtigten – hat man gleichberechtigte, nebenläufige Module, die jeweils eine Schnittstelle tragen.
- Betrachtet man eine Aufrufhierarchie, so muss jedes aufgerufene Modul eine Abstraktion – meist eine Schnittstelle – bereitstellen, gegen welche der Aufrufer und der Aufgerufene programmiert werden. Siehe hierzu das "Dependency Inversion Principle" (DIP)[46] in Kapitel 4.5.

4.2 Information Hiding

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

[42] Die Semantik der Funktionen kann in der Regel nicht in den gängigen Programmiersprachen ausgedrückt werden. Das Konzept "Design by Contract" (siehe Kapitel 5.1) setzt an dieser Stelle an: Die Verträge von "Design by Contract" stellen ein Mittel dar, um die Semantik besser zu beschreiben, als es mit einer natürlichen Sprache möglich ist.

[43] siehe Kapitel 5.1

[44] Da jedes System eine andere Performance hat, können für die Performance nur gewünschte, geschätzte Werte angegeben werden. Soll es jedoch beispielsweise einen Timeout nach einer gewissen Zeitspanne geben, so muss garantiert werden, dass der Client der API nach dieser Zeit eine Antwort in Form eines Timeouts erhält.

[45] Auch wenn Security oftmals gerne als nicht funktional (im Sinne der Anwendungsfunktionen) angesehen wird, stecken dennoch hinter der Security dedizierte Sicherheitsfunktionen. Gerade Web-Schnittstellen können mit verschiedensten Sicherheitsmechanismen geschützt werden. Der verwendete Mechanismus ist in der Schnittstellenbeschreibung zu benennen.

[46] Das DIP lässt auch abstrakte Klassen und damit Abhängigkeiten von Implementierungen zu.

4.2.1 Historie

David L. Parnas erfand das Prinzip "Information Hiding". Dieses Entwurfsprinzip stammt noch aus der prozeduralen Welt, wurde später aber auch zu einem zentralen Prinzip der Objektorientierung. Es wurde durch die Veröffentlichung "On the criteria to be used in decomposing systems into modules" von David L. Parnas im August 1971 an der Carnegie Mellon University in Pittsburgh, Pennsylvania, in der Fachwelt bekannt [Par71]. Hierbei schrieb Parnas, wie man Module finden sollte:

"… it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

***"Every module is characterized by its knowledge of a design decision which it hides from all others.** Its interface or definition was chosen to reveal as little as possible about its inner workings."*

Die Schnittstellen sollen so wenig wie möglich über das Innere der Module freigeben. Sie sollen **Designentscheidungen verbergen**.

Auch wenn der Begriff einer **"einzigen Verantwortlichkeit"** eines Moduls damals noch nicht bekannt war, so schlug David L. Parnas doch bereits im Jahre 1971 eine **Kapselung auf Grund von Änderungswahrscheinlichkeiten** vor.

Im Jahre 2002 verknüpfte Robert C. Martin die **Änderungswahrscheinlichkeit eines Moduls** mit dessen innerer Kohäsion und legte fest, dass ein Modul aus Stabilitätsgründen nur eine **einzige Verantwortlichkeit** und somit auch nur einen einzigen Grund für eine Änderung haben darf.

Durch die Arbeiten von Robert C. Martin zum "Single Responsibility Principle" (siehe Kapitel 4.7) wurde der Begriff der "Änderungswahrscheinlichkeit" später dann populär und allgemein akzeptiert.

4.2.2 Ziel

Zur Erreichung von Stabilität sollen nach Parnas Designentscheidungen in Modulen gekapselt und damit Implementierungen verborgen werden.

4.2.3 Bewertung

"Information Hiding" ist für die Objektorientierung so zentral, dass objektorientierte Programmiersprachen eine Kapselung durch die Vergabe von Schutzrechten wie privat oder geschützt für die Elemente einer Klasse, d. h. für die Variablen und Methoden, unterstützen.

Der **Vorteil** von "Information Hiding" ist:

- **Unabhängigkeit von der Implementierung und damit leichter Austausch der Implementierung**

  Die Abhängigkeit von Implementierungen entfällt. Bleibt eine Schnittstelle unverändert, dann kann dennoch die Implementierung problemlos von Version zu Version geändert werden, da sie gekapselt ist.

Auch Frameworks oder Datenbanken stellen ihre Leistungen über APIs zur Verfügung und verbergen dabei ihre Implementierung.

4.3 Separation of Concerns

Das Prinzip **"Separation of Concerns"**[47] von Edsger W. Dijkstra ist zwar verwandt mit dem später aufgestellten "Single Responsibility Principle"[48] von Robert C. Martin (siehe Kapitel 4.7), ist aber grundsätzlich anders einzuordnen. Im Folgenden wird der Zusammenhang zwischen diesen beiden Prinzipien beschrieben.

"Separation of Concerns" ist ein allgemeines Denkprinzip, das besagt, dass ein "concern"[49] eines Systems sauber von den anderen "concerns" dieses Systems abzugrenzen ist. Ein **"concern"** repräsentiert dabei ein bestimmtes Interesse in einem System, einen **Belang**.

"Separation of Concerns" gilt nicht nur für Software, sondern generell für beliebige Systeme.[50]

Gefordert wird, dass die verschiedenen Belange eines Systems voneinander sauber abgetrennt sind. Es wird jedoch nicht gefordert, dass sie eigene physische Module darstellen.

4.3.1 Historie

In der Veröffentlichung "On the role of scientific thought" postulierte Edsger W. Dijkstra [Dij74]:

"It is what I sometimes have called the separation of concerns, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by focussing one's attention upon some aspect: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant."

---

47 abgekürzt als SoC
48 abgekürzt als SRP
49 Ein "concern" kann mit Belang, Anliegen oder Verantwortung übersetzt werden.
50 Das "Single Responsibility Principle" (siehe Kapitel 4.7) gilt hingegen nur für das Finden von Modulen im Lösungsbereich eines Softwaresystems.

4.3.2 Ziel

"Separation of Concerns" will die verschiedenen Belange eines Systems voneinander trennen, damit diese jeweils getrennt für sich betrachtet werden können.

Für die Abbildung in einen Belang werden die anderen Belange des Systems nicht betrachtet. Diese werden jeweils abgetrennt[51], damit man sich voll auf eine einzige Aufgabe konzentrieren kann und nicht durch die Wechselwirkung mit anderen Aufgaben verwirrt wird. Das Prinzip "Separation of Concerns" von Edsger W. Dijkstra [Dij74] teilt also ein System in verschiedene voneinander getrennte **Belange** auf, die sich in ihrer Funktionalität nicht überlappen sollen.

Das Ziel von "Separation of Concerns" ist es, ein System so in Belange zu schneiden, dass diese Teile unabhängig von den anderen Teilen entwickelt und verändert werden können. Ein Fehler in einem Teil soll sich dabei nicht auf die anderen Teile auswirken.

Der Einsatz von "Separation of Concerns" trennt im Falle von Programmcode beispielsweise fachlichen von technischem Code, aber insbesondere auch unterschiedliche fachliche Belange innerhalb einer Domäne bzw. eines Fachgebiets.

4.3.3 Umsetzung in die Programmierung

Umgesetzt werden kann "Separation of Concerns" beim Programmieren beispielsweise über

- Schichten in Schichtenmodellen,
- Klassen in objektorientierten Sprachen,
- Funktionen bzw. Prozeduren in prozeduralen Sprachen,
- Services in einer serviceorientierten Architektur (SOA) oder
- Klassen und Aspekte in der aspektorientierten Programmierung.

4.3.4 Bewertung

Ohne eine saubere Abgrenzung der verschiedenen Belange entstehen zu viele Abhängigkeiten zwischen den verschiedenen Belangen eines Systems. Eine saubere Trennung erlaubt es, diese Belange jeweils einzeln für sich zu betrachten.

[51] Eine saubere Abgrenzung ist aber nur dann möglich, wenn jeweils eine starke Kohäsion innerhalb der verschiedenen Belange eines Systems besteht.

Vorteile von Separation of Concerns sind:

- **Fokussierung auf getrennte Belange**

  Dies reduziert die Komplexität und stellt das jeweils Wichtige in den Vordergrund.

- **Erhöhung der Wiederverwendbarkeit**

  Sauber getrennte Belange können wiederverwendet werden.

4.4 Law of Demeter[52]

Nach K. Lieberherr, I. Holland und A. Riel wird mit dem **"Law of Demeter"** (dt. **"Gesetz von Demeter"**) eine programmiersprachenunabhängige Richtlinie, welche die Idee einer losen Kopplung von Modulen und deren Kapselung in der objektorientierten Welt aufgreift, beschrieben [Lie88, p. 323].

Das "Law of Demeter" kann grundsätzlich als eine Projektion des Ansatzes von "Loose Coupling and Strong Cohesion" auf die **Objektorientierung** betrachtet werden [liebla].

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

4.4.1 Historie

Das **"Law of Demeter"** wurde erstmals im Herbst 1987 von Ian Holland an der Northeastern University in Boston im Rahmen des Forschungsprojektes "Demeter"[53] aufgestellt [liebla]. Andere Namen für dieses Prinzip sind:

- **"Principle of Least Knowledge"** [lieber],
- **"Law of Goodstyle"** [Lie88, p. 323] bzw.
- **"Law of Demeter for Functions/Methods"** (LoD-F) [appint].

Das "Law of Demeter" betrachtet Methodenaufrufe und gibt eine Richtlinie vor, welche Aufrufe erwünscht sind und welche unterlassen werden sollten. Es existieren die folgenden **Ausprägungen** des "Law of Demeter" durch Lieberherr, Holland und Riel:

- "Weak Law of Demeter" (siehe Kapitel 4.4.6.1),
- "Strong Law of Demeter" (siehe Kapitel 4.4.6.2),
- Objektform (siehe Kapitel 4.4.6.3) und
- Klassenform (siehe Kapitel 4.4.6.4).

[52] abgekürzt als LoD

[53] ein Projekt zur Entwicklung verschiedener Tools, welche die Softwareentwicklung unter Einhaltung des "Law of Demeter" leichter machen

4.4.2 Ziel

K. Lieberherr, I. Holland und A. Riel [liebob, p. 325] formulierten den Zweck des "Law of Demeter" erstmals in ihrer 1988 erschienenen Veröffentlichung mit den Worten:

*"The motivation behind this law is to ensure that the software is as **modular** as possible."*

Ziel des "Law of Demeter" ist es, Abhängigkeiten zwischen den einzelnen Komponenten zu reduzieren [liebfo] und so ein **modulares**, objektorientiertes Softwaresystem mit minimalen Abhängigkeiten bereitzustellen.[54]

Die folgenden Unterkapitel beschreiben die konzeptionelle Umsetzung dieses Ziels.

4.4.3 "Schüchterner" Code – Einschränkung verketteter Methodenaufrufe

Spricht man in der objektorientierten Softwareentwicklung von Modularität und einer schwachen Kopplung, so setzt dies eine Einschränkung der gegenseitigen Bekanntheit der einzelnen Softwarekomponenten und damit eine Einschränkung ihrer Kommunikation voraus.[55]

Jedes Objekt darf nur seinen direkten Nachbarn kennen, nicht aber den Nachbarn des Nachbarn [liebla].

Die direkten Nachbarn werden als **Freunde** bezeichnet, die anderen Objekte sind die **Fremden**. Karl Lieberherr [lieber] schreibt hierzu:

*"Each unit should only talk to its **friends**. **Don't talk to strangers**."*

Methoden, welche mit dem Gesetz von Demeter konform sind, dürfen nur ihre direkten Nachbarn kennen. Indirektionen zur Verkettung "fremder" Klassen dürfen nicht verwendet werden (**Geheimnisprinzip**).

Im Zuge der Einhaltung des Entwurfsprinzips "Information Hiding" wird durch das "Law of Demeter" die Zahl der **zulässigen Verkettungen** von Methodenaufrufen drastisch limitiert.

---

[54] Damit kann das "Law of Demeter" auch als ein Spezialfall des Entwurfprinzips "loose coupling" betrachtet werden kann [liebla].

[55] Die Strategie, Beziehungen über mehrere Klassen hinweg nicht zu erlauben, wird auch beispielsweise in David Bocks Paperboy-Programm [bockda] demonstriert. Kein Kunde würde dem Zeitungsjungen den Geldbeutel in die Hand geben, damit dieser sich selbst seinen Lohn entnehmen kann. Stattdessen entnimmt der Kunde selbst das Geld aus seinem Geldbeutel und reicht es dem Zeitungsjungen.

Code, der nach dem "Law of Demeter" geschrieben wird, bei dem ein Objekt nicht über ein zweites Objekt auf ein drittes Objekt zugreifen darf, wird als **"schüchterner" Code** bezeichnet.

Die konkrete Umsetzung des Gesetzes von Demeter variiert je nach verwendeter Programmiersprache.[56] Lieberherr, Holland und Riel formulierten das "Law of Demeter" deshalb gesondert für verschiedene Programmiersprachen (Smalltalk-80, CLOS, Eiffel und C++) [Hol89].

Bei Programmiersprachen, welche als Zugriffsoperator auf Methoden oder Attribute einen Punkt oder auch einen Pfeil einsetzen wie z. B. C++, wird das "Law of Demeter" auf die einfache Vorschrift beschränkt, nur einen einzigen Punkt beziehungsweise nur einen einzigen Pfeil pro Anweisung zu verwenden.

4.4.4 Konzeptionelles Beispiel für das "Law of Demeter"

Die in Abbildung 4-3 dargestellte Klassenkonstellation soll als Beispiel für erlaubte und unerlaubte Methodenaufrufe dienen:

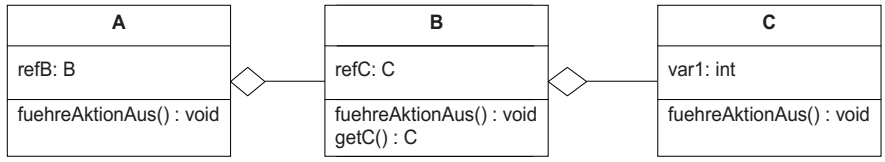


Abbildung 4-3 Beispiel

Innerhalb der Methode `fuehreAktionAus()` der Klasse `A` darf auf die befreundete Klasse `B` über die Referenz `refB` und deren Methoden zugegriffen werden. Ein Aufruf wie beispielsweise der von `refB.fuehreAktionAus()` ist demnach vollkommen legitim, denn `A` spricht nur mit dem Freund `B`.

Erfolgt innerhalb der Methode `fuehreAktionAus()` der Klasse `A` jedoch ein Aufruf von `refB.getC().fuehreAktionAus()`, so wird über die Klasse `B` hinweg auf eine der Klasse `A` unbekannte Klasse `C` zugegriffen. Ein derartiger Aufruf ist ein Beispiel für die Verletzung des "Law of Demeter": Das **Geheimnisprinzip** und folglich das Law of Demeter wird durch einen derartigen Methodenaufruf verletzt.

Eine Verletzung kann beispielsweise dadurch vermieden werden, indem die Klasse `B` einen Dienst anbietet, welcher die Funktionalität der zuvor in unachtsamer Weise aufgerufenen Methode der Klasse `C` kapselt (siehe Kapitel 4.4.5).

Neben der Verletzung des "Law of Demeter" wird im dargestellten Beispiel auch das Prinzip des "Information Hiding" verletzt, da die Klasse `B` ihre eigentlich private Refe-

[56] Eine Übertragung des "Law of Demeter" auf die Programmiersprache Java fehlt in den ursprünglichen Formulierungen. Aus diesem Grunde wird in Kapitel 4.4 versucht, das "Law of Demeter" möglichst im Sinne der Erfinder auf Java zu übertragen.

renz vom Typ der Klasse C anderen Klassen über eine get-Methode öffentlich zur Verfügung stellt.

Die achtlose Verwendung von **get-** und **set-Methoden** verletzt das Prinzip des **"Information Hiding"**. Daher sollten derartige Methoden nur dann angeboten werden, wenn dies zwingend notwendig ist.

4.4.5 Wrapper-Methoden zur Kommunikation mit Fremden

Soll das Wissen über andere Komponenten und deren Struktur auf Freunde beschränkt werden, so muss die Kommunikation zweier Fremder (hier der Klassen A und C) durch eine zusätzliche Methode – hier in der dazwischenstehenden Klasse B – gekapselt werden. Diese Methoden werden als **Wrapper-Methoden**[57] bezeichnet [App96].

Wrapper-Methoden[58] zur Kommunikation mit Fremden ermöglichen auf indirektem Weg die **Kommunikation zwischen zwei nicht direkt benachbarten Objekten**.

Eine solche Wrapper-Methode stellt einen "höheren" Dienst mit mehr Logik als etwa eine einfache get-Methode dar.

4.4.6 Formen des "Law of Demeter"

Beim Gesetz von Demeter gibt es verschiedene **Ausprägungen**, Interpretationen und Entwicklungsstufen. Zunächst unterschieden die Erfinder des "Law of Demeter", K. Lieberherr, I. Holland und A. Riel im Jahre 1988 in ihrer Veröffentlichung [Lie88] ein "**Weak Law of Demeter**" von einem "**Strong Law of Demeter**". Im darauffolgenden Jahr sprachen K. Lieberherr und I. Holland hingegen von zwei weiteren Kategorien, einer "**Objektform**" und einer "**Klassenform**" [Lie89]. Die folgenden Unterkapitel befassen sich mit diesen Ausprägungen.

4.4.6.1 "Weak Law of Demeter"

Das "**Weak Law of Demeter**" wurde im Jahre 1988 von K. Liebherr, I. Holland und A. Riel wie folgt definiert [Lie88, p. 329]:

"The Weak Law of Demeter defines instance variables as being BOTH the instance variables that make up a given class AND any instance variables inherited from other classes."

Beim "Weak Law of Demeter" darf innerhalb einer Methode sowohl auf die eigenen Instanzvariablen als auch auf die von der Basisklasse geerbten Variablen zugegriffen werden.

[57] auch als Vermittler-Methoden bezeichnet

[58] Das englische Wort "to wrap" bedeutet "umwickeln". Der Begriff "Wrapper-Methoden" bzw. "Wrapper-Klassen" ist auch in anderem Zusammenhang gebräuchlich.

Wird das "Weak Law of Demeter" angewandt, so wirken sich Änderungen der Attribute einer Basisklasse auch auf die Methoden von abgeleiteten Klassen aus, welche die von den Änderungen betroffenen Instanzvariablen der Basisklasse direkt verwenden [liebob, p. 329].

4.4.6.2 "Strong Law of Demeter"

Während das "Weak Law of Demeter" den Zugriff von Methoden einer abgeleiteten Klasse auf geerbte Variablen zulässt, schränkt das "Strong Law of Demeter" den Zugriff auf geerbte Variablen ein [Lie88, p. 329]:

"The Strong Law of Demeter defines instance variables as being ONLY the instance variables that make up a given class. Inherited instance variable types may not be passed messages."

Gemäß dem "Strong Law of Demeter" wird eine Variable nur dann als Instanzvariable bezeichnet, wenn sie direkter Bestandteil der betrachteten Klasse ist.

Wrapper-Methoden für Attribute der Basisklasse beim "Strong Law of Demeter"

Ein Zugriff einer Methode einer abgeleiteten Klasse auf eine Variable, welche durch Vererbung an diese abgeleitete Klasse weitergereicht wurde, ist nach dem "Strong Law of Demeter" nicht legitim.

Deshalb fordert das "Strong Law of Demeter" in der Basisklasse zusätzliche Methoden, die ebenfalls Wrapper-Methoden genannt werden. Diese **kapseln** die **konkrete Struktur** einer Basisklasse und erlauben dennoch den Zugriff auf deren Attribute durch eine abgeleitete Klasse.

Wrapper-Methoden beim "Strong Law of Demeter" für die Attribute einer Basisklasse fungieren als Vermittler zwischen diesen Attributen der Basisklasse und den Methoden einer abgeleiteten Klasse.

Dabei ist es wichtig, dass diese Wrapper-Methoden in der Basisklasse implementiert sind und nicht von den abgeleiteten Klassen überschrieben werden.

Kommt es beim "Strong Law of Demeter" zu Änderungen der Attribute der Basisklasse, so betrifft dies nur die entsprechende Wrapper-Methode, nicht aber die Methoden der abgeleiteten Klassen.

Das "Strong Law of Demeter" unterstützt damit vor allem das Prinzip des **"Information Hiding"** stärker als das "Weak Law of Demeter" und reduziert somit die Abhängigkeiten zwischen einer Basisklasse und ihren abgeleiteten Klassen.

4.4.6.3 Objektform

Die Formulierung des Gesetzes von Demeter in der **Objektform** ist die am häufigsten verwendete Form.

Mit der Objektform legten Lieberherr, Holland und Riel die Grenzen fest, innerhalb derer Methodenaufrufe auf Objekten mit dem "Law of Demeter" konform sind. Es wurde definiert, welche Objekte als **befreundet** betrachtet werden dürfen.

Im Folgenden wird die Definition der **Objektform** nach I. Holland [Hol89] beschrieben. Eine Methode `m` eines Objektes `o` darf nach dem "Law of Demeter" nur die folgenden Methoden aufrufen:

• des Objektes `o` selbst	(1) im folgenden Beispiel
• der an die Methode `m` übergebenen Objekte	(2) im folgenden Beispiel
• von lokal im Objekt `o` neu erzeugten Objekten	(3) und (4) im folgenden Beispiel
• eines globalen Objektes des Objekts `o`	Im folgenden Beispiel gibt es kein globales Objekt. Ein solches wäre in Java `public static`

Folgender Programmcode[59] zeigt hierfür einige konkrete Beispiele in der Programmiersprache Java:

```
// Datei: LawOfDemeterObjektform.java

public class LawOfDemeterObjektform
{
   private A aRef = new A();

   private void fuehreAktionAus()
   {
   }

   public void beispielAufrufe (B bRef)
   {
      C cRef = new C();
      this.fuehreAktionAus();    //(1) -> eigenes Objekt
      bRef.fuehreAktionAus();    //(2) -> übergebenes Objekt
      cRef.fuehreAktionAus();    //(3) -> selbsterzeugtes Objekt
                                 //(immediate part class)
      aRef.fuehreAktionAus();    //(4) -> selbsterzeugtes Objekt
   }
}
```

Nach Karl Lieberherr [Lie95] sagt das "Law of Demeter" im Wesentlichen:

*"[…] you should only talk to yourself (**current class**), to close relatives (**immediate part classes**), and **friends** who visit you (**argument classes**). But you never talk to strangers." [Lie95, p. 203]*

[59] angelehnt an [Fri04]

4.4.6.4 Klassenform

Die **Klassenform** ist der Objektform sehr ähnlich, wird jedoch nicht nur auf Objekte, sondern allgemein auf Klassen bezogen. Hierbei findet eine Unterscheidung in eine strikte und eine minimierte Form statt.

Die Definitionen der strikten und der minimierten Form nach [Hol89] und [Ber11] werden im Folgenden vorgestellt.

Strikte Form: Eine Methode `m` der Klasse `A` darf folgende Methoden aufrufen:

- Methoden der Klasse `A` selbst,
- Methoden der Klasse eines Übergabeparameters von `m`,
- Methoden von Klassen, deren Instanzen lokal neu erzeugt wurden, oder
- Methoden einer Klasse eines globalen[60] Objektes.

Minimierte Form: Die minimierte Form lockert die Einschränkungen der strikten Form auf. Sie erlaubt den Zugriff auf Methoden weiterer Klassen, nämlich auf

- Methoden einer Klasse, welche stabil ist oder eine stabile Schnittstelle implementiert,
- Methoden einer Klasse, welche die zuvor bei der strikten Form genannten Bedingungen nicht erfüllen, sofern der direkte Zugriff aus Laufzeit-Effizienzgründen nötig ist, und
- Methoden zur Objekterzeugung.

Diese Form des "Law of Demeter" ist die schwächste Ausprägung des Gesetzes und bringt somit die wenigsten Einschränkungen für erlaubte Methodenaufrufe mit sich.

4.4.7 Programmierbeispiel für das "Weak" und "Strong Law of Demeter"

K. Lieberherr, I. Holland und A. Riel erläuterten den Unterschied des "weak" und "strong" LoD in "Object-Oriented Programming: An Objective Sense of Style" [Lie88, pp. 329,330] beispielhaft an einem Obstkorb. Hierbei befinden sich in einem Obstkorb (Klasse `Obstkorb`) jeweils ein einziger Apfel, eine einzige Orange und eine einzige Pflaume. Die Klassen `Apfel`, `Orange` und `Pflaume` sind von der Klasse `Frucht` als Basisklasse abgeleitet, welche ein Attribut `gewicht` besitzt.

Zur Berechnung des Gewichts der unterschiedlichen Früchte des Obstkorbs sind unterschiedliche Formeln nötig, da sich der eigentliche Fruchtanteil je nach Sorte unterscheidet. So muss beispielsweise bei einer Pflaume das Gewicht des Kerns abgezogen werden und bei einer Orange das Gewicht der Schale. Es soll das Gewicht jeder einzelnen Fruchtsorte mit Hilfe der entsprechenden Formel ermittelt werden.

[60] In Java kann ein Objekt mithilfe von `public static` als global definiert werden.

Wird das **"Weak Law of Demeter"** angewandt, so besitzen die Klassen `Apfel`, `Orange` und `Pflaume` jeweils eine eigene Methode mit der Bezeichnung `berechneGewicht()`, welche das Gewicht anhand des von der Klasse `Frucht` geerbten Attributes `gewicht` und eines Prozentsatzes berechnet. Das folgende Beispiel zeigt die Implementierung der abstrakten Klasse `Frucht`, der Klasse `Obstkorb` und – beispielhaft für eine konkrete Frucht – der Klasse `Apfel` unter Berücksichtigung des **"Weak Law of Demeter"**:

```
// Datei: Frucht.java

public abstract class Frucht
{
   protected float gewicht = 1.0f;
   public abstract float berechneGewicht();
}

// Datei: Apfel.java

public class Apfel extends Frucht
{
   public float berechneGewicht()
   {
      return gewicht*0.85f;   // willkürlich gewählt
   }
}

// Datei: Obstkorb.java

import java.util.ArrayList;
import java.util.List;

public class Obstkorb
{
   private List<Frucht> fruechte = new ArrayList<Frucht>();

   public Obstkorb()
   {
      fruechte.add (new Apfel());
   }

   public float berechneGewicht()
   {
      float obstkorbGewicht = 0.0f;

      for(Frucht frucht : fruechte)
      {
         obstkorbGewicht += frucht.berechneGewicht();
      }
      return obstkorbGewicht;
   }
}
```

Während der Zugriff der abgeleiteten Klasse `Apfel` auf das Attribut `gewicht` der Basisklasse `Frucht` nach dem "Weak Law of Demeter" erlaubt ist, ist gerade dieser Zugriff nach dem "Strong Law of Demeter" unzulässig. Wird das "Strong Law of

Demeter" befolgt, so kapselt die Wrapper-Methode `berechneProzentualesGewicht()`, welche sich in der Basisklasse `Frucht` befindet und von den abgeleiteten Klassen aufgerufen wird, den Zugriff auf das Attribut `gewicht`. Die abgeleiteten Klassen verfügen über keine Informationen bezüglich der Implementierung dieser Wrapper-Methode und der Attribute der Basisklasse.

Folgender Programmcode zeigt die Implementierung des Obstkorb-Beispiels unter Einhaltung des **"Strong Law of Demeter"**:

```
// Datei: Frucht.java

public abstract class Frucht
{
   private float gewicht = 1.0f;
   public abstract float berechneGewicht();
   protected final float berechneProzentualesGewicht (float prozent)
   {
      return gewicht * prozent;
   }
}

// Datei: Apfel.java

public class Apfel extends Frucht
{
   public float berechneGewicht()
   {
      return this.berechneProzentualesGewicht (0.85f);
   }
}

// Datei: Obstkorb.java

import java.util.ArrayList;
import java.util.List;

public class Obstkorb
{
   private List<Frucht> fruechte = new ArrayList<Frucht>();

   public Obstkorb()
   {
      fruechte.add (new Apfel());
   }

   public float berechneGewicht()
   {
      float obstkorbGewicht = 0.0f;

      for(Frucht frucht : fruechte)
      {
         obstkorbGewicht += frucht.berechneGewicht();
      }

      return obstkorbGewicht;
   }
}
```

An diesem Beispiel wird deutlich, dass eine durch das "Law of Demeter" eingeführte Wrapper-Methode weit mehr Funktionalität und Logik beinhaltet als eine reine get-Methode.

4.4.8 Bewertung

Das Gesetz von Demeter setzt sich wie die meisten Software-Entwurfsprinzipien mit der Abschwächung von Abhängigkeiten auseinander. Hierbei versucht es, durch das Verbot bestimmter Methodenzugriffe diese Abhängigkeiten abzuschwächen.

Ein unter Berücksichtigung des "Law of Demeter" entwickeltes Softwaresystem profitiert von einer besseren Strukturierung, welche eine Verringerung der Abhängigkeiten bewirkt.

Weitreichende Kopplungen werden vermieden, da sich nur direkt benachbarte Objekte kennen dürfen. Überdies werden die **Übersichtlichkeit**, **Wandelbarkeit**, **Testbarkeit** und **Wartbarkeit** stark erhöht. Passend hierzu trägt das Gesetz von Demeter oftmals auch die Bezeichnung "Law of Goodstyle" [Lie88, p. 323].

Mit der Aussage:

"[…] any object-orientated program written in bad style can be transformed systematically into a structured program obeying the Law of Demeter."

postulierten Lieberherr, Holland und Riel [Lie88, p. 324], dass das "Law of Demeter" allgemein auf jedes schlechte Softwaredesign mit Erfolg angewandt werden kann.

Das "Law of Demeter" ist nicht als überall geltend und als unanfechtbar anzusehen. Es ist eher als eine Art Richtlinie zu verstehen [Hol89, p. 13], an die man sich – sofern möglich – halten sollte.[61] Es gibt jedoch auch Fälle, in denen die Anwendung des "Law of Demeter" nicht sinnvoll ist. Beispielsweise dann, wenn die Anforderungen an die Performance eine höhere Priorität haben als zum Beispiel die Wandelbarkeit oder Wartbarkeit.

Vorteile des "Law of Demeter" sind:

- **Stabilität**

  Das Gesetz von Demeter hilft Softwareentwicklern, eine stabile und gut strukturierte Software zu erstellen.

- **Reduktion der Komplexität**

  Es gibt eine geringere Kopplung von Objekten. Dies führt zu einer besseren Verständlichkeit der Software, einer besseren Wandelbarkeit, Testbarkeit und Wartbarkeit.

[61] Dies gilt nicht nur für das "Law of Demeter", sondern auch für andere Prinzipien.

Nachteile des "Law of Demeter" sind:

- **Große Zahl an Übergabeparametern bei Wrapper-Methoden**

  Wrapper-Methoden zur Kommunikation erfordern nicht selten eine große Anzahl an Übergabeparametern [Hol89].

- **Mehraufwand für Wrapper-Methoden**

  Das Schreiben von Wrapper-Methoden führt zu einem Mehraufwand in der Entwicklung.

- **Performance/Speicherbedarf**

  Die Implementierung einer Software unter Einhaltung des Gesetzes von Demeter kann zu einem geringfügigen Performance-Verlust oder einem etwas erhöhten Speicherbedarf führen [Hol89].[62]

4.5 Dependency Inversion Principle

Das "Dependency Inversion Principle", abgekürzt als DIP, von Robert C. Martin betrachtet, wie man in einer Aufrufhierarchie modularisieren sollte.

4.5.1 Historie

Die erste Fassung des "Dependency Inversion Principle" erschien 1996 im Rahmen mehrerer Kolumnen zum Software Engineering mit C++ [Mar96] in folgender Form:

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

Dieses Prinzip wurde durch das Buch "Agile Software Development: Principles, Patterns, and Practices" von Robert C. Martin im Jahre 2002 [Mar02] populär und ist seitdem in der Praxis weit verbreitet. Mitgeholfen hat bei dieser Entwicklung die starke Verbreitung von "Dependency Injection Containern" wie beispielsweise dem Spring Framework, welche Systeme unterstützen, die zur Kompilierzeit Abstraktionen verwenden und zur Laufzeit Objekte, welche diese Abstraktionen implementieren.

4.5.2 Ziel

Das "Dependency Inversion Principle" betrachtet die Abhängigkeit von Modulen in Aufrufhierarchien. Dieses Prinzip bedeutet auf Deutsch:

- Ein Modul einer höheren Ebene soll nicht von einem Modul einer tieferen Ebene abhängig sein.

[62] Viele Entwurfsprinzipien führen zum Verlust von Performance oder zu einem erhöhten Speicherverbrauch.

- Hingegen soll ein Modul einer tieferen Ebene wie auch die Module der höheren Ebene von einer **Abstraktion** abhängen.

Die Aussage des "Dependency Inversion Principle" ist, dass ein in der Hierarchie höher stehendes Modul nur die Abstraktion eines tiefer stehenden Service-Moduls kennen und benutzen darf, nicht aber dessen Implementierung.

Als Abstraktion einer Klasse der tieferen Ebene kommt eine Schnittstelle oder eine abstrakte Klasse in Frage. Diese Abstraktion soll wiederum von der Klasse der tieferen Ebene implementiert werden. Die höhere Ebene wiederum soll nur die Abstraktion benutzen.

Im Folgenden werden die Überlegungen von Robert C. Martin zu einem hierarchischen System dargelegt. Hierbei verwendet Robert C. Martin eine Abbildung von Grady Booch [Boo95]:

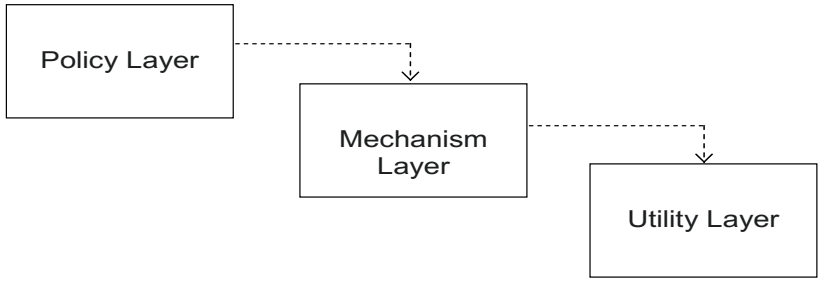


Abbildung 4-4 Module einer höheren Ebene rufen Module einer tieferen Ebene auf

Nach dem Prinzip von "Teile und Herrsche" entsteht komplexe Software durch die Zerlegung in immer feinere Module. Module werden oft in Ebenen angeordnet, welche die schrittweise Verfeinerung charakterisieren. Dabei beschreibt nach Booch

- die oberste Ebene die Politik der Geschäftsprozesse,
- die nächste Ebene die Ebene der sogenannten "Mechanismen" und
- die letzte Ebene die Ebene der Hilfsdienste.

In einem klassischen Entwurf, sei er prozedural oder objektorientiert, würde ein Modul einer höheren Ebene direkt eine Methode eines Moduls der untergeordneten Ebene aufrufen und würde dadurch von ihr abhängig werden.

Dies könnte dazu führen, dass eine Änderung, die auf einer der unteren Ebenen durchgeführt werden muss, sich direkt auf die darüber liegenden Ebenen auswirkt und diese dann ebenfalls geändert werden müssen. Umgekehrt könnten die Module einer höheren Ebene nicht ohne die Kenntnisse der Implementierung der Methoden der tieferen Ebene diese Methoden aufrufen.

Übergeordnete und untergeordnete Module sollen nach dem "Dependency Inversion Principle" von Robert C. Martin nur von der Abstraktion des benutzten Moduls abhängen. Dies bedeutet, dass zwischen einem höher angesiedelten und einem tiefer liegenden Modul eine **Abstraktionsschicht** als Zwischenschicht eingezogen wird.

Die Abhängigkeitsbeziehung zwischen den Modulen eines klassischen Entwurfs wird also durch zwei Abhängigkeiten ersetzt:

1. die Abhängigkeit eines Moduls der höheren Ebene von der Abstraktion des benutzten Moduls der tieferen Ebene (Benutzungsabhängigkeit) und
2. die Abhängigkeit des Moduls der tieferen Ebene von seiner eigenen Abstraktion (Realisierungsabhängigkeit).

Die folgende Abbildung zeigt die eingeschobene **Abstraktionsschicht**:

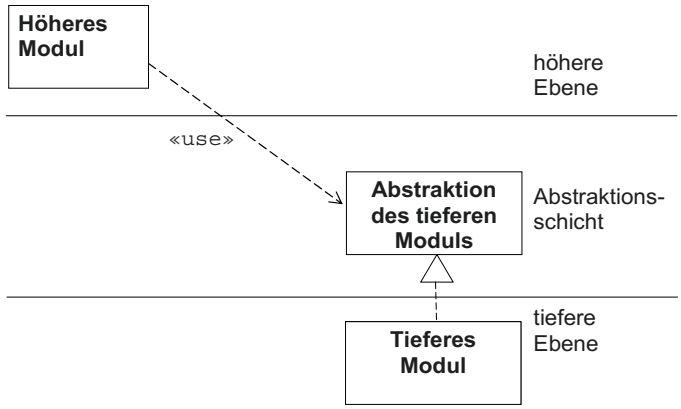


Abbildung 4-5 Einführung einer Abstraktionsschicht

Die Verwendung einer Abstraktion lässt sich auch als Aggregation der entsprechenden Abstraktion darstellen:

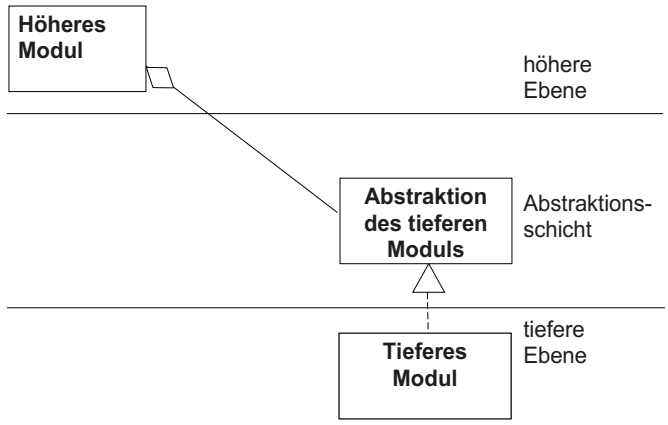


Abbildung 4-6 Formulierung des "Dependency Inversion Principle" mit Aggregation

An die Stelle einer Abstraktion können in Programmen nach dem liskovschen Substitutionsprinzip (siehe Kapitel 5.2) alle Module treten, die diese Abstraktion implementieren und deren Verträge nicht brechen.

Die tieferen Module können nur dann unbemerkt ausgetauscht werden, wenn sie den **Vertrag**[63] **der Abstraktion** erfüllen.

Das "Dependency Inversion Principle" – ebenso wie alle anderen Entwurfsprinzipien, welche mit Abstraktionen arbeiten – erlaubt verschiedene Implementierungen einer Abstraktion.

Das "Dependency Inversion Principle" wird insbesondere für das Testen verwendet, da auf Grund dieses Prinzips **Stub-** oder **Mock-Objekte** als Stellvertreter für echte Objekte verwendet werden können.

4.5.3 Bewertung

Das "Dependency Inversion Principle" ist ein grundlegender Mechanismus zur Abschwächung von Abhängigkeiten in Aufrufhierarchien. Es führt dazu, dass Module einer Ebene nicht von konkreten Implementierungen, sondern nur von der Abstraktion des benutzten Moduls der tieferen Ebene abhängen. Hinter einer Abstraktion können aber verschiedene Implementierungen stecken.

Kommentar des Autors:

Der Name "Dependency Inversion" ist nicht korrekt. Die höhere Ebene ist nach wie vor von der tieferen Ebene abhängig, allerdings nur noch von der Abstraktion der tieferen Ebene und nicht mehr von deren Implementierung. Es wurde aber ferner eine neue Abhängigkeit der Implementierung der tieferen Ebene zu der Spezifikation ihrer Abstraktion eingeführt. Es gibt jedoch keine Umkehrung einer Abhängigkeit, wie es der Name "Dependency Inversion" andeutet.

Vorteile des "Dependency Inversion Principle" sind:

- **Kapselung der Implementierung**

  Module einer höheren Ebene hängen nicht von Implementierungen, sondern von Abstraktionen ab.

- **Erhöhung der Stabilität**

  Bleibt eine Abstraktion bei Änderungen der Implementierung stabil, so bleibt auch die entsprechende Aufrufhierarchie in sich stabil.

[63] siehe Kapitel 5.1

- **Austausch der Implementierung**

  Bleibt die Abstraktion gleich, so können Implementierungen ausgetauscht werden. Der Austausch der Implementierungen bringt insbesondere beim Testen mit Stubs und Mock-Objekten einen Nutzen.

4.6 Interface Segregation Principle

Das "Interface Segregation Principle", abgekürzt als ISP, befasst sich mit der **Kohäsion von Schnittstellen.** Eine Schnittstelle sollte nur solche Methodenköpfe enthalten, welche zusammengehörig sind. Das "Interface Segregation Principle" verlangt, dass große Schnittstellen in kleinere Schnittstellen aufgeteilt werden, wobei die Schnittstellen sich lediglich auf die Anforderungen ihrer jeweiligen Clients beschränken. Die Clients erhalten also nur Schnittstellen, die sie jeweils tatsächlich nutzen. Damit hängen sie nicht von nicht benötigten Schnittstellen ab. Dieses Prinzip stammt von Robert C. Martin.

In [Mar02, p. 137] lautet die Formulierung:

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

Clients sollten nicht gezwungen werden, von Methoden abhängig zu sein, die sie gar nicht brauchen.

Sogenannte "fette" oder "polluted" Schnittstellen, die zu breit sind und vom Client nicht benötigte Methoden enthalten, sind zu vermeiden.

Die Anwendung dieses Prinzips erfordert die Analyse jeder einzelnen Methode einer Schnittstelle. Es ist hierbei zu prüfen, ob eine betrachtete Methode wirklich von jedem Client benötigt wird. Ist dies nicht der Fall, so muss überlegt werden, welche Methoden in welcher Kombination von einem Client oder einer Gruppe von Clients verwendet werden sollen. Jedem Client oder jeder Gruppe von Clients wird daraufhin eine individuell angepasste **Rollen-Schnittstelle**[64] zur Verfügung gestellt.

4.6.1 Historie

Das "Interface Segregation Principle" wurde von Robert C. Martin während eines Auftrags bei der Firma XEROX, die u. a. Drucker und Multifunktionsgeräte herstellt, entwickelt. Veröffentlicht wurde es 1996 in [Ma296].

4.6.2 Ziel

Wandelbare Anwendungen müssen in verschiedene Features aufgeteilt werden mit so wenig Überlapp wie möglich. Das Ziel ist die Minimierung der Wechselwirkung der verschiedenen Features, um eine lose Kopplung und eine starke Kohäsion zu erreichen.

[64] Eine Schnittstelle, die genau an den Bedürfnissen eines bestimmten Clients ausgerichtet ist, wird auch als Rollen-Schnittstelle bezeichnet (siehe Martin Fowler in [Fow06]).

Um dieses Ziel zu erreichen, sollte eine Schnittstelle nur eng zusammenhängende Methodenköpfe enthalten. Je weniger Methoden in einer von einem Client benötigten Schnittstelle enthalten sind, desto schwächer ist die Kopplung zwischen diesen beiden Komponenten sowie potentielle Wechselwirkungen mit anderen Clients.

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

4.6.3 Konzeptionelles Beispiel für das "Interface Segregation Principle"[65]

Das Beispiel in Abbildung 4-7 ist an die Problemstellung angelehnt, die Ursache dafür war, dass Robert C. Martin das "Interface Segregation Principle" während eines Auftrags bei XEROX entwickelte.

Im folgenden Beispiel soll die Klasse `Multifunktionsdrucker` Methoden für den Zugriff auf einen Multifunktionsdrucker in einer Schnittstelle bereitstellen. Dieser Drucker sollte die Funktionen Drucken, Scannen und Kopieren beinhalten. Ein Client greift über die Schnittstelle `IMultifunktionsdrucker` auf die implementierten Methoden zu. Dies zeigt die folgende Abbildung:

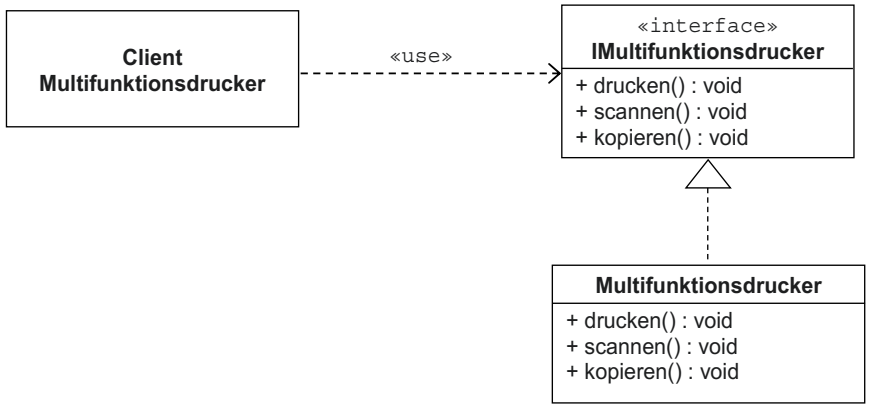


Abbildung 4-7 Beispiel mit einem einzigen Client und einer einzigen Schnittstelle

Hier kann man dem "Client Multifunktionsdrucker" problemlos eine Schnittstelle mit drei Methodenköpfen für das Drucken, Scannen und Kopieren anbieten, da er diese Methoden auch tatsächlich nutzt.

Im Folgenden soll das System um zwei weitere Clients erweitert werden, einen "Client Drucker" und einen "Client Scanner". Der "Client Drucker" soll allerdings nur die Druckfunktion des Multifunktionsdruckers verwenden und der "Client Scanner" nur die Scan-Funktion. Es wäre zwar möglich, dass der "Client Drucker" und "Client Scanner" ebenfalls die Schnittstelle `IMultifunktionsdrucker` verwenden. Eine solche

[65] Ein Programmierbeispiel zum ISP ist auf dem begleitenden Webauftritt zu finden.

Lösung würde jedoch gegen das "Interface Segregation Principle" verstoßen, da der "Client Drucker" die Methoden `kopieren()` und `scannen()` überhaupt nicht verwenden würde, genauso wenig wie der "Client Scanner" die Methoden `drucken()` und `kopieren()`. Nach dem "Interface Segregation Principle" soll jedem Client eine eigene **Rollen-Schnittstelle** zur Verfügung gestellt werden, welche lediglich die von ihm benötigten Methodenschnittstellen anbietet.

Der Zugriff von "Client Multifunktionsdrucker", "Client Drucker" und "Client Scanner" auf die jeweils tatsächlich von ihnen benötigten Schnittstelle(n) ist in folgender Abbildung zu sehen:

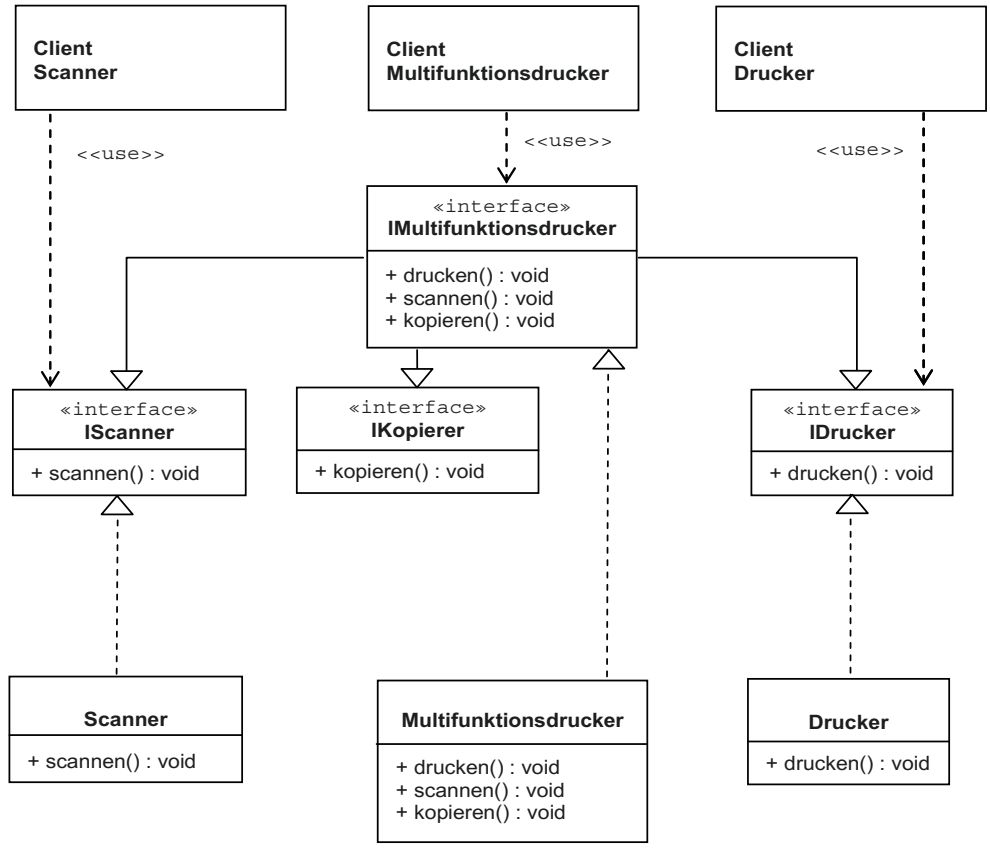


Abbildung 4-8 Beispiel für die Einhaltung des "Interface Segregation Principle"

In dieser Abbildung sind zwei neue Implementierungsklassen zu sehen, die eigentlich nicht notwendig wären. Sie zeigen jedoch, dass man aus der Einführung von Rollen-Schnittstellen weitere Vorteile ziehen kann: Nun sind auch schmälere Implementierungsklassen möglich wie beispielsweise die zwei neu eingeführten Klassen `Scanner` und `Drucker`. Würden diese Klassen das Interface `IMultifunktionsdrucker` (vgl. Abbildung 4-7) implementieren, müssten sie leere Methoden enthalten oder Methoden, die nur Exceptions werfen, um formal die Bedingung zu erfüllen, dass alle Methoden des Interface implementiert sind.

4.6.4 Bewertung

Durch die Anwendung des "Interface Segregation Principle" verwenden Clients stets nur solche Rollen-Schnittstellen, deren Methoden sie auch tatsächlich nutzen.

Das "Interface Segregation Principle" führt zu folgenden **Vorteilen**, welche zu einem einfacheren Code mit weniger Abhängigkeiten führen:

- **Leichte Identifikation der benötigten Methoden**

  Ein nutzender Client wird genau gegen diejenige Rollen-Schnittstelle entwickelt, deren Methoden der Client benötigt und auch tatsächlich aufruft. Ein mühsames Heraussuchen der relevanten Methoden aus einer zu breiten Schnittstelle wird vermieden und es werden nicht fälschlicherweise von den Entwicklern die falschen Methoden bearbeitet.

- **Gliederung von Schnittstellen nach Aufgabenbereichen**

  Schmälere Schnittstellen können nach Aufgabenbereichen gegliedert sein. Solche Schnittstellen können treffende Namen erhalten.

- **Abschwächung von Abhängigkeiten**

  Die Wechselwirkungen werden reduziert. Dies führt zu "Loose Coupling and Strong Cohesion". Wenn mehrere Clients eine große, gemeinsame Schnittstelle hätten, könnte die Änderung eines Methodenkopfes für einen Client einen anderen Client beschädigen.

- **Schmälere Implementierungsklassen**

  Wenn man die Schnittstellen der Clients aufteilt, kann man auch "schmälere" Implementierungsklassen bauen.

- **Vermeiden leerer Methoden im Implementierer einer Schnittstelle**

  Ohne Rollen-Schnittstellen ist jede implementierende Klasse gezwungen, das "fette" Interface komplett zu implementieren. Ggf. müssen dabei leere Methoden bzw. Exceptions eingeführt werden, welche einem Client keinen Nutzen bringen, aber formal die Bedingung erfüllen, dass alle Methoden des Interface implementiert sind.

Ein **Nachteil** ist aber

- **die Erhöhung der Anzahl der Schnittstellen**

  Die Einführung von individuellen Rollen-Schnittstellen für die Clients erhöht die Anzahl der Schnittstellen.

4.7 Single Responsibility Principle

Das "Single Responsibility Principle"[66] nach Robert C. Martin [marsrp] ist das erste der bekannten SOLID-Prinzipien (siehe auch Kapitel 10.3).

[66] "Responsibility" bedeutet auf Deutsch "Verantwortlichkeit".

Nach Robert C. Martin sollten Dinge, die sich aus verschiedenen Gründen **ändern**, getrennt werden: Für die Veränderung eines Software-Moduls sollte es nur einen einzigen Grund geben.

4.7.1 Historie

Dieses Prinzip wurde nicht wie die anderen SOLID-Prinzipien von Robert C. Martin im Rahmen von "The C++ Record" [marsrp] im Jahre 1996 veröffentlicht, sondern wurde getrennt davon erst im Jahre 2002 publiziert [Mar02].

Robert C. Martin leitete das "Single Responsibility Principle" nach eigenen Angaben aus der **"Strong Cohesion"** (siehe Kapitel 4.1) ab, welche auf die Arbeiten von Larry Constantine in den sechziger Jahren zu "Loose Coupling and Strong Cohesion" zurückgeht. Hierbei führte Martin das "Single Responsibility Principle" auf die Arbeiten von Tom DeMarco [DeM79] und Meilir Page-Jones [Pag88] zurück. Sowohl Tom DeMarco als auch Meilir Page-Jones analysierten "Loose Coupling" und "Strong Cohesion". Martin bezieht sich jedoch in [Mar12] explizit auf deren Arbeit zur Kohäsion.

Das Qualitätsmerkmal "Strong Cohesion" ist begrifflich generell schwer zu charakterisieren. Daher gab es verschiedene Ansätze, den Begriff zu präzisieren bzw. durch ausdrucksstärkere Wörter zu ersetzen. So ist auch der Ansatz von Robert C. Martin zu sehen.

Im Zusammenhang mit dem "Single Responsibility Principle" verschob Martin den ursprünglichen Begriff der "Kohäsion" etwas von dem inneren Zusammenhang der Elemente eines Moduls hin zu den Kräften, die eine Klasse beziehungsweise ein Modul zu einer Änderung bringen.

Martin stellte für dieses Prinzip im Jahre 2002 die folgende Regel auf:

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

Mit dem **einzigen Änderungsgrund** einer Klasse liegt durch das "Single Responsibility Principle" eine neue Interpretation der **starken Kohäsion** vor.

War das "Single Responsibility Principle" in seiner Ursprungsform nur für Klassen gedacht, bezog Robert C. Martin im Mai 2014 dieses Prinzip nicht mehr nur speziell auf Klassen, sondern im Blog-Eintrag der Firma "8th Light" allgemein auf Software-Module [Mar14]:

"The Single Responsibility Principle (SRP) states that each software module should have one and only one reason to change."

Martin schrieb dort ferner:

"Gather together the things that change for the same reasons. Separate those things that change for different reasons."

Dinge, die sich aus verschiedenen Gründen ändern, müssen getrennt werden.

"A reason to change" charakterisiert eine sogenannte **Verantwortlichkeit** (engl. **responsibility**) einer Klasse [Mar12]. Gibt es mehrere Gründe, warum ein Modul geändert werden kann, so bedeutet das, dass sich mehrere Verantwortlichkeiten in diesem Modul befinden.

Nach dem Blog von Robert C. Martin [Mar14] steckt hinter jeder Verantwortlichkeit im Sinne eines Geschäftsprozesses stets das Interesse einer einzelnen Person oder einer Gruppe von Personen. So schrieb Martin:

". . . remember that the reasons for change are people. It is people who request changes."

4.7.2 Ziel

Ein Modul hat nach dem "Single Responsibility Principle" nur eine **einzige Verantwortlichkeit** und ist dann stark zusammenhängend, wenn es nur eine **einzige Kraft** gibt, die auf das Modul wirkt und damit zur Änderung dieses Moduls führen kann.

Mehrere Verantwortlichkeiten innerhalb eines Moduls führen zu einer Erhöhung des Fehlerrisikos, da bei einer Änderung einer Verantwortlichkeit leicht eine andere Verantwortlichkeit beschädigt werden könnte.

Mehrere Verantwortlichkeiten innerhalb eines Moduls dürfen nicht vorhanden sein.

Hat in der Praxis eine Klasse zu viele Beziehungen zu anderen Klassen, so sollte überprüft werden, ob die betrachtete Klasse nicht zu viele Verantwortlichkeiten enthält.

4.7.3 Vergleich des "Single Responsibility Principle" mit "Separation of Concerns"

Nach dem "Single Responsibility Principle" sollte jedes Modul nur eine einzige Verantwortlichkeit haben. Alle Teile eines Moduls sollen zur Erfüllung ein und derselben Aufgabe beitragen. Demnach sollte man Module aufgrund ihrer Verantwortlichkeiten

unterscheiden, um Änderungen der Verantwortlichkeiten leichter in verschiedenen Modulen umsetzen zu können.

Das ist im Sinne des Prinzips "Separation of Concerns". Dieses Prinzip ist allgemeingültig und befasst sich nicht speziell mit Programmen.

Diese beiden Prinzipien sind jedoch eng miteinander verwandt.

Man kann **"Separation of Concerns"** als einen **elementaren Prozess der Zerlegung** komplexer Systeme in einzelne Belange betrachten.

Das **"Single Responsibility Principle"** kann als ein **Designprinzip** für die **Modularisierung** von Programmen angesehen werden.

Das "Single Responsibility Principle" gilt für Klassen bzw. Module, "Separation of Concerns" hingegen ist ein **elementares Ordnungsprinzip**.

4.7.4 Bewertung

Die folgenden **Vorteile** des "Single Responsibility Principle" werden gesehen:

- **Fehlervermeidung**

  Unbeabsichtigte Beschädigungen anderer Module werden vermieden.

- **Begriff der Kohäsion**

  Der Wert der Arbeit von Robert C. Martin zum "Single Responsibility Principle" ist auch darin zu sehen, dass Martin die Kohäsion eines Moduls über den Begriff einer "einzigen Verantwortlichkeit" für die Änderungswahrscheinlichkeit eines Moduls in verständlicher Form formulierte. Damit wurde in neuer Form gedeutet, was man unter einer starken Kohäsion versteht.

- **Weltweite Verbreitung**

  Robert C. Martin verstand es, seine Idee in der Informatik weltweit zu vermitteln.

Die folgenden **Nachteile** werden gesehen:

- **Fassbarkeit der Änderungswahrscheinlichkeit**

  Die Änderungswahrscheinlichkeit ist oft sehr schwer zu beurteilen. Nur wenn die Änderungswahrscheinlichkeiten klar umrissen sind, kann das Prinzip überhaupt angewandt werden.

- **Einseitigkeit**

  Andere Aspekte einer starken Köhäsion werden nicht gesehen.

- **Zu starke Modularisierung**

  Eine zu konsequente Anwendung des "Single Responsibility Principle" könnte zu einer zu starken Modularisierung führen.

- **Schwierigkeit der Umsetzung**

  Inhaltlich kann das "Single Responsibility Principle" als eines der einfachsten Entwurfsprinzipien verstanden werden, bezüglich seiner Umsetzung gehört es jedoch zu den schwierigsten [Mar12, p. 98]. Es stellt sich dabei nämlich immer die Frage, in welcher Granularität man zerlegen soll.

Ist man sich ganz sicher, dass die Verantwortlichkeiten niemals getrennt voneinander geändert werden müssen, dann ist es nicht nötig, diese aufzuteilen. Eine unnötige Trennung der Verantwortlichkeiten könnte im Gegenteil zum Auftreten einer überflüssigen Komplexität[67] führen [Mar12, p. 97].

4.8 Die Konzepte "Dependency Lookup" und "Dependency Injection"

Beide Konzepte sind keine Entwurfsprinzipien, sind aber von großer Bedeutung bei der Abschwächung von Abhängigkeiten, welche bei der direkten Erzeugung von Objekten durch ein anderes Objekt entstehen.

4.8.1 Ziel

Wenn ein Objekt einer nutzenden Klasse das von ihm benötigte Objekt einer anderen Klasse selbst erzeugt, dann entsteht eine weitere Abhängigkeit, nämlich eine sogenannte create-Abhängigkeit, da die Klasse des erzeugenden Objekts Informationen über die Klasse des erzeugten Objekts statisch im Quellcode halten muss. Dadurch wird jedoch die erzeugende Klasse von der erzeugten Klasse abhängig.

Die folgende Abbildung zeigt eine solche create-Abhängigkeit:

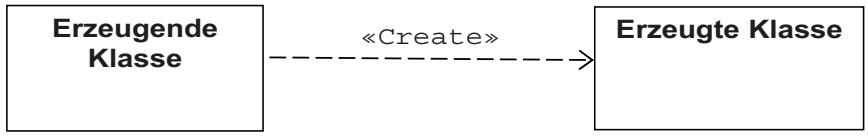


Abbildung 4-9 Neue Abhängigkeit durch die Erzeugung eines Objekts

Solche create-Abhängigkeiten sollen explizit vermieden werden.

[67] Überflüssige Komplexität ("Needless Complexity") ist eine der "smells", die Robert C. Martin einführte, um schlechte Eigenschaften von Software zu beschreiben [Mar12, p. 88].

4.8.2 Lösungsalternativen

Die Konzepte "Dependency Lookup" und "Dependency Injection" verfolgen beide das Ziel, direkte create-Abhängigkeiten in einem Programm zu vermeiden und die Erzeugung von Objekten an eine außenstehende Instanz zu delegieren.

Für "Dependency Lookup" und "Dependency Injection" gilt:

Damit ein Objekt ein benötigtes Objekt überhaupt verwenden kann, ist es erforderlich, dass eine **Abstraktion der Klasse des benötigten Objekts** im Programm des benutzenden Objekts enthalten ist. Das benutzende Objekt kann die Methoden des benutzten Objekts nur dann aufrufen, wenn es dessen Abstraktion kennt.

In beiden Fällen hat das benutzende Objekt keine Kontrolle darüber, wann ein Objekt und welches konkrete Objekt erzeugt wird. Beim Injektor der "Dependency Injection" erfolgt das Erzeugen ohne irgendeinen Einfluss durch das benutzende Objekt. Beim "Dependency Lookup" hingegen erfolgt das Suchen dann, wenn das benutzende Objekt den Suchauftrag erteilt. Damit kann natürlich auch verbunden sein, dass erst in diesem Moment das benötigte Objekt erzeugt wird. Aber das entscheidet die mit der Suche beauftragte Instanz und nicht das benutzende Objekt.

Im Folgenden werden die Konzepte "Dependency Lookup" und "Dependency Injection" kurz beschrieben:

- **Dependency Lookup**

  Bei einem "Dependency Lookup" sucht ein Objekt, das ein anderes Objekt braucht, nach diesem Objekt etwa in einem Register (engl. registry), um die Verknüpfung mit diesem Objekt herzustellen. Durch den Suchvorgang behält das suchende Objekt die Kontrolle, wann gesucht wird. Zur Suche braucht es beispielweise nur den Namen des gesuchten Objekts zu kennen und ist von diesem Objekt weitgehend entkoppelt. Aber natürlich hängt es dann vom Register ab.

- **Dependency Injection**[68]

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

Der Injektor bestimmt, wann ein Objekt erzeugt wird. Der Injektor ist von allen beteiligten Objekten abhängig. Die Objekte sind untereinander selbst nicht abhängig.

[68] Der Begriff "Dependency Injection" wurde zum ersten Mal von M. Fowler in [Fow04] benutzt, um dieses Konzept von der "Inversion of Control" abzugrenzen. "Dependency Injection" gilt in der Literatur oft als Entwurfsmuster, "Dependency Lookup" jedoch nicht.

Bei "Dependency Injection" ist ein benutzendes Objekt vom Injektor unabhängig. Es kennt ihn gar nicht. Es ist aber von der **Abstraktion** der **Klasse des benutzten Objekts** abhängig. Sonst könnte es dieses Objekt gar nicht benutzen.

4.8.3 Das Konzept "Dependency Lookup"

Die ursprüngliche create-Abhängigkeit zwischen der Klasse des nutzenden Objekts und der Klasse des benötigten Objekts wird bei "Dependency Lookup" durch eine schwächere Abhängigkeitsbeziehung zwischen der Klasse des nutzenden Objekts und der **Abstraktion der Klasse des benötigten Objekts** ersetzt.

Die Abstraktion wird in der Regel – wie auch im Falle der "Dependency Injection" – mit Hilfe einer Schnittstelle oder einer abstrakten Basisklasse definiert. [69]

Um das benötigte Objekt zu finden und die Verknüpfung mit diesem herzustellen, benötigt das suchende Objekt lediglich einen **Schlüssel**. Als Schlüssel kann beispielsweise der Name des Objekts oder die Abstraktion der Klasse des gesuchten Objekts verwendet werden. Allerdings muss das suchende Objekt nicht mehr die konkrete Klasse des gesuchten Objekts kennen.

"Dependency Lookup" bedeutet, dass ein Objekt seine **Verknüpfung** mit einem anderen Objekt **zur Laufzeit** erstellt, indem es nach diesem anderen Objekt zur Laufzeit mit einem **Schlüssel** in einer zentralen Instanz wie einer Registratur sucht. Hierzu muss es die Registratur kennen.

Das suchende Objekt ist aber weiterhin von der Abstraktion der Klasse des gesuchten Objektes abhängig, die es einhalten muss, um das gesuchte Objekt überhaupt verwenden zu können.

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

4.8.3.1 Realisierung über eine Registry

Zur Umsetzung des Konzepts "Dependency Lookup" gibt es beispielsweise die Möglichkeit, als zentrale Instanz eine Registratur anzulegen. Hierauf wird im Folgen-

[69] Andere Typen wie z. B. Delegates in C# sind auch möglich.

den eingegangen. Eine Registratur ermöglicht das Registrieren der Objekte unter einem Schlüssel. Die erzeugten Objekte werden registriert, also in die Registratur eingetragen.

Die Beziehung zwischen einem suchenden und einem gesuchten Objekt wird aber nicht statisch verlinkt, sondern durch den Schlüssel wird diese Beziehung **dynamisch zur Laufzeit** hergestellt, was zu einer gewissen Entkopplung führt und Abhängigkeiten abschwächt.

Wird ein Objekt für eine Verknüpfung benötigt, wird in der Registratur zum Beispiel über den Objektnamen nach einem entsprechenden Objekt gesucht. Die Registratur liefert eine Referenz auf das gesuchte Objekt zurück, womit dann eine Verknüpfung hergestellt werden kann.

4.8.3.2 Programmierbeispiel "Dependency Lookup" mit Hilfe einer Registry

Im Folgenden wird die Idee des "Dependency Lookup" anhand eines Beispiels verdeutlicht. Als Beispiel wird ein Plotter betrachtet, dessen Aufgabe es ist, eine Reihe von Daten, die von einer Datenquelle – also einem Objekt – erzeugt werden, aufzubereiten und grafisch darzustellen.

Die Klasse `Plotter` aggregiert die Schnittstelle `IDatenquelle`, welche auf verschiedene Arten realisiert werden kann. In diesem Beispiel wird die Schnittstelle `IDatenquelle` von der Klasse `Datenquelle` realisiert. Alle vorhandenen Realisierungen einer Datenquelle werden in einer Registratur verwaltet. Die Klasse `Plotter` verwendet die Registratur, um eine konkrete Datenquelle auszuwählen, damit die darin enthaltene Datenreihe dann dargestellt werden kann. Hier die Schnittstelle `IDatenquelle`:

```
// Datei: IDatenquelle.java

import java.awt.Point;
import java.util.List;

public interface IDatenquelle
{
   List<Point> holeDatenreihe();
}
```

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

```
// Datei: Datenquelle.java

import java.awt.Point;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Datenquelle implements IDatenquelle
{
```

```
   @Override
   public List<Point> holeDatenreihe()
   {
      List<Point> datenreihe = new ArrayList<>();
      Random zufallszahlenGenerator = new Random();

      // Erzeuge mindestens 2, maximal 31 Punkte
      int anzahlPunkte = zufallszahlenGenerator.nextInt (30) + 2;

      // Startpunkt liegt zwischen -29 und 0
      int startPunkt = -zufallszahlenGenerator.nextInt (30);

      for (int x = startPunkt; x < anzahlPunkte; x++)
      {
         // Zufälliger Y-Wert im Bereich [-20,19]
         datenreihe.add (
               new Point (
                     x,
                     zufallszahlenGenerator.nextInt (40) - 20
               )
         );
      }
      return datenreihe;
   }
}
```

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

```
// Datei: Registratur.java

import java.util.HashMap;

/***
 * Verwaltet beliebig viele Objekte. Jedes Objekt wird durch einen
 * eindeutigen String-Schluessel registriert und kann ueber diesen
 * angefordert werden.
 */

public class Registratur
{
   private static Registratur registraturInstanz = null;
   private HashMap<String, Object> objekte = new HashMap<>();

   private Registratur(){}

   public static Registratur holeInstanz()
   {
      if (registraturInstanz == null)
      {
         registraturInstanz = new Registratur();
      }
      return registraturInstanz;
   }
```

```
   public Object frageObjektAb (String bezeichner)
   {
      return objekte.get (bezeichner);
   }

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

In der folgenden Klasse `Plotter` wird die Ausgabe einer Datenreihe auf der Konsole generiert. Hier soll im Wesentlichen gezeigt werden, wie ein Objekt der Klasse `Plotter` mit Hilfe einer Registratur eine Datenquelle finden und dann die von dieser Datenquelle erzeugte Datenreihe ausgeben kann. Die Ausgabe ist in der Methode `plot()` der Klasse `Plotter` zu sehen:

```
// Datei: Plotter.java

import java.awt.Point;
import java.util.List;

public class Plotter
{
   private List<Point> datenreihe = null;

   public void plot()
   {
      Registratur registratur = Registratur.holeInstanz();

      IDatenquelle datenquelle;
      datenquelle = (IDatenquelle)
            registratur.frageObjektAb ("Datenquelle");

      if (datenquelle == null)
      {
         return;
      }

      datenreihe = datenquelle.holeDatenreihe();

      for (Point punkt : datenreihe)
      {
         String formatierteAusgabe = String.format
         (
               "(%d,%d)",
               (int)punkt.getX(),
               (int)punkt.getY()
         );
         System.out.println (formatierteAusgabe);
      }
   }
}
```

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

Bezeichnung auf die Datenquelle zugreifen und dieses Objekt benutzen. Hier der Quellcode der Klasse `TestPlotter`:

```
// Datei: TestPlotter.java

public class TestPlotter
{
   static public void main (String[] args)
   {
      Registratur registry = Registratur.holeInstanz();

      registry.registriereObjekt
      (
            "Datenquelle",
            new Datenquelle()
      );

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

Ein Ausschnitt der Programmausgabe ist beispielsweise:

```
(-8,-10)
(-7,16)
(-6,0)
(-5,13)
(-4,-8)
```

Auf dem begleitenden Webauftritt ist ein weiteres Beispiel, in welchem ein Plotter die Datenreihe in grafischer Form ausgibt, enthalten. Die folgende Abbildung zeigt eine von diesem Plotter gezeichnete Datenreihe:

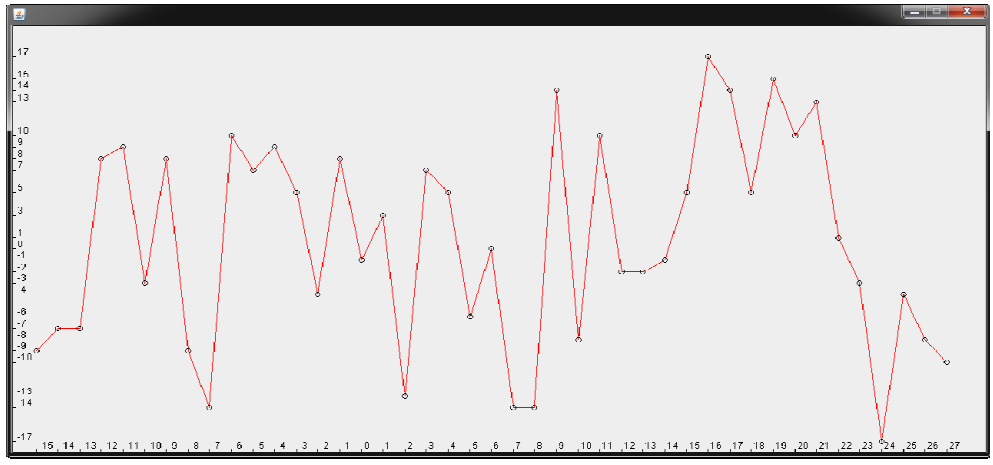


Abbildung 4-10 Geplottete Datenreihe

4.8.4 Das Konzept "Dependency Injection"

Das Konzept "Dependency Injection" wurde von Martin Fowler entwickelt.

Auch wenn das Konzept "Dependency Injection" zur Erzeugung von Objekten in der Literatur oftmals mit dem Begriff "Inversion of Control" verbunden wird, folgen wir diesem Gedanken nicht. Wir folgen hier Martin Fowler [Fow04]:

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

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

Objekte müssen oft Kenntnisse ihrer Umgebung mitbringen und sind dadurch fest mit anderen Objekten verdrahtet. Ein Vorgehen, diese festen Verdrahtungen zu lösen, ist "Dependency Injection". Die Abhängigkeiten zwischen Komponenten sollen dadurch abgeschwächt werden, indem eine neue Komponente – oftmals **Injektor** genannt – zur Verwaltung der vorhandenen Komponenten eingeführt wird.

Eine **Injektion** ist das Übergeben einer Referenz auf ein anderes Objekt an ein Objekt, welches dieses andere Objekt nutzt.

Der Injektor hat das Wissen über die benötigten Verbindungen und steuert dann zur Laufzeit, welche Objekte für die einzelnen Abhängigkeiten erzeugt werden. Dabei kennt ein Objekt den Injektor nicht. Ein benutzendes Objekt muss aber grundsätzlich die Abstraktion des benötigten Objekts kennen. Als Abstraktion kann entweder eine Schnittstelle oder eine abstrakte Klasse verwendet werden, wobei in der Praxis Schnittstellen bevorzugt werden.

Bei "Dependency Injection" kennt weder das benutzende Objekt, das ein anderes Objekt injiziert bekommt, noch das injizierte Objekt den Injektor.

Beispielsweise soll die Verbindung eines Objekts `:A` zu einem Objekt `:B`, das von `:A` benutzt werden soll, erstellt werden. Es ist dabei einem Objekt nicht erlaubt, benötigte Objekte selbst zu erstellen oder nach ihnen zu suchen. Die neue Komponente des **Injektors** übernimmt hierbei komplett eigenständig die Kontrolle von außen.

Mit "Dependency Injection" können einem Objekt beim Hochfahren des Systems alle benötigten anderen Objekte, die es braucht, zur Verfügung gestellt werden. Auf diese Weise kann ein Objekt die von ihm benötigten anderen Objekte zur Laufzeit benutzen, ohne zur Kompilierzeit deren konkrete Klassen zu kennen.

"Dependency Injection" löst keineswegs Abhängigkeiten vollkommen auf, ist aber ein Hilfsmittel für die Abschwächung von Abhängigkeiten.

Es besteht immer noch eine Abhängigkeit des nutzenden Objekts von der **Abstraktion der Klasse des benötigten Objektes**, denn sonst könnte das nutzende Objekt nicht auf das benötigte Objekt zugreifen.

Man unterscheidet zwischen drei **Varianten der "Dependency Injection"**:

- **Constructor Injection**

  Die abhängige Klasse definiert einen Konstruktor zum direkten Setzen der Abhängigkeiten durch den Injektor.

- **Setter Injection**

  Die abhängige Klasse stellt Methoden zur Verfügung, mit denen man Abhängigkeiten festlegen kann.

- **Interface Injection**

  Die abhängige Klasse muss eine Schnittstelle implementieren, um zur Laufzeit Abhängigkeiten, die sie benötigt, vom Injektor zur Verfügung gestellt zu bekommen.

Für diese drei Varianten werden in den folgenden Unterkapiteln Beispiele gegeben, die zeigen, wie der Ansatz der "Dependency Injection" eingesetzt werden kann, um Abhängigkeiten beim Erzeugen von Objekten innerhalb eines Programms abzuschwächen. Im folgenden Unterkapitel wird die erste Variante "Constructor Injection" anhand eines Beispiels erläutert. Anschließend wird dieses Beispiel in den darauffolgenden Unterkapiteln modifiziert, um auch die beiden anderen Varianten "Interface Injection" und "Setter Injection" jeweils getrennt für sich zu erklären.

4.8.4.1 Programmierbeispiel für "Constructor Injection"

In diesem Beispiel wird die Klasse `AusgabeSteuerung` eingeführt. Sie soll die Ausgabe eines Strings auf verschiedene Weisen durchführen können. Wie eine konkrete Ausgabe funktioniert, soll der Klasse `AusgabeSteuerung` zur Kompilierzeit nicht bekannt sein, sondern soll erst zur Laufzeit von einem Injektor durch Übergabe des passenden konkreten Objektes der Klasse `AusgabeDienst` festgelegt werden.

Im Folgenden die Klasse `AusgabeSteuerung`:

```
// Datei: AusgabeSteuerung.java

public class AusgabeSteuerung
{
   private IAusgabeDienst ausgabeDienst;

   // Konstruktor für Constructor Injection
   public AusgabeSteuerung (IAusgabeDienst konkreterDienst)
   {
      ausgabeDienst = konkreterDienst;
   }
```

```
   public void gebeTextAus (String text)
   {
      ausgabeDienst.gebeStringAus (text);
   }
}
```

Um in der Methode `gebeTextAus()` eine Ausgabe des übergebenen Textes durchführen zu können, besitzt die Klasse `AusgabeSteuerung` im Konstruktor eine Referenz auf eine Schnittstelle `IAusgabeDienst`, welche zur Laufzeit auf eine konkrete AusgabeDienst-Klasse zeigen muss. Die Schnittstelle `IAusgabeDienst` gibt den Methodenkopf für die Methode `gebeStringAus()` vor, welche von allen Klassen, welche die Schnittstelle implementieren, zur Verfügung gestellt werden muss. Hier die Schnittstelle `IAusgabeDienst`:

```
// Datei: IAusgabeDienst.java

public interface IAusgabeDienst
{
   void gebeStringAus (String ausgabe);
}
```

Diese Schnittstelle dient aber nur als Platzhalter, damit die Klasse `AusgabeSteuerung` kompiliert werden kann. Damit das Programm lauffähig wird, werden konkrete AusgabeDienst-Klassen benötigt, die in der Lage sind, eine Ausgabe durchzuführen. Damit diese zur Laufzeit an die Stelle der Schnittstelle treten können, müssen sie die Schnittstelle `IAusgabeDienst` implementieren. Im Folgenden zwei Beispiele für konkrete AusgabeDienste:

```
// Datei: KonsolenAusgabeDienst.java

public class KonsolenAusgabeDienst implements IAusgabeDienst
{
   @Override
   public void gebeStringAus (String ausgabe)
   {
      System.out.println (ausgabe);
   }
}
```

```
// Datei: DateiAusgabeDienst.java

import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class DateiAusgabeDienst implements IAusgabeDienst
{
   @Override
   public void gebeStringAus (String ausgabe)
   {
      try(PrintWriter out = new PrintWriter ("ausgabe.txt"))
      {
         out.println (ausgabe);
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      }
   }
}
```

Die gezeigten Klassen `KonsolenAusgabeDienst` und `DateiAusgabeDienst` unterscheiden sich in der Implementierung ihrer Methode `gebeStringAus()`. Ein Objekt der Klasse `KonsolenAusgabeDienst` gibt den der Methode `gebeStringAus()` übergebenen String an der Konsole aus, während ein Objekt der Klasse `DateiAusgabeDienst` den String in einer neuen Textdatei ablegt.

Die Zuordnung der Abhängigkeiten zur Laufzeit übernimmt in diesem Beispiel die Klasse `Injektor`. Sie verwendet den Konstruktor, welchen die Klasse `AusgabeSteuerung` zur Verfügung stellt, um ein Objekt einer konkreten AusgabeDienst-Klasse zur Laufzeit zu injizieren. Im Folgenden die Klasse `Injektor`:

```
// Datei: Injektor.java

public class Injektor
{
   public static void main (String[] args)
   {
      IAusgabeDienst dienst = new KonsolenAusgabeDienst();

      // Constructor Injection
      AusgabeSteuerung steuerung = new AusgabeSteuerung (dienst);
      steuerung.gebeTextAus ("Constructor Injection");
   }
}
```

Die Klasse `Injektor` weist dem Objekt, auf das die Referenz `steuerung` der Klasse `AusgabeSteuerung` zeigt, eine Referenz auf ein Objekt der konkreten Klasse `KonsolenAusgabeDienst` zu.[70] Damit wird das Objekt, auf das die Referenz `steuerung` zeigt, lauffähig und die Ausgabe erfolgt über die Konsole.

Die Ausgabe des Programms ist:

```
Constructor Injection
```

Durch die Injektion des benötigten Objekts von außen durch den Injektor verschiebt sich die create-Abhängigkeit (vgl. Kapitel 4.8.1) hin zum Injektor. Dieser hat nun die Verantwortung für die Erzeugung des benötigten Objekts und dadurch muss die nutzende Klasse die Klasse des benötigten Objekts nicht mehr kennen. Stattdessen kennt die nutzende Klasse nur eine **Abstraktion der Klasse des benötigten Objekts**. Diese Abstraktion wird meist als Schnittstelle implementiert wie auch in diesem Beispiel.

Die ursprüngliche create-Abhängigkeit zwischen der nutzenden Klasse und dem Objekt der benötigten Klasse wird durch eine schwächere Abhängigkeitsbeziehung zwischen der nutzenden Klasse und der Schnittstelle als Abstraktion der Klasse des benötigten Objekts ersetzt. Somit wird die ursprüngliche Abhängigkeit zwar abgeschwächt, löst sich jedoch nicht vollständig auf.

[70] In diesem Beispiel ist die Wahl des zu erzeugenden Objekts im Injektor statisch festgelegt. Häufig erhält der Injektor die Information, welches konkrete Objekt erzeugt werden soll, über eine Konfigurationsdatei.

4.8.4.2 Programmierbeispiel für eine "Setter Injection"

Das Beispiel aus Kapitel 4.8.4.1 wird in diesem Kapitel verändert, um die Implementierung einer "Setter Injection" zu zeigen. Bei einer "Setter Injection" muss die von weiteren Objekten abhängige Klasse eine oder mehrere Methoden zur Verfügung stellen, mit deren Hilfe der Injektor die Abhängigkeiten zur Laufzeit festlegen kann.

In dem betrachteten Beispiel wird die bereits bekannte Klasse `AusgabeSteuerung` nun durch die Methode `setAusgabeDienst()` erweitert:

```
// Datei: AusgabeSteuerung.java

public class AusgabeSteuerung
{
   private IAusgabeDienst ausgabeDienst;
   //...
   public AusgabeSteuerung()
   {
     ...
   }

   public void setAusgabeDienst (IAusgabeDienst konkreterDienst)
   {
      ausgabeDienst = konkreterDienst;
   }
   //...
}
```

Diese `set`-Methode wird von der Methode `main()` der Klasse `Injektor` benutzt, um zur Laufzeit ein Objekt einer konkreten Klasse `DateiAusgabeDienst` zu injizieren:

```
// Datei: Injektor.java

public class Injektor
{
   public static void main (String[] args)
   {
      AusgabeSteuerung steuerung = new AusgabeSteuerung();

      // Setter Injection
      steuerung.setAusgabeDienst (new DateiAusgabeDienst());
      steuerung.gebeTextAus ("Setter Injection");
   }
}
```

Die Klasse `Injektor` injiziert in diesem Beispiel eine Referenz auf ein Objekt der Klasse `DateiAusgabeDienst`. Die Ausgabe erfolgt also in einer Textdatei im Projektpfad des Programms.

Die Ausgabe des Programms in die Textdatei ist:

```
Setter Injection
```

Auch hier findet eine Verschiebung der create-Abhängigkeit hin zum Injektor statt. Die nutzende Klasse ist nun von der Schnittstelle als **Abstraktion der Klasse des benötigten Objekts** abhängig anstatt von der konkreten Klasse eines benötigten Objekts. Dadurch wird die Abhängigkeit wie bei der "Constructor Injection" abgeschwächt, jedoch nicht vollständig entfernt.

4.8.4.3 Programmierbeispiel für eine "Interface Injection"

In diesem Kapitel wird das vorliegende Beispiel aus Kapitel 4.8.4.1 abgeändert, um die Durchführung einer "Interface Injection" zu ermöglichen. Für eine "Interface Injection" wird eine zusätzliche Schnittstelle benötigt, welche einen Methodenkopf für eine Methode vorgibt, die es dem Injektor ermöglicht, eine Injektion der Abhängigkeiten zur Laufzeit durchzuführen. Die Klasse, welche eine Abhängigkeit zu einem benötigten Objekt aufweist, muss diese Schnittstelle implementieren und die Methode `injiziereDienst()` zur Verfügung stellen.

Im vorliegenden Beispiel wird die Schnittstelle `IAusgabeDienstInjektion` eingeführt[71], welche den Methodenkopf für die Methode `injiziereAusgabeDienst()` vorgibt:

```
// Datei: IAusgabeDienstInjektion.java

public interface IAusgabeDienstInjektion
{
   void injiziereAusgabeDienst (IAusgabeDienst ausgabeDienst);
}
```

Die Klasse `AusgabeSteuerung` implementiert diese Schnittstelle und muss die Methode `injiziereAusgabeDienst()` zur Verfügung stellen:

```
// Datei: AusgabeSteuerung.java

public class AusgabeSteuerung implements IAusgabeDienstInjektion
{
   private IAusgabeDienst ausgabeDienst;
   //...

   @Override
   public void injiziereAusgabeDienst (IAusgabeDienst ausgabeDienst)
   {
      this.ausgabeDienst = ausgabeDienst;
   }
   //...
}
```

Die Klasse `Injektor` kann nun diese Methode benutzen, um ein Objekt einer konkreten Klasse eines AusgabeDiensts zur Laufzeit zu injizieren:

[71] Da der Injektor diese Schnittstelle nutzt, wird die Schnittstelle in der Regel auch vom Injektor vorgegeben. Dadurch wird die Abhängigkeit zwischen dem Injektor und dem Objekt, in das injiziert werden soll, abgeschwächt.

```
// Datei Injektor.java

public class Injektor
{
   public static void main (String[] args)
   {
      AusgabeSteuerung steuerung = new AusgabeSteuerung();

      // Interface Injection
      steuerung.injiziereAusgabeDienst (new KonsolenAusgabeDienst());
      steuerung.gebeTextAus ("Interface Injection");
   }
}
```

Hier injiziert die Klasse `Injektor` nun wieder eine Referenz auf ein Objekt der Klasse `KonsolenAusgabeDienst`. Die Ausgabe erfolgt also auf der Konsole.

Die Ausgabe des Programms ist:

```
Interface Injection
```

Auch bei der "Interface Injection" wird die ursprüngliche create-Abhängigkeit (vgl. Kapitel 4.8.1) zwischen der nutzenden Klasse und dem Objekt einer benötigten Klasse aufgelöst. Stattdessen entsteht durch die Einführung einer Schnittstelle für die **Klasse des benötigten Objekts als Abstraktion** eine schwächere Abhängigkeitsbeziehung zwischen der nutzenden Klasse und dieser Abstraktion.

4.8.5 Vergleichende Bewertung "Dependency Lookup" und "Dependency Injection"

"Dependency Lookup" und "Dependency Injection" helfen, Abhängigkeiten bei der Erzeugung von Objekten abzuschwächen. Beide Ansätze verfolgen die Strategie, die Erzeugung von benötigten Objekten an eine außenstehende Instanz zu delegieren. In beiden Fällen ist jedoch im Programm eine Abhängigkeit von der Abstraktion der Klasse des gesuchten Objekts notwendig, damit ein Objekt die Methoden des benötigten Objekts aufrufen kann. Es handelt sich also in beiden Fällen um keine vollständige Beseitigung der Abhängigkeiten.[72]

Das Objekt einer nutzenden Klasse behält bei **"Dependency Lookup"** die Kontrolle darüber, wann es nach einem Objekt einer benötigten Klasse sucht. Bei "Dependency Lookup" sucht ein Objekt seine Abhängigkeiten über einen ihm zur Verfügung stehenden Dienst (z. B. über eine Registry). Hierdurch entsteht eine Abhängigkeit zwischen dem Dienst und dem suchenden Objekt. Um diese Suche durchführen zu können, braucht das suchende Objekt jedoch eine spezifische Schlüsselinformation für das Auffinden des gesuchten Objekts. Diese Schlüsselinformation darf sich nicht ändern und muss sowohl dem suchenden Objekt als auch dem Dienst, welcher das gesuchte Objekt zur Verfügung stellt, bekannt sein. Der Dienst muss nach der Übergabe des Schlüssels das zugehörige Objekt zurückliefern. Damit wird die Abhängigkeit

[72] Würden Abhängigkeiten vollständig beseitigt, könnten die betreffenden Objekte nicht wechselwirken.

von der **Abstraktion der Klasse des gesuchten Objekts** durch die Kenntnis der Schlüsselinformation noch verstärkt. Hinzu kommt noch, dass das suchende Objekt auch den Dienst kennen muss.

Bei **"Dependency Injection"** gibt das Objekt einer nutzenden Klasse die Kontrolle über die Zuweisung der Abhängigkeit zu einem benötigten Objekt einer konkreten Klasse vollständig an den Injektor ab. Auch bei "Dependency Injection" bleibt eine schwächere Abhängigkeit zwischen einer nutzenden Klasse und dem Objekt einer benötigten Klasse bestehen. Es findet eine Verschiebung der create-Abhängigkeit hin zum Injektor statt und die nutzende Klasse ist stattdessen von einer **Abstraktion der Klasse eines benötigten Objekts** abhängig. Diese Abstraktion wird meist als Schnittstelle implementiert. Dadurch kann zur Laufzeit ohne Aufwand jedes beliebige Objekt, dessen Klasse diese Schnittstelle zur Kompilierzeit implementiert, den Platz der Schnittstelle als Abstraktion einnehmen, wodurch bei "Dependency Injection" ein hohes Maß an Flexibilität erreicht wird.

Dabei haben die Umsetzungsmöglichkeiten "Setter Injection" und "Interface Injection" gegenüber der "Constructor Injection" den Vorteil, dass das injizierte, benötigte Objekt durch einen entsprechenden Methodenaufruf ausgetauscht werden kann. "Constructor Injection" wird dagegen eingesetzt, wenn eben diese Austauschbarkeit zur Laufzeit von den Entwicklern nicht gewünscht ist.

Bei "Dependency Lookup" bestimmt nicht nur die Abstraktion die Auswahl der gesuchten Objekte, sondern zusätzlich auch der zur Suche verwendete statische Schlüssel und die Kenntnis beispielsweise der Registratur, wodurch eine stärkere Abhängigkeit entsteht als bei "Dependency Injection". "Dependency Injection" wird in der Praxis auch gerne bei automatisierten Tests wie z. B. Unit Tests eingesetzt. Durch "Dependency Injection" ist es sehr einfach, bei zu testenden Klassen Mock-Objekte anstatt echter benötigter Objekte zu injizieren, wodurch sich die Entwickler das Aufsetzen einer aufwendigen Testumgebung sparen.

Beide Ansätze, "Dependency Lookup" und "Dependency Injection", sind hilfreich bei der Abschwächung von Abhängigkeiten bei der Erzeugung von Objekten. In der Praxis wird heute vor allem der Ansatz "Dependency Injection" verwendet, da bei diesem Ansatz die benutzenden Objekte unabhängig vom Injektor bleiben.

"Dependency Lookup" hat gegenüber "Dependency Injection" inzwischen an Bedeutung verloren. Selbst bei Frameworks wie **Spring** wurde zusätzlich zu "Dependency Lookup" auch "Dependency Injection" angeboten. "Dependency Lookup" genießt inzwischen mehr ein historisches Interesse.

4.9 Zusammenfassung

Durch "Loose Coupling and Strong Cohesion" (siehe Kapitel **4.1**) wird gefordert, dass die verschiedenen Module eines Systems aufgrund ihrer innewohnenden starken Kohäsion separierbar sind und untereinander eine schwache Kopplung aufweisen. Aus diesen Forderungen resultiert letztendlich der Zugriff auf andere Module über Schnittstellen. "Loose Coupling and Strong Cohesion" stammt noch aus der prozeduralen Welt, gilt aber allgemein. Für die Konstruktion von Systemen ist "Loose Coupling and Strong Cohesion" nach wie vor von höchster Bedeutung. Die Prinzipien

- "Single Responsibility Principle" und
- "Dependency Inversion Principle"

kann man als Konsequenz von "Loose Coupling and Strong Cohesion" betrachten, das "Law of Demeter" als dessen objektorientierte Ausprägung.

Das Prinzip "Information Hiding" (siehe Kapitel **4.2**) beruht auf der Absicht, Programmcode, der zusammengehörig ist und eventuell noch im Fluss ist, in eigene Module zu kapseln. Damit werden Design-Entscheidungen verborgen. Das Verbergen der Implementierung hinter einer schmalen Schnittstelle erlaubt es, Implementierungen abzuändern, ohne andere Module zu beeinträchtigen.

"Separation of Concerns" (siehe Kapitel **4.3**) ist ein allgemeines Ordnungsprinzip und fordert, die verschiedenen Concerns (dt. Belange) eines Systems sauber gegeneinander abzugrenzen.

Das "Law of Demeter" (siehe Kapitel **4.4**) projiziert "Loose Coupling and Strong Cohesion" auf die objektorientierte Welt. Es spezifiziert letztendlich, welche Methodenzugriffe erlaubt sind, und sorgt hierdurch für einen "schüchternen" Code.

Das "Dependency Inversion Principle" (siehe Kapitel **4.5**) verlangt allgemein für die Module einer Hierarchie, dass die Module einer tieferen Ebene nur über ihre eigenen Abstraktionen angesprochen werden dürfen. Dieses Prinzip dient damit zur Abschwächung von Abhängigkeiten unter Modulen, die zueinander in einer hierarchischen Aufrufbeziehung stehen. Ein Modul einer höheren Ebene ist nur von der Abstraktion eines Moduls der tieferen Ebene abhängig und nicht von dessen Implementierung. Dies erlaubt in Aufrufhierarchien den Einsatz verschiedener Implementierungen der Abstraktion und die Verwendung von Stubs oder Mock-Objekten für das Testen.

Das "Interface Segregation Principle" (siehe Kapitel **4**.**6**) fordert, dass Clients nur Schnittstellen erhalten sollen, deren Methoden sie auch tatsächlich nutzen (Rollen-Schnittstellen).

Das "Single Responsibility Principle" (siehe Kapitel **4**.**7**) fordert, dass ein Modul nur eine einzige Verantwortlichkeit hat. Damit werden bei Änderungen einer Verantwortlichkeit unbeabsichtigte Beschädigungen anderer Module vermieden.

"Dependency Lookup" und "Dependency Injection" (siehe Kapitel **4.8**) sind beides Konzepte, um bei der Erzeugung von Objekten Abhängigkeiten abzuschwächen. Aber eine Restabhängigkeit zu dem erzeugten Objekt bleibt auf jeden Fall bestehen: Das nutzende Objekt muss die Abstraktion der Klasse des erzeugten Objektes kennen.

Bei Verwendung des Konzepts "Dependency Lookup" werden Informationen über die Objekte in einer zentralen Instanz gespeichert. Diese Informationen sind bei Kenntnis der zentralen Instanz über den Suchschlüssel wie beispielsweise den Objektnamen oder die Schnittstelle der Klasse des gesuchten Objekts zugänglich. Damit muss das suchende Objekt nicht mehr alle Details des gesuchten Objekts kennen wie beispielsweise dessen konkrete Klasse, was zu einer Abschwächung der Abhängigkeiten führt.

Bei "Dependency Injection" generiert der Injektor alle Objekte und Verbindungen. Ein Objekt ist vom Injektor total unabhängig. Es kennt ihn überhaupt nicht.

Kapitel 5

Entwurfsprinzipien für korrekte Programme

5 Entwurfsprinzipien für korrekte Programme

Bei dem Begriff "Korrektheit" kann man zwei Dimensionen unterscheiden:

- einerseits die **korrekte Konstruktion der Software** durch die Entwickler (innere Qualität),
- andererseits aber auch den **Erfüllungsgrad der Kundenforderungen** (äußere Qualität).

Dass Software korrekt – also ohne Fehler – konstruiert wird, ist die Pflicht der Entwickler.

Maßnahmen, um die **Korrektheit der Konstruktion eines System**s zu erreichen, sind im Wesentlichen:

- ein "shared understanding" im Team für den betrachteten Problembereich, die Vision des zukünftigen Systems, die Architektur des Systems und den jeweils aktuellen Stand des Projekts,
- Fehlervermeidung z. B. durch die Verwendung gemeinsamer Methoden im Team,
- Einfachheit der Programme,
- Überprüfungen von Spezifikationen bzw. Programmen z. B. durch Reviews oder Tests und
- die Einhaltung des liskovschen Substitutionsprinzips für Programme, die polymorphe Objekte verwenden können sollen.

Es kann sein, dass der Kunde zufälligerweise Konstruktionsfehler anhand von Laufzeitfehlern erkennt. Im Prinzip jedoch weiß der Kunde bis zur Wartung nichts über die Konstruktion der Software. Der Kunde kann daher in der Regel nur den Erfüllungsgrad seiner eigenen Forderungen überprüfen (**Korrektheit als äußere Qualität**). Ein System soll das tun, was von ihm verlangt wird. Aber selbst wenn ein Konstruktionsfehler der Software vom Kunden zunächst gar nicht sofort bemerkt werden würde, muss ein System intern korrekt konstruiert werden, da es ansonsten mit Sicherheit zu einem späteren Zeitpunkt zu Fehlern kommt.

Korrektheit als innere Qualität ist die wichtigste aller Eigenschaften einer Software. Das Erfüllen weiterer Qualitätsmerkmale ist nutzlos, wenn ein System falsch konstruiert ist.

Um objektorientierte Programme korrekt zu schreiben, ist es wichtig, dass ein Programm die gewünschten Beziehungen zwischen seinen Klassen einhält. Dazu muss man aber erst in der Lage sein, die gewünschten Beziehungen zwischen Klassen überhaupt formulieren zu können.

Bertrand Meyer hat sich um die Beziehungen zwischen Klassen gekümmert.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8_5

Bertrand Meyer hat im Rahmen von "**Design by Contract"** sogenannte **Verträge** als Spezifikation der Beziehungen zwischen Klassen eingeführt.

Verträge werden in Kapitel 5.1 erläutert.

Verträge werden durch sogenannte Vorbedingungen, Nachbedingungen und Invarianten definiert.

Das liskovsche Substitutionsprinzip (siehe Kapitel 5.2) befasst sich mit der Fragestellung, unter welchen Bedingungen Objekte in korrekter Weise polymorph verwendet werden dürfen. Voraussetzung hierfür ist, dass die Klassen der Objekte in einer Vererbungshierarchie liegen.

Das liskovsche Substitutionsprinzip fordert, dass eine Referenz vom Typ einer Basisklasse nicht nur auf ein Objekt der Basisklasse, sondern auch auf ein Objekt einer von der Basisklasse abgeleiteten Klasse zeigen kann. Damit muss nicht bei jedem Zugriff einzeln geprüft werden, von welchem Typ das referenzierte Objekt einer Vererbungshierarchie eigentlich ist.

Um das liskovsche Substitutionsprinzip zu erfüllen, muss eine abgeleitete Klasse die Verträge ihrer Basisklasse mit Kundenklassen einhalten, damit ein Kundenobjekt nicht bemerkt, dass an der Stelle eines Objekts einer Basisklasse plötzlich ein Objekt einer davon abgeleiteten Klasse steht.

Kapitel 5.3 befasst sich mit dem "Principle of Least Astonishment". Das "Principle of Least Astonishment" will die korrekte Interpretation der Programme sicherstellen.

In seiner urspünglichen Form will das "Principle of Least Astonishment" verhindern, dass Oberflächen zur Laufzeit falsch interpretiert werden und zu allergrößter Verwunderung verbunden mit Bedienfehlern führen.

Dieses Prinzip kann in seiner erweiterten Form auf alle Erzeugnisse eines Systems bezogen werden.

5.1 Das Konzept "Design by Contract"

Das Konzept "**Design by Contract**" (dt. **Entwurf durch Verträge**) ist von hoher Bedeutung, um zu spezifizieren, welche Beziehungen Aufrufer und Aufgerufener allgemein in Programmen zueinander haben.

Man braucht "Design by Contract" auch, um zu beschreiben, wie die Bedingungen lauten, damit abgeleitete Klassen in Klassenhierarchien an die Stelle von Basisklassen treten können.

5.1.1 Historie

"Design by Contract", abgekürzt als DbC, wurde 1992 von Bertrand Meyer [Mey92] als Entwurfstechnik eingeführt.

5.1.2 Ziel

Das Konzept "Design by Contract" wurde von Bertrand Meyer exemplarisch in der von ihm entwickelten Programmiersprache Eiffel umgesetzt, stellt aber ein allgemeingültiges Konzept dar, welches beim objektorientierten Entwurf unabhängig von der einzusetzenden Programmiersprache angewandt werden kann.

Eine Klasse wird in einem Programm von anderen Klassen – hier Kunden (engl. clients) genannt – benutzt und hat damit Beziehungen zu all ihren Kunden.

Das Konzept "Design by Contract" fordert für die Beziehungen zwischen Objekten formale Übereinkünfte in Form von sogenannten **Verträgen** zwischen den beteiligten Partnern, in denen präzise definiert wird, unter welchen Umständen die Beziehungen zwischen den Partnern korrekt sind.

Bei einem Vertrag[73] haben in der Regel beide Seiten Rechte und Pflichten.

"Design by Contract" sorgt dafür, dass beim Aufruf einer Methode Aufrufer und aufgerufene Methode sich gegenseitig aufeinander verlassen können. Dies erfolgt durch die Prüfung der Verträge zur Laufzeit.

In dem Konzept "Entwurf durch Verträge" von Bertrand Meyer werden **Verträge** mit Hilfe von sogenannten **Zusicherungen** spezifiziert.

Zusicherungen sind boolesche Ausdrücke über Zustände von Objekten und Werte von Variablen an bestimmten Programmstellen, die niemals falsch werden dürfen.

5.1.3 Zusicherungen

Ein Vertrag kann drei verschiedene Arten von Zusicherungen enthalten:

[73] Eine Klasse kann mit verschiedenen Kundenklassen verschiedene Verträge haben.

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

So wie im Alltag ein Vertrag die Beziehungen zwischen Parteien (Personen, Organisationen) regelt, beschreiben Vor- und Nachbedingungen die Verträge einer Methode mit ihren Kunden, den Aufrufern.

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

Wie bei einem guten Vertrag im täglichen Leben haben Aufrufer und Aufgerufener Pflichten und Vorteile. Der Aufgerufene hat den Vorteil, dass er unter den gewünschten Bedingungen ablaufen kann. Dies wird durch die Vorbedingungen geregelt. Der Aufrufer hat den Vorteil, dass gewisse Bedingungen, die Nachbedingungen, nach dem Aufruf einer Methode erfüllt sind.

Invarianten beziehen sich nicht auf einzelne Methoden, sondern beziehen sich immer auf eine Klasse. Eine Invariante muss also für jedes einzelne Objekt einer Klasse erfüllt sein, damit ein System korrekt arbeitet oder in einem korrekten Zustand ist. Da die Invarianten von allen Methoden einer Klasse, die von einem Kunden aufgerufen werden können, eingehalten werden müssen, spricht man auch von **Klasseninvarianten**.

Der **Vertrag einer Klasse** umfasst

- die Verträge der Methoden, also deren Vor- und Nachbedingungen, sowie
- die Invarianten der Klasse.

5.1.3.1 Vorbedingung

Eine **Vorbedingung** (engl. **precondition**) ist eine Einschränkung, unter der eine Methode korrekt aufgerufen wird. Sie stellt eine Verpflichtung für einen Aufrufer dar. Hierbei ist es unerheblich, ob der Aufruf innerhalb der eigenen Klasse erfolgt oder von einer Kundenklasse kommt.

Ein korrekt arbeitendes System führt nie einen Aufruf durch, der nicht die Vorbedingungen einer aufgerufenen Methode erfüllen kann.

Eine Vorbedingung bindet also einen Aufrufer. Sie definiert die Bedingung, unter der ein Aufruf zulässig ist.

Der Aufrufer hat die Pflicht, die Vorbedingungen des Aufzurufenden zu erfüllen. Er muss prüfen, ob diese erfüllt sind. Den Nutzen hat der Aufgerufene.

Von der Theorie her ist nach Bertrand Meyer der Aufrufer für die Überprüfung des Einhaltens von Vorbedingungen verantwortlich, sodass man davon ausgehen kann, dass es keine fehlerhaften Aufrufe gibt.

Wenn sehr viele Aufrufer jedoch ein und dieselbe Methode aufrufen, wird aus Gründen der Einfachheit oft die Prüfung einer Vorbedingung im Aufgerufenen vorgenommen. Der Aufgerufene wirft dann, wenn eine Vorbedingung nicht eingehalten wurde, eine entsprechende Exception.

Erfolgt die Prüfung einer Vorbedingung im Aufgerufenen, so muss dieser im Fehlerfall eine Exception werfen und darf seinen Programmcode nicht ausführen.

5.1.3.2 Nachbedingung

Nachbedingungen (engl. **postconditions**) stellen den Zustand nach dem Aufruf einer Methode dar. Sie binden eine Methode einer Klasse und stellen Bedingungen dar, die von der Methode eingehalten werden müssen.

Mit einer Nachbedingung wird garantiert, dass der Aufrufer nach Ausführung einer Methode einen Zustand mit gewissen Eigenschaften vorfindet, natürlich immer unter der Voraussetzung, dass beim Aufruf der Methode die Vorbedingungen erfüllt waren.

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

5.1.3.3 Invariante

Eine **Invariante** (engl. **invariant**) ist eine **Zusicherung** bezüglich einer Klasse. Sie muss also von allen Objekten dieser Klasse eingehalten werden.

Die Eigenschaften einer Invariante gelten für alle Methoden einer Klasse und nicht individuell nur für eine einzelne Methode. Sie sind damit Klasseneigenschaften im Gegensatz zu Vor- und Nachbedingungen von Methoden, welche die zulässigen Wertebereiche für den Aufruf bzw. den Rückgabewert, das Werfen von Exceptions und Seiteneffekte einzelner Methoden regeln.

Invarianten müssen von allen nach außen sichtbaren Methoden einer Klasse vor und nach dem Methodenaufruf eingehalten werden.

Invarianten gelten während der gesamten Lebensdauer der Objekte einer Klasse. Invarianten gelten ab dem Zeitpunkt des abgeschlossenen Konstruktoraufrufs.

Beim Aufruf von internen bzw. privaten Methoden einer Klasse – sogenannten **Service-Methoden** – müssen die **Invarianten** einer Klasse nicht unbedingt beachtet werden und können **auch mal nicht erfüllt** sein, da der Kunde hiervon nichts bemerkt.

Eine Invariante **kann während der Ausführung einer Schnittstellenmethode temporär verletzt werden**. Dies stellt kein Problem dar, da eine Invariante erst nach Ausführung einer Schnittstellenmethode dem Aufrufer wieder zur Verfügung steht.

5.1.3.4 Konzeptionelles Beispiel

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

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

Diese Klasse hat die Invariante, dass die Zahl der Elemente auf dem Stack bei allen Operationen größer gleich Null und kleiner gleich der gegebenen maximalen Stackgröße sein muss. Wie bereits bekannt ist, ist eine Invariante für alle Methoden einer Klasse gültig.

Die Methode `push()` hat die Vorbedingung, dass die Zahl der Elemente auf dem Stack kleiner als die maximale Stackgröße sein muss.[74] Die Nachbedingung von `push()` lautet, dass die Zahl der Elemente auf dem Stack um eins größer als vor dem Aufruf ist.

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

5.1.4 Einhalten der Verträge beim Überschreiben

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

[74] Damit ein Aufrufer eine Vorbedingung gewährleisten kann, muss ihm eine Methode `size()` zur Verfügung stehen.

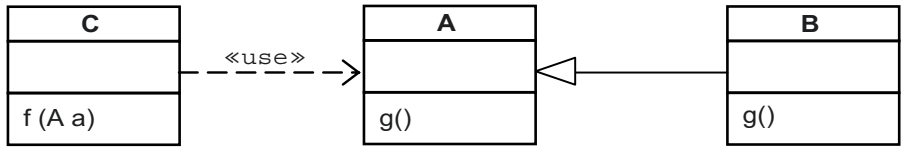


Abbildung 5-1 Überschreiben einer Methode

Die Klasse `B` sei von der Klasse `A` abgeleitet und soll die Methode `g()` aus `A` überschreiben. Aufrufer von `g()` sei eine Methode `f()` einer Kundenklasse `C`. Die Methode `f()` soll die folgende Aufrufschnittstelle besitzen: `f (A a)`. Mit anderen Worten: An `f()` kann beispielsweise eine Referenz auf ein Objekt der Klasse `A` oder eine Referenz auf ein Objekt der von der Klasse `A` abgeleiteten Klasse `B` übergeben werden.

Der Kunde `f()` kann zur Laufzeit nicht wissen, ob ihm eine Referenz auf ein Objekt der Klasse `A` oder der Klasse `B` übergeben wird (liskovsches Substitutionsprinzip, siehe Kapitel 5.2). **Dem Kunden `f()` ist auf jeden Fall nur die Basisklasse `A` bekannt und daran richtet er sich aus**. Also kann `f()` beim Aufruf von `g()` nur den Vertrag der Methode `g()` aus der Basisklasse `A` beim Aufruf beachten. `f()` stellt also die Vorbedingungen für `g()` aus `A` sicher und erwartet im Gegenzug, dass `g()` seine Nachbedingungen erfüllt.

Wie im täglichen Leben auch, darf ein Vertrag übererfüllt werden, er darf aber nicht verletzt werden! Dies hat zur Konsequenz, dass `g()` aus der abgeleiteten Klasse `B` die **Vorbedingungen nicht verschärfen** kann, denn darauf wäre der Kunde `f()` überhaupt nicht eingerichtet. `g()` aus `B` darf aber die Vorbedingungen aufweichen. Dies stellt für `f()` überhaupt kein Problem dar, denn aufgeweichte Vorbedingungen kann `f()` sowieso mühelos einhalten.

Eine **Vorbedingung** kann beim Überschreiben nur aufgeweicht werden, denn eine aufgeweichte Vorbedingung kann der Kunde problemlos erfüllen.

`g()` aus `B` darf die **Nachbedingungen nicht aufweichen**, denn der Kunde `f()` erwartet die Ergebnisse in einem bestimmten Bereich. Da ein Kunde sich auf die Basisklasse eingestellt hat, muss sichergestellt werden, dass sich der überschreibende Anteil korrekt verhält.

Eine **Nachbedingung** kann nur verschärft werden (kleinerer Wertebereich), denn auf eine aufgeweichte Nachbedingung (größerer Wertebereich) wäre der Kunde gar nicht vorbereitet, da er sich an der Basisklasse orientiert.

Eine überschreibende Methode einer abgeleiteten Klasse darf

- eine Nachbedingung der überschriebenen Methode nicht aufweichen und
- eine Vorbedingung der überschriebenen Methode nicht verschärfen.

Im Folgenden wird ein **konzeptionelles Beispiel** gegeben:

Hat beispielsweise eine Methode einen Rückgabewert vom Typ `int` und garantiert, dass sie nur Werte zwischen 1 und 5 liefert, so darf die überschreibende Methode keine Werte außerhalb dieses Bereichs liefern. Besitzt diese Methode zudem beispielsweise einen formalen Parameter vom Typ `int`, welcher in einem Wertebereich von 1 bis 10 definiert ist, so darf die überschreibende Methode diesen Wertebereich nicht einschränken.

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

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

Abgeleitete Klassen dürfen Invarianten nur verschärfen.

5.1.5 Bewertung

Die folgenden **Vorteile** von Design by Contract werden gesehen:

- **Korrektheit und Stabilität der Programme**

  Mit "Design by Contract" können Verträge zwischen Klassen spezifiziert werden. Bei Einhaltung dieser Verträge kann sich eine Klasse auf eine andere Klasse verlassen. Die Klassen verhalten sich dann "ordnungsgemäß". Damit wird ein Programm stabiler.

- **Praktikabilität des liskovschen Substitutionsprinzips**

  Ohne die Einhaltung der Verträge der Basisklassen können Objekte abgeleiteter Klassen nicht an die Stelle von Objekten der entsprechenden Basisklasse treten, denn sonst würde es die Kundenklasse bemerken und das liskovsche Substitutionsprinzip (siehe Kapitel 5.2) wäre verletzt.

- **Hilfe bei der Dokumentation**

  Durch die Formulierung der Vor- und Nachbedingungen ist bereits ein Teil der Dokumentation erledigt.

- **Hilfe beim Testen**

  Durch die Festlegung der Verträge wird das gültige Verhalten und das ungültige Verhalten festgelegt.

Der folgende **Nachteil** wird gesehen:

- Der Programmieraufwand wird größer.

5.2 Liskovsches Substitutionsprinzip

Das liskovsche Substitutionsprinzip (abgekürzt als LSP) gilt nur in der Objektorientierung, bezieht sich auf die Vererbung und ist dort für die Verwendung polymorpher Objekte[75] von allerhöchster Bedeutung.

Das liskovsche Substitutionsprinzip fordert, dass Referenzen auf Objekte einer Basisklasse auch auf Objekte einer davon abgeleiteten Klasse zeigen können.

Wenn das liskovsche Substitutionsprinzip erfüllt ist, merkt es ein Kunde nicht, wenn an die Stelle eines referenzierten Objekts der Basisklasse ein Objekt einer abgeleiteten Klasse tritt. Damit hat man den Vorteil, dass ein Programm für beide Typen (Basisklasse und abgeleitete Klasse) verwendet werden kann und dass hierbei nicht bei jedem Objekt einzeln geprüft werden muss, von welchem dieser beiden Typen das Objekt denn ist.

Das liskovsche Substitutionsprinzip stellt nur diese Forderung auf, gibt aber keine Handlungsanweisungen und hat deshalb eigentlich gar keinen Prinzip-Charakter.

Das LSP hat mehr den Charakter eines Konzeptes und verlangt das Unterlassen der Bildung gewisser Subtypen in einer Vererbungshierarchie.

Wird das LSP nicht eingehalten, können ernsthafte Probleme bei der polymorphen Verwendung von Objekten abgeleiteter Klassen anstelle der im Programm verwendeten Objekte einer Basisklasse auftreten.

Polymorphie (Vielgestaltigkeit) ist ein zentrales Konzept der objektorientierten Programmierung. Polymorphie betrifft sowohl Methoden als auch Objekte. Ehe im Folgenden auf das liskovsche Substitutionsprinzip eingegangen wird, sollen zunächst die beiden Formen der Polymorphie kurz vorgestellt werden.

5.2.1 Polymorphie von Methoden

Bei der **Polymorphie von Methoden** stellt jede Klasse, die sich **nicht in einer Vererbungshierarchie** befindet, einen eigenen Namensraum dar. Derselbe Methodenkopf kann in jeder Klasse in einer jeweils anderen Ausprägung implementiert werden. Diese Ausprägung der Polymorphie ist für das liskovsche Substitutionsprinzip irrelevant.

5.2.2 Polymorphie von Objekten

Polymorphie von Objekten gibt es nur im Falle von **Vererbungshierarchien**.

[75] Streng genommen sind die Referenzvariablen polymorph.

Diese Ausprägung der Polymorphie ist für das liskovsche Substitutionsprinzip von Bedeutung.

Polymorphie von Objekten bedeutet, dass ein Objekt einer **Vererbungshierarchie** sich in Gestalt verschiedener Typen zeigen kann. Ein Objekt einer abgeleiteten Klasse kann auch als Objekt der zugehörigen Basisklasse betrachtet werden.

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

Wichtig ist, dass im Falle des **Überschreibens** einer Methode der Basisklasse durch eine Methode einer abgeleiteten Klasse die überschreibende Methode gleichartig wie die überschriebene Methode der Basisklasse aufgerufen wird und auch gleichartig antwortet. Hierbei nimmt die Implementierung der Methode in der abgeleiteten Klasse jedoch eine andere Ausprägung an als in der Basisklasse. Nur wenn Aufrufe und Rückgabe in gleicher Weise erfolgen, d. h. die Verträge der Basisklasse in der abgeleiteten Klasse beim Überschreiben eingehalten werden, merkt ein Kunden-Objekt nicht, dass an der Stelle eines Objekts einer Basisklasse plötzlich ein Objekt einer abgeleiteten Klasse steht.

Findet ein Überschreiben – also die Neudefinition einer Methode in einer abgeleiteten Klasse – statt, so muss darauf geachtet werden, dass in der abgeleiteten Klasse die Verträge der Basisklasse mit ihren Kunden eingehalten werden, wenn ein Objekt einer abgeleiteten Klasse an die Stelle eines Objekts der Basisklasse treten soll.

5.2.3 Historie

Barbara Liskov hat 1987 in [Lis87] die Forderung formuliert, bei deren Einhaltung ein Programm die Verwendung polymorpher Objekte aus Vererbungshierarchien gefahrlos unterstützt.

5.2.4 Ziel

Das **liskovsche Substitutionsprinzip** fordert:

Will man Polymorphie von Objekten für ein Programm haben, so muss im Programm eine Referenz auf ein Objekt einer abgeleiteten Klasse an die Stelle einer Referenz auf ein Objekt der Basisklasse treten können.

Das bedeutet:

Überschreibende Methoden einer abgeleiteten Klasse dürfen die **Verträge**[76] einer Basisklasse mit Kundenklassen nicht brechen, denn sonst würde eine Kundenklasse etwas vom Austausch der Objekte bemerken.

Solange bei der Ableitung von einer Basisklasse der Vertrag der entsprechenden Basisklasse nicht gebrochen wird, ist es möglich, ein für Basisklassen geschriebenes Programm auch für Klassen zu verwenden, die eventuell erst zu einem späteren Zeitpunkt abgeleitet werden.

Wenn das liskovsche Substitutionsprinzip nicht gelten würde, könnten Referenzen vom Typ einer Basisklasse nicht auf Objekte abgeleiteter Klassen zeigen. Es müsste dann stets geprüft werden, von welchem Typ das aktuelle Objekt ist, auf welches die Referenz gerade verweist.

Oftmals wird das liskovsche Substitutionsprinzip dadurch verletzt, indem eine überschreibende Methode in einer abgeleiteten Klasse keine Funktionalität oder weniger Funktionalität aufweist als die entsprechende überschriebene Methode der Basisklasse [Mar12, p. 124].

Kommentar des Autors:

Zufälligerweise kann ein Programm auch laufen, wenn das liskovsche Substitutionsprinzip verletzt wird, dennoch ist es nicht korrekt. Tritt eine abgeleitete Klasse, die das liskovsche Substitutionsprinzip nicht einhält, an die Stelle einer Basisklasse und das Programm läuft nicht mehr, darf auf keinen Fall die Kundenklasse "repariert" werden. Dieser Anfängerfehler sollte unbedingt vermieden werden!

Man muss das liskovsche Substitutionsprinzip unter zwei Gesichtspunkten betrachten:

Fall 1: Reines Erweitern einer Basisklasse[77]

Erweitert eine Klasse eine andere Klasse nur und überschreibt nichts, sondern fügt nur zusätzliche Attribute und Methoden hinzu, dann kann im Programm eine Instanz dieser abgeleiteten Klasse problemlos an die Stelle einer Instanz ihrer Basisklasse treten. Zeigt eine Referenz vom Typ der Basisklasse in einem Programm auf ein Objekt einer abgeleiteten Klasse, so sieht man das Objekt aus Sicht der Basisklasse. Der erweiternde Anteil der abgeleiteten Klasse fällt somit weg.

[76] Der Vertrag einer Klasse umfasst die Vor- und Nachbedingungen der Methoden und die Invarianten einer Klasse. In Kapitel 5.1.4 wird erläutert, was es bedeutet, dass ein Vertrag durch eine abgeleitete Klasse nicht gebrochen werden darf.

[77] Wird nur erweitert, so wird bei der Ableitung keine Methode überschrieben.

Zusätzliche, durch Erweiterung hinzugefügte Methoden abgeleiteter Klassen werden von einem Programm, das nur Instanzen der Basisklasse kennt, sowieso nicht angesprochen.

Fall 2: Überschreiben in einer abgeleiteten Klasse

Eine Projektion auf die Sicht der Basisklasse erfolgt natürlich auch, wenn die abgeleitete Klasse Methoden der Basisklasse überschreibt. Im Fall einer späten Bindung von Instanzmethoden wie in Java wird dann die überschreibende Methode der abgeleiteten Klasse an Stelle der überschriebenen Methode der Basisklasse aufgerufen. Dies sollte jedoch keinen schädlichen Einfluss auf den Ablauf des Programms haben! Das liskovsche Substitutionsprinzip fordert, dass ein Programmierer beim Überschreiben dafür sorgen muss, dass eine überschreibende Instanzmethode der abgeleiteten Klasse den Vertrag der entsprechenden überschriebenen Instanzmethode der Basisklasse einhält.

Werden in einem Programm mit polymorphen Objekten Instanzmethoden einer Basisklasse in einer abgeleiteten Klasse überschrieben, so tritt im Objekt der abgeleiteten Klasse die überschreibende Instanzmethode an die Stelle der geerbten, überschriebenen Methode.

Es wird somit für jedes Objekt die konkrete Methodenimplementierung seines speziellen Typs aufgerufen.

5.2.5 Konzeptionelles Beispiel

Abbildung 5-2 zeigt als Beispiel eine Klassifikation von Vögeln. Die abstrakte Klasse `Vogel` deklariert die abstrakten Operationen `essen()` und `fliegen()`.

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

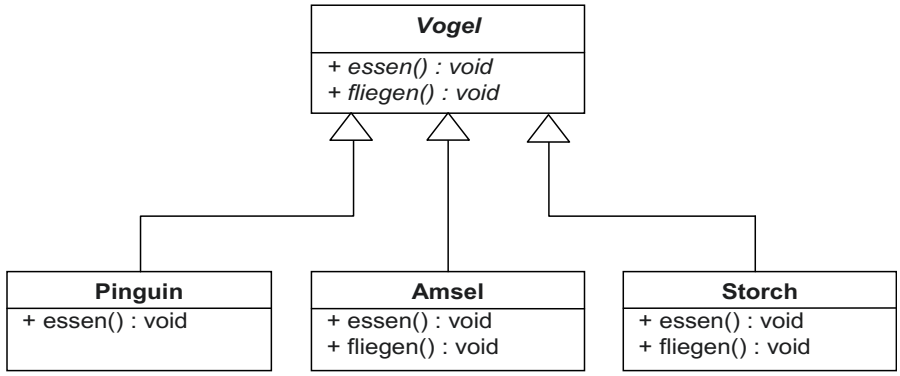


Abbildung 5-2 Fehlerhafte Ableitung

Sämtliche Methoden einer abstrakten Klasse müssen in einer davon abgeleiteten, konkreten Klasse implementiert sein.

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

Durch eine Einteilung der Vögel in flugfähige und flugunfähige Tiere kann – wie in folgender Abbildung gezeigt – das Problem gelöst werden. Denn mit dieser Einteilung muss die Klasse `Pinguin` das Fliegen nicht implementieren. Hier das entsprechende Klassendiagramm:

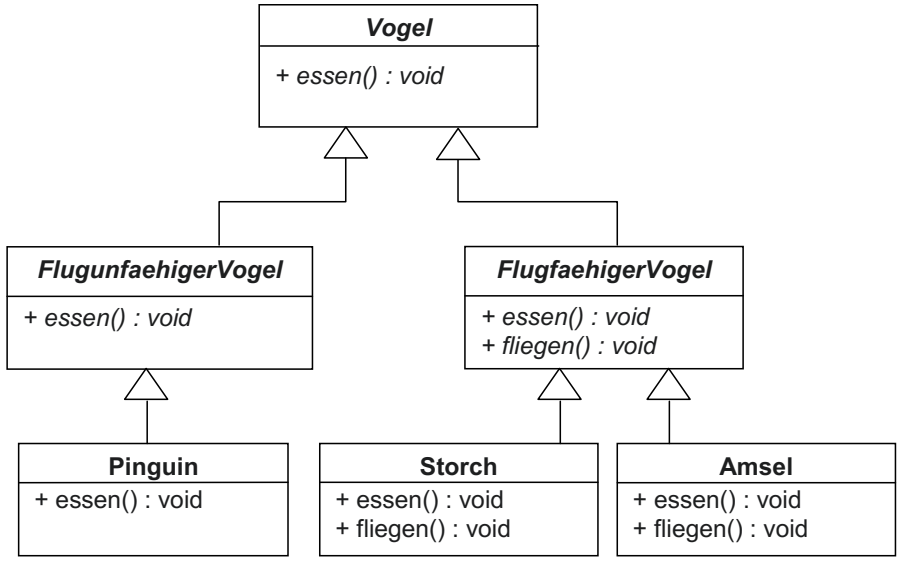


Abbildung 5-3 Einteilung der Vögel in unterschiedliche Gruppen

5.2.6 Bewertung

Das liskovsche Substitutionsprinzip hat die folgenden **Vorteile**:

- **Entfall von Typprüfungen**

  Die Einhaltung des liskovschen Substitutionsprinzips führt dazu, dass man nicht jedesmal prüfen muss, von welchem Typ ein Objekt einer Vererbungshierarchie eigentlich ist.

- **Vermeidung semantischer Probleme**

  Das liskovsche Substitutionsprinzip führt bei seiner Einhaltung zur korrekten Verwendung polymorpher Objekte. Schwerwiegende semantische Probleme bei der Verwendung von Subtypen werden vermieden.

Das liskovsche Substitutionsprinzip hat den **Nachteil**:

- **es ist eigentlich keine Handlungsanweisung**

  Das liskovsche Substitutionsgesetz ist keine Handlungsanweisung – also kein Prinzip –, sondern empfiehlt das Unterlassen gewisser Subtypen.

5.3 Principle of Least Astonishment

Nach dem "Principle of Least Astonishment"[78], abgekürzt zu PLA, von Geoffrey James [Jam86] sollten Programme so entworfen werden, dass ihr Verhalten den Benutzer nicht überrascht. Mit anderen Worten:

> Programme sollen sich gegenüber ihrem Nutzer so verhalten, wie sie es andeuten. Sie sollen den Benutzer nicht täuschen, da damit Bedienfehler provoziert würden.

Dieses Prinzip trägt damit zur **Verständlichkeit** und zur **Fehlervermeidung** bei.

5.3.1 Historie

Geoffrey James formulierte das Prinzip 1987 in seinem Buch "Tao of Programming“ [Jam86], in welchem verschiedene Programmierprinzipien im Stil chinesischer Tao-Texte beschrieben werden. Er schreibt in seinem Buch 4 – einem Unterkapitel des "Tao of Programming" – unter der Überschrift "A well-written program is its own heaven; a poorly-written program is its own hell" unter anderem:

"A program should follow the 'Law of Least Astonishment'. What is this law? It is simply that the program should always respond to the users in the way that least astonishes them."

Samuel Keene [keenes] schreibt:

"What exactly is the 'law of least astonishment'? From a reliability point of view, the 'law of least astonishment' is a guidepost to designers and programmers of computer interfaces, propagating a 'mental checks and balances' archetype, affirmed by the ideal that interfaces should always perform in a manner originally intended, and not in a manner that will astonish (negatively) those 'interfaced'."

[78] manchmal auch "Principle/Law/Rule of Least Surprise" oder auf Deutsch "Prinzip der geringsten Überraschung/des geringsten Erstaunens" genannt

5.3.2 Ziel

Überraschungen führen zu Unterbrechungen des Arbeitsflusses und zu Fehlern. Das gilt insbesondere für eine Fehlbedienung des Systems im laufenden Betrieb. Durch das "Principle of Least Astonishment" soll die Bedienbarkeit der Programme zur Laufzeit verbessert werden.

Schnittstellen eines Programms sollen sich so verhalten, wie der Benutzer es erwartet. Das bedeutet beispielsweise, dass die Funktionen von Buttons in einer Benutzeroberfläche auf den ersten Blick erkennbar sein sollten.

Je geringer die Überraschung über das tatsächliche Verhalten ist, desto einfacher lässt sich ein System verstehen, korrekt verwenden und warten.

Das "Principle of Least Astonishment" wurde ursprünglich nur für Benutzeroberflächen formuliert.

Das "Principle of Least Astonishment" kann jedoch auch in dem Sinne verwendet werden, Überraschungen generell zu vermeiden, um mögliche Interpretationsfehler zur verhüten. Alle Erzeugnisse eines Systems müssen in eindeutiger Weise verstanden werden können.

So dürfen beispielweise Programme eines Programmierers nicht für die anderen Programmierer missverständlich sein. Sie müssen intuitiv korrekt erfasst werden können. Dies kann Programmierrichtlinien und Absprachen erfordern.

5.3.3 Umsetzung

Das Prinzip in seiner ursprünglichen Form will die Eindeutigkeit der Bedienung einer Oberfläche sicherstellen. Es kann umgesetzt werden, indem gängige Konventionen eingehalten werden und Absprachen mit den Nutzern getroffen werden, damit das lauffähige Programm im Betrieb nicht zu unliebsamen Überraschungen und Bedienfehlern führt.

Um unliebsame Überraschungen zu vermeiden, ist es beispielsweise üblich, dass Buttons mit einem "X" ein Fenster schließen. Es würde die Benutzer zutiefst überraschen, wenn ein solcher Button plötzlich eine andere Funktion hätte.

Generell sollte versucht werden, die Erwartungshaltung der Benutzer zu treffen. Dazu hilft es, sich in die Benutzer hineinzuversetzen oder sie zu befragen und die Oberfläche dementsprechend zu designen.

5.3.4 Bewertung

Die folgenden **Vorteile** des Prinzips "Principle of Least Astonishment" in seiner ursprünglichen Form werden gesehen:

- **Treffen der Intuition der Benutzer und damit Vermeidung von Fehlern**

  Überraschung tritt auf, wo die Intuition der Benutzer nicht getroffen und damit enttäuscht wird. Überraschungen zu vermeiden, hilft dabei, intuitiv verständliche Oberflächen und Programme zu entwerfen, damit keine Bedienfehler provoziert werden.

Die folgenden **Nachteile** werden gesehen:

- **Unkenntnis der Erwartung der Benutzer**

  Problematisch bei der Umsetzung dieser Regel ist, dass ein Programmierer oft nicht weiß, was den Benutzer überrascht, da er systemnah denkt und sich in der Welt der Anwender nur bedingt auskennt.

- **Diversität der Benutzergruppen**

  Verschiedene Benutzergruppen können gänzlich verschiedene Erwartungshaltungen haben. Daher sollte bereits während der Entwicklung einer Bedienoberfläche das Gespräch mit den zukünftigen Nutzergruppen gesucht werden.

5.4 Zusammenfassung

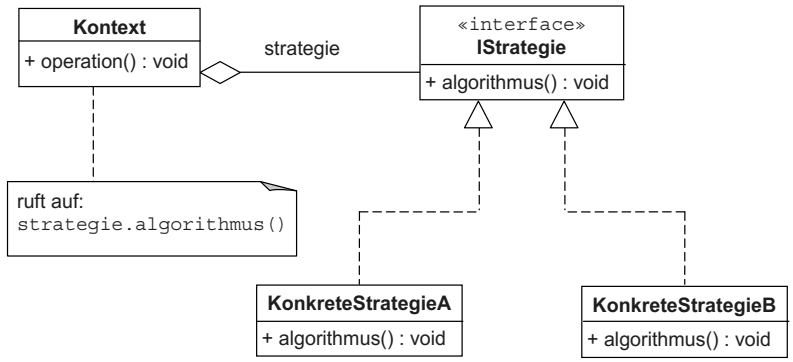
Das Konzept "Design by Contract" (siehe Kapitel **5.1**) fordert für die Beziehungen von Klassen eines Programms mit ihren Kundenklassen eine formale Übereinkunft zwischen den beteiligten Partnern. In dieser Übereinkunft wird durch Verträge präzise definiert, unter welchen Umständen die genannten Beziehungen korrekt ablaufen.

Nach dem liskovschen Substitutionsprinzip (siehe Kapitel **5.2**) kann in einem Programm mit polymorphen Objekten eine Referenz vom Typ der Basisklasse nicht nur auf ein Objekt der Basisklasse zeigen, sondern auch auf ein Objekt einer abgeleiteten Klasse, wenn die abgeleitete Klasse die Verträge der Basisklasse einhält. Das liskovsche Substitutionsprinzip im Verbund mit "Design by Contract" ist von höchster Bedeutung, damit Objekte abgeleiteter Klassen an die Stelle von Objekten einer Basisklasse in einem Programm treten können.

Das "Principle of Least Astonishment" (siehe Kapitel **5.3**) in seiner ursprünglichen Ausprägung will verhindern, dass die Bedienung der Oberflächen gegen gängige Konventionen oder die Absprachen mit den Nutzern verstößt, damit nicht auf Grund einer falschen Interpretation Fehlersituationen entstehen. Dieses Prinzip kann aber auch in einem erweiterten Sinne verwendet werden, dass Überraschungen generell zu vermeiden sind, um die korrekte Interpretation aller Teile des Systems wie beispielsweise des Programmcodes in eindeutiger Weise sicherzustellen.

Kapitel 6

Entwurfsprinzipien für die Stabilität bei Programm-erweiterungen



6 Entwurfsprinzipien für die Stabilität bei Programmerweiterungen

Alle Prinzipien, die für eine schwache Wechselwirkung zwischen verschiedenen Modulen und eine starke Kohäsion innerhalb eines Moduls sorgen, tragen zur **Stabilität** bei. Denn damit bleiben Störungen durch Fehler oder Fehleingaben im Betrieb des Systems oftmals lokal begrenzt.

Dementsprechend könnte unter Stabilität jedes Entwurfsprinzip genannt werden, welches für eine schwache Kopplung der Module und eine starke Kohäsion innerhalb eines Moduls sorgt.

In diesem Kapitel geht es jedoch nicht allgemein um die Stabilität, sondern um die **Stabilität bei Programmerweiterungen**. Hierbei sollen nach dem "Open-Closed Principle" (siehe Kapitel 6.1) stabile "Fertigteile" verwendet werden. Das "Open-Closed Principle" sorgt für Stabilität bei Erweiterungen, da getesteter Code einfach um zusätzliche Funktionalität erweitert, aber selbst nicht abgeändert wird.[79]

Erweiterungen können beispielsweise durch

- Vererbung (siehe Kapitel 6.2) oder
- die sogenannte "Objektkomposition" (siehe Kapitel 6.3)

erfolgen.

Eine andere Möglichkeit zur Erweiterung ist das Arbeiten in einer Plugin-Architektur, die im Rahmen dieses Buchs jedoch nicht behandelt wird. Zur Plug-in Architektur siehe [Gol14].

6.1 Open-Closed Principle

Nach dem "Open-Closed Principle" sollen Änderungen der Anforderungen nicht dadurch bewältigt werden, indem man den vorhandenen Code abändert. Stattdessen sollen vorhandene stabile Module erweitert werden. Auf diese Weise soll die Stabilität der Programme erhöht werden.

6.1.1 Historie

Dieses Prinzip stammt von Bertrand Meyer [Mey88]. Als eines der SOLID-Prinzipien (siehe Kapitel 10.3) von Robert C. Martin [Ma396] wurde es populär.

[79] Getesteter Code führt aber nur dann zur Stabilität, wenn er tatsächlich bereits mit den Daten der Anwendung getestet wurde.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8_6

6.1.2 Ziel

Das "Open-Closed Principle" (OCP) fordert, dass Module offen und geschlossen sein sollen.

Dieser scheinbare Widerspruch lässt sich folgendermaßen erklären:

- Ein Modul sollte **offen** sein in dem Sinne, dass es erweiterbar ist.
- Ein Modul ist **geschlossen**, wenn es zur Benutzung in unveränderter Form durch andere Module im Rahmen einer Bibliothek zur Verfügung steht. Änderungen dürfen nicht im Modul selbst durchgeführt werden! Dies bedeutet, dass ein Modul als stabile Einheit wiederverwendet werden soll.

Eine Klasse sollte nur in Form von Erweiterungen angepasst und abgeändert werden. Der Code und die Schnittstelle der ursprünglichen Klasse sollen unberührt bleiben!

6.1.3 Umsetzung

In Kapitel 6.1.3.1 und Kapitel 6.1.3.2 wird auf das "Open-Closed Principle" für Code und Spezifikationen eingegangen.

6.1.3.1 "Open-Closed Principle" für Code

Auf der Implementierungsebene bedeutet Geschlossenheit, dass der Code, der wiederverwendet werden soll, geschlossen für Veränderungen ist. Der bereits vorhandene Code eines Moduls kann kompiliert, gebunden und getestet sowie in eine Bibliothek in Form von Quellcode oder Binärcode aufgenommen und benutzt werden.

Durch die **Geschlossenheit des Quellcodes** bzw. des lauffähigen Codes gegenüber Veränderungen erreicht man eine höhere Stabilität der vorhandenen Programme einer Bibliothek.

Durch die **Offenheit des Quellcodes** bzw. des lauffähigen Codes gewinnt man eine Wiederverwendbarkeit des vorhandenen Quellcodes, der unverändert bleibt, als Teil von neuen Modulen.

6.1.3.2 "Open-Closed Principle" für Spezifikationen

Bei Spezifikationen ergibt sich keine prinzipielle Verschiedenheit zu Kapitel 6.1.3.1. Im Falle eines Entwurfs bzw. einer Spezifikation eines Moduls bedeutet die Geschlossenheit, dass der entsprechende Baustein offiziell abgenommen ist und selbst als wiederverwendbarer Baustein zur Verfügung steht.

Durch die **Geschlossenheit** von **Spezifikationen** gegenüber Veränderungen wird ihr Inhalt stabil. Durch die **Offenheit** von **Spezifikationen** gewinnt man ihre Wiederverwendbarkeit als Teil einer neuen Einheit.

6.1.4 Bewertung

Das "Open-Closed Principle" zählt zu den wichtigsten Grundprinzipien eines guten objektorientierten Designs.

Das "Open-Closed Principle" hat die folgenden **Vorteile**:

- **Erhöhung der Stabilität**

  Die Stabilität der Bauteile wird erhöht.

- **Erhöhung der Wiederverwendbarkeit**

  Das Erweitern von Fertigteilen nach dem "Open-Closed Principle" verbessert die Wiederverwendbarkeit.

Das "Open-Closed Principle" weist den folgenden **Nachteil** auf:

- **Höherer Abstraktionsgrad erforderlich**

  Der Entwurf erfordert einen höheren Abstraktionsgrad, um wiederverwendbare Teile zu identifizieren.

  Man muss sich im Klaren darüber sein, welche Teile eines Programmes offen für Veränderungen sein müssen [Mar12, pp. 108 - 109].

6.2 Erweiterungen durch Vererbung

Vererbung ist ein Beispiel dafür, dass der Quellcode einer Klasse zwar geschlossen gegenüber Veränderungen, aber dennoch offen für Erweiterungen ist. Der lauffähige Code einer Basisklasse wird nicht verändert. Trotzdem ist diese Basisklasse erweiterbar.

Bei Einhaltung des liskovschen Substitutionsprinzips können Referenzen auf Objekte der Basisklasse zur Laufzeit auf Objekte abgeleiteter Klassen zeigen. Dabei muss es diese Objekte zum Zeitpunkt der Erstellung des Quellcodes noch gar nicht geben.

6.2.1 Konzeptionelles Beispiel für das "Open-Closed Principle" mit Vererbung

Die folgende Abbildung zeigt ein Beispiel für eine Vererbung:

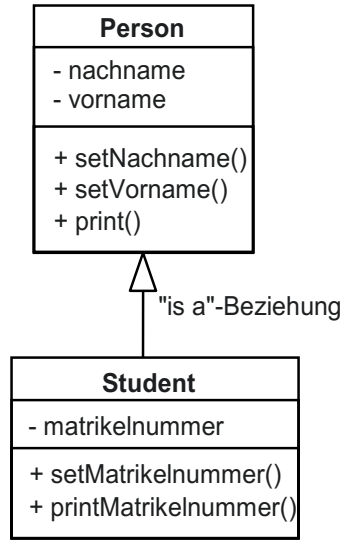


Abbildung 6-1 Ableitung der Klasse `Student` von der Klasse `Person`

Die Basisklasse `Person` hat hier zwei Attribute: `nachname` und `vorname` und die drei Methoden `setNachname()`, `setVorname()` und `print()`. Die set-Methoden werden benötigt, um die als `private` gekennzeichneten Attribute von außen setzen zu können. Die `print()`-Methode gibt den Vornamen gefolgt vom Nachnamen aus.

Die abgeleitete Klasse `Student` steht in einer "is a"-Beziehung zur ihrer Basisklasse. Gesprochen wird dies als "ein Student ist eine (engl. 'is a') Person", das heißt, dass ein Student auch eine Person ist. Bei der Vererbung erbt die abgeleitete Klasse `Student` die Struktur und das Verhalten der Basisklasse `Person`, also deren Attribute und Methoden. Die Klasse `Student` erweitert die Klasse `Person` um ein zusätzliches Attribut und um zwei zusätzliche Methoden. Davon merkt die Klasse `Person` jedoch nichts.

Um das liskovsche Substitutionsprinzip einzuhalten, darf der Vertrag der Basisklasse beim Ableiten nicht gebrochen werden. Der Aufrufer darf es nicht merken, wenn er zur Laufzeit beispielsweise ein Objekt der Klasse `Student` vor sich hat anstelle eines Objekts der Klasse `Person`.

6.2.2 Programmierbeispiel für das "Open-Closed Principle" mit Vererbung

Im folgenden Beispielprogramm symbolisiert die abstrakte Basisklasse `GrafischesElement` ein Framework, das nicht verändert, aber erweitert werden kann.

Geschlossen (engl. closed) ist der Quellcode der abstrakten Basisklasse `GrafischesElement`. Eine Erweiterung dieser Basisklasse erfolgt im folgenden Beispiel in den abgeleiteten Klassen `Kreis` und `Rechteck`, die anschließend von einem grafischen Editor verwendet werden sollen.

Die folgende Abbildung zeigt diese Situation:

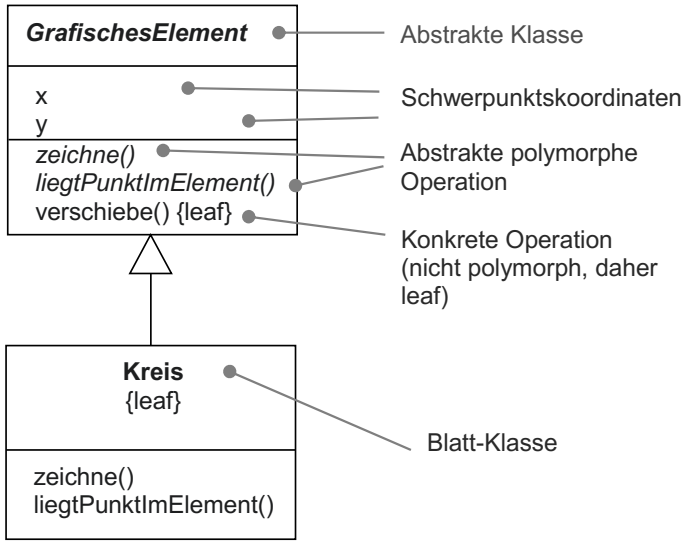


Abbildung 6-2 Abstrakte Klasse `GrafischesElement` symbolisch als Framework

Jedes grafische Element weist bestimmte Eigenschaften und Methoden auf, die in ihrer Funktionsweise unabhängig von der speziellen Ausprägung des Elementes sind, zum Beispiel, ob es sich um einen Kreis oder ein Rechteck handelt. Dazu gehört die Schwerpunktskoordinate eines Elements und die zugehörige Verschiebungsmethode für den Schwerpunkt.

Im Gegensatz dazu gibt es Methoden, die von den abgeleiteten Klassen selbst implementiert werden müssen – beispielsweise die Darstellungsroutine eines Elements.

Die für die abgeleiteten Klassen invarianten Anteile werden in der Klasse `GrafischesElement` abgebildet. Diese Basisklasse enthält die von allen Elementen in gleicher Weise genutzten Methoden und Attribute. Sie gibt ferner Methodenköpfe für die Implementierung in den abgeleiteten Klassen vor.

Aufgrund der Abstraktheit einiger Methodenköpfe ist diese Basisklasse abstrakt. Die von allen Elementen genutzten Methoden sind vor Veränderungen – beispielsweise durch Überschreiben – zu schützen. Ein Beispiel hierfür ist in Abbildung 6-2 die Methode `verschiebe()`. Solche Methoden werden in UML `leaf`-Methoden genannt und werden in Java durch das Schlüsselwort `final` deklariert.

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

Die abstrakte Klasse `GrafischesElement` könnte wie folgt aussehen:

```
// Datei: GrafischesElement.java

import java.awt.Graphics;

public abstract class GrafischesElement
{
   // Schwerpunktskoordinaten
   private int x, y;
   public GrafischesElement (int x, int y)
   {
       this.x = x;
       this.y = y;
   }

   // final macht eine Methode zu einer leaf-Methode
   final public void verschiebeSchwerpunkt (int x, int y)
   {
      this.x += x;
      this.y += y;
   }

   final public int getX()
   {
      return x;
   }

   final public int getY()
   {
      return y;
   }

   // Graphics erlaubt das Auftragen von Grafiken auf Komponenten
   public abstract void zeichne (Graphics g);

   public abstract boolean liegtPunktImElement (int x, int y);
}
```

Die Anwendung soll Kreise und Rechtecke als grafische Elemente enthalten, welche die abstrakte Basisklasse `GrafischesElement` nicht abändern, aber erweitern. Zum Zeichnen eines Rechtecks wird in der folgenden Klasse `Rechteck` die in AWT[80] vorhandene Klasse `java.awt.Rectangle` benutzt:

```
// Datei: Rechteck.java

import java.awt.Graphics;
import java.awt.Rectangle;

public class Rechteck extends GrafischesElement
{
   Rectangle rechteck;
```

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

```
   public Rechteck (int x, int y, int laenge, int hoehe)
   {
      super (x, y);

      // Eckpunkte für das Anlegen des Rechtecks ermitteln
      int eckpunktX = this.getX() - (int) (0.5 * laenge);
      int eckpunktY = this.getY() - (int) (0.5 * hoehe);

      rechteck = new Rectangle (eckpunktX, eckpunktY, laenge, hoehe);
   }

   public void zeichne (Graphics g)
   {
       // Rechteck auf Graphics-Objekt zeichnen
       g.fillRect (
                   rechteck.x,
                   rechteck.y,
                   rechteck.width,
                   rechteck.height
       );
   }

   @Override
   public boolean liegtPunktImElement (int x, int y)
   {
      return rechteck.contains (x, y);
   }
}
```

Als weiteres grafisches Element wird eine Klasse `Kreis` eingeführt:

```
// Datei: Kreis.java

import java.awt.Graphics;
import java.awt.geom.Ellipse2D;

public class Kreis extends GrafischesElement
{
   private Ellipse2D kreis;

   public Kreis (int x, int y, int radius)
   {
      super (x, y);

      int xEckpunkt = x - radius;
      int yEckpunkt = y - radius;

      kreis = new Ellipse2D.Float
      (
            xEckpunkt,
            yEckpunkt,
            radius,
            radius
      );
   }
```

```
    @Override
    public void zeichne (Graphics g)
    {
       g.fillOval (
             (int) kreis.getX(),
             (int) kreis.getY(),
             (int) kreis.getWidth(),
             (int) kreis.getWidth()
       );
    }

    @Override
    public boolean liegtPunktImElement (int x, int y)
    {
       return kreis.contains (x, y);
    }
}
```

Die Klassen `Kreis` und `Rechteck` werden anschließend von einem grafischen Editor verwendet. Dieser Editor kann Kreise, Rechtecke oder alle von der Basisklasse `GrafischesElement` abgeleiteten Formen erstellen und verschieben. Eine einfache Editor-Implementierung könnte dann wie folgt aussehen:

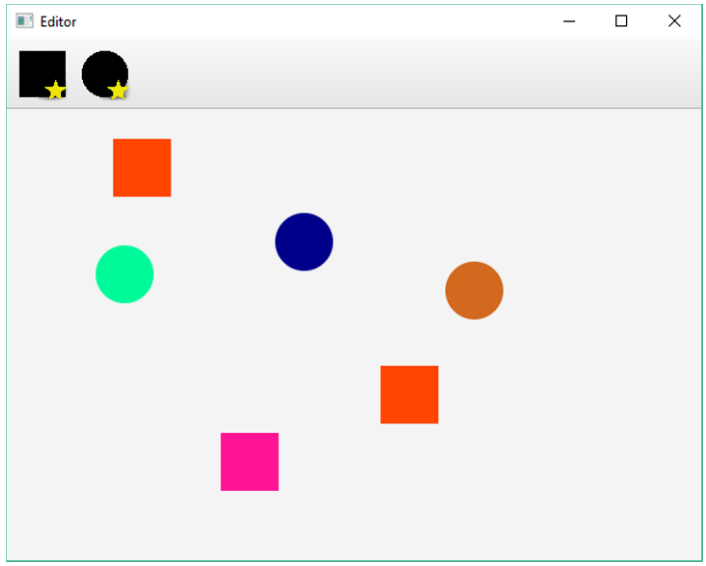


Abbildung 6-3 Screenshot der grafischen Oberfläche des Editors

6.3 Ziehe Objektkomposition der Klassenvererbung vor

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

6.3.1 Historie

Das Prinzip "Ziehe Objektkomposition der Klassenvererbung vor" wird im berühmten Entwurfsmuster-Buch vom Gamma et al. [Gam94] besonders hervorgehoben. Auf Englisch heißt dieses Prinzip "Favour ‘object composition’ over ‘class inheritance‘." Daher wird der Name dieses Prinzips auch abgekürzt zu FCOI.

Zusammen mit dem Prinzip "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" ist FCOI eine zentrale Grundlage für die berühmten Entwurfsmuster von Gamma et al. Es dient zur Reduktion von Abhängigkeiten.

6.3.2 Ziel

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

Dieses Prinzip ist eine Konkurrenztechnik zur Vererbung bei der Erzeugung von Programmen mit polymorphen Objekten.

Bei einer Vererbung kann die Basisklasse ihre Elemente für eine abgeleitete Klasse sichtbar machen. Die abgeleitete Klasse wird dadurch von den geerbten Elementen abhängig (**White-Box-Verhalten**), wenn diese in ihr sichtbar sind. Anders formuliert bedeutet das, dass in diesem Falle das Prinzip der Kapselung gebrochen wird. Diese Gefahr besteht bei FCOI nicht. Der Inhalt der beteiligten Objekte ist bei FCOI nicht sichtbar (**Black-Box-Verhalten**). Was bleibt, ist die Abhängigkeit von der Abstraktion.

6.3.3 Diskussion des Namens des Prinzips

Kommentar des Autors:

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

So schreiben Gamma et al. [Gam09, p. 27]: *"Objektkomposition wird dynamisch zur Laufzeit festgelegt, indem die Objekte Referenzen auf andere Objekte erhalten."*

Aggregation bedeutet nach UML, dass die beteiligten Objekte unabhängig voneinander weiterbestehen können. Konträr hierzu funktioniert nach UML eine **Komposition**. Hierbei leben beide Objekte stets gleich lang (siehe Kapitel 2.1.1).

Man braucht nach UML bei "Ziehe Objektkomposition der Klassenvererbung vor" eine Aggregation und keine Komposition, um das liskovsche Substitutionsprinzip anwenden zu können.

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

Kommentar des Autors:

Man liest oft, dass das Prinzip "Ziehe Objektkomposition der Klassenvererbung vor" eine "is a"-Beziehung durch eine "has a"-Beziehung ersetzt. Dies wäre zu hundert Prozent eine korrekte Formulierung im Falle einer Komposition im Sinne von UML. Im Falle der vorliegenden Aggregation – siehe auch die Beschreibung von Gamma et al. [Gam09, p. 27] – können aber theoretisch auch mehrere "Groß"-Objekte auf dasselbe Objekt verweisen, so dass man im strengen Sinne nicht von "has a" sprechen kann. Überdies sind bei einer Komposition das besitzende Objekt und das eingeschlossene Objekt fest verknüpft, so dass das liskovsche Substitutionsprinzip nicht angewandt werden kann.

6.3.4 Bekannte Beispiele für den Einsatz einer Aggregation

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

6.3.4.1 Strategiemuster

Das Verwenden einer Schnittstelle als aggregierte Abstraktion ist beispielsweise vom Strategiemuster bekannt.

Hier das entsprechende Klassendiagramm:

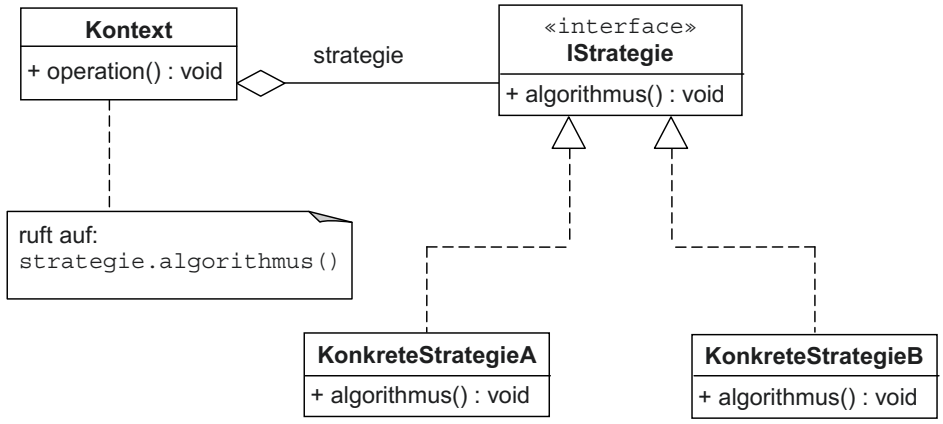


Abbildung 6-4 Klassendiagramm des Strategiemusters

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

Enthält der Kontext eine Referenz auf eine Abstraktion, dann kann das liskovsche Substitutionsprinzip verwendet werden.

In Abbildung 6-4 gibt der Kontext die Abstraktion (die Schnittstelle `IStrategie`) vor und nutzt sie mittels einer Aggregation. Bei Einhaltung des liskovschen Substitutionsprinzips kann zur Laufzeit an die Stelle der Schnittstelle in polymorpher Weise eine konkrete Strategie treten. Dabei ist der Vertrag der Schnittstelle einzuhalten.

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

6.3.4.2 Dekorierer

Ein weiteres bekanntes Beispiel ist das Dekorierermuster. Ein Dekorierer soll eine beliebige Komponente "dekorieren" können. Die folgende Abbildung zeigt eine der möglichen Formulierungen des bekannten Klassendiagramms des Dekorierermusters:

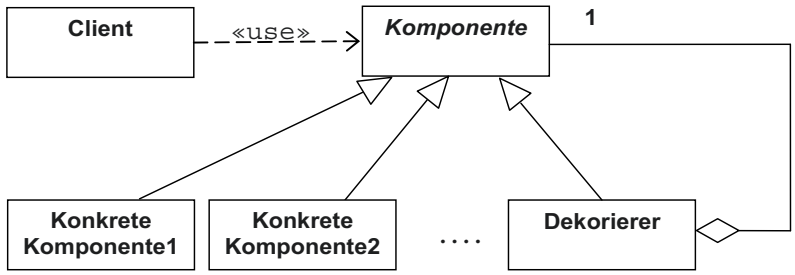


Abbildung 6-5 Klassendiagramm des Dekorierermusters

Der Client benutzt in Abbildung 6-5 eine Komponente in Form einer abstrakten Basisklasse als Abstraktion. An die Stelle der referenzierten Komponente kann nach Liskov auch eine konkrete Komponente bzw. ein Dekorierer treten. Der Dekorierer verwendet den geerbten Anteil nicht! Er verwendet den aggregierten Anteil. Damit kann auch eine konkrete Komponente dekoriert werden, da jederzeit bei Einhaltung der Verträge eine konkrete Komponente an die Stelle einer aggregierten Komponente treten kann.

6.3.5 Vergleich Vererbung und Objektkomposition

Für die Erzeugung von Polymorphie kennt die Objektorientierung zwei generelle Mechanismen:

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

Objektkomposition ist eine Alternative zur Vererbung und ist der Vererbung nach Gamma et al. [Gam09, p. 27] vorzuziehen. Deshalb heißt dieses Prinzip auch *"Ziehe Objektkomposition der Klassenvererbung vor".*

Gamma et al. [Gam09, p. 27] sagen:

"Entwürfe können durch den verstärkten Einsatz von Objektkomposition häufig einfacher und leichter wiederverwendbar gemacht werden."

Eine Klasse referenziert bei der Objektkomposition eine Abstraktion, die implementiert werden muss. Da bei einer Objektkomposition keine internen Details der Objekte der referenzierenden Klasse sichtbar sind[81] und die Objekte als Black-Boxes[82] erscheinen, wird dieser Stil der Wiederverwendung auch **Black-Box-Wiederverwendung** genannt [Gam09, p. 26].

Wiederverwendung durch Unterklassenbildung wird oft auch als **White-Box-Wiederverwendung** bezeichnet. Der Begriff "White-Box" bezieht sich auf die Sichtbarkeit, nämlich wenn infolge von **Vererbung** die Daten bzw. Methoden einer Oberklasse für eine Unterklasse (abgeleitete Klasse) sichtbar sind [Gam09, p. 26]. Bei der **Vererbung** wird zur Kompilierzeit festgelegt, dass ein Objekt einer abgeleiteten Klasse quasi ein Objekt der Basisklasse – nämlich den geerbten Anteil – enthält. Dies kann ebenfalls als eine Art der Komposition betrachtet werden, legt bei Sichtbarkeit jedoch die Implementierung von Elementen der Basisklasse offen. Die Beziehung zwischen den beiden Klassen ist statisch und kann zur Laufzeit nicht mehr geändert werden. Da bei der Vererbung die abgeleitete Klasse die Daten und das Verhalten einer Basisklasse erbt, besteht durch die Ableitung eine starke Abhängigkeit einer abgeleiteten Klasse von ihrer zugehörigen Basisklasse, wenn die Elemente der Basisklasse in der abgeleiteten Klasse sichtbar sind. Vererbung als Technik der Wiederverwendung wird nach Gamma et al. [Gam09, p. 27] in der Praxis überstrapaziert.

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

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

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

---

[81] Nur die Methodenköpfe sind in Abstraktionen sichtbar.

[82] Das bedeutet, dass nur die Schnittstellenmethoden sichtbar sind.

6.3.6 Bewertung

Vorteile von "Ziehe Objektkomposition der Klassenvererbung vor" sind:

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

  Es wird nicht wie bei der Vererbung die Kapselung gebrochen. Bei der Vererbung sind die Struktur bzw. das Verhalten einer Basisklasse auch in der abgeleiteten Klasse oft direkt sichtbar, nämlich dann, wenn die Daten bzw. Methoden der Basisklasse nicht mit dem Zugriffsmodifikator privat geschützt sind und nicht überschrieben werden. In diesem Fall schlägt jede Änderung der Implementierung einer Basisklasse auf die abgeleitete Klasse direkt durch. Damit hat eine abgeleitete Klasse eine starke Abhängigkeit von ihrer Basisklasse. Werden hingegen Objekte ausschließlich über die Implementierung aggregierter Abstraktionen verwendet wie bei "Ziehe Objektkomposition der Klassenvererbung vor", so wird die Kapselung der ursprünglichen Implementierung nicht aufgebrochen. Bei der Objektkomposition besteht nur eine schwache Abhängigkeit der Klasse eines Objekts zu dem Vertrag der referenzierten Abstraktion.

- **Keine komplexen Klassenhierarchien**

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

- **Simulation von Mehrfachvererbung**

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

- **Leichteres Testen**

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

- **Flexibilität**

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

- **Einfacher Gebrauch von Dependency Injection**

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

Nachteile von "Ziehe Objektkomposition der Klassenvererbung vor" sind:

- **Prinzip oft unbekannt**

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

- **Potenzielle Zunahme der Anzahl der Schnittstellen**

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

- **Keine direkte Wiederverwendung von Code**

  Vererbung führt zu einer direkten Wiederverwendung von Code, da geerbte Methoden direkt Bestandteil der abgeleiteten Klasse werden – falls sie dort sichtbar sind und nicht überschrieben werden. Bei der Objektkomposition ist hingegen der vorhandene Code in einer Black-Box. Die aggregierte Abstraktion wird zur Laufzeit durch ein Objekt, dessen Klasse die Abstraktion implementiert, ersetzt.

Vererbung ist sinnvoll, wenn:

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

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

6.4 Zusammenfassung

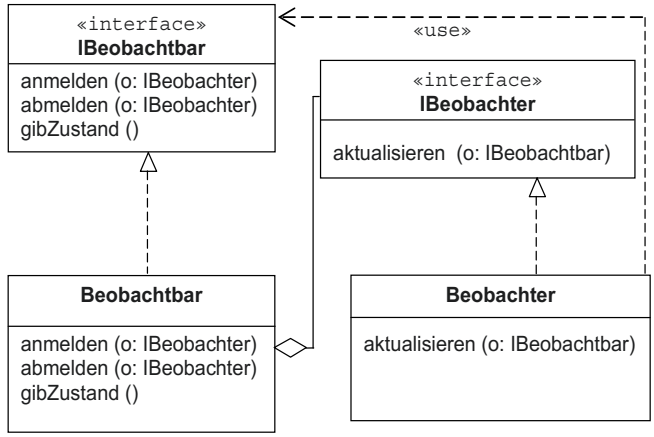
Das "Open-Closed Principle" (siehe Kapitel **6.1**) fordert, dass Module entworfen werden, die sich niemals ändern. Änderungen sollen dadurch realisiert werden, indem der Code der bestehenden Module nicht abgeändert, sondern nur erweitert wird.

Die Vererbung (siehe Kapitel **6.2**) ist für "is a"-Beziehungen gedacht und nicht für eine reine Code-Wiederverwendung.

Bei Erweiterungen von stabilen Modulen ist das Prinzip "Ziehe Objektkomposition der Klassenvererbung vor" (siehe Kapitel **6.3**) von höchster Bedeutung. Es verringert die durch eine Vererbung entstehende starke Kopplung zwischen Basisklasse und abgeleiteter Klasse, indem es eine sogenannte "Objektkomposition" verwendet.

Kapitel 7

Das Konzept "Inversion of Control"



7 Das Konzept "Inversion of Control"

"Inversion of Control"[83] dient – wie der Name schon sagt – der Umkehr der Kontrolle. Bei diesem Konzept wird die Kontrolle des Programmablaufs zwischen zwei zusammenwirkenden Programmteilen umgedreht, beispielsweise dann, wenn ein selbst programmiertes Modul die Steuerung des Kontrollflusses an ein Framework als wiederverwendbares Modul abgeben soll.

Die gezeigten Beispiele sind objektorientiert. Das Konzept "Inversion of Control" kann jedoch auch bei anderen Programmierparadigmen angewandt werden.

Zu beachten ist, dass durch das Konzept "Inversion of Control" keine Abhängigkeiten umgedreht oder gar beseitigt werden.

Die Umkehr der Kontrolle darf nicht mit der Umkehr der Abhängigkeiten verwechselt werden.

Das Konzept "Inversion of Control" wird auch als **"Hollywood-Prinzip"** bezeichnet, nach dem gilt: "Don't call us, we'll call you".

UI-Frameworks wie WPF oder JavaFX verwenden diese Technik, um auf Benutzerinteraktionen zu reagieren.

Kapitel 7.1 betrachtet die Historie der "Inversion of Control".

Kapitel 7.2 listet die Ziele von "Inversion of Control" auf.

Kapitel 7.3 stellt das Beobachtermuster als eine Technik zur Implementierung von "Inversion of Control" vor. Wird beispielsweise nach diesem Muster ereignisorientiert programmiert, so muss ein Beobachter nicht mehr pollen, um Informationen über Daten- bzw. Zustandsänderungen des beobachteten Objekts zu erhalten. Hingegen wird er vom beobachteten Objekt selbst informiert. In beiden Fällen – ob gepollt oder ereignisorientiert gearbeitet wird – werden dessen Daten vom Beobachter benötigt.

Auf Callback-Schnittstellen von Frameworks wird in Kapitel 7.4 eingegangen. Hierbei gibt ein Framework eine Abstraktion vor, welche ein selbst programmiertes Modul implementieren muss. Damit kann ein Framework die hinter der Callback-Schnittstelle stehende Funktion aufrufen. Anstatt, dass ein selbst geschriebenes Modul Routinen eines Framework verwendet und wie früher die Steuerung des Programms selbst durchführt, wird bei Einsatz von "Inversion of Control" das selbst geschriebene Modul zur Laufzeit vom Framework beim Eintreffen bestimmter Ereignisse über die implementierte Callback-Schnittstelle aufgerufen. Damit das Framework das selbst geschriebene Modul kennt, muss das selbst geschriebene Modul dem Framework bekannt gemacht werden. Dies kann beispielsweise nach dem Konzept der "Dependency Injection" oder durch ein formales Anmeldeverfahren erfolgen.

[83] abgekürzt als IoC

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8_7

Kapitel 7.5 diskutiert die Vor- und Nachteile von "Inversion of Control".

Anschließend wird in Kapitel 7.6 ein ausführliches Programmbeispiel gegeben.

7.1 Historie

Martin Fowler [Fow05] führt den Begriff "Inversion of Control" zurück auf das Papier "Designing Reusable Classes" von Johnson und Foote im Journal of Object-Oriented Programming im Jahre 1988 (siehe [Joh88]).

Der synonyme Begriff **"Hollywood Principle"** scheint nach Martin Fowler [Fow05] aus einer Veröffentlichung von Richard E. Sweet über die Programmierumgebung MESA im Jahre 1985 zu stammen [Swe85].

7.2 Ziele

Hinter dem Entwurfsprinzip "Inversion of Control" stehen die folgenden Ziele:

- **Vertauschen der Kontrolle**

  Der Kontrollfluss wird nicht mehr vom selbst geschriebenen Programm vorgegeben, sondern oftmals durch die Ereignisse eines Framework (**ereignisorientierte Programmierung**). Eine ereignisorientierte Programmierung beruht auf der Umkehr der Kontrolle.

- **Modularität und Wartbarkeit**

  Das Programm soll modularer und wartbarer werden, indem die Programmteile des Auslösens eines Ereignisses und seine Verarbeitung getrennt werden.

- **Erhöhung der Testbarkeit**

  Durch die Entkoppelung von "Was wird getan" und "Wann wird es getan" wird die Testbarkeit gesteigert.

Kommentar des Autors:

Durch die Entkoppelung von "Was wird getan" und "Wann wird es getan" wird der Code nicht mehr rein sequenziell von oben nach unten, sondern ereignisorientiert abgearbeitet.

Um den Programmablauf – besonders in Verbindung mit Frameworks – nachzuvollziehen, müssen die Auslösung einer Aktion und das Doing der Aktion selbst getrennt betrachtet werden.

7.3 Ereignisorientierte Programmierung anstelle von Pollen

Zuerst soll hier ein **pollender**[84] **Kunde** betrachtet werden, der als **Listener** Werte von einem zu pollenden Programm abfragt, wie die folgende Abbildung zeigt:

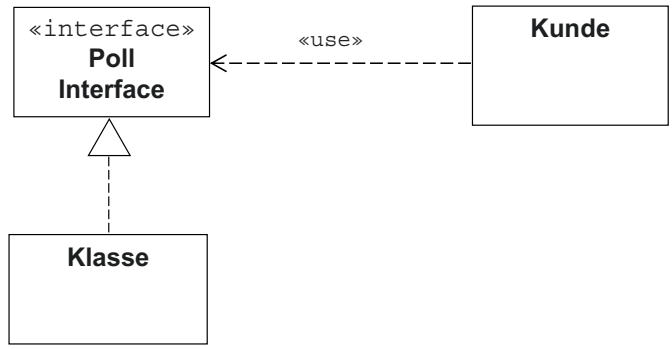


Abbildung 7-1 Pollender Kunde

Pollen verbraucht durch das zyklische Abfragen viel Rechenzeit. Der Gepollte muss darüber hinaus einem pollenden Kunden eine Schnittstelle zum Abrufen von Daten bzw. zum Abfragen einer Zustandsänderung anbieten.

Im Folgenden wird das **Beobachtermuster** (engl. **observer pattern**) als potenzielle Lösung des Polling-Problems vorgestellt. Das Beobachtermuster setzt "Inversion of Control" um.

Damit das beobachtbare Objekt einen Beobachter ereignisorientiert aufrufen kann und dabei nicht vom Beobachter abhängig wird, muss nach dem Beobachter-Muster jeder Beobachter eine **Callback-Schnittstelle**, welche das zu beobachtende Objekt vorgibt, implementieren.

Dies führt zu der folgenden Architektur:

Abbildung 7-2 Architektur mit einer Callback-Schnittstelle

Die Architektur zur Erreichung einer "Inversion of Control" ist dabei von der Konstruktion her ähnlich zum "Dependency Inversion Principle" in Hierarchien. In beiden Fällen wird eine Abstraktion aggregiert, um Nutzen aus dem liskovschen Substitutionsprinzip zu ziehen. Allerdings muss bei "Inversion of Control" die Abstraktion im Gegensatz zum "Dependency Inversion Principle" vom Beobachtbaren selbst vorge-

[84] Polling bezeichnet in der Informatik die Methode, ein Ereignis oder eine Werteänderung mittels zyklischem Abfragen der Quelle zu ermitteln.

geben werden. Beim "Dependency Inversion Principle" hingegen gibt das Modul der tieferen Ebene die Abstraktion vor (siehe Kapitel 4.5).

Die Klassen `Beobachtbar` und `Beobachter` des Beobachtermusters stehen jedoch nicht in einer hierarchischen Beziehung zueinander, sondern befinden sich auf derselben Ebene. Berücksichtigt man noch, dass ein Beobachter auf den Beobachtbaren über die von diesem vorgegebene Schnittstelle `IBeobachtbar` zugreift, beispielsweise um sich als Beobachter an- und abzumelden, so kommt man zum Klassendiagramm des Beobachtermusters. Dies ist im Folgenden dargestellt:

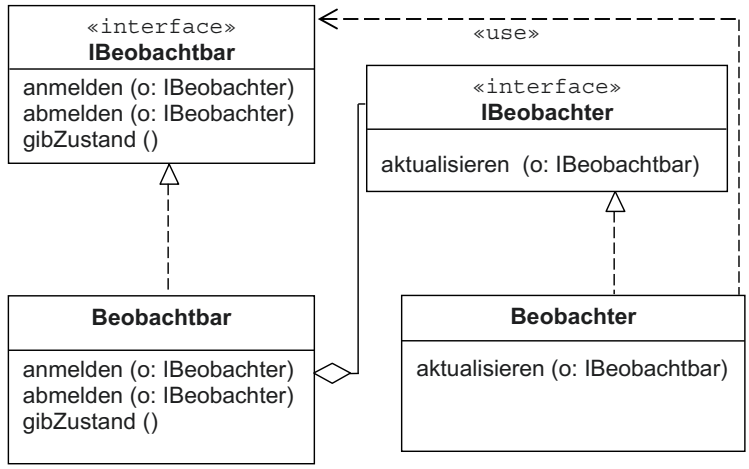


Abbildung 7-3 Klassendiagramm des Beobachtermusters

Durch die Vorgabe der Callback-Schnittstelle `IBeobachter` mit dem Methodenkopf `aktualisieren (o: IBeobachtbar)` durch den Beobachtbaren ist der Beobachtbare vom Beobachter unabhängig und ist damit mehrfach verwendbar.

Der Beobachtbare ruft seine Beobachter über die von ihm selbst vorgegebene Callback-Schnittstelle auf. Dem Beobachtbaren bleibt die Implementierung der Callback-Schnittstelle unbekannt, da er ja selber nur deren Abstraktion vorgibt.

Beim Beobachter-Muster gibt es zwei Varianten:

- das Push-Prinzip und
- das Pull-Prinzip.

Verwendet man das **Pull-Prinzip**, das in Abbildung 7-3 dargestellt ist, so informiert der Beobachtbare den Beobachter über die Änderung seiner Daten bzw. seines Zustands mit Hilfe der Methode `aktualisieren()`. Daraufhin holt sich der Beobachter durch Aufruf der Methode `gibZustand()` beim Beobachtbaren die geänderten Daten ab. Beim **Push-Prinzip** sendet der Beobachtbare gleich die geänderten Daten an den Beobachter mit. Die Methode `aktualisieren()` muss dann entsprechend parametrisiert werden und die Methode `gibZustand()` kann entfallen.

7.4 "Inversion of Control" bei Frameworks

Bei Anwendung des Konzepts **"Inversion of Control"** auf Frameworks ist ein selbst geschriebenes Modul von der Callback-Schnittstelle, welche das wiederverwendbare Framework vorgibt, abhängig.

Diese Schnittstelle muss von einem selbstgeschriebenen Modul implementiert werden. Das Modul kann sich daraufhin zur Laufzeit beim Framework anmelden und von diesem über die vom Framework spezifizierte, jedoch vom Modul implementierte Callback-Schnittstelle aufgerufen werden.

Statt dass das selbst geschriebene Modul den Kontrollfluss steuert und lediglich Funktionen einer Bibliothek aufruft, wird die Steuerung der Ausführung bestimmter Programmteile des selbst geschriebenen Moduls an das Framework abgegeben, welches die Kontrolle übernimmt. Bei Vorliegen eines Ereignisses wird die entsprechende Funktion des Moduls über die vorgegebene Callback-Schnittstelle vom Framework aufgerufen.

"Inversion of Control" bedeutet in der Praxis oft, dass ein selbst geschriebenes Modul die Steuerung des Kontrollflusses an ein wiederverwendbares Modul abgibt. Dabei werden der wiederverwendbare und der problemspezifische Code getrennt voneinander entwickelt. Beide Programme arbeiten aber trotzdem in derselben Anwendung zusammen.

7.5 Bewertung

"Inversion of Control" ist ein Konzept, um eine Umkehr der Kontrolle zu erreichen, falls dies erwünscht ist.

Die folgenden **Vorteile** werden gesehen:

- **Wandelbarkeit durch Trennung des Auslösens eines Ereignisses und der Reaktion auf dieses Ereignis**

  Der Kontrollfluss eines Programms legt die Abarbeitungsreihenfolge von Anweisungen fest. Er bestimmt, "was" "wann" getan wird. In vielen Programmen werden diese beiden Aspekte an einer einzigen Stelle im Programmcode definiert. Beispielsweise erledigt eine Funktion eine bestimmte Aufgabe und ruft dann je nach Erfolg dieser Aufgabe selbst eine definierte Folge-Funktion auf. Hierbei werden die beiden Aspekte des "Wann" – also der Zeitpunkt des Auftretens des Ereignisses – und der Ausprägung der Reaktion – je nach Ergebnis wird eine definierte Folge-Funktion ausgeführt – an einer einzigen Stelle festgelegt.

  Bei "Inversion of Control" liegt der Aufruf der Reaktion im Beobachtbaren, die Festlegung der Reaktion erfolgt jedoch im Beobachter.

  Durch diese Trennung steigert sich die Wandelbarkeit. So muss die aufrufende Funktion bei einer Änderung der Reaktion auf das Ereignis selbst nicht angepasst

und neu getestet werden. Sie bleibt somit stabil und erfüllt somit auch das "Open-Closed Principle".

- **Wiederverwendbarkeit durch Trennung des Auslösens eines Ereignisses und der Reaktion auf dieses Ereignis**

  Zusätzlich steigt auch die Wiederverwendbarkeit, da sowohl die Aufgabe als auch die Reaktion voneinander unabhängig sind. Bei Frameworks, welche das Prinzip "Inversion of Control" umsetzen, definiert das Framework, "wann" ein Ereignis ausgelöst wird. Der Programmierer definiert, "was" getan wird, d. h. wie die Reaktion aussieht.

Doch das Konzept "Inversion of Control" hat auch einen **Nachteil**:

- **Unterbrochener Lesefluss**

  Bei der Verwendung von "Inversion of Control" wird der von der sequenziellen Programmierung gewohnte Lesefluss gestört. Um den Programmablauf zu verstehen, muss ein Entwickler an zwei verschiedenen Stellen der Programme nachschauen.

Im Folgenden wird für "Inversion of Control" ein ausführliches Programmierbeispiel gegeben.

7.6 Beispielprogramm

Das folgende Programm zeigt ein einfach gehaltenes Beispiel für "Inversion of Control" in Anlehnung an nachrichtenbasierte Frameworks wie Apache ActiveMQ[85] oder RabbitMQ[86]. Diese Frameworks verwenden "Inversion of Control", um – vergleichbar mit dem Beobachtermuster – Nachrichtenempfänger ereignisorientiert über neue Nachrichten zu informieren, sodass diese nicht pollend das Vorliegen neuer Nachrichten überprüfen müssen. Als zentrales Element stellen nachrichtenbasierte Frameworks einen Vermittler bereit. An diesem Vermittler können sich Nachrichtenempfänger für bestimmte Themen registrieren beziehungsweise registriert werden. Zudem können Nachrichtensender unkompliziert Nachrichten für bestimmte Themen veröffentlichen, indem sie diese an den Vermittler schicken. Der Vermittler leitet sie dann an die angemeldeten Nachrichtenempfänger weiter. Auf diese Weise wird erreicht, dass Nachrichtenempfänger und Nachrichtensender nur lose über den Vermittler gekoppelt sind.

Um das Prinzip "Inversion of Control" in Zusammenhang mit dem Vermittler darzulegen, wird das Beobachtermuster aufgegriffen (vergleiche Abbildung 7-4). Die Klasse des Vermittlers gibt dabei eine Schnittstelle `IAbonnent` vor, welche Abonnenten implementieren müssen. Solche abonnierenden Objekte werden im folgenden Beispiel repräsentiert durch die Klasse `KonsolenAbonnent`. Ein Objekt vom Typ `KonsolenAbonnent` kann sich durch die Schnittstelle `IVermittler` beim Vermittler für bestimmte Themen anmelden bzw. angemeldet werden. Nachfolgend das Klassendiagramm:

[85] https://activemq.apache.org
[86] https://www.rabbitmq.com

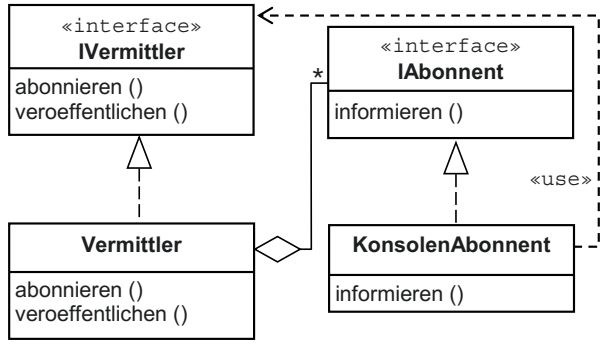


Abbildung 7-4 Klassendiagramm für den Vermittler nach dem Beobachtermuster

Durch die Schnittstelle `IAbonnent` stellt der Vermittler eine Callback-Schnittstelle zur Verfügung. Dadurch wird es möglich, interessierte Abonnenten über das Vorliegen von Nachrichten zu einem bestimmten Thema zu informieren. Die Schnittstelle ist nachfolgend gegeben:

```
// Datei: IAbonnent.java

public interface IAbonnent
{
   void informieren (String thema, String nachricht);
}
```

Die einzige konkrete Implementierung der Schnittstelle `IAbonnent` ist in diesem Beispiel die Klasse `KonsolenAbonnent`. Bei einem gemäß "Inversion of Control" durch den Vermittler initiierten Aufruf der Methode `informieren()` gibt dieser Konsolenabonnent die dabei übermittelte Nachricht einfach nur auf die Konsole aus. Es könnten aber auch weitere Implementierungen von `IAbonnent` existieren wie zum Beispiel eine Klasse `DateiAbonnent`, welche die empfangene Nachricht in eine Datei schreibt. Nachstehend die Klasse `KonsolenAbonnent`:

```
// Datei: KonsolenAbonnent.java

public class KonsolenAbonnent implements IAbonnent
{
   @Override
   public void informieren (String thema, String nachricht)
   {
      System.out.println ("Neue Nachricht erhalten.\r\n" +
                          "  Thema : " + thema + "\r\n" +
                          "  Inhalt: " + nachricht);
   }
}
```

Das Herzstück dieses Beispiels ist der Vermittler. Er stellt über die Schnittstelle `IVermittler` zwei Methoden zur Verfügung, um Abonnenten zu registrieren und um Nachrichten zu versenden bzw. zu veröffentlichen. Nachrichten, die über die Methode `veroeffentlichen()` an einen Vermittler geschickt werden, leitet dieser dann an

die interessierten Abonnenten weiter (vergleichbar mit dem Push-Prinzip des Beobachtermusters). Das Abmelden von Abonnenten wird zur Vereinfachung des Beispiels nicht betrachtet. Die Schnittstelle `IVermittler` ist nachfolgend dargestellt:

```
// Datei: IVermittler.java

public interface IVermittler
{
   // Hiermit koennen Abonnenten Interesse an einem Thema anmelden.
   void abonnieren (String thema, IAbonnent abonnent);

   // Hiermit werden alle Abonnenten mit Interesse an dem
   // konkreten Thema ueber eine neue Nachricht informiert.
   void veroeffentlichen (String thema, String nachricht);
}
```

Die konkrete Implementierung hinter der Schnittstelle `IVermittler` kann je nach Einsatzzweck bzw. verwendetem Framework unterschiedlich ausfallen. Eine einfache Implementierung eines Vermittlers kann beispielsweise wie folgt aussehen:

```
// Datei: Vermittler.java

import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

public class Vermittler implements IVermittler
{
   Map<String, List<IAbonnent>> abonnenten = new HashMap<>();

   @Override
   public void veroeffentlichen (String thema, String nachricht)
   {
      if (abonnenten.containsKey (thema))
      {
         for (IAbonnent abonnent : abonnenten.get(thema))
         {
            abonnent.informieren (thema, nachricht);
         }
      }
   }

   @Override
   public void abonnieren (String thema, IAbonnent abonnent)
   {
      if (!abonnenten.containsKey (thema))
      {
         abonnenten.put (thema, new ArrayList<IAbonnent>());
      }
      abonnenten.get (thema).add (abonnent);
    }
}
```

Wie Nachrichten veröffentlicht und anschließend über die Callback-Schnittstelle übermittelt werden können, soll durch das nachfolgende Hauptprogramm aufgezeigt werden. Dieses Hauptprogramm besitzt allerdings aus Vereinfachungsgründen drei Verantwortlichkeiten. Die folgenden Anregungen zeigen, wie diese Verantwortlichkeiten getrennt werden könnten:

- **Instanziierung des Vermittlers**

  Das Hauptprogramm instanziiert u. a. einen konkreten Vermittler. Anstelle dessen können Vermittler beispielsweise über "Dependency Lookup" in einer Registratur gesucht werden. Wird wie in dem vorliegenden Beispiel nur ein einziger Vermittler benötigt, so bietet sich auch das Singelton-Pattern an.

- **Anmeldung von Abonnenten**

  An dem Vermittler registriert das Hauptprogramm einen Abonnenten für zwei unterschiedliche Themen. Es ist aber auch denkbar und üblich, dass sich Abonnenten selbst bei einem Vermittler für Themen anmelden.

- **Veröffentlichung von Nachrichten**

  Neben der Instanziierung und der Anmeldung übernimmt das Hauptprogramm auch das Veröffentlichen von Nachrichten. Allerdings werden normalerweise Nachrichtenquellen als eigenständige Objekte realisiert.

Nachstehend das Hauptprogramm:

```
// Datei: Hauptprogramm.java

public class Hauptprogramm
{
   public static void main (String[] args)
   {
      // Instanziiere Objekte
      IVermittler vermittler = new Vermittler();
      IAbonnent   abonnent   = new KonsolenAbonnent();

      // Melde Interesse fuer die Themen Fehler und Warnungen an.
      vermittler.abonnieren ("Fehler",  abonnent);
      vermittler.abonnieren ("Warnung", abonnent);

      // Versende Nachrichten fuer die Themen Fehler und Info.
      vermittler.veroeffentlichen ("Fehler", "Fehlernachricht");
      vermittler.veroeffentlichen ("Info",   "Infonachricht");
   }
}
```

Der sequenzielle Ablauf des Hauptprogramms wird nachfolgend dargestellt:

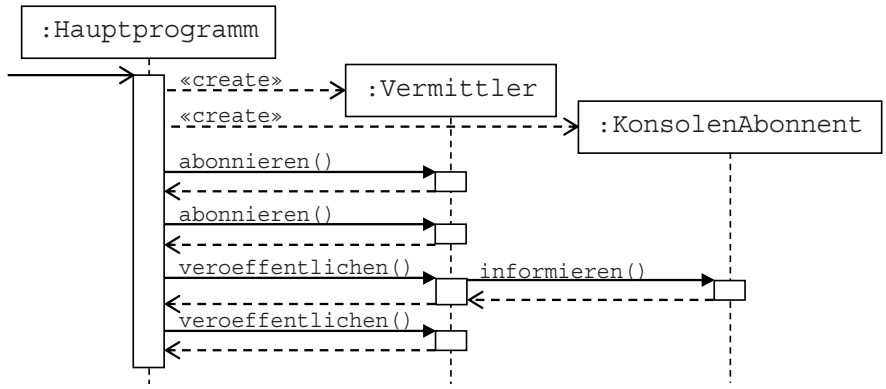


Abbildung 7-5 Sequenzdiagramm für die Information eines Abonnenten

Beim ersten Aufruf der Methode `veroeffentlichen()` soll der Vermittler eine Nachricht zum Thema "Fehler" weiterleiten. Der Vermittler findet zu diesem Thema einen Abonnenten und übermittelt die Nachricht an ihn (Methode `informieren()`). Der dadurch informierte Konsolenabonnent gibt die Nachricht anschließend aus, wie in der folgende Ausgabe des Programms zu sehen ist. Beim zweiten Aufruf der Methode `veroeffentlichen()` erhält der Vermittler eine Nachricht mit dem Thema "Info". Zu diesem Thema hat der Vermittler keine angemeldeten Abonnenten, daher kann er auch niemand informieren. Folglich gibt es in Abbildung 7-5 keinen zweiten Aufruf der Methode `informieren()` und in der Programmausgabe erscheint auch die Nachricht zum Thema "Info" nicht.

Die Ausgabe des Programms ist:

```
Neue Nachricht erhalten.
  Thema : Fehler
  Inhalt: Fehlernachricht
```

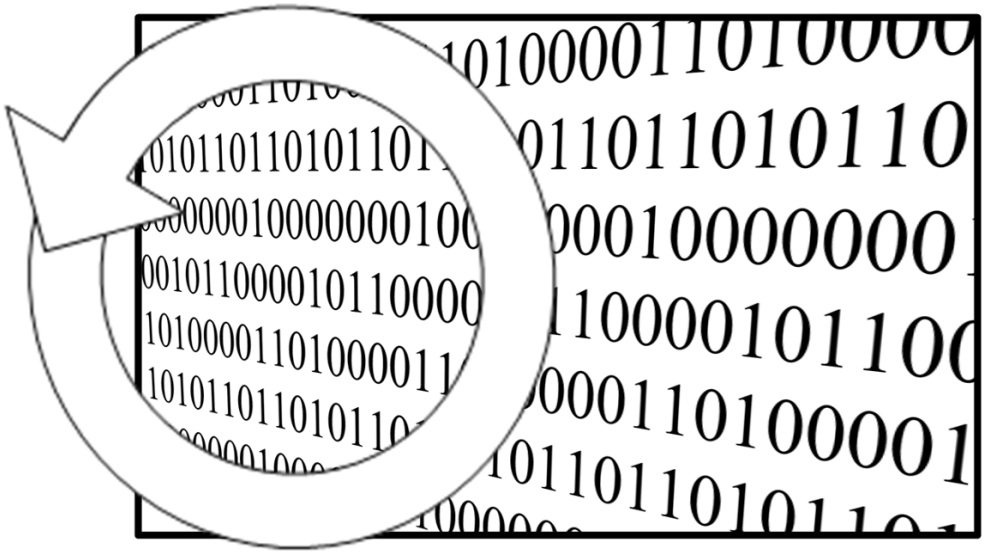
Durch die Verwendung des Vermittlers besteht keine direkte Referenz zwischen Nachrichtensender und Nachrichtenempfänger. Beide müssen jedoch den Vermittler kennen, denn durch diesen werden Nachrichtenempfänger über das Vorliegen von Nachrichten benachrichtigt und müssen nicht selbst nach neuen Nachrichten fragen. Somit werden durch "Inversion of Control" Sender und Empfänger nur lose gekoppelt und zudem können Ressourcen für das alternative zyklische Nachfragen eingespart werden.

7.7 Zusammenfassung

"Inversion of Control" führt zu einer Umkehr der Kontrolle. In der Praxis bedeutet "Inversion of Control", dass ein selbst programmiertes Modul die Steuerung des Kontrollflusses an ein Framework abgibt.

Kapitel 8

Entwurfsprinzipien nahe am Code



8 Entwurfsprinzipien nahe am Code

Das Prinzip "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" von Gamma et al. [Gam94] ist ein Prinzip der Objektorientierung und stellt die Erweiterbarkeit der Programme in den Vordergrund (siehe Kapitel 8.1). Dieses Prinzip stammt aus dem Jahre 1994.

Gamma et al. konzentrieren sich auf eine **Schnittstellen-basierte Programmierung**[87]. Hier werden keine konkreten Klassen als Datentypen verwendet, sondern abstrakte Basisklassen bzw. Schnittstellen in Java. Dadurch wird der Code wiederverwendbar.

Das "Single Level of Abstraction Principle" (siehe Kapitel 8.2) will die Verständlichkeit von Programmen verbessern.

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

8.1 Programmiere gegen Schnittstellen, nicht gegen Implementierungen

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

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

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

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

[87] siehe "Schnittstelle" im Begriffsverzeichnis

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8_8

Alle von einer abstrakten Klasse abgeleiteten Klassen sind auch vom Typ der abstrakten Klasse. Denn eine solche Unterklasse kann zwar neue Methoden hinzufügen, muss aber die in der Basisklasse definierten abstrakten Methoden implementieren[88]. Ebenso ist eine Klasse, die eine Java-Schnittstelle implementiert, auch vom Typ dieser Schnittstelle. Somit gilt:

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

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

Die berühmten, objektorientierten Entwurfsmuster von Gamma et al. beruhen auf den beiden Entwurfsprinzipien

- "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" und
- "Ziehe Objektkomposition der Klassenvererbung vor" (siehe Kapitel 6.3).

Die Bedeutung dieser Prinzipien geht aber weit über die Entwurfsmuster von Gamma et al. hinaus.

8.1.1 Historie

Nach Gamma et al. [Gam94] ist das Prinzip "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" eines der wichtigsten Konzepte der Programmierung (siehe auch [Gam09, p. 25]).

Wie bereits in Kapitel 4.1 erwähnt, ist der Ansatz, Schnittstellen für Module zu verwenden, bereits durch das Prinzip "Loose Coupling and Strong Cohesion" implizit bekannt, denn ohne die Einführung von Schnittstellen gibt es keine schwache Kopplung von Modulen. Die Trennung von Schnittstelle und Implementierung der Module stellt die Voraussetzung für eine Benutzungsabstraktion (siehe Kapitel 1.1.1) und das Einhalten von "Information Hiding" (siehe Kapitel 4.2) dar. Die Trennung von Schnittstelle und Implementierung wird u. a. auch bei Schichtenmodellen (siehe Kapitel 2.2.2) oder dem "Dependency Inversion Principle" (siehe Kapitel 4.5) gefordert.

Gamma et al. gehen jedoch über die Verwendung von Schnittstellen für Module hinaus. Sie betrachten Schnittstellen als Typen, die bei der Typisierung von Variablen eingesetzt werden sollen. Im Original [Gam94] lautet die Formulierung des Prinzips:

"Program to an interface, not an implementation."

[88] In Java werden in diesem Falle die Methoden mit der Annotation `@override` versehen. Der Compiler stellt dann sicher, dass eine so gekennzeichnete Methode implementiert wird, und gibt einen Fehler aus, wenn dies nicht der Fall ist. Daher findet man auch die Sprechweise, dass eine abstrakte Methode in der Unterklasse überschrieben wird.

Diese etwas abstrakte Formulierung wird im anschließenden Satz präzisiert:

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

Das Prinzip fordert also, dass für die Typisierung von Variablen nur abstrakte Klassen (in Java auch Interfaces) benutzt werden. Die Verwendung von konkreten Klassen soll unterbleiben, um möglichst keine Implementierungsabhängigkeiten zu erzeugen.

8.1.2 Ziel

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

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

Programmieren gegen eine Schnittstelle bedeutet letztendlich, dass die eingeführten abstrakten Schnittstellen als Datentypen verwendet werden, wo immer das möglich ist, sei es in den Methodenköpfen, bei Attributen oder bei lokalen Variablen.

Wird gegen eine Abstraktion beispielsweise in Form einer abstrakten Basisklasse programmiert, so kann eine Klasse bei Einhaltung des liskovschen Substitutionsprinzips problemlos gegen eine andere Klasse, welche dieselbe Abstraktion implementiert, ausgetauscht werden. Entsprechendes gilt für Schnittstellen in Java. Dadurch werden Implementierungsabhängigkeiten vermieden und auf diese Weise wird das eigentliche Ziel des Prinzips erreicht, nämlich die Wiederverwendbarkeit des so geschriebenen Programms.

8.1.3 Bewertung

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

- **Abhängigkeit der Clients nur von abstrakten Klassen bzw. Schnittstellen**

  Clients (Kunden) kennen nur die abstrakten Klassen bzw. Schnittstellen in Java und sind damit auch nur von diesen abhängig und nicht von konkreten Objekten bzw. deren Klassen. Clients wissen nichts über die konkreten Klassen der von ihnen verwendeten Objekte, solange diese Objekte der von den Clients erwarteten Schnittstelle genügen. Die konkreten Objekte können beispielsweise durch "Dependency Injection" (siehe Kapitel 4.8.4) einem Client bekannt gemacht werden oder durch "Dependency Lookup" (siehe Kapitel 4.8.3) vom Client gefunden werden.

- **Wiederverwendbarer Code**

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

- **Breiter Anwendungsbereich**

  Dieses Prinzip kann in statisch typisierten objektorientierten Programmiersprachen verwendet werden.

Ein **Nachteil** ist:

- **Einführung einer zusätzlichen Schicht**

  Die Schicht der abstrakten Klassen bzw. Schnittstellen muss als zusätzliche Schicht im Entwurf eingeführt werden. Durch die neuen Elemente wird der Umfang von Entwurf und Implementierung vergrößert.

8.2 Single Level of Abstraction Principle

Um die Verständlichkeit eines Programms zu erhöhen und die Komplexität zu verringern, soll der Programmierer nach dem "Single Level of Abstraction Principle"[89], abgekürzt als SLA, sein Programm in Methoden verschiedener Abstraktionsniveaus einteilen. Alle Anweisungen einer Methode sollen dasselbe Abstraktionsniveau aufweisen.

Ein Programm hat verschiedene Abstraktionsstufen, beispielsweise

- die Ebene der Klassen,
- die Ebene der öffentlichen Methoden oder
- die Ebene der Servicemethoden.

Die Zuweisung eines Wertes an eine Variable steht in der Regel auf einer niedrigeren Abstraktionsebene als beispielsweise ein Methodenaufruf, da sich hinter einem Methodenaufruf viel Logik befinden kann. Doch auch zwei verschiedene Methodenaufrufe können auf zwei unterschiedlichen Abstraktionsebenen stehen. So steht beispielsweise eine get-Methode auf einer niedrigeren Ebene als eine Methode, die einer komplexen User Story[90] entspricht.

Anweisungen, deren Abstraktionsniveau von dem der anderen Anweisungen innerhalb einer Methode abweichen, sollten in eine eigene Methode ausgelagert werden.

Damit hat ein Leser des Programms je nach Interesse die Möglichkeit, Details oder Wesentliches zu betrachten.

[89] auch bekannt als "One Level of Abstraction per Function"
[90] Eine User Story ist eine semiformal formulierte Anforderung des Kunden.

Für das Verständnis des Codes auf dem höheren Abstraktionsniveau ist es in der Regel ausreichend, wenn der Name einer Methode aus Anweisungen eines niederen Niveaus expressiv gewählt wird, da auf der höheren Ebene die Details der niederen Ebene in der Regel überhaupt nicht interessieren. Ein Entwickler möchte beim Lesen einer Methode nämlich oft nur wissen, was in dieser Methode getan wird, und nicht, wie es getan wird.

8.2.1 Historie

Das Prinzip "Single Level of Abstraction" wurde das erste Mal von Robert C. Martin in seinem Buch zu Clean Code [Mar08] unter dem Namen "One Level of Abstraction per Function" erwähnt. Die Bezeichnung "Single Level of Abstraction Principle" stammt aus dem Buch "The Productive Programmer" von Neil Ford [For08].

8.2.2 Ziele

Zwischen den verschiedenen Abstraktionsniveaus eines Codes zu wechseln, macht das Verstehen eines solchen Codes schwierig. Ein Ziel von "Single Level of Abstraction" ist es deshalb, zu vermeiden, dass man beim Lesen eines Programms zuallererst im Kopf die Anweisungen des Programms gruppieren muss, um für jede Gruppe dasselbe Abstraktionsniveau der Anweisungen zu erhalten. Ferner soll durch das Ausweisen getrennter Methoden für die verschiedenen Abstraktionsniveaus der Code besser strukturiert werden.

8.2.3 Bewertung

Der **Vorteil** von SLA ist:

- **Erhöhung der Verständlichkeit von Quellcode**

  Das "Single Level of Abstraction" hilft, den Code besser zu strukturieren und damit die Verständlichkeit zu erhöhen. So wird dem Entwickler geholfen, wichtige Konzepte von unwichtigen Details zu trennen. Zusätzlich erhält der Entwickler einen schnelleren Überblick über das Programm, wenn er ggf. nicht bis zur tiefsten Abstraktionsebene vordringen muss, sondern nur die höheren Abstraktionsebenen betrachtet.

Der **Nachteil** von SLA ist:

- **Potenzielles Inlining**

  Wenn der Code auf zu vielen Ebenen verteilt ist, leidet die Verständlichkeit darunter, da der Entwickler eventuell ein "Inlining" im Kopf betreiben muss, es sei denn, der Name einer Methode ist expressiv gewählt, so dass die Methode bereits durch ihren Namen erklärt wird. Dies sollte eigentlich der Standardfall sein. Für klare Methodennamen hat ein Programmierer zu sorgen!

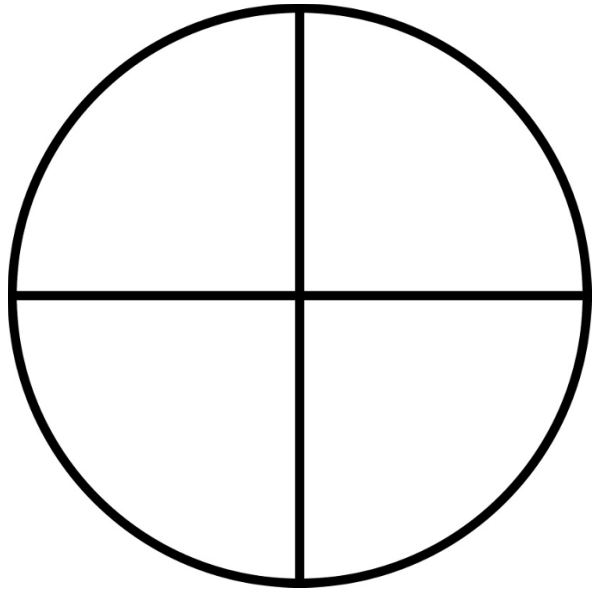
8.3 Zusammenfassung

Das Prinzip ″Programmiere gegen Schnittstellen, nicht gegen Implementierungen″ ist Inhalt von Kapitel **8.1**. Dieses Prinzip fordert, dass Abstraktionen in Form von abstrakten Klassen bzw. Schnittstellen in Java anstelle konkreter Klassen als Datentypen verwendet werden, und schafft dadurch einen wiederverwendbaren Code.

Das Prinzip "Single Level of Abstraction" (siehe Kapitel **8.2**) will verhindern, dass in einem Code zwischen verschiedenen Abstraktionsniveaus gewechselt werden muss. Anweisungen sollen gruppiert werden, um für jede Gruppe dasselbe Abstraktionsniveau der Anweisungen zu erhalten. Diese Gruppen sollen als getrennte Methoden ausgewiesen werden. Man darf aber nicht vergessen, dass diese Strategie beim Lesen des Programms zu einem "Inlining" im Kopf führen muss, falls kein expressiver Methodenname gewählt wurde.

Kapitel 9

Entwurfsprinzipien auf Systemebene



9 Entwurfsprinzipien auf Systemebene

Diese Prinzipien betrachten ganze Systeme und dienen dazu, komplexe Systeme beherrschbar zu machen.

Das Prinzip **"Teile und Herrsche"** (siehe Kapitel 9.1) ist ein wichtiges Grundprinzip der Informatik, welches dann angewandt wird, wenn die **Komplexität** des betrachteten Systems zu hoch ist. Nach diesem Prinzip wird ein System in beherrschbare Abstraktionen als Systemteile aufgeteilt.

Das Prinzip **"Design to Test"** (siehe Kapitel 9.2) betont die hohe Bedeutung einer **einfach testbaren Systemarchitektur**. Das betrachtete System muss leicht testbar sein! Ist die gefundene Architektur nicht leicht testbar, weil sie zu komplex ist, so wird sie verworfen.

9.1 Teile und Herrsche

Das Prinzip **"Teile und Herrsche"** (engl. **"divide and conquer"**, lat. **"divide et impera"**) wird in der Informatik auf vielen Gebieten eingesetzt. Nach diesem Prinzip kann man beispielsweise Algorithmen entwerfen oder ein Programm in Prozeduren, Funktionen, Objekte oder Module als kleinere Einheiten einteilen, die getrennt für sich einer Lösung zugeführt werden können.

Die Methode der schrittweisen Verfeinerung (engl. stepwise refinement) von Wirth (1971) bei der Programmerstellung beruht auf dieser Entwurfsstrategie [Wir71].

Die folgende Abbildung symbolisiert die rekursive Zerlegung eines Problems in Teilprobleme:

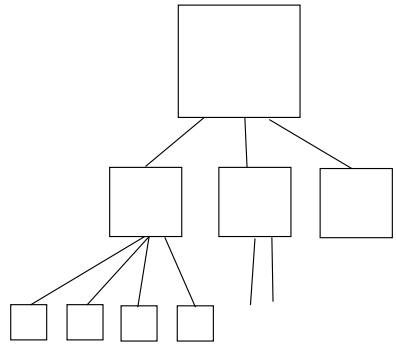


Abbildung 9-1 Rekursive Zerlegung eines Problems in Teilprobleme

9.1.1 Ziel

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

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8_9

Durch Zusammensetzen der Teillösungen ergibt sich dann die Lösung des Gesamtproblems. Die Zerlegung kann rekursiv wiederholt werden, bis ein Teilproblem so klein ist, dass es gelöst werden kann.

9.1.2 Bewertung

Die folgenden **Vorteile** von "Teile und Herrsche" werden gesehen:

- **Brechen der Komplexität**

  Eine Zerlegung bricht die ursprünglich zu hohe Komplexität des Problems.

- **Geringe Komplexität auf der terminalen Ebene**

  Auf der terminalen Ebene der Zerlegung wird nur ein Problem relativ geringer Komplexität gelöst.

- **Atomare Bausteine**

  Die gefundene Lösung wird als atomarer Baustein für die Lösung des übergeordneten Problems verwendet.

Der folgende **Nachteil** wird gesehen:

- **"Top-Down"-Zerlegung passt nicht immer**

  "Teile und Herrsche" bzw. "stepwise refinement" ist eine "Top-Down"-Vorgehensweise. Das kann problematisch sein, wenn eine untere Ebene, beispielsweise Bibliotheken, vorgegeben sind: Dann muss die Zerlegung so erfolgen, dass man diese Ebene auch "trifft". Um das Ziel (die zu treffende Ebene) nicht aus den Augen zu verlieren, geht man häufig gemischt vor: "bottom-up" und "top-down" (Jo-Jo-Ansatz).

9.2 Design to Test

Oftmals bestimmen Performance-Gesichtspunkte in hohem Maße den Entwurf eines Produktes. Dies bedeutet in der Regel, dass dann die Gesichtspunkte der Testbarkeit zu kurz kommen. Dies hat zur Konsequenz, dass sich im Nachhinein der Gesamttest des Systems nicht in dem gewünschten Grade automatisieren lässt. Hohe Kosten für den Gesamtsystemtest sowohl bei der Integration des Systems als auch bei Weiterentwicklungen oder der Beseitigung von Fehlern in der Wartung sind die Folge. Damit wird eine oft enorme Verteuerung nicht nur der Entwicklung, sondern insbesondere auch der Wartung hervorgerufen. Ein Gesamtsystemtest ist leichter, wenn komponentenweise getestet werden kann, da die Komplexität dann geringer ist ("divide et impera" beim Testen).

9.2.1 Ziel

Nach dem Prinzip "Design to Test" wird eine Architektur so entworfen, dass sie leicht zu testen ist. Sollte dies wider Erwarten nicht der Fall sein, so ist die gefundene Architektur zu verwerfen und eine gut testbare Architektur zu entwickeln.

9.2.2 Bewertung

Das Prinzip "Design to Test" hat die folgenden **Vorteile**:

- **Unabhängig testbare Komponenten**

  Es wird auf unabhängig testbare Komponenten geachtet.

- **Automatisierung der Tests**

  Unabhängig testbare Komponenten erlauben eine Automatisierung der Tests.

- **Verringerung der Kosten für das Testen**

  Die bessere Testbarkeit der Teile verringert die Kosten für das Testen.

Der folgende **Nachteil** wird gesehen:

- **Ggf. Erhöhung der Kosten für den Entwurf**

  Stellt man erst nach dem Entwurf fest, dass eine Architektur schlecht testbar ist, steigen die Entwicklungskosten durch einen Neuentwurf stark an. Aus diesem Grund sollte während des gesamten Entwicklungsprozesses die Testbarkeit im Vordergrund stehen, sodass ggf. nur wenige Entscheidungen mit geringer Auswirkung auf die Architektur rückgängig gemacht werden müssen.

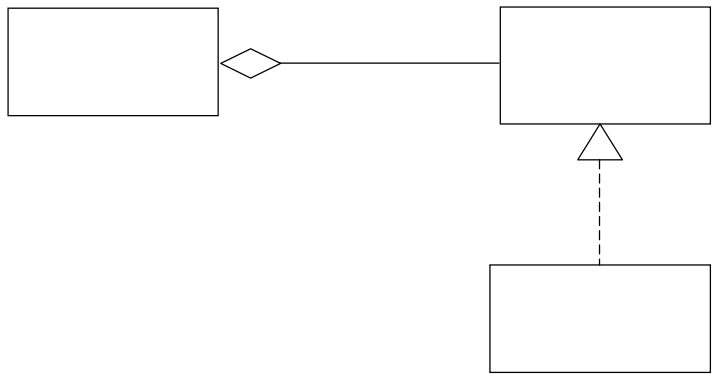
9.3 Zusammenfassung

Das Prinzip "Teile und Herrsche" in Kapitel **9.1** bedeutet, dass ein großes, nicht direkt lösbares Problem so lange in kleine Probleme aufgeteilt wird, bis das entsprechende Teilproblem gelöst werden kann. Die gefundenen Teillösungen sind Bausteine der Gesamtlösung.

Das Prinzip "Design to Test" (siehe Kapitel **9.2**) betont die Wichtigkeit, dass eine Systemarchitektur leicht testbar sein muss, und fordert einen leicht testbaren Systementwurf. Sollte sich wider Erwarten herausstellen, dass eine gute Testbarkeit nicht erreicht wird, so ist die Architektur erneut zu entwerfen.

Kapitel 10

Übersicht über die vorgestellten Entwurfsprinzipien und Konzepte



10 Übersicht über die vorgestellten Entwurfsprinzipien und Konzepte

Kapitel 10.1 klassifiziert die in diesem Buch beschriebenen Prinzipien bzw. Konzepte nach ihrer Bedeutung für die Architektur.

Kapitel 10.2 analysiert deren Einfluss auf die Qualitätsmerkmale

- Korrektheit,
- Einfachheit und Verständlichkeit,
- Stabilität,
- Wandelbarkeit sowie
- Testbarkeit[91].

Kapitel 10.3 diskutiert den Stellenwert der bekannten SOLID-Prinzipien[92] von Robert C. Martin.

10.1 Klassifikation nach der Bedeutung für die Architektur

Betrachtet werden die Kategorien:

- Entwurfsprinzipien zur Komplexität (siehe Kap. 10.1.1),
- Prinzipien und Konzepte für den Entwurf einzelner Klassen (siehe Kapitel 10.1.2) sowie
- Prinzipien und Konzepte für den Entwurf miteinander kooperierender Klassen (siehe Kapitel 10.1.3).

10.1.1 Entwurfsprinzipien zur Komplexität

Diese Prinzipien umfassen das **Vermeiden von Überflüssigem** (siehe Kapitel 3) sowie **Ratschläge zum Bau von Systemen** (siehe Kapitel 9).

Vermeiden von Überflüssigem betrachtet die Prinzipien:

- KISS (siehe Kapitel 3.1),
- YAGNI (siehe Kapitel 3.2) und
- DRY (siehe Kapitel 3.3).

Die eigentliche Bedeutung dieser Prinzipien liegt darin, dass man gewisse Dinge unterlassen soll, nämlich:

[91] Ob ein System leicht testbar ist, erkennt nicht nur der Entwickler des Systems bei seiner Arbeit, sondern auch der Kunde, nämlich genau dann, wenn das System in die Wartung übergeht und Änderungen oder Erweiterungen getestet werden müssen. Die Eigenschaft der Wartbarkeit ist eine externe Qualität, die aus inneren Qualitäten der Software abgeleitet werden kann.

[92] Die SOLID-Prinzipien werden in diesem Buch jeweils einzeln aufgeführt.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8_10

- unnötige Komplexitäten im System,
- vom Kunden nicht benötigte Funktionen und Generalisierungen sowie
- Replikate.

Die genannten Prinzipien gelten – wie bereits in Kapitel 8 erwähnt – generell und nehmen keinen Einfluss auf die Konstruktion der Architektur eines Systems.[93] Sie sorgen aber für

- eine bessere Verständlichkeit der Programme,
- mehr Stabilität und
- eine Abschwächung von wechselseitigen Abhängigkeiten,

da diese Prinzipien die Komplexität des Aufbaus von Systemen einschränken und die Zahl der Elemente eines Systems reduzieren wollen.

Auf der **Systemebene** sorgen die Prinzipien "Teile und Herrsche" sowie "Design to Test" für eine Erleichterung der Strukturierung eines Systems (siehe Kapitel 9). Das Prinzip "Teile und Herrsche" wird verwendet, wenn die Komplexität des vorhandenen Systems zu groß für eine unmittelbare Bearbeitung ist. Man zerlegt dann das System in Teilsysteme, löst die Teilprobleme und fügt die Lösungen der Teilsysteme als Bausteine zum Ganzen zusammen. Das Prinzip "Design to Test" führt durch sein Ziel einer guten Testbarkeit zu einer Architektur eines Systems, die einen verringerten Testaufwand hat.

10.1.2 Prinzipien für den Entwurf einzelner Klassen

Diese Prinzipien betrachten die Konstruktion einzelner Klassen. Zu diesen Prinzipien gehören:

- Abstraktion und "Information Hiding**"** (siehe Kapitel 1.2.3 und 4.2),
- "Separation of Concerns" (siehe Kapitel 4.3),
- das "Dependency Inversion Principle" (siehe Kapitel 4.5),
- das "Interface Segregation Principle" (siehe Kapitel 4.6) sowie
- das "Single Responsibility Principle" (siehe Kapitel 4.7).

10.1.3 Prinzipien und Konzepte für den Entwurf miteinander kooperierender Klassen

In die Kategorie "Prinzipien und Konzepte für den Entwurf miteinander kooperierender Klassen" werden die folgenden Prinzipien und Konzepte, welche die Wechselwirkung von Klassen betreffen, eingestuft:

- "Loose Coupling and Strong Cohesion" (siehe Kapitel 4.1),
- "Law of Demeter" (siehe Kapitel 4.4),
- "Dependency Lookup" (siehe Kapitel 4.8),
- "Dependency Injection" (siehe Kapitel 4.8),

[93] siehe "Architektur eines Systems" im Begriffsverzeichnis

- "Design by Contract" (siehe Kapitel 5.1),
- liskovsches Substitutionsprinzip (siehe Kapitel 5.2)[94],
- "Open-Closed Principle" (siehe Kapitel 6.1)[95],
- "Ziehe Objektkomposition der Klassenvererbung vor" (siehe Kapitel 6.3),
- "Inversion of Control" (siehe Kapitel 7) und
- "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" (siehe Kapitel 8.1).

10.2 Einfluss der Entwurfsprinzipien und Konzepte auf die Qualität

Durch den Einsatz von Entwurfsprinzipien bzw. Konzepten soll eine hohe innere Qualität der Software sichergestellt werden.

In den folgenden Unterkapiteln werden die behandelten Prinzipien und Konzepte für den Entwurf in Verbindung mit den Qualitätsmerkmalen

- Korrektheit,
- Einfachheit und Verständlichkeit,
- Stabilität,
- Wandelbarkeit sowie
- Testbarkeit

gebracht.

10.2.1 Einfluss auf die Korrektheit

Korrektheit betrifft

- den **Erfüllungsgrad der Forderungen des Kunden**,
- die **Korrektheit der Konstruktion des Systems** und
- die **Korrektheit durch das Vermeiden von Überraschungen**.

Kundenspezifische Forderungen betreffen die Sicht des Kunden wie die Forderungen nach Funktionalitäten oder nach Systemeigenschaften.

Als Maßnahmen, um die **Korrektheit der Konstruktion eines Systems** zu erreichen, werden in den folgenden Unterkapiteln 10.2.1.1 bis 10.2.1.4 betrachtet:

- der Einfluss der **Fehlervermeidung** auf die Korrektheit (siehe Kapitel 10.2.1.1),
- die Erleichterung der Korrektheit durch **Einfachheit und Verständlichkeit** (siehe Kapitel 10.2.1.2),

[94] Das LSP regelt die Beziehung zwischen einer abgeleiteten Klasse und ihrer Basisklasse.
[95] Das OCP regelt die Wiederverwendung eines vorhandenen Objekts als Bauteil eines anderen Objekts.

- **Design by Contract** (siehe Kapitel 10.2.1.3) sowie
- die Korrektheit von **Programmen mit polymorphen Objekten** (siehe Kapitel 10.2.1.4).

Auf die Korrektheit durch das Vermeiden von Überraschungen wird in Kapitel 10.2.1.5 eingegangen.

10.2.1.1 Einfluss der Fehlervermeidung auf die Korrektheit

Werden Fehler bei der Entwicklung vermieden, so unterstützt das die Korrektheit eines Systems. Die Verwendung von Entwurfsprinzipien und Konzepten trägt zur Fehlervermeidung bei.

10.2.1.2 Erleichterung der Korrektheit durch Einfachheit und Verständlichkeit

Hat man die Übersicht, so begeht man weniger Fehler. Welche Prinzipien die Einfachheit bzw. die Verständlichkeit und damit die Korrektheit fördern, wird in Kapitel 10.2.2 detailliert analysiert.

10.2.1.3 Design by Contract

Das Konzept **"Design by Contract"** (siehe Kapitel 5.2) fordert für die Beziehungen von Programmen zu ihren Kundenprogrammen eine formale Übereinkunft zwischen den beteiligten Partnern, in welcher präzise definiert wird, unter welchen Umständen ein korrekter Ablauf des Programms erfolgt. Verträge müssen auch bei polymorphen Erweiterungen von Basisklassen eingehalten werden. Dies betrifft eine statische Ableitung oder die Realisierung einer aggregierten polymorphen Abstraktion, die sogenannte "Objektkomposition", zur Laufzeit.

10.2.1.4 Korrektheit von Programmen mit polymorphen Objekten

Das **liskovsche Substitutionsprinzip** (siehe Kapitel 5.2) **in Verbindung mit "Design by Contract"** trägt wesentlich zur Korrektheit von Programmen mit polymorphen Objekten[96] bei.

Bei der Anwendung des liskovschen Substitutionsprinzips wird der lauffähige Code eines Programms aus Basisklassen nicht verändert. Dennoch ist dieses Programm erweiterbar, da bei Einhaltung des liskovschen Substitutionsprinzips die Referenzen auf Objekte einer Basisklasse zur Laufzeit auch auf Objekte abgeleiteter Klassen zeigen können sollen. Bekanntermaßen ist eine abgeleitete Klasse auch vom Typ einer Basisklasse. Die abgeleitete Klasse muss es zum Zeitpunkt der Erstellung des Quellcodes aus Basisklassen noch gar nicht geben.

Das liskovsche Substitutionsprinzip (siehe Kapitel 5.2) postuliert also, dass eine Referenz auf ein Objekt einer Basisklasse jederzeit auch auf ein Objekt einer abgeleiteten Klasse zeigen können soll. Damit eine solche Ersetzung korrekt funktioniert, ist die Einhaltung des Konzepts "Design by Contract" (siehe Kapitel 5.2) absolut erforderlich. Das Kundenprogramm darf es nicht merken, dass an der Stelle eines Objekts der

[96] zur Polymorphie siehe Kapitel 5.2

Basisklasse plötzlich ein Objekt einer abgeleiteten Klasse steht. Daher muss sich die abgeleitete Klasse gleich verhalten und die Verträge der Basisklasse einhalten.

10.2.1.5 Korrektheit durch das Vermeiden von Überraschungen

Hierfür ist das "Principle of Least Astonishment" von sehr großer Bedeutung, welches dafür sorgt, dass die Anwender bzw. die Programmierer sich auf das Einhalten gängiger Konventionen verlassen können und nicht getäuscht werden. Nicht durch Konventionen festgelegte Interpretationen müssen im Projekt festgelegt werden.

10.2.2 Einfluss auf die Einfachheit und Verständlichkeit

Um eine unnötige Komplexität zu vermeiden, sollte ein Programm und seine Dokumentation so einfach wie möglich und gut verständlich sein.

Zur **Einfachheit und Verständlichkeit** tragen wesentlich bei:

- KISS, YAGNI und DRY (siehe Kapitel 3),
- "Loose Coupling and Strong Cohesion" (siehe Kapitel 4.1),
- Abstraktion und **"**Information Hiding**"** (siehe Kapitel 1.2.3 und Kapitel 4.2),
- das "Law of Demeter" (siehe Kapitel 4.4),
- "Separation of Concerns" (siehe Kapitel 4.3) und das "Single Responsibility Principle" (siehe Kapitel 4.7),
- das "Principle of Least Astonishment" (siehe Kapitel 5.3),
- "Teile und Herrsche" (siehe Kapitel 9.1),
- "Design to Test" (siehe Kapitel 9.2) sowie
- das Prinzip "Single Level of Abstraction" (siehe Kapitel 8.2)

Nach KISS (siehe Kapitel 3.1) soll ein System so einfach wie möglich gebaut werden. Teile, die man nur spekulativ brauchen könnte, sollen nach YAGNI (siehe Kapitel 3.2) weggelassen werden. Dies erhöht auch die Einfachheit. Redundante Teile erschweren die Pflege von Änderungen und sollen nach DRY (siehe Kapitel 3.3) entfallen. Dadurch werden Programme weniger komplex und damit einfacher.

Das Prinzip "Loose Coupling and Strong Cohesion" (siehe Kapitel 4.1) fordert, ein System in schwach wechselwirkende Teilysteme zu zerlegen, die einzeln betrachtet werden können, wobei die einzelnen Teile in sich eine hohe Kohäsion haben sollen. Dieses Prinzip macht die Anwendung des Prinzips **"**Teile und Herrsche**"** (siehe Kapitel 9.1) erst sinnvoll möglich.

Abstraktion (siehe Kap. 1.2.2) erlaubt es, sich auf das Wesentliche zu konzentrieren und das Unwesentliche wegzulassen. Das Prinzip "Information Hiding" (siehe Kapitel 4.2) erfordert zum Verstecken der Implementierung schmale Schnittstellen nach außen als Abstraktion der Implementierung.

Das "Law of Demeter" (siehe Kapitel 4.4) führt in der Objektorientierung zu einfachen Kommunikationsstrukturen. Ein Objekt darf nur seinen direkten Nachbarn kennen, es darf aber keineswegs eine mehrfache Verkettung über mehrere Objekte hinweg erfolgen. Der Code hat "schüchtern" zu sein.

Die Prinzipien "Separation of Concerns" (siehe Kapitel 4.3) und "Single Responsibility Principle" (siehe Kapitel 4.7) wollen beide Komplexität und damit verbundene Abhängigkeiten vermeiden. Spricht das Prinzip "Separation of Concerns" nur von einer sauberen Trennung der Belange, geht das "Single Responsibility Principle" noch einen Schritt weiter. Es verlangt, dass in einem physischen Modul nur eine einzige Aufgabe bearbeitet werden darf, um die Komplexität und die damit verbundenen Abhängigkeiten der Module zu reduzieren. Es spricht von einer einzigen Verantwortlichkeit pro Modul. Eine Verantwortlichkeit eines Moduls wird dabei als Grund angesehen, ein Modul abzuändern. Verschiedene Verantwortlichkeiten erfordern verschiedene Module.

Nach dem "Principle of Least Astonishment" (siehe Kapitel 5.3) sollten Oberflächen so entworfen werden, dass ihre Reaktion den Benutzer nicht überrascht. Mit anderen Worten: Sie sollen sich so verhalten, wie sie es andeuten, und sollen den Benutzer nicht täuschen, da damit Fehler provoziert würden. Dieses Prinzip trägt damit zur Verständlichkeit und zur Korrektheit bei. Dieses Prinzip kann auch im erweiterten Sinne, dass Überraschungen generell zu vermeiden sind, praktiziert werden.

Das Prinzip "Teile und Herrsche" führt zu überschaubaren, weniger komplexen Teilsystemen (siehe Kapitel 9.1), die für das System wiederum als Bausteine dienen.

Das Prinzip "Design to Test" (siehe Kapitel 9.2) fordert einfach zu testende Systeme und führt damit zur Verringerung von Komplexitäten.

Das "Single Level of Abstraction Principle" (siehe Kapitel 8.2) will die Verständlichkeit von Programmen verbessern, indem es fordert, dass alle Anweisungen einer Methode dasselbe Abstraktionsniveau aufweisen sollen.

10.2.3 Einfluss auf die Stabilität der Software

Entwurfsprinzipien und Konzepte befassen sich bei der Stabilität mit zwei verschiedenen Aspekten, nämlich mit

- der **Stabilität der Software im Betrieb** durch die Vermeidung der Ausbreitung von Fehlern und
- der **Stabilität der Software bei Änderungen und Erweiterungen** durch die Wiederverwendung stabiler Teile nach dem "Open-Closed Principle".

Auf diese beiden Fälle wird in den beiden folgenden Unterkapiteln eingegangen.

10.2.3.1 Stabilität im Betrieb

Um eine Stabilität der Software im Betrieb zu erreichen, wird der Ausbreitung von Fehlern durch das Vermeiden von starken Kopplungen zwischen den Komponenten entgegengewirkt. Hierbei ist es nicht von Belang, ob diese Fehler durch Programmierfehler oder durch Fehleingaben zustande kommen.

10.2.3.2 Stabilität durch Wiederverwendung vorhandener stabiler Teile

Die Stabilität von Software bei Änderungen und Erweiterungen von Programmen wird im Folgenden betrachtet. Die Wiederverwendung stabiler Teile nach dem "Open-Closed Principle" spielt dabei eine wichtige Rolle. Das "Open-Closed Principle" fordert, dass Module offen für Erweiterungen und gleichzeitig geschlossen für Veränderungen sind.

Offenheit bedeutet, dass stabile Module als unveränderte Bauteile wiederverwendet werden können, wobei die Änderungen sich auf Erweiterungen beschränken. Eine Änderung soll nicht im bereits bestehenden Code, sondern soll als objektorientierte Erweiterung erfolgen. Diese kann beispielsweise über eine Vererbung oder eine sogenannte "Objektkomposition" umgesetzt werden. Vererbung und das Prinzip "Ziehe Objektkomposition der Klassenvererbung vor" arbeiten beide mit objektorientierten Erweiterungen. "Ziehe Objektkomposition der Klassenvererbung vor" betont aber, dass eine Aggregation von Abstraktionen günstiger als die Vererbung für den Bau eines Systems ist, da schwächere Abhängigkeiten erzeugt werden als bei einer Vererbung.

Geschlossenheit eines Moduls gegenüber Veränderungen heißt, dass ein Modul als stabiles Bauteil wiederverwendet werden kann, ohne seinen Code anpassen zu müssen. Da ein solches Modul bereits getestet[97] ist, kann es zur Stabilität beitragen, wenn es wiederverwendet wird.

10.2.4 Einfluss auf die Wandelbarkeit

Die Wandelbarkeit wird wesentlich durch eine Reduktion der Abhängigkeiten gefördert.

Zu einer Verringerung der Zahl der Abhängigkeiten bzw. zur Abschwächung von Abhängigkeiten führen:

- die Prinzipien KISS, YAGNI und DRY (siehe Kapitel 3),
- "Loose Coupling and Strong Cohesion" (siehe 4.1),
- Abstraktion und "Information Hiding" bei Kapselung (siehe Kapitel 1.2.3 und 4.2),
- "Separation of Concerns" (siehe Kapitel 4.3),
- "Law of Demeter" (siehe Kapitel 4.4),
- "Dependency Inversion Principle" (siehe Kapitel 4.5),
- "Interface Segregation Principle" (siehe Kapitel 4.6),
- "Single Responsibility Principle" (siehe Kapitel 4.7),
- "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" (siehe Kapitel 8.1)

und die Konzepte

- "Dependency Lookup" (siehe Kapitel 4.8.3) sowie
- "Dependency Injection" (siehe Kapitel 4.8.4).

[97] Dies ist durchaus kritisch zu sehen. Testen allein ist schon gut, aber ein Modul sollte mit relevanten Daten getestet sein.

Mit der Abschwächung von Abhängigkeiten gegenüber der Vererbung befasst sich das Entwurfsprinzip:

- "Ziehe Objektkomposition der Klassenvererbung vor" (siehe Kapitel 6.3).

Im Folgenden werden die Auswirkungen der genannten Prinzipien und Konzepte kurz zusammengefasst:

- Das Vermeiden von Überflüssigem wird in Kapitel 3 betrachtet. Je weniger Komponenten ein System nach KISS hat, umso weniger Abhängigkeiten kann es zwischen dessen Komponenten geben. Werden nicht geforderte Abhängigkeiten nach YAGNI vermieden, so entfallen unnötige Abhängigkeiten. Gibt es infolge des Prinzips DRY keine Replikate, so müssen diese bei Änderungen ihrer Quelle nicht nachgeführt werden. Damit entfallen Abhängigkeiten.

- Das Prinzip "Loose Coupling and Strong Cohesion" (siehe Kapitel 4.1) fordert in relativ abstrakter Weise eine geringe Kopplung von Modulen und indirekt den Zugriff auf andere Module über Schnittstellen. Dies erlaubt es, Teilsysteme einzuführen, die nicht von der Implementierung anderer Teilsysteme abhängig sind. Das ist eine Grundvoraussetzung für Änderbarkeit und Stabilität.

- Abstraktion und "Information Hiding" (siehe Kapitel 1.2.3 und Kapitel 4.2) ermöglichen schmale Schnittstellen und das Verstecken der Implementierungsdetails. Bei Objekten bedeutet dies nicht nur eine schmale Schnittstelle, sondern nach dem Geheimnisprinzip auch das Verstecken

  – der Daten,
  – von Servicemethoden des eigenen Objekts und
  – der Methodenrümpfe der Schnittstellenmethoden eines Objekts

  im nicht zugänglichen Inneren einer Kapsel.

  Bei Einhalten von "Information Hiding" sind die verschiedenen Objekte über schmale Schnittstellen als Abstraktion der Schnittstellenmethoden nur schwach gekoppelt ("Loose Coupling").

- "Separation of Concerns" (siehe Kapitel 4.3) ist ein Prinzip, das besagt, dass im Falle eines Programms ein Belang (engl. concern) sauber von den anderen Belangen abzugrenzen ist. Ein Belang repräsentiert dabei ein bestimmtes Interesse in einem Programm. Mit "Separation of Concerns" werden Abhängigkeiten verringert.

- Das "Law of Demeter" (siehe Kapitel 4.4) konkretisiert "Loose Coupling and Strong Cohesion" für die Objektorientierung und spezifiziert, welche Methodenzugriffe im Hinblick auf die Reduktion der Abhängigkeiten erwünscht sind und welche nicht. Nach dem "Law of Demeter" wird ein "schüchterner" Code geschrieben, bei dem ein Objekt nicht über ein zweites Objekt auf ein drittes Objekt zugreifen darf.

- Das "Dependency Inversion Principle" (siehe Kapitel 4.5) dient zur Abschwächung von Abhängigkeiten unter Modulen, die in einer hierarchischen Aufrufbeziehung zueinander stehen. Dabei hängt eine Klasse einer aufrufenden höheren Ebene nur von der Abstraktion der Klasse der tieferen Ebene ab. Dies unterstützt in Aufrufhier-

archien den Austausch von Implementierungen sowie die Verwendung von Stubs und Mock-Objekten beim Testen.

- Nach dem "Interface Segregation Principle" (siehe Kapitel 4.6) sollen einem Client nur Methoden einer Schnittstelle angeboten werden, die von diesem wirklich benötigt werden. Das "Interface Segregation Principle" will vermeiden, dass Schnittstellen zu "mächtig" sind. Sie sollen schmal sein und nur die vom jeweiligen Client benötigten Methoden verwenden (Rollen-Schnittstellen).

- Das "Single Responsibility Principle" (siehe Kapitel 4.7) fordert, dass ein Modul nur eine einzige Verantwortlichkeit hat. Durch Anwendung dieses Prinzips werden bei Änderungen einer Verantwortlichkeit unbeabsichtigte Beschädigungen anderer Verantwortlichkeiten vermieden, da andere Verantwortlichkeiten sich in anderen Modulen befinden. Dies erhöht die Stabilität.

- "Programmiere gegen Schnittstellen, nicht gegen Implementierungen" (siehe Kapitel 8.1) unterstützt durch die Verwendung abstrakter Basisklassen bzw. von Schnittstellen in Java anstelle konkreter Klassen die Wiederverwendbarkeit von Programmen bei Einhaltung des liskovschen Substitutionsprinzips. Damit werden Abhängigkeiten von konkreten Klassen vermieden.

- "Dependency Lookup" (siehe Kapitel 4.8.3) ist kein Entwurfsprinzip. Es ist ein Konzept, um bei der Erzeugung von Objekten unliebsame Abhängigkeiten abzuschwächen. Bei "Dependency Lookup" muss ein nutzendes Objekt, das ein weiteres Objekt benötigt, die Abstraktion der Klasse des benötigten Objekts kennen, um die Methoden des gesuchten Objekts aufrufen zu können. Bei "Dependency Lookup" werden Informationen über Objekte in einer zentralen Instanz gespeichert. In der zentralen Instanz gespeicherte Objekte sind beispielsweise über den Objektnamen zugänglich.

- Das Konzept "Dependency Injection"[98] (siehe Kapitel 4.8.4) stellt ebenfalls kein Entwurfsprinzip dar. Es handelt sich hingegen auch um ein nützliches Konzept, um Abhängigkeiten bei der Erzeugung von Objekten abzuschwächen. Zur Kompilierzeit kennt ein Objekt einer nutzenden Klasse statt der konkreten Klasse nur eine Abstraktion der Klasse des von ihm benötigten Objekts. Der Vertrag dieser Abstraktion muss vom Objekt, das an die Stelle der Abstraktion tritt, eingehalten werden, damit die nutzende Klasse das Objekt, welches der Injektor zur Laufzeit an die Stelle der Abstraktion setzt, verwenden kann.

  Eine außenstehende Instanz, die Injektor genannt wird, erzeugt alle Objekte und deren Verknüpfungen. Alle beteiligten Objekte kennen den Injektor nicht und sind daher von diesem unabhängig. Der Injektor dagegen braucht alle Informationen über die zu erzeugenden Objekte sowie deren Verknüpfungen und ist daher von diesen abhängig.

- Das Prinzip "Ziehe Objektkomposition der Klassenvererbung vor" (siehe Kapitel 6.3) verringert Abhängigkeiten durch den Verzicht auf Vererbung einer Basisklasse. Dieses Prinzip erhöht damit die Wandelbarkeit.

---

[98] Dieses Konzept wird oftmals als Entwurfsmuster bezeichnet, "Dependency Lookup" dagegen aber nicht.

10.2.5 Einfluss auf die Testbarkeit

Zur Testbarkeit trägt ganz wesentlich das Prinzip "Loose Coupling and Strong Cohesion" bei. Nur bei Vorliegen schwach gekoppelter Teilsysteme[99] kann man isolierte Unit Tests für die Teilsysteme durchführen. Eine schwache Kopplung der Teilsysteme kann wiederum nur durch die Abstraktion der Implementierung der Teilsysteme mit Hilfe schmaler Schnittstellen nach außen erreicht werden (Benutzungsabstraktion). Das Prinzip "Loose Coupling and Strong Cohesion" verlangt letztendlich "Information Hiding" durch schmale Schnittstellen, auch wenn "Loose Coupling and Strong Cohesion" es selbst nicht explizit fordert.

Hingegen kann das "Dependency Inversion Principle" als eine Spezialisierung von "Loose Coupling and Strong Cohesion" für Aufrufhierarchien angesehen werden. Das "Dependency Inversion Principle" verlangt klare Schnittstellen zwischen den Modulen einer höheren Ebene und den Modulen einer tieferen Ebene. Ein Modul einer höheren Ebene wird dabei gegen die Abstraktion des Moduls der tieferen Ebene programmiert. Damit können an die Stelle von Objekten einer tieferen Ebene beispielsweise Mock-Objekte treten, welche die Schnittstelle erfüllen und das Testen durch die Rückgabe simulierter, sinnvoller Werte unterstützen.

Die Verwendung von Mock-Objekten beim Testen anstelle noch nicht vorliegender echter Objekte wird durch das Konzept der "Dependency Injection" immens erleichtert. Das Setzen der Abhängigkeiten eines Testobjektes zu Mock-Objekten kann dann durch wenige Methodenaufrufe im Test-Setup bewerkstelligt werden. Somit kann in einfacher Weise sichergestellt werden, dass Modultests auch vor der Fertigstellung derjenigen Module, von denen das getestete Modul abhängig ist, lauffähig werden.

Ganz entscheidend für die Testbarkeit der Architektur eines Systems insgesamt ist aber das Prinzip "Design to Test". Nach diesem Prinzip muss beim Entwurf auf eine gute Testbarkeit geachtet werden. Bei Vorliegen des Entwurfs kann beurteilt werden, ob das Design des Systems eine leicht testbare Systemarchitektur geliefert hat oder nicht. Ist die gefundene Architektur wider Erwarten nicht leicht testbar, so muss sie verworfen werden, da das Testen des Systems zu aufwendig wäre.

10.3 Stellenwert der SOLID-Prinzipien

Robert C. Martin fasste eine wichtige Gruppe von Prinzipien zur Erzeugung stabiler, wartbarer, korrekter und langfristig erweiterbarer Software unter dem Begriff "SOLID" zusammen. Der Begriff "SOLID" umfasst die Prinzipien "**S**ingle Responsibility Principle" (siehe Kapitel 4.7), "**O**pen-Closed Principle" (siehe Kapitel 6.1), das **l**iskovsche Substitutionsprinzip (siehe Kapitel 5.2), das "**I**nterface Segregation Principle" (siehe Kapitel 4.6) sowie das "**D**ependency Inversion Principle" (siehe Kapitel 4.5). Der Begriff "SOLID" soll andeuten, dass diese Prinzipien für das Schreiben hochwertiger Software unabdingbar sind. Durch Anwendung der SOLID-Prinzipien kann die Lebensdauer und die Wiederverwendbarkeit von Software erhöht werden. Software, die nach diesen Prinzipien entwickelt wird, lebt nicht nur länger, sondern ist infolge der Abschwächung von Abhängigkeiten auch leichter zu testen.

---

[99] Eine schwache Kopplung wird beispielsweise auch durch das Prinzip "Separation of Concerns" und durch das "Single Responsibility Principle" unterstützt.

Robert C. Martin erklärte die SOLID-Prinzipien zu den wichtigsten Entwurfsprinzipien [Mar02, p. 86]. Das ist natürlich seine persönliche Sicht. Man kann genauso beispielweise die Entwurfsprinzipien "Loose Coupling and Strong Cohesion", "Separation of Concerns" oder "Ziehe Objektkomposition der Klassenvererbung vor" als besonders wertvoll ansehen. Dies würde aber das Wortspiel "SOLID" zerstören.

Durch SOLID wird in mehrfachem Sinne Qualität erzeugt. Das Design

- wirkt dem Altern der Software durch die Abschwächung von Abhängigkeiten entgegen ("Single Responsibility Principle", "Interface Segregation Principle", "Dependency Inversion Principle"),
- erhöht die Testbarkeit durch Mock-Objekte oder Stubs bei Aufrufhierarchien ("Dependency Inversion Principle"),
- sorgt für technische Korrektheit bei Polymorphie (liskovsches Substitutionsprinzip) und
- kann durch Verwendung des "Open-Closed Principle" Stabilität und Erweiterbarkeit erzeugen.

10.4 Zusammenfassung

Kapitel **10.1** klassifiziert die in diesem Buch beschriebenen Prinzipien und Konzepte nach ihrer Bedeutung für die Architektur eines Systems.

Kapitel **10.2** betrachtet, welchen Beitrag die vorgestellten Prinzipien und Konzepte für die Qualitätsmerkmale

- Korrektheit (siehe Kapitel 10.2.1),
- Einfachheit und Verständlichkeit (siehe Kapitel 10.2.2),
- Stabilität (siehe Kapitel 10.2.3),
- Wandelbarkeit (siehe Kapitel 10.2.4) sowie
- Testbarkeit (siehe Kapitel 10.2.5)

erbringen.

In Kapitel **10.3** wird die Bedeutung der SOLID-Prinzipien besprochen.

Literaturverzeichnis

Abkürzungen erhalten bei Büchern und bei Internetquellen, die eine Jahreszahl aufweisen, 5 Zeichen. Die ersten drei Buchstaben werden aus dem ersten Namen der Autoren gebildet, wobei der erste Buchstabe groß geschrieben wird. Die Zeichen 4 und 5 speichern die letzten beiden Ziffern des Erscheinungsjahrs. Gibt es von einem Autor mehrere Veröffentlichungen im selben Jahr, so wird sein Name in der 3. Stelle eindeutig abgeändert.

Der Name von Internetquellen ohne Jahreszahl besteht aus 6 klein geschriebenen Zeichen.

App96 Appleton, B.: "(OTUG) Law of Demeter" 24 10 1996.
Online verfügbar:
http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/AppletonExplainsLoD.txt
[Zugriff am 05 12 2017].

appint Appleton, B.: "Introducing Demeter and its Laws".
Online verfügbar:
http://www.bradapp.com/docs/demeter-intro.html
[Zugriff am 05 12 2017].

Bar68 Barnett, T. O. and Constantine, L. L., eds: "Segmentation and Design Strategies for Modular Programming." In "Modular Programming: Proceedings of a National Symposium". Cambridge, Mass., Information & Systems Press, 1968.

Bec97 Beck, K.: "Make it Run, Make it Right: Design Through Refactoring. The Smalltalk Report", 6, pp. 19-24, 1997.

Bec99 Beck, K: "Extreme Programming Explained: Embrace Change". Addison-Wesley Longman, Amsterdam, 1999.

Ber11 Berens, D., "Ein Refaktorisierungswerkzeug zur Umsetzung des Law of Demeter". Masterarbeit, Okt. 2011.
Online verfügbar:
https://www.fernuni-hagen.de/imperia/md/content/ps/masterarbeit-berens.pdf
[Zugriff am 05 12 2017].

bockda Bock., D.: "The Paperboy, The Wallet, and The Law Of Demeter".
Online verfügbar:
http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf
[Zugriff am 05 12 2017].

Boo95 Booch, G.: "Object Solutions: Managing the Object-Oriented Project". Addison-Wesley, 1995.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8

Dav95 Davis, A. M.: "201 Principles of Software Development". Mc Graw-Hill, 1995.

DeM79 DeMarco, T.: "Structured Analysis and System Specification". Prentice Hall, 1979.

Dij74 Dijkstra, E.: "On the role of scientific thought". In: Dijkstra, E.: "Selected Writings on Computing: A Personal Perspective". Springer-Verlag, 1982, pp. 60-66.
Das entsprechende Originalpapier stammt von 1974.
Auch online verfügbar:
https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html
[Zugriff am 05 12 2017].

For08 Ford, N.: "The Productive Programmer". O'Reilly Media, 2008.

Fow04 Fowler, M.: "Inversion of Control Containers and the Dependency Injection Pattern", 23.01.2004.
Online verfügbar:
http://www.martinfowler.com/articles/injection.html
[Zugriff am 05 12 2017].

Fow05 Fowler, M.: "Inversion of Control", 26.6.2005.
Online verfügbar:
https://martinfowler.com/bliki/InversionOfControl.html
[Zugriff am 05 12 2017].

Fow06 Fowler, M: "RoleInterface", 22.12.2006.
Online verfügbar:
https://martinfowler.com/bliki/RoleInterface.html
[Zugriff am 05 12 2017].

Fow15 Fowler, M.: "Yagni", 26. 05. 2015.
Online verfügbar:
http://martinfowler.com/bliki/Yagni.html
[Zugriff am 05 12 2017].

Fri04 Frick, S. empros gmbh: "Testgetriebene Entwicklung – ein Leitfaden" 9.10.2004.
Online verfügbar:
http://www.empros.ch/downloads/testgetriebeneentwicklung041009.pdf
[Zugriff am 05 12 2017].

Gam09 Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software". Addison-Wesley, 2009

Gam94 Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1994.

Gau70 Gauthier, R. L., Ponto, S. D.: "Designing System Programs". Prentice Hall, 1970.

Gol14 Goll, J.: "Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java": Springer Vieweg, 2014.

Hol89 Holland, K. J. L. u. I.: "Assuring Good Style for Object-Oriented Programs", 1989.
Online verfügbar:
http://homepages.cwi.nl/~storm/teaching/reader/LieberherrHolland89.pdf
[Zugriff am 05 12 2017].

Jam86 Geoffrey, J.: "The Tao of Programming". Info Books, 1986.
Online verfügbar:
http://www.mit.edu/~xela/tao.html
[Zugriff am 05 12 2017].

Jef00 Jeffries, R., Anderson A., Hendrickson Ch.: "Extreme Programming Installed". Addison-Wesley, 2000.

Joh88 Johnson, R. E., Foote, B.: "Designing Reusable Classes". Journal of Object-Oriented Programming, June/July 1988.
Online verfügbar:
https://www.cse.msu.edu/~cse870/Input/SS2002/MiniProject/Sources/DRC.pdf
[Zugriff am 05 12 2017].

keenes Keene, S: "Reliability, Law of Least Astonishment and the Interoperability Imperative".
Online verfügbar:
https://www.hawaii.edu/csati/summit/SKeene_Reliability_Society_Interoperability.pdf
[Zugriff am 05 12 2017].

Lie88 Lieberherr, K., Holland, I., Riel, A. "Object-Oriented Programming: An Objective Sense of Style". OOPSLA '88 Proceedings, Sept. 88.
Online verfügbar:
http://www.ccs.neu.edu/research/demeter/papers/law-of-demeter/oopsla88-law-of-demeter.pdf
[Zugriff am 05 12 2017].

Lie89 Lieberherr, K. J., Holland, L. : "Assuring Good Style by Object-Oriented Programs".
Online verfügbar:
http://homepages.cwi.nl/~storm/teaching/reader/LieberherrHolland89.pdf
[Zugriff am 05 12 2017].

Lie95 K. J. Lieberherr, "Adaptive Object-Oriented Software – The Demeter Method With Propagation Patterns". PWS Publishing company, 1995.

lieber K.J. Lieberherr, "Law of Demeter: Principle of Least Knowledge".
Online verfügbar:
http://www.ccs.neu.edu/home/lieber/LoD.html
[Zugriff am 05 12 2017].

liebfo I. H. Karl J. Lieberherr, "Formulations and Benefits of the Law of Demeter".
Online verfügbar:
http://www.ccs.neu.edu/research/demeter/papers/law-of-demeter/law-formulations/revision1/ss.tex
[Zugriff am 05 12 2017].

liebla Lieberherr, K. "Law of Demeter (LoD) ".
Online verfügbar:
http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html
[Zugriff am 05 12 2017].

Lie89 Lieberherr, K. J., Holland, L. : "Assuring Good Style by Object-Oriented Programs".
Online verfügbar:
http://homepages.cwi.nl/~storm/teaching/reader/LieberherrHolland89.pdf
[Zugriff am 05 12 2017].

Lis87 Liskov, B.: "Data Abstraction and Hierarchy". ACM SIGPLAN Notices, 23(5): pp. 17-34, May 1987.

Mar02 Martin, R. C.: "Agile Software Development: Principles, Patterns, and Practices". Prentice Hall, 2002.

Mar08 Martin, R. C.: "Clean Code: A Handbook of Agile Software Craftsmanship". Prentice Hall, 2008.

Mar12 Martin, R. C.: "Agile Software Development: Principles, Patterns and Practises". Pearson Education, 2012.

Mar14 Martin, R. C." The Single Responsibility Principle"
Online verfügbar:
https://8thlight.com/blog/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html
[Zugriff am 05 12 2017].

Mar96 R. C. Martin: "The Dependency Inversion Principle", in C++ Report, 1996.
Online verfügbar:
http://www.cs.utexas.edu/users/downing/papers/DIP.1996.pdf
[Zugriff am 05 12 2017].

Ma296 Martin, R. C.: "The Interface Segregation Principle"; in: C++ Report, 1996.
Online verfügbar:
http://www.cs.utexas.edu/users/downing/papers/ISP.1996.pdf
[Zugriff am 05 12 2017].

Ma396 R. C. Martin: "The Open-Closed Principle". In C++ Report, 1996.
Online verfügbar:
http://www.cs.utexas.edu/users/downing/papers/OCP.1996.pdf
[Zugriff am 05 12 2017].

marsrp Martin, Robert C.,: "SRP: The Single Responsibility Principle".
Online verfügbar:
http://www.guillemette.org/uqam/mgl7361/assets/documents/SOLID_principles.pdf
[Zugriff am 05 12 2017].

Mey14 Meyer, B.: "Agile!: The Good, the Hype and the Ugly". Springer, 2014.

Mey88 Meyer, B: Object-Oriented Software Construction. Prentice Hall, 1988.

Mey92 Meyer, B: Applying "Design by Contract". IEEE Computer 25 (1992), Nr. 10, p. 40–51

Pag88 Page-Jones, Meilir: "The Practical Guide to Structured Systems Design". 2d ed. Yourdon Press Computing Series, 1988.

Par71 Parnas, D. L.:"On the criteria to be used in decomposing systems into modules". Carnegie-Mellon University in Pittsburgh, Pennsylvania, 1971.
Online verfügbar:
http://repository.cmu.edu/cgi/viewcontent.cgi?article=2979&context=compsci
[Zugriff am 05 12 2017].

Ste74 Stevens, W. P., Myers, G. J., Constantine, L. L.: "Structured design". IBM Systems Journal 13 (2), pp. 115–139, 1974.

Swe85 Sweet, R. E.:"The Mesa Programming Environment". Xerox Palo Alto Research Center, 1985 veröffentlicht in SIGPLAN (ACM Special Interest Group on Programming Languages): SLIPE '85 Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments, pp. 216-229, Seattle, Washington, USA – June 25 - 28, 1985, ACM, New York, 1985.
Online verfügbar:
http://www.digibarn.com/friends/curbow/star/XDEPaper.pdf
[Zugriff am 05 12 2017].

Tho03 Thomas, D., Hunt, A.: "Der Pragmatische Programmierer". Hanser, 2003.

Wes10 Westphal, R: "Abhängigkeiten bewusster wahrnehmen". Blog Ralph Westphal, 7.9.2010.
Online verfügbar:
http://blog.ralfw.de/2010/09/abhangigkeiten-bewusster-wahrnehmen.html
[Zugriff am 05 12 2017].

Wes13 Westphal, R.: "Messaging as a programming model: Doing OOP as if you meant it". CreateSpace Independent Publishing Platform, 2013.

Wir71 Wirth, N.: "Program development by stepwise refinement". Communications of the ACM, 14(4), 1971.

You79 Yourdon, Edward; Constantine, Larry L.: "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design". Prentice Hall, 1979.

Abkürzungsverzeichnis

API	Application Programming Interface
AWT	Abstract Window Toolkit
DbC	Design by Contract
DI	Dependency Injection
DIP	Dependency Inversion Principle
DRY	Don’t repeat yourself
FCOI	Favour Composition over Inheritance
GUI	Graphical User Interface
IEC	International Electrotechnical Commission
IoC	Inversion of Control
ISO	International Organization for Standardization
ISP	Interface Segregation Principle
KISS	Keep it simple, stupid
LoD	Law of Demeter
LSP	Liskovsches Substitutionsprinzip
PLA	Principle of Least Astonishment
OCP	Open-Closed Principle
SLA	Single Level of Abstraction
SOA	Service-Oriented Architecture
SoC	Separation of Concerns
SOLID	Akronym für SRP, OCP, LSP, ISP, DIP
SRP	Single Responsibility Principle

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8

UI	User Interface
UML	Unified Modeling Language
XP	Extreme Programming
YAGNI	You aren't gonna need it

Begriffsverzeichnis

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

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

- **Abstract Window Toolkit**
  Die Klassenbibliothek AWT ist die Vorgängerin der Klassenbibliothek Swing. AWT-GUI-Komponenten sind in Aussehen und Verhalten abhängig vom Betriebssystem. Aufgrund ihrer Abhängigkeit vom Betriebssystem werden sie als "schwergewichtig" bezeichnet. "Leichtgewichtige" Swing-GUI-Komponenten werden hingegen mit Hilfe der Java 2D-Klassenbibliothek durch die Java Virtual Machine selbst auf den Bildschirm gezeichnet und sind damit in Aussehen und Verhalten unabhängig vom Betriebssystem.

- **Abstrakte Klasse**
  Von einer abstrakten Klasse können keine Instanzen gebildet werden. Ein Grund dafür ist oftmals, dass die Klasse abstrakte (nicht implementierte) Methoden enthält.

- **Abstraktion**
  Unter Abstraktion versteht man das Weglassen irrelevanter Einzelheiten und die Konzentration auf das Wesentliche.

- **Aggregation**
  Eine Aggregation ist eine spezielle Assoziation, die eine Beziehung zwischen einer referenzierenden Komponente und einer referenzierten Komponente ausdrückt. Bei einer Aggregation ist – im Gegensatz zu einer Komposition – die Lebensdauer eines referenzierten Objekts nicht mit der Lebensdauer des referenzierenden Objekts gekoppelt.

- **Architektur eines Systems**
  Die Beschreibung der Architektur eines Systems (Systemarchitektur) umfasst:

  – die Beschreibung der Zerlegung des Systems in seine physischen Komponenten,
  – eine Beschreibung, wie durch das Zusammenwirken der Komponenten die verlangten Funktionen erbracht werden, sowie
  – eine Beschreibung der Strategie für die Architektur, d. h. für die Zerlegung (Statik) und für das Verhalten (Dynamik), damit im Team ein "shared understanding" der Architektur gegeben ist.

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8

- **Assoziation**
  Eine Beschreibung eines Satzes von Verknüpfungen (Links) zwischen Objekten. Dabei verbindet eine Verknüpfung bzw. ein Link zwei oder mehr[100] Objekte als Peers (Gleichberechtigte). Eine Assoziation ist prinzipiell eine symmetrische Strukturbeziehung zwischen Klassen. Man kann aber die Navigation auf eine einzige Richtung einschränken.

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

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

- **Äußere Qualität**
  Die äußere Qualität ist die Sicht des Kunden auf die Qualität einer Software.

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

- **Belang**
  Zusammenhängende Funktionen bilden in der Sprechweise von "Separation of Concerns" einen Belang (engl. concern). Verschiedene Belange sollen nach dem Prinzip "Separation of Concerns" sauber getrennt voneinander geführt werden.

- **Benutzungsabstraktion**
  Der Zugang zu einem Modul bzw. einer Komponente erfolgt über eine definierte, schmale Schnittstelle, welche die Leistung eines Moduls abstrahiert und dessen Services anbietet. Benutzer müssen nur wissen, welche Leistung durch ein Modul erbracht wird, nicht aber, wie dessen Implementierung erfolgt. Bei Vorliegen einer Benutzungsabstraktion kann die Verwendung eines Moduls erfolgen, ohne dass man dessen Aufbau kennt.

- **Black-Box-Wiederverwendung**
  Da keine internen Details der Objekte sichtbar sind und die Objekte als Black-Boxes erscheinen, wird "Objektkomposition" auch Black-Box-Wiederverwendung genannt [Gam09, p. 26].

- **Classifier**
  Classifier ist ein Begriff aus dem Metamodell von UML. Die Metaklasse `Classifier` ist eine abstrakte Metaklasse.

[100] Man kann Multiplizitäten einführen.

Es gibt eine Hierarchie von Classifiern. Die Wurzel ist die Metaklasse `Classifier` des Pakets `Classes::Kernel`. Von dieser Wurzel wird abgeleitet und ein ganzer Baum von Classifiern aufgebaut. Eine der Spezialisierungen der Metaklasse `Classifier` ist die Metaklasse `Class`, die Abstraktion einer Klasse. Instanzen der Metaklasse `Class` sind die Klassen.

Jedes Modellelement von UML, von dem eine Instanz gebildet werden kann, ist ein Classifier. Ein Classifier außer der abstrakten Metaklasse `Classifier` kann instanziiert werden.

Ein Classifier besitzt in der Regel eine Struktur und ein Verhalten. Schnittstellen besitzen als einzige Ausnahme meist keine Attribute, d. h., sie haben keine Struktur.

- **Client**
  Siehe Kunde

- **Concern**
  Das engl. Wort "concern" kann mit Belang, Anliegen, Interesse oder Verantwortung übersetzt werden.

- **Dependency Injection**
  Die Erzeugung von Objekten und die Zuordnung von Abhängigkeiten zwischen Objekten wird an eine dafür vorgesehene Instanz delegiert. Diese Instanz, die auch Injektor genannt wird, erzeugt zur Laufzeit die Verknüpfungen zwischen den Objekten. Damit wird die Abhängigkeit zwischen nutzendem und benötigtem Objekt stark abgeschwächt, aber nicht vollständig aufgelöst.

  Zur Kompilierzeit kennt ein Objekt einer nutzenden Klasse statt der konkreten Klasse nur eine Abstraktion der Klasse des von ihm benötigten Objekts. Der Vertrag dieser Abstraktion muss vom Objekt der benutzten Klasse, welches der Injektor an die Stelle der Abstraktion setzt, eingehalten werden.

- **Dependency Lookup**
  Bei dem Konzept "Dependency Lookup" sucht ein Objekt, das ein anderes Objekt braucht, nach diesem anderen Objekt z. B. in einem Register, um die Verknüpfung mit dem benötigten Objekt herzustellen. Zur Suche braucht das suchende Objekt nur einen Schlüssel wie den Namen des gesuchten Objekts zu kennen und ist beim Suchen von dem anderen Objekt weitgehend entkoppelt.

  Ein suchendes Objekt muss nicht mehr die konkrete Klasse des von ihm benötigten Objekts, aber die Abstraktion der Klasse des benötigten Objekts kennen und einhalten, um die Methoden des gesuchten Objekts aufrufen zu können.

  Das suchende Objekt ist aber ferner auch vom Register abhängig, da es von diesem seine benötigten Objekte bezieht.

- **Design**
  Wird hier im Sinne von Entwurf verwendet.

- **Design Pattern**
  Siehe Entwurfsmuster

- **Domäne**
  Eine Domäne umfasst die Aufgaben einer bestimmten Anwendung, auch Fachkonzept genannt.

- **Entwurfsmuster** (engl. **design pattern**)
  Klassen oder Objekte in Rollen, die in einem bewährten Lösungsansatz zusammenarbeiten, um gemeinsam die Lösung eines wiederkehrenden Problems zu erbringen.

- **Extreme Programming**
  Extreme Programming (XP) von Kent Beck [Bec99] ist eine agile Methode, die im Gegensatz zu spezifikationsorientierten Methoden das Programmieren in den Vordergrund eines Projekts stellt sowie die Planung und die Erstellung von Dokumenten auf das Allernötigste beschränkt.

- **Feature**
  Ein Feature ist eine kleine, nützliche Funktion für den Kunden.

- **Framework**
  Ein Framework offeriert dem nutzenden System Klassen, von welchen das System abgeleitet werden kann und somit deren Funktionslogik erben kann. Ein Framework bestimmt die Architektur der Anwendung, also die Struktur im Großen. Es definiert weiter die Unterteilung in Klassen und Objekte, die jeweiligen zentralen Zuständigkeiten, die Zusammenarbeit der Klassen und Objekte sowie den Kontrollfluss [Gam09, p. 37].

- **Geheimnisprinzip**
  Siehe Information Hiding

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

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

- **Information Hiding**
  Wird eine Einheit wie ein Modul oder eine Klasse gekapselt, werden nach außen nur wenige Informationen über die Services dieser Einheit freigegeben (schmale

Schnittstelle). Die Implementierung wird gekapselt oder verborgen ("Information Hiding").

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

- **Innere Qualität**
  Der Entwickler sieht die innere Qualität einer Software. Software muss eine korrekte, stabile und verlässliche Konstruktion darstellen, wobei der Code leicht änderbar und erweiterbar sein muss.

- **Instanz**
  Eine Instanz ist ein Objekt einer Klasse.

- **Instanziierung**
  Das Erzeugen einer Instanz einer Klasse.

- **Inversion of Control**
  Bei der Umkehrung des Kontrollflusses gibt ein selbst geschriebenes Modul die Steuerung des Kontrollflusses an ein anderes – meist wiederverwendbares – Modul ab. Oft ruft dann ein mehrfach verwendbares Modul wie etwa ein Framework das selbst geschriebene Modul auf.

- **Kapselung**
  Die Kapselung eines Moduls umfasst "Information Hiding" und den Zugriff auf ein Modul über eine Schnittstelle als Abstraktion der Leistung eines Moduls. Die Implementierung der Module ist verborgen und ist nur über deren Schnittstellen zugänglich (Benutzungsabstraktion).

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

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

- **Komponente**
  Siehe Modul

- **Komposition**
  Eine Komposition ist ein Spezialfall einer Assoziation. Bei einer Komposition gehört ein Teil genau zu einem zusammengesetzten Ganzen. Ein Teil lebt genauso lange wie das enthaltende Ganze.

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

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

- **Kunde**
  Hier: Teil eines Computerprogramms, welches gewisse Objekte oder Funktionalitäten benutzt.

- **Logisch zusammenhängend**
  Aus Sicht des Problembereichs stark zusammenhängende Module werden als "logisch zusammenhängend" bezeichnet.

- **Mock-Objekt**
  Ein Mock-Objekt ist ein Objekt, das als Platzhalter für echte Objekte innerhalb eines Komponententests verwendet wird. Der Begriff kommt aus dem Englischen und kann mit "etwas vortäuschen" übersetzt werden. Ein Mock-Objekt hat eine Logik implementiert. Ein Mock-Objekt kann beim Testen wie das tatsächliche Objekt aufgerufen werden und liefert vorher festgelegte, sinnvolle Werte zurück.

- **Modul**
  Es gibt keine einheitliche Definition. In diesem Buch wird ein Modul folgendermaßen verwendet:

  Ein Modul kapselt seine Implementierung. Solange die Schnittstelle eines Moduls nicht verändert wird, kann man im Inneren eines Moduls beliebige Änderungen durchführen. Die Leistung eines Moduls steht über schmale Schnittstellen nach außen zur Verfügung. Die Verwendung eines Moduls kann also erfolgen, ohne dass man den Aufbau eines Moduls kennt (Benutzungsabstraktion).

  Wenn die Schnittstellen der Module feststehen, können die Module unabhängig voneinander entwickelt werden (Parallelität der Entwicklung der Module). Module sind in sich logisch sehr stark zusammenhängend und tragen nach dem "Single Responsibility Principle" von Robert C. Martin nur eine einzige Verantwortlichkeit. Module sollen getrennt getestet werden können.

- **Oberklasse**
  Siehe Basisklasse

- **Objekt**
  Siehe Klasse

- **Öffentliche Methode**
  Siehe Schnittstellenmethode

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

- **Paket in Java**
  Ein Paket in Java dient zur Gruppierung von Klassen und Schnittstellen sowie von Paketen in einem Paket. Ein Paket stellt einen Namensraum dar, ist eine Einheit für den Zugriffsschutz und erleichtert die Übersicht.

- **Paradigma**
  Ein Paradigma ist ein Denkkonzept.

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

- **Realisierung**
  Eine Realisierung ist eine statische Beziehung zwischen zwei Elementen, in der das eine Element einen Vertrag spezifiziert und das andere Element sich verpflichtet, diesen Vertrag bei der Realisierung einzuhalten. Realisierungsbeziehungen gibt es beispielsweise bei Klassen, die Schnittstellen implementieren.

- **Schnittstelle**
  Eine Schnittstelle (engl. interface) stellt das Bindeglied zwischen den Nutzern eines Moduls und der Implementierung dieses Moduls dar. Nur die Schnittstelle eines Moduls ist nach außen sichtbar. Eine Schnittstelle erlaubt es, auf die Implementierung eines Moduls zuzugreifen, ohne die Implementierung dieses Moduls zu kennen. Sind Module nicht gekoppelt, können sie nicht zusammenarbeiten.

  Eine Schnittstelle stellt eine Abstraktion des entsprechenden Moduls und dessen angebotenen Leistungen dar. Auf der Ebene von Klassen beinhaltet eine Schnittstelle die Methodenköpfe – also Methodennamen, Rückgabetypen und Übergabeparameter – von öffentlichen Methoden der Klasse. Eine bestimmte Schnittstelle kann alle Operationen oder aber auch nur einen Teil der Operationen einer Klasse repräsentieren. Eine Schnittstelle spezifiziert einen Vertrag, den das realisierende Element erfüllen muss.

  Schnittstellen können in objektorientierten Programmiersprachen oftmals mit einer formalen Syntax beschrieben werden wie beispielsweise in Form eines Interface in Java.

In der Vergangenheit sah ein Interface in Java keine syntaktische Möglichkeit vor, Methoden zu definieren. Dies ist erst seit Java 8 möglich. Schnittstellen in Java können jetzt auch den Charakter einer abstrakten Klasse[101] haben und dedizierte Methoden vorgeben. Die Einführung von Implementierungen in einer Schnittstelle untergräbt jedoch die eigentliche Bedeutung einer Schnittstelle.

Solche Zwischenstufen zwischen einer rein abstrakten Schnittstelle und der Vorgabe zusätzlicher konkreter Implementierungen werden in diesem Buch nicht betrachtet. Eine Schnittstelle im Sinne dieses Buchs enthält grundsätzlich keine Implementierungen.

- **Schnittstellenmethode**
  Über den Methodenkopf einer Schnittstellenmethode (öffentlichen Methode) kann man auf die Daten eines Objektes zugreifen.

- **Schüchterner Code**
  Bei schüchternem Code nach dem "Law of Demeter" darf ein Objekt nicht über ein zweites Objekt auf ein drittes Objekt zugreifen.

- **Shared understanding**
  "Shared understanding" ist ein Begriff aus der Literatur, siehe beispielsweise [Pat14]. Er bedeutet, dass das wesentliche Wissen über das System und das Projekt zwischen allen Projektbeteiligten geteilt ist.

- **Spezialisierung**
  Siehe Generalisierung

- **Struktur**
  Die Struktur eines Systems ist sein statischer Aufbau.

- **Stub**
  Stub bedeutet Platzhalter. Ein Stub ist ein vorläufiger, einfacher Ersatz für eine andere Komponente, die noch nicht erstellt ist, aber benötigt wird, damit ein Programm ablauffähig wird und getestet werden kann. Ein Stub bildet die noch fehlende Komponente in einfachster Weise nach. Er hat keine Logik. Die Aufgabe des Stubs ist es dabei nur, die Durchführung eines Aufrufs zu gewährleisten. Damit ist die Lauffähigkeit eines Programmes hergestellt. Ein Stub gibt beim Aufruf einer seiner Methoden einen festen Wert zurück.

- **Subklasse**
  Siehe abgeleitete Klasse

- **Superklasse**
  Siehe Basisklasse

[101] Abstrakte Klassen dürfen im Allgemeinen neben abstrakten Methoden auch für einige Methoden Implementierungen enthalten – wie neuerdings auch die Interfaces in Java.

- **Unterklasse**
  Siehe abgeleitete Klasse

- **Verantwortlichkeit**
  "A reason to change" (Robert C. Martin) charakterisiert eine sogenannte Verantwortlichkeit (engl. responsibility) einer Klasse.

- **Vererbung**
  In der Objektorientierung kann eine Klasse von einer anderen Klasse statisch zur Kompilierzeit erben bzw. von ihr abgeleitet werden. Durch die Vererbung besitzt die abgeleitete Klasse automatisch alle Attribute und Methoden der Klasse, von der sie abgeleitet wird.

- **Vererbungshierarchie**
  Durch die Vererbungsbeziehung zwischen Klassen entsteht eine Hierarchie: Abgeleitete Klassen werden ihren Basisklassen untergeordnet bzw. Basisklassen sind ihren abgeleiteten Klassen übergeordnet.

- **Verhalten**
  Im Verhalten eines Systems kommt seine Funktionalität zum Ausdruck.

- **Vertrag**
  Ein Vertrag regelt insbesondere die Beziehungen zwischen Aufrufer und Aufgerufenem.

- **White-Box-Wiederverwendung**
  Wiederverwendung durch Unterklassenbildung wird oft auch White-Box-Wiederverwendung genannt, da die öffentlichen und geschützten Elemente der Basisklasse in der abgeleiteten Klasse direkt sichtbar sind.

Index

J. Goll, *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*, https://doi.org/10.1007/978-3-658-20055-8

Lizenz zum Wissen.

Sichern Sie sich umfassendes Technikwissen mit Sofortzugriff auf tausende Fachbücher und Fachzeitschriften aus den Bereichen: Automobiltechnik, Maschinenbau, Energie + Umwelt, E-Technik, Informatik + IT und Bauwesen.

Exklusiv für Leser von Springer-Fachbüchern: Testen Sie Springer für Professionals 30 Tage unverbindlich. Nutzen Sie dazu im Bestellverlauf Ihren persönlichen Aktionscode **C0005406** auf *www.springerprofessional.de/buchaktion/*

Jetzt 30 Tage testen!

Springer für Professionals.
Digitale Fachbibliothek. Themen-Scout. Knowledge-Manager.

- Zugriff auf tausende von Fachbüchern und Fachzeitschriften
- Selektion, Komprimierung und Verknüpfung relevanter Themen durch Fachredaktionen
- Tools zur persönlichen Wissensorganisation und Vernetzung

www.entschieden-intelligenter.de

Springer für Professionals

Springer