Objekt-orientiertes Software-Engineering

Karl Friedrich Gebhardt

Auflage vom 10. Oktober 2019

Prof. Dr. K. F. Gebhardt

Tel: 0711-667345-11(16)(15)(12)
Fax: 0711-667345-10
email: kfg@lehre.dhbw-stuttgart.de

Vorwort

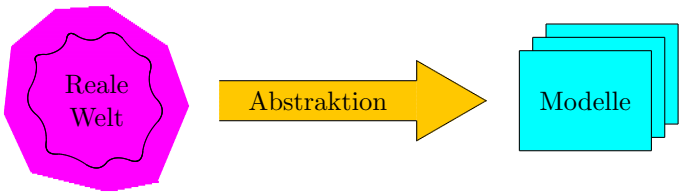
Das Skriptum ist ein Torso und weist daher noch viele Lücken auf. Aber nur das Umvollkommene ist kreativ.

Auf die wichtigste Notation für objekt-orientierte Entwicklung – UML Unified Modeling Language – wird hier nicht eingegangen. Für UML gibt es ein eigenes Skriptum.

Inhaltsverzeichnis

Kapitel 1

Einleitung



Software-Engineering bedeutet Modellierung der realen Welt. Dabei müssen vorerst künstliche Sprachen verwendet werden, weil uns sonst die Hardware nicht versteht. Allerdings sind Texte in natürlichen Sprachen auch Modelle der realen Welt, die von menschlichen Partnern i.a. besser verstanden werden. Weil auch die künstlichen Sprachen von Menschen gelesen werden, gilt vieles, was für die Verwendung von natürlichen Sprachen gilt, auch für künstliche Sprachen. Man sollte z.B. **treffende** Namen oder Begriffe verwenden.

Modellierung ist die Erstellung einer abstrakten Darstellung der realen Welt. Abstraktion bedeutet dabei, nur die für die Anwendung wesentlichen Aspekte der realen Welt zu erfassen. Genauigkeit der Abbildung wird nur soweit getrieben, als es für die Anwendung nötig ist. Vernünftigerweise sollten dabei nicht nur die gerade vorliegende Anwendung oder Anforderung berücksichtigt werden, sondern auch naheliegende, künftige Anwendungen und Entwicklungen antizipiert werden.

Modellierung hat also die Aufgabe, den nahezu unendlichen Informationsgehalt der realen Welt auf ein handhabbares Maß zu reduzieren.

Software-Engineering ist **Systemanalyse** und **Systemdesign** (ältere Begriffe). Das Design dürfte der wichtigste Aspekt sein, dem wir uns im folgenden widmen:

Design (Formgebung, künstliche Gestaltung, Entwurf (Das Bertelsmann Lexikon)) von Software entspricht der Tätigkeit des Architekten beim Hausbau. Das fertige Produkt sollte dementsprechend folgende Qualitäten haben ("De Architectura Libri Dece", Vitruvius, 29 aD), die wir hier als Ziele formulieren.

Design-Ziele:

- ***commodity***, Benutzbarkeit, erfüllt Anforderung des Kunden (Aspekt des Kunden)
- ***firmness***, Zuverlässigkeit, Korrektheit, Effizienz (Aspekt des Engineerings)
- ***delight***, Ästhetische Freude, Stil (Künstlerischer Aspekt)

Commodity und *delight* sind manchmal sich widersprechende Ziele. Aber *firmness* und *delight* scheinen korreliert zu sein.

Design ist keine Wissenschaft, aber verwendet durchaus wissenschaftliche Erkenntnisse um Software zu erstellen.

Wissenschaftliche **Grundlagen** von Design sind:

- Theorien über Struktur und Repräsentation
- Theorien über Effizienz
- Strenge und formale Techniken um die Eigenschaften eines Designs zu untersuchen und das beste (oder ein akzeptables) Design zu ermitteln
- Theorien über Benutzerschnittstellen
- Theorien über Ästhetik

Design ist ein Prozess, eine Tätigkeit, eine Vorgehensweise. Es wird gemacht, weil es schlicht billiger ist, eine Skizze zu ändern oder zu verwerfen als ein fertiges Software-Produkt.

Allgemeine **Methodologie** oder Vorgehensweise:

- Design ist vorläufig. Geschieht *vor* der Konstruktion.
- Design benutzt Skizzen, Modelle. Modelle sind Abstraktionen. Dabei wird eine **Notation** benutzt.
- Komplexität wird durch **Dekomposition** (Zerlegung) und **Abstraktion** (Vereinfachung, Verzicht auf Irrelevantes) beherrscht.
- Design ist ein **Prozess**, d.h. in einer bestimmten Reihenfolge werden Modelle mit eventuell unterschiedlicher Notation verwendet. Der Prozess ist wahrscheinlich iterativ.
- Design benutzt Heuristiken (Faustregeln, Erfahrungen).

Software-Design ist fundamental schwieriger als das Design von Gebäuden, Brücken, Maschinen. Denn Software ist

- komplex (Viele Teile, d.h. Codezeilen. Die Anzahl der Verwendungs-Szenarien ist praktisch unbegrenzt. Jede Änderung oder Ergänzung kann neues Verhalten, neue Bedingungen bringen.)
- ohne Prinzipien (Es gibt keine Gesetze, die für korrekte Programme erfüllt sein müssten.)

- leicht zu ändern (Deshalb und, weil sich die reale Welt ändert, wird Software dauernd geändert und angepasst. Software bietet von Natur aus nahezu unbegrenzte Flexibilität. Ein Design muss robust gegenüber solchen Änderungen sein, d.h. sein Gesicht nicht völlig verlieren.)
- unsichtbar. (Maschinenteile, Gebäudeteile sind leicht visualisierbar, vorstellbar, Programmteile nicht.)

Beim prozeduralen (oder auch strukturierten) Design ist der primäre Abstraktions-Mechanismus das **Unterprogramm** ***(subroutine)***. Man spricht auch von **Arbeitseinheiten** ***(units of work)***. Der Kontrollfluss geht von Oberprogramm zu Unterprogramm.

Beim objekt-orientierten Design ist der primäre Abstraktions-Mechanismus das **Objekt** und die **Klassifikation** ***(classification)*** von Objekten. Die Klasse ist eine Kombination von Daten und Operationen auf den Daten, wobei das Prinzip der **Datenkapselung** ***(encapsulation)*** beachtet wird. Der Kontrollfluss ergibt sich aus Objekten, die miteinander agieren, indem sie einander Botschaften senden.

Die Branchenbezeichnung "Datenverarbeitung" ***(data processing)*** lenkt den Augenmerk auf eine prozedurale Vorgehensweise in der Softwareentwicklung. Software-Systeme transformieren Eingangsdaten in Ausgangsdaten.

Die objektorientierte Sicht dagegen sieht Software-Systeme als Modelle der realen Welt. Software-Objekte entsprechen real-weltlichen Objekten, die allerdings eventuell auch Datentransformationen durchführen.

Kapitel 2

Objekt-Orientierung

Aristoteles war wohl der erste, der den Begriff Klasse verwendete, indem er Fische oder Vögel jeweils als zu einer Klasse gehörig sah. Die erste objekt-orientierte Sprache war Simula-67 mit dem Schlüsselwort `class`, das die Einführung neuer Typen in ein Programm erlaubte.

Software-Entwicklung bedeutet, eine Abbildung zwischen den Elementen des **Problem-Bereichs** ***(problem space, problem domain)*** und des **Lösungs-Bereichs** ***(solution space)*** zu schaffen. Mit objekt-orientierter Programmierung ist es relativ leicht eine Eins-zu-Eins-Abbildung zu erreichen.

Vier fundamentale Konzepte bestimmen objekt-orientierte Programme:

1. **Objekt** ***(object)***:

   - Hat eine Identität. Zwei gleiche Tische sind zwei unterschiedliche Objekte.
   - Enthält Daten, die seinen Zustand beschreiben.
   - Hat ein Verhalten, d.h. eine Menge von Prozeduren, Funktionen oder Methoden, die das Verhalten des Objekts modellieren und über Botschaften angesprochen werden können.

2. **Klasse** ***(class)***:
   Die Klasse ist eine Abstraktion des Objekts. Es ist i.a. unmöglich, alle Daten und das ganze Verhalten für jedes einzelne Objekt zu definieren. Stattdessen wird für Objekte, die bezüglich der vorliegenden Anwendung gleich strukturiert sind, d.h. die gleiche Art von Daten und dasselbe Verhalten haben, eine Klasse definiert, die diese Objekte nur soweit beschreibt, wie es für die Anwendung erforderlich ist.

   Kapselung ***(encapsulation)***:
   Prinzipiell wird jede Klasse in zwei Teile geteilt: eine **Schnittstelle** ***(interface)*** und eine **Implementation** ***(implementation)***. Die Schnittstelle beschreibt vollständig, wie ein Benutzer *(user, client)* der Klasse auf Objekte der Klasse zugreifen kann. Normalerweise gehören die Datenelemente einer Klasse nicht zur Schnittstelle, sondern zur Implementation. Damit werden Klassen *robust* gegenüber Änderungen. D.h. ihre Implementation kann relativ leicht geändert und neuen Anforderungen angepasst werden. Das hilft, dem Problem der sich laufend ändernden Software Herr zu werden.

3. **Vererbung** ***(inheritance)***, besser **Erweiterung** ***(extension)***:
   Vererbung ist ein mächtiges Softwareorganisationsinstrument, indem Gemeinsamkeiten von Klassen herausgezogen und in allgemeinere Superklassen gestellt werden.

4. **Polymorphismus** ***(polymorphism)***:
   Jedes Objekt einer Subklasse kann diesselben Botschaften empfangen, die ein Objekt der Superklasse empfängt. Die Reaktion auf diese Botschaften kann aber durchaus unterschiedlich sein ("vielgestaltig"). Der Polymorphismus vermeidet umständliche und fehlerträchtige Fallunterscheidungen an vielen, oft verstreuten Stellen. Damit wird der Kontrollfluss wesentlich vereinfacht.

Alan Kay hat im Hinblick auf Smalltalk fünf charakteristische Eigenschaften genannt, die einen reinen Angang objektorientierten Programmierens darstellen:

Everything is an object. Ein Objekt ist eine besonders gut ausgestattete Variable, die Daten enthält, die man aber auch Operationen auf den Daten ausführen lassen kann. Jede konzeptionelle Komponente eines Problem-Bereichs kann durch ein Objekt im Programm repräsentiert werden.

A program is a bunch of objects telling each other what to do by sending messages. Ein Ersuchen *(request)* wird an das Objekt gestellt, indem man ihm eine Botschaft sendet. Konkret bedeutet das den Aufruf einer Funktion oder Methode, die zu dem Objekt gehört.

Each object has its own memory made up of other objects. Man kann eine neue Art von Objekten machen, indem man ein Paket aus existierenden Objekten schnürt. Auf diese Weise kann man Komplexität in einem Programm repräsentieren, die hinter einfachen Objekten verborgen bleibt.

Each object has a type. Jedes Objekt ist die Instanz einer Klasse, wobei Typ synonym mit Klasse verwendet wird. Die Klasse bestimmt, welche Botschaften einem Objekt gesendet werden können. Wenn man Typ und Klasse unterscheidet, dann bestimmt der Typ nur die Schnittstelle, während die Klasse eine spezielle Implementation dieser Schnittstelle ist.

All objects of a particular type can receive the same messages. Diese Aussage hat weitreichende Konsequenzen. Da ein Objekt auch vom Typ der geerbten Klassen ist, kann es auch deren Botschaften empfangen. Ein Objekt kann Objekte geerbter Klassen substituieren. Das Substitutionsprinzip ist eines der mächtigsten Konzepte objekt-orientierter Programmierung.

Prinzip: Alles und jedes ist ein Objekt (und nicht eine Prozedur).

Damit wird alles viel konkreter, greifbarer, sichtbarer.

Das Problem für einen Computer ist, dass sein "Denken" relativ "kontextfrei" ist. Das bedeutet, man muss ihm seine Umgebung immer ganz genau spezifizieren. Wenn das nicht gemacht wird, macht er Fehler. Durch die Objekt-Orientierung ist es viel leichter, die Umgebung eines Objekts zu spezifizieren. Jedes Objekt hat als Kontext seine Klasse und deren Beziehungen.

Jedes Objekt kann man als ein eigenständiges "Programm" auffassen, das durch Methodenaufrufe in seiner durch das Objekt spezifizierten Umgebung zum Laufen gebracht wird. Während man es bei der prozeduralen Programmierung mit der Komplexität eines riesigen Programms zu tun

hat, wird diese Komplexität in der Objektorientierung durch Zerlegung in viele kleine Programme vermieden.

Im folgenden wird noch einmal ausführlich auf die vier fundamentalen Aspekte der Objekt-Orientierung eingegangen.

2.1 Objekte und ihre Identität

Identität ***(identity)*** bedeutet, dass Daten als diskrete, unterscheidbare Einheiten, sogenannte Objekte gesehen werden. Ein Objekt ist z.B. ein spezieller Abschnitt in einem Text, eine spezielle Wiedergabe eines Musikstücks, das Sparkonto Nr.2451, ein Kreis in einer Zeichnung, ein Eingabekanal, ein Sensor, ein Aktor. Objekte können konkreter oder konzeptioneller Natur sein (z.B. Herstellungsanleitung, Rezeptur, Verfahrensweise). Prozesse können auch Objekte sein. Jedes Objekt hat seine Identität. Selbst wenn die Werte aller Attribute von zwei Objekten gleich sind, können sie doch verschiedene Objekte sein. In der Datenverarbeitung werden solche Objekte dadurch unterschieden, dass sie verschiedene Speicherplätze belegen.

Ein besonderes Problem für die Datenverarbeitung ist die Tatsache, dass reale Objekte i.a. eine Lebensdauer haben, die über die Programmlaufzeit hinausgeht (**Persistenz** von Objekten).

Der Zugang zur realen Welt geschieht beim objektorientierten Ansatz über Objekte, nicht über Funktionen. Für Objekte wird ein **Verhalten nach außen (Interface, Methoden)** und eine **interne Repräsentation (Datenstruktur)** definiert. Das Verhalten von Objekten ist im Laufe der Zeit sehr konstant oder kann sehr konstant gehalten werden. Wenn sich das Verhalten ändert, dann äußert sich das meistens durch eine schlichte Erweiterung des Interfaces mit weiteren Methoden. Alte Methoden können oft erhalten bleiben.

Eine Verhaltensänderung hat manchmal zur Folge, dass die interne Repräsentation (Datenstruktur) von Objekten geändert werden muss. Da aber das bisherige Verhalten fast immer auch mit der neuen Datenstruktur emuliert werden kann, ist die Datenstruktur von Objekten insgesamt weniger konstant als ihr Verhalten. Das Verhalten von Objekten hat einen allgemeinen Charakter unabhängig von speziellen Datenverarbeitungsaufgaben.

Funktionen sind Lösungen konkreter Automatisierungsaufgaben. Diese verändern sich laufend, oder es kommen neue Aufgaben dazu. Beruhen Funktionen direkt auf der Datenstruktur, dann müssen alle Funktionen geändert werden, wenn es notwendig wird, die Datenstruktur zu ändern. Beruhen Funktionen aber auf dem Verhalten, muss an den Funktionen nichts geändert werden.

2.2 Klasse und Klassifizierung

Klassifizierung ***(classification)*** bedeutet, dass Objekte mit denselben Attributen (Datenstruktur) und demselben Verhalten (Operationen, Methoden) als zu einer Klasse gehörig betrachtet werden. Die Klasse ist eine Abstraktion des Objekts, die die für die gerade vorliegende Anwendung wichtigen Eigenschaften des Objekts beschreibt und den Rest ignoriert. Die Wahl von Klassen ist letztlich willkürlich und hängt von der Anwendung ab. Jede Klasse beschreibt eine möglicherweise unendliche Menge von Objekten, wobei jedes Objekt eine **Instanz** seiner Klasse ist. Attribute und Verhalten werden in *einer* Klasse zusammen verwaltet, was die Wartung von Software wesentlich erleichtert.

Jede Methode, die für eine Klasse geschrieben wird, steht überall dort zur Verfügung, wo ein Objekt der Klasse verwendet wird. Hierin liegt die Ursache für den Gewinn bei der Software-Entwicklung. Denn der Entwickler wird dadurch gezwungen, die Methoden allgemein anwendbar und "wasserdicht" zu schreiben, da er damit rechnen muss, dass die Methode auch an ganz anderen Stellen angewendet wird, als wofür sie zunächst entworfen wurde. Der Schwerpunkt liegt auf dem Problem**bereich** (**problem domain**) und nicht auf dem gerade zu lösenden (Einzel-)Problem.

Abstrakte Datentypen ***(data abstraction)***, **Datenkapselung** ***(information hiding)*** bedeutet, dass kein direkter Zugriff auf die Daten möglich ist. Die Daten sind nur über Zugriffsfunktionen zugänglich. Damit ist eine nachträgliche Änderung der Datenstruktur relativ leicht möglich durch Änderung der Software nur in einer lokalen Umgebung, nämlich der Klasse. Die Zugriffsfunktionen sollten so sorgfältig definiert werden, dass sie ein wohldefiniertes und beständiges Interface für den Anwender einer Klasse bilden.

Die Trennung von **Implementation** und **Schnittstelle** ist eines der wichtigsten Prinzipien des Software-Engineering. Beim objekt-orientierten Ansatz wird diese Trennung auf der Objektebene, nicht auf Prozess- oder Funktionsebene durchgeführt. Die Trennung auf Objektebene ist wesentlich verantwortlich für die Stabilität objektorientierter Software.

2.3 Vererbung

Vererbung ***(inheritance)*** ist die gemeinsame Nutzung von Attributen und Operationen innerhalb einer Klassenhierarchie. Girokonto und Sparkonto haben die Verwaltung eines Kontostands gemeinsam. Um die Wiederholung von Code zu vermeiden, können beide Arten von Konten von einer Klasse `Konto` *erben*. Oder alle Sensoren haben gewisse Gemeinsamkeiten, die an spezielle Sensoren wie Temperatursensoren, Drucksensoren usw weitervererbt werden können. Durch solch ein Design werden Code-Wiederholungen vermieden. Wartung und Veränderung von Software wird erheblich sicherer, da im Idealfall nur an *einer* Stelle verändert oder hinzugefügt werden muss. Ohne Vererbung ist man gezwungen, Code zu kopieren. Nachträgliche Änderungen sind dann an vielen Stellen durchzuführen. Mit Vererbung wird von einer oder mehreren Basisklassen geerbt. Das andere Verhalten der erbenden Klasse wird implementiert, indem Daten ergänzt werden, zusätzliche Methoden geschrieben werden oder geerbte Methoden überschrieben ("überladen" ist hier falsch) werden.

2.4 Polymorphismus

Polymorphismus ***(polymorphism)*** bedeutet, dass dieselbe Operation unterschiedliche Auswirkung bei verschiedenen Klassen hat. Die Operation "lese" könnte bei einer Klasse `Textabschnitt` bedeuten, dass ein Textabschnitt aus einer Datenbank geholt wird. Bei einer Klasse `Sparkonto` wird durch "lese" der aktuelle Kontostand ausgegeben. Bei einer Klasse `Sensor` bedeutet "lese", dass der Wert des Sensors angezeigt wird, bei einer Klasse `Aktor`, dass ein Stellwert eingegeben werden soll.

Eine spezifische Implementation einer Operation heißt **Methode**. Eine Operation ist eine Abstraktion von analogem Verhalten verschiedener Arten von Objekten. Jedes Objekt "weiß", wie

es seine Operation auszuführen hat. Der Anwender von solchen Objekten muss sich nicht darum kümmern.

Polymorphismus von Objekten bedeutet, dass ein Objekt sowohl als Instanz seiner Klasse alsauch als Instanz von Basisklassen seiner Klasse betrachtet werden kann. Es muss möglich sein, Methoden dynamisch zu binden. Hierin liegt die eigentliche Mächtigkeit objekt-orientierter Programmierung.

Ohne Objektorientierung wurde Polymorphismus durch `case`-Konstrukte implementiert. Das bedeutet, dass jedes Vorkommen solcher Konstrukte nach einer Anwendung des Polymorphismus von Objekten ruft.

2.5 Beispiele

2.5.1 Beispiel Messtechnik

Bei der Programmierung einer Prozessdatenverarbeitung müssen Messdaten, z.B. Temperaturen erfasst werden.

Bei einem **funktionalen** Angang des Problems wird man versuchen, eine Funktion zur Erfassung einer Temperatur zu entwickeln, die eine Analog/Digital-Wandler-Karte anzusprechen hat, die Rohdaten eventuell linearisiert und skaliert.

Beim **objekt-basierten** Ansatz muss man zunächst geeignete **Objekte** bezüglich der Temperaturerfassung suchen. Hier bietet sich der **Temperatursensor** als Objekt an, für den die Klasse Temperatursensor definiert werden kann. Der nächste Schritt ist, die **Repräsentation** eines Temperatursensors zu definieren, d.h. festzulegen, durch welche Daten oder Attribute ein Temperatursensor repräsentiert wird. Als Datenelemente wird man wohl den aktuellen Temperaturwert, verschiedene Skalierungsfaktoren, Werte für den erlaubten Messbereich, Eichkonstanten, Kenngrößen für die Linearisierung, Sensortyp, I/O-Kanal-Adressen oder -Nummern definieren.

Erst im zweiten Schritt wird man die Funktionen (jetzt **Methoden** genannt) zur Skalierung, Linearisierung und Bestimmung des Temperaturwerts und eventuell Initialisierung des Sensors definieren und schließlich programmieren. Ferner wird man dem Konzept der **Datenkapselung** folgend Zugriffsfunktionen schreiben, mit denen die Daten des Sensors eingestellt und abgefragt werden können.

Beim **objekt-orientierten** Denken wird man versuchen, zunächst einen allgemeinen Sensor zu definieren, der Datenelemente hat, die für jeden Sensor (Temperatur, Druck usw) relevant sind. Ein Temperatursensor wird dann alles erben, was ein allgemeiner Sensor hat und nur die Temperaturspezifika ergänzen. Weiter könnte der Tempertursensor relativ allgemein gestaltet werden, so dass bei Anwendung eines NiCrNi-Thermoelements das Thermoelement vom allgemeinen Temperatursensor erbt.

Dieses Beispiel soll zeigen, dass der objekt-basierte oder -orientierte Ansatz den Systementwickler veranlasst (möglicherweise gar zwingt), eine Aufgabe allgemeiner, tiefer, umfassender zu durchdringen. Ferner ist beim funktionalen Ansatz nicht unmittelbar klar, was z.B. beim Übergang von einem Sensor zum anderen zu ändern ist. Im allgemeinen muss man die Messfunktion neu schreiben. Bei einer gut entworfenen Objektklasse sind nur Werte von Datenelementen zu ändern. Ferner wird es bei einer gut angelegten Klasse offensichtlich sein, welche Daten zu ändern sind.

Kapitel 3

Objekt-orientierte Systementwicklung

In diesem Kapitel werden wir eine Methodologie oder Vorgehensweise vorstellen, deren Ziel es ist, gute Software zu produzieren. Wenn die einzelnen Schritte zuverlässig durchgeführt werden, dann wird das Ziel sehr wahrscheinlich erreicht.

Aber warum ist dann Software in der Praxis so schlecht? Warum gibt es eine sogenannte Software-Krise? Das liegt nicht daran, dass es keine geeigneten Methoden gibt oder die Methoden den Entwicklern unbekannt wären. Es liegt daran, dass der Wettbewerb auf dem Software-Markt heutzutage die Entwicklungs-Teams zu unmöglichen Terminplänen zwingt, die dazu verleiten, nicht systematisch vorzugehen [19]. Das hat schlechte Software zur Folge, die die Termin-Situation noch weiter verschärft. Es ist ein circulus viciosus: Mit jedem entdeckten Fehler werden genügend neue Fehler erzeugt, so dass fast kein Fortschritt festzustellen ist.

Auf einen Software-Manager, der "nein" zu einem Terminplan sagt, kommen zehn andere, die "ja" sagen. Damit ist er aus dem Geschäft. Die Termin-Problematik ist das schlimmste Problem der Software-Entwicklung, alle anderen Probleme sind dagegen harmlos. Zugegeben: Wir werden uns im folgenden mit den harmloseren Problemen beschäftigen.

Systementwicklung ist trotzdem noch schwierig genug, weil es dafür keine Patentrezepte gibt. Systementwicklung heißt, sich ein Modell von der realen Welt zu machen. Ein Modell ist grundsätzlich jede Sicht auf ein reales System, ist das, was wir in unserem Gehirn aus der realen Welt aufgrund unserer Sinneseindrücke machen und evtl. auf einem Speichermedium (Papier oder IT-Hardware) "persistieren".

Ein gutes Modell kann Fragen an ein System beantworten, ohne dass man mit dem System selbst arbeiten muss.

Bei der objekt-orientierten Systementwicklung sehen wir die Realität in erster Linie als ein aus Objekten zusammengesetztes Modell. Erst an zweiter Stelle interessieren wir uns für die konkreten Automatisierungsaufgaben, was sich wiederum mit den Zielen des Auftraggebers reibt, der konkrete Automatisierungsaufgaben gelöst sehen will.

Eine **Software-Engineering-Methodologie** ***(software engineering methodology)*** ist ein Prozess zur **organisierten** Entwicklung von Software, der definierte Techniken und Notationen

benutzt. Wir beschränken uns hier auf Systeme, deren wesentlicher Teil Software ist. D.h. wir beschäftigen uns nicht oder nur am Rande mit der sogenannten **Unternehmensentwicklung *(enterprise development)***, die auch andere Bereiche als nur Software berücksichtigt. Ein Unternehmen kann man als aus einzelnen Systemen aufgebaut betrachten, wobei diese einzelnen Systeme durchaus in das hier behandelte Gebiet passen.

Folgende Prinzipien gelten als ***best practices*** der Software-Entwicklung. Moderne Prozesse versuchen diese Prinzipien zu realisieren [45]:

- Entwickle Software iterativ. *(Develop software iteratively.)*
- Verwalte Anforderungen. *(Manage requirements.)*
- Verwende eine Komponenten-basierte Architektur. *(Use component-based architectures.)*
- Modelliere Software visuell. *(Visually model software.)*
- Verifiziere die Software-Qualität kontinuierlich. *(Continuously verify software quality.)*
- Verwalte Änderungen an der Software. *(Control changes to software.)*

Die Entwicklung eines objekt-orientierten Systems erfolgt gemäß dem **Unified Process (UP)**, früher Rational Unified Process RUP) in **Phasen *(phase)***. Die Phasen sind:

- **Konzeption *(inception, conception)***
- **Entwurf, Ausarbeitung *(elaboration, design)***
- **Konstruktion *(construction)***
- **Auslieferung, Einführung, Übergang *(transition)***

Eine Phase besteht aus **Iterationen *(iteration)***, in denen Entwicklungs-**Prozesskomponenten *(process component)*** mit unterschiedlicher Intensität durchgeführt werden. Eine Iteration dauert zwischen zwei und sechs Wochen. Zu lange Iterationen werden der Flexibilität dieses Prozesses nicht gerecht. Das iterative Vorgehen, hat den wesentlichen Vorteil, das man eigentlich immer nur auf den Stand der letzten Iteration zurückfallen kann. Ein "totaler Absturz" (mit tolalem Neustart) ist wesentlich unwahrscheinlicher. Kleine Schritte, Feedback und Anpassung sind Schlüsselkonzepte.

Die Phasen sind typischerweise die Sicht des Managements auf ein Projekt.

Das typische Verhältnis der Phasen zueinander und die Anzahl der jeweiligen Iterationen gibt folgende Tabelle:

Phase	Inception	Elaboration	Construction	Transition
Aufwand	5%	20%	65%	10%
Iterationen:				
Kurzes Projekt:	0	1	1	1
Typisches Projekt:	1	2	2	1
Komplexes Projekt:	1	3	3	2
Riskantes Projekt:	2	3	3	2

Die Prozesskomponenten sind:

- **Management** ***(management)***
- **Architektur** ***(architecture)***
- **Anforderungsanalyse** ***(requirements)***
- **Analyse** ***(analysis)***
- **Design** ***(design)***
- **Implementierung** ***(implementation)***
- **Testen** ***(testing)***
- **Systemintegration** ***(system integration)***
- **Einsatz** ***(deployment, operation, service, use, application)***

Die Phasen werden **sequentiell** durchgeführt, die Prozesskomponenten können **parallel** durchgeführt werden. Prinzipiell kann in **jeder** Phase **jede** Prozesskomponente durchgeführt werden.

Folgende Matrix soll das veranschaulichen, wobei die Symbole · ○ ● ein Maß für die Intensität der jeweiligen Prozesskomponente sind.

	Phasen															
Prozesskomponenten ▽	Konzeption				Entwurf				Konstruktion				Auslieferung			
	Iter.1		Iter.	...	Iter.1		Iter.	...	Iter.1		Iter.	...	Iter.1		Iter.	...
Management	●	●	●	○	○	○	·	·	·	·	·	○	●	○	○	·
Architektur	·	○	○	●	●	●	●	○	●	○	·	·	○	·	·	·
Anforderungsanalyse	·	○	○	●	●	○	·	·		·			○	·		
Analyse		·	·	○	●	●	○	·	·	·	·	·	○			
Design			·	·	○	●	●	●	●	○	·					
Implementierung							·	○	●	●	●	●	·	○	·	
Testen		·	·	○	·	○	○	●	○	●	●	●	●	○	○	·
Systemintegration			○							·	○	●	●	○	·	
Einsatz												·	○	●	●	●

Die Systementwicklung kann als Datenfluss begriffen werden, wobei als die zu bearbeitenden Daten das Systemmodell selbst zu sehen ist, das durch die verschiedenen Prozesskomponenten transformiert wird.

Jede Komponente des Entwicklungsprozesses bedeutet die Erstellung einer Beschreibung *(description)* eines Modells des Systems. Diese Beschreibungen können von verschiedenen Leuten auf verschiedenen Stufen verstanden werden. Z.B. ist der Source-Code eine Beschreibung des Systems, die von Programmierern und von Compilern verstanden wird.

Der Übergang von einer Phase in die andere ist fließend. Eine einheitliche Meinung über genaue Schnitte gibt es nicht. Wahrscheinlich haben Sie schon mal etwas anderes gehört. Das kann aber nicht schaden. Es gibt nicht nur eine Lösung, sondern mehrere.

Im folgenden geben wir einen Überblick über die einzelnen Prozesskomponenten. Beinahe alle Prozesskomponenten bestehen aus Unterkomponenten. Zu jeder Prozesskomponente folgt später ein eigenes Kapitel.

Management

Architektur

- **Systemhierarchie**
  Bei großen Systemen wird hier versucht in Teilsysteme zu unterteilen, für die jeweils ein eigenes Vorgehens-Schema durchlaufen wird.

Anforderungsanalyse

Analyse (Was ist das System?)

- **Systembeschreibung**
  Das zu entwickelnde System (Aufgabe oder Problem, ***domain***) wird umgangssprachlich beschrieben. Resultat ist ein Text, der das notwendige Fachwissen über das System enthält. Anforderungen kommen vor, spielen aber eine sehr untergeordnete Rolle.
- **Anwendungsfälle** ***(use case)***
  Mit den Anwendungsfällen wird beschrieben, wie das System von den verschiedenen Benutzern verwendet wird.
- **Analyse von Entitäten**
  Entitäten, d.h. Objekte und Klassen, deren Attribute, Verhalten und Beziehungen werden identifiziert.
- **Dynamisches Verhalten des Systems**
  Welche Abläufe gibt es? Zu welchen Objekten gehören diese Abläufe? Welche Beziehungen gibt es zwischen Abläufen? Parallelität, Serialität.
- **Funktionales Verhalten des Systems**
  Welche wichtigen Datentransformationen und Datenflüsse gibt es in dem System?

Design (Entwurf des Systems) Beim objekt-orientierten Design geht man von den Daten aus und folgt damit einem datenorientierten Ansatz, wie er bei Datenbankanwendungen üblich ist.

- **Klassifikation von Entitäten**
  Entitäten werden in (Entitäts-)Typen klassifiziert, indem Gemeinsamkeiten von Entitäten festgestellt werden. Es wird eine Typenhierarchie aus Basistypen und Subtypen gebildet.
- **Objektmodell**
  Ein Objektmodell (z.B. E/R- oder UML-Diagramm) wird erstellt.
- **Muster**
  Es wird versucht, Muster ***(patterns)*** zu erkennen.
- **Dynamisches Modell**
  Ein dynamisches Modell wird erstellt.
- **Funktionales Modell**
  Ein funktionales Modell wird erstellt, wenn das wirklich erforderlich ist. Jedenfalls können Überlegungen in dieser Richtung nicht schaden.

Implementierung in Java, C#, C++, Go, ... (Konstruktion des Systems)

Entitäts-Typen werden Klassen. Entitäts-Instanzen werden Objekte von Klassen. Attribute werden Datenelemente von Klassen. Verhalten wird eine Menge von Methoden von Klassen.

Testen

Testen betrifft Korrektheit und Qualität. Dazu mehr im Kapitel "Qualität".

Systemintegration

Systemintegration wird nur dann notwendig, wenn das Gesamtsystem vorher konzeptionell in Teilsysteme unterteilt wurde (siehe Atchitektur).

- **Zusammensetzung aus Teilsystemen**
  Das System wird aus seinen Teilsystemen zusammengebaut und getestet. Das System wird selbst als Objekt einer Klasse behandelt.

Charakteristika von stabilen Systemen (Herbert Simon):

- Stabile Systeme sind *hierarchisch* strukturiert. Systeme bestehen aus Teilsystemen. Für Teilsysteme gilt dasselbe wie für Systeme.
- Stabile Systeme sind *fast-zerlegbar*. Man kann die Teile, aus denen ein System besteht, identifizieren und man kann unterscheiden zwischen Interaktionen zwischen den Teilen und innerhalb der Teile. Es gibt *zwischen* den Teilen wesentlich weniger Interaktion.
- Stabile Systeme bestehen aus *wenigen* Teilen unterschiedlichen Typs.
- Stabile Systeme haben sich aus einfachen funktionierenden Systemen entwickelt.

Einsatz

- **Auslieferung, Installation** *(deployment, installation)*
- **Einweisung und Schulung** *(training)*
- **Produktionseinsatz, Operation des Systems** *(operation)*
- **Wartung** *(maintenance)*
  Die Philosophie langfristiger Unterstützung *(long-term support)* ist bei SW *absolut* notwendig wegen der inhärenten Änderungen (Flexibilität) von SW.
- **Erweiterung** *(extension)*
  Bei größeren Erweiterungen muss der ganze Entwicklungszyklus durchlaufen werden. Versionen sind zu verwalten *(delta requirements specification)*.

Bei den meisten Software-Systemen beansprucht Wartung den größeren Teil der gesamten Softwarekosten.

Anhang A ist eine Dokumentations-Schablone für diesen Entwicklungsprozess.

Kapitel 4

Management der Software-Entwicklung

4.1 Aspekte

Stephen R. Palmer: *"Those who work in real estate industry tell us that the three most important aspects of real estate are location, location, and location.*
The software development process equivalent is communication, communication, and communication."

Das Management steuert, koordiniert und überwacht die Entwicklung.

Das Management muss Entscheidungen treffen, auch wenn die vorliegenden Informationen unvollständig oder widersprüchlich sind.

Dazu sind notwendig

- Erfahrung: Das bedeutet, ein gutes Einschätzungsvermögen zu haben bezüglich der Erfolgswahrscheinlichkeit von Lösungsalternativen.
- Führungspersönlichkeit: Das bedeutet, die natürliche – nicht institutionelle – Autorität zu haben, Entscheidungen auf Grund mangelhafter Informationen durchzusetzen.
- Gute Kommunikationseigenschaften
- Aktivität und Zielorientierung

Die schiere Größe eines Software-Projekts beeinflusst dessen Erfolg. Je größer und komplexer, desto schwerer ist es zu beherrschen und desto wahrscheinlicher ist sein Scheitern. Man sollte daher unbedingt versuchen, klein anzufangen. Moderne Entwicklungs-Prozesse sind in der Tat **evolutionär** ausgerichtet, wobei die Entwicklung in kleinen, überschaubaren Schritten erfolgt.

Es gibt sicher keinen Entwicklungs-Prozess, der sich für alle Systeme eignet. Daher sollte man sich nicht sklavisch an ein Vorgehensmodell klammern, sondern pragmatisch vorgehen.

Das Management muss im allgemeinen drei Faktoren zusammenbringen, um ein Produkt zu entwickeln. Das sind:

Personen
Entwicklungs-Prozess
Technologie

Das Management eines Software-Systems umfasst typischerweise folgende Aspekte:

Technisches Management: Wird meistens als Architektur bezeichnet und ist im entsprechenden Kapitel beschrieben.

Hier werden eventuell die Phasen des UP definiert und verfolgt.

Kunden-Management: Hier unterscheiden wir die

- **Anwender, Nutzer** der Software, die schließlich die Use-Cases liefern, und die dann das System anwenden.

  Möglicherweise wird ihre Tätigkeit durch die zu entwickelnde Software wesentlich rationalisiert. Oft ist es schwierig, Zugang zu den eigentlichen Nutzern eines Systems zu finden.

- **Geldgeber**. Sie haben strategische Ziele wie Reaktionsfähigkeit, Rationalisierung, Motivation für Mitarbeiter.

  Alle Mitglieder der Geschäftsführung müssen hinter den Entwicklungszielen eines Software-Projekts stehen.

Team-Management: Hier geht es um die Motivation eines Entwickler-Teams. Das Team kann gefordert werden, darf aber nicht überlastet werden. Das Management muss ihm den Rücken frei halten. Motivation ist der wichtigste Faktor für das Gelingen eines Projekts.

Entwickler haben die Tendenz, das Ziel aus den Augen zu verlieren.

Für das Team-Management eignen sich eventuell Prozesse, wie sie unter der agilen Software-Methodologie Scrum definiert sind.

Organisations-Management: Ein Projekt ist normalerweise nicht das einzige innerhalb einer Organisation, sondern steht in einem konkurrierenden und/oder synergetischen Verhältnis zu anderen Projekten. Ferner gibt es Stabsstellen wie Controlling, Qualitätssicherung, Revision, die auch ihren Einfluss auf Projekte haben.

4.2 Agile Projekte

Moderne Projekte sind dadurch charakterisiert, dass sie sehr schnell auf die Änderungen ihrer Umgebung reagieren. Sie heißen daher auch "agile" Projekte. Als Managementstrategie hat sich ein Team-orientierter Ansatz bewährt, der eher auf Kompetenz und Vertrauen basiert, statt auf Kontrolle. Jens Coldewey hat für diese Strategie ein Manifest erstellt:

> Wir finden bessere Wege, Software zu entwickeln, indem wir es tun und anderen dabei helfen. Durch diese Arbeit sind wir zu der folgenden Wertegewichtung gelangt:

- *Einzelperson und Interaktion* bewerten wir höher als Prozesse und Werkzeuge.
- *Laufende Systeme* bewerten wir höher als umfangreiche Dokumentation.
- *Zusammenarbeit mit dem Kunden* bewerten wir höher als Vertragsverhandlungen.
- *Die Fähigkeit, auf Veränderungen zu reagieren,* bewerten wir höher als das Verfolgen eines Plans.

D.h. wir messen den Punkten auf der linken Seite der Liste eine höhere Bedeutung bei, obwohl auch die Punkte auf der rechten Seite wichtig sind.

4.3 Management von Kommunikation

Bei zwei Entwicklern gibt es genau *einen* Kommunikationspfad. Bei 10 Entwicklern gibt es 45 Pfade, bei 100 gibt es 4950 Kommunikationspfade. Das bedeutet, dass Kommunikation hier unbedingt gemanaged werden muss.

Zur Kommunikation werden Sprachen verwendet. Ferner findet oft eine Übersetzung von einer Sprache in eine andere Sprache statt. Hier gibt es Fehlerquellen. Schließlich stellt sich die Frage, ob die richtige Information geliefert wird. Ein guter Kommunikations-Prozess besteht daher aus zahlreichen Validierungs- und Verifikations-Zyklen.

Psychologische Probleme – z.B. die Furcht, falsche oder fehlerhafte Information zu liefern – behindern den Kommunikationsfluss. Die Beziehungen müssen vertrauensvoll sein und nicht etwa auf Furcht, Misstrauen und Repressalien basieren. Und trotzdem soll der Einzelne Verantwortung für seine Resultate übernehmen.

4.4 Management von Komplexität

Die Komplexität eines Software-Systems wächst mindestens mit dem Quadrat seiner Größe (Gerald Weinberg: size/complexity dynamic).

Das kann man mathematisch etwa folgendermaßen ausdrücken: Wenn die Größe g_a des Projekts um den Faktor f ansteigt

$$g_n = f \cdot g_a$$

und seine ursprüngliche Komplexität k_a war, dann ist jetzt die neue Komplexität:

$$k_n = k_a^f$$

Wenn seine ursprüngliche Erfolgswahrscheinlichkeit p_a war, dann ist jetzt die neue Erfolgswahrscheinlichkeit:

$$p_n = p_a^f$$

Die beste Strategie ist rekursive oder hierarchische Zerlegung, bis jedes Teil behandelbar wird. Dabei muss die Integration der Teile zum Gesamtsystem immer im Auge behalten werden.

4.5 Scrum

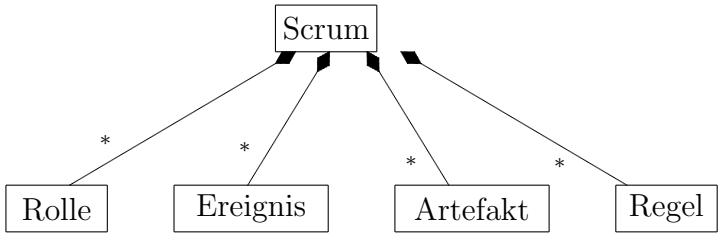
Ken Schwaber und Jeff Sutherland haben Scrum, ein Rahmenwerk zur Entwicklung und Erhaltung komplexer Produkte entwickelt.

Wichtige Aspekte oder Komponenten des Prozesses sind:

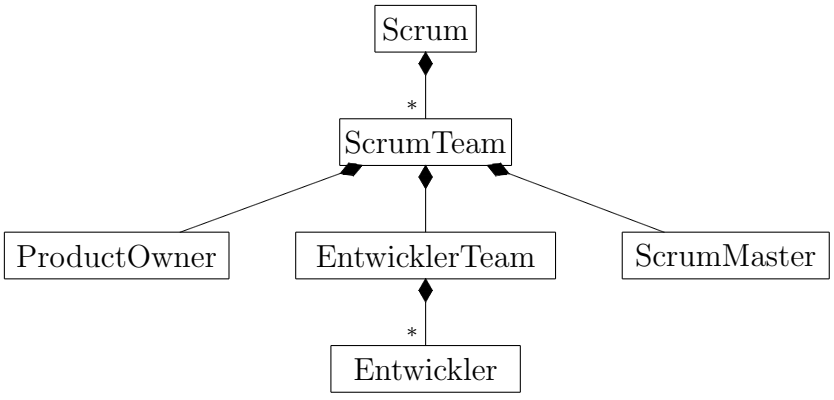
- Transparenz
- Überprüfung
- Anpassung

Diese Prozesskomponenten müssen gewährleistet sein, damit der Scrum-Prozess erfolgreich durchgeführt werden kann.

4.5.1 Bestandteile von Scrum

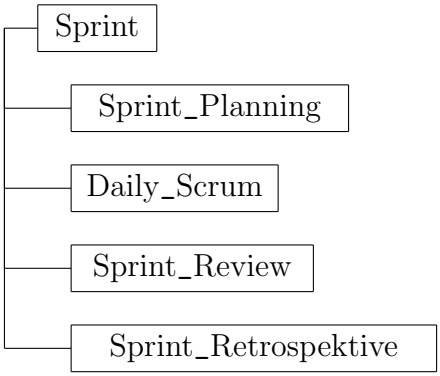


4.5.2 Rollen in Scrum

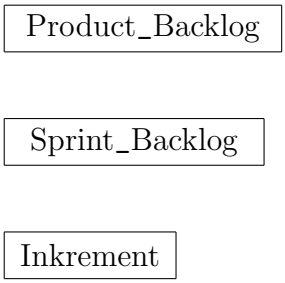


Teams in Scrum sind **selbstorganisierend** und **interdisziplinär**.

4.5.3 Ereignisse in Scrum



4.5.4 Artefakte in Scrum



4.5.5 Regeln in Scrum

- Nur das Entwicklungsteam kann sein Sprint Backlog während des Sprints ändern.
- Am Ende eines Sprints muss das Inkrement **"Done"** sein.
- Artefakte müssen transparent sein.
- Jedes Ereignis hat eine **maximale Dauer**. Ausnahme ist der Sprint, der eine Dauer hat, die weder gekürzt noch verlängert werden darf.

Kapitel 5

Architektur von Software-Systemen

Jedes Software-System hat eine Architektur. Aber nur wenn die Architektur **explizit** gemacht wird, erlaubt sie die Planung und Steuerung der Entwicklung. Wichtigstes Merkmal ist dann, dass Entscheidungen nicht zufällig entstehen. "Es wird nichts dem Zufall überlassen." **Regeln** bestimmen, wie Aufgaben im System gelöst werden.

Die Software-Architektur befasst sich nicht nur mit der Struktur und dem Verhalten eines Systems, sondern auch mit Anwendung, Funktionalität, Performanz, Wartbarkeit, Wiederverwendbarkeit, Verständlichkeit, ökonomischen und technischen Randbedingungen und mit der Ästhetik des Systems.

Allerdings wird die Architektur eines Software-Systems oft enger gefasst: *The software architecture of a program or computing system is the structure or the structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.* [1]

Architektur beschäftigt sich mit der Frage, wie ein System aussehen soll.

Architektur ist aufwendig und teuer und dauert lange. Allerdings können Teile einer Architektur i.a. oft wiederverwendet werden.

Die Architektur sollte auf der Grundlage eines Analysemodells entwickelt werden.

Die Software-Architektur beeinflusst insbesondere folgende Punkte:

- Änderbarkeit
- Wartbarkeit
- Wiederverwendbarkeit
- Risiken

Daher ist der Einfluss der Software-Architektur bei kurzlebigen Projekten deutlich geringer als bei langlebigen Projekten.

Architektur greift auf Erfahrungen zurück: Ähnliche Problemstellungen werden auf ähnliche, bewährte Weise gelöst.

5.1 Architektonische Stile oder Muster

5.1.1 Zentralistische Architektur

5.1.2 Dezentrale Architektur

5.1.3 Service Oriented Architecture (SOA)

5.1.4 Client/Server

Client/Server–Konzepte (C/S) haben sich aus der Einsicht entwickelt, dass man Oberfläche und Implementierung trennen muss. Diese Einsicht ist natürlich auch die Basis für OOP. Eine typische C/S–Umgebung hat drei Kategorien von Diensten:

Benutzerdienste *(user services, user interface, UI)***:** Sie stellen Information für den Endbenutzer dar und sammeln Information vom Endbenutzer. Sie bieten dem Endbenutzer die Möglichkeit, Geschäftsprozesse anzustoßen. (Benutzerschnittstelle)

Geschäftsprozessdienste *(business services, business processes, BP)***:** Sie treffen Entscheidungen auf der Grundlage der vorhandenen Information und den Geschäftsregeln *(business rules)*, um verschiedene Funktionen auszuführen. Sie sind für die sinnvolle Verknüpfung oder Zerlegung von Information verantwortlich. (Anwendungslogik, Geschäftslogik)

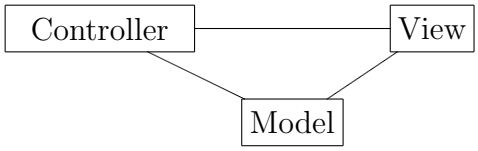
Datendienste *(data services, database, DB)***:** Sie ermöglichen den Zugang zu in Datenbanken gespeicherten Daten. (DB-Server)

5.1.5 Architekturmuster *Model-View-Controler*

Die MVC-Architektur trennt

***Model* (Modell, Repräsentation)**,
***View* (Präsentation, Darstellung)** und
***Controller* (Steuerung, Manipulation)**

von einander, so dass sie unabhängig voneinander entwickelt werden können.



Im Gegensatz zum MVC-Entwurfsmuster fassen wir hier die einzelnen Komponenten des Musters als Teilsysteme auf, die i.a. aus sehr vielen Klassen bestehen. Dabei gibt es grundsätzlich immer nur ein Modell, aber i.a. mehrere Views und Controller. Die Views und die Controller beziehen sich auf dasselbe Modell.

Diese Teilsysteme treten typischerweise folgendermaßen in Wechselwirkung:

1. Ein View verwendet das Modell, um eine Sicht auf das Modell darzustellen.
2. In den meisten Systemen verfügt der View über Eingabemöglichkeiten durch den Benutzer (Buttons, Auswahl- und Eingabefelder). Diese Eingaben werden als Ereignisse an den Controller geschickt.
3. Auf Grund der Ereignisse verändert der Controller eventuell das Modell und/oder möglicherweise auch die Views oder anderen Controller.
4. Wenn sich das Modell geändert hat, werden alle Views (und eventuell alle Controller) davon benachrichtigt.

Vergleicht man mit der Client/Server-Architektur, dann ergeben sich die Entsprechungen:

Model = BP + DB
Controller + View = UI

5.2 Entwurfsentscheidungen

- Welche Entwicklungsumgebung soll verwendet werden?
  Welche Programmiersprachen sollen verwendet werden?
  Was sind die Kodier-Konventionen?
- Wird ein Satz von Entwurfsregeln verwendet? Wenn ja, welcher?
- Welche Klassenbibliotheken sollen verwendet werden?

  Klassenbibliotheken

  - haben sehr viele Schnittstellen und Klassen,
  - sind allgemein anwendbar.
- Einkauf oder Erstellung von Komponenten

  Komponenten

  - haben nur eine, möglichst kleine Schnittstelle,
  - sind allgemein einsetzbar,
  - decken eventuell eine Teilanwendung ab,
  - bestehen aus 20 bis 30 Klassen.

- Einkauf oder Erstellung von Frameworks

  Frameworks (Rahmenwerk) sind leere Anwendungen mit einer Ablauflogik, die mit speziellen Anwendungen ***(plugins)*** gefüllt werden müssen.

  Z.B. ist die MFC von MicroSoft ein Framework. (MFC ist außerdem auch eine Klassenbibliothek.)

- System/Subsystem-Struktur

  Subsysteme

  - haben eine kleine Schnittstelle,
  - decken einen Spezialfall ab und sind nicht allgemein einsetzbar,
  - sind eventuell sehr groß.

- Fehlerbehandlung

  - eigene Fehler
  - Fehler des Anwenders
  - Netzprobleme, Hardwareprobleme, Datenbankprobleme usw.?
    Das sind entweder eigene Fehler oder Fehler des Anwenders abhängig davon, wer die Verantwortung für die Problembereiche hat. Wahrscheinlich sollten sie bis zu einem gewissen Grad behandelt werden.

- Persistenz
- Testen
- Verteilung von Objekten
- Sicherheit
- Benutzeroberfläche
- Systemstart und -ende

5.3 Teamstruktur

Das Architektur-Team besteht aus Architekten, von denen die meisten auch einem Entwicklerteam angehören. Der Architekt vermittelt die Entscheidungen des Architektur-Teams in seinem Entwickler-Team.

Entwickler-Teams werden einzelnen (z.B. UML-)Paketen zugeordnet.

Darüberhinaus gibt es ein Unterstützungs-Team, das die Entwicklungsumgebung pflegt. Auch dieses Team stellt einen Architekten zur Verfügung.

Ferner arbeitet das Architektur-Team sehr eng mit dem Management zusammen.

Kapitel 6

Anforderungsanalyse von Software-Systemen

In diesem Kapitel geht es um die Ermittlung und Spezifikation der **Anforderungen** ***(requirements)*** an das System. Welche Dienste muss das System dem Anwender zur Verfügung stellen ***(service packages)***?

6.1 Problematik

Die Anforderungen können die Form eines Lasten- oder Pflichtenhefts haben. Für die Systementwicklung ist Präzision hier nicht so wichtig, kann sogar hinderlich sein, da dann ein Trend zu speziellen, schlecht modifizierbaren und kaum wiederverwendbaren Lösungen besteht. Allerdings wird die Vertragsgestaltung eine bestimmte Präzision bei den Anforderungen fordern. Hier sind Kompromisse einzugehen.

Der Kunde fühlt sich mit präzisen Anforderungen wohler und der Systementwickler hat es zunächst leichter, weil er nur eine spezielle Lösung zu liefern hat. Aber der Kunde wird selten mit dem Ergebnis zufrieden sein, da es sehr schwierig ist, die eigentlichen Bedürfnisse als präzise Anforderungen zu formulieren. Wenn das System zu speziell ausgelegt ist, werden Anpassungen an die eigentlichen Wünsche des Kunden schwierig, gerade weil man die ursprünglichen, sehr konkreten Wünsche des Kunden möglichst genau realisieren wollte.

Wichtig ist, die *wirklichen* Bedürfnisse ***(desire, wish)*** des Kunden herauszufinden. Das ist aber eine sehr schwierige Aufgabe. Für diese Aufgabe müssten noch Methoden entwickelt werden. Praktisch ist es unmöglich, alle Anforderungen zu ermitteln, weil der Kunde diese oft (noch) nicht kennt. Der Kunde wird immer sogenannte **späte Anforderungen** ***(creeping requirements)*** stellen, also Anforderungen, die zu Beginn des Projekts nicht bekannt sind. Sich schnell ändernde Marktanforderungen machen häufig späte Änderungen erforderlich.

Sich ändernde Anforderungen müssen daher als **zentrale, treibende Kraft** im Entwicklungsprozess begriffen werden.

Insgesamt sollte man also die Anforderungen minimieren, d.h. nur soviele Anforderungen definieren wie eben unbedingt – für Vertragsverhandlungen – notwendig sind, aber nicht mehr.

6.2 Anforderungen

Anforderungen sollten als einfache, kurze, **aktive** Sätze formuliert werden. Das "aktiv" sorgt dafür, dass man dann gleich weiß, *wer* was zu tun hat.

Der Detailierungs-Grad von Anforderungen hängt von vielen Faktoren ab:

- Verfügbarkeit von Experten
- Eigenes Wissen
- Vorgehensmodell. Der Trend geht zu frühem Codieren.
- Preisgestaltung: Wenn ein Festpreis angeboten wird, muss der Detailierungs-Grad zwangsläufig sehr groß sein.

Man muss auf den **Sprachstil** achten:

- **muss** ***(must have)***: Das muss das System bei Auslieferung können.
- **kann** ***(nice to have)***: Das ist optional.
- **wird** ***(will have)***: Das wird in einer nächsten Release realisiert.

Anforderungen:

- Funktionale Anforderungen
- Nicht-Funktionale Anforderungen
  - Qualitätsanforderungen
  - Randbedingungen
- Leistungskriterien
- Kostenrahmen, -schätzung
- Zeitrahmen, -schätzung

6.3 Requirements-Engineering

Eine systematische Durchführung der Anforderungsanalyse ist als Requirements-Engineering bekannt. Hier wird versucht, diese frühe, schwer definierbare Prozess-Komponente der Systementwicklung als Schlüssel-Werkzeug zu etablieren, das das System in einen technischen, wissenschaftlichen und sozialen Zusmmenhang stellt.

Als wichtigste Aufgaben des Requirements-Engineering werden folgende drei Punkte gesehen:

1. Die Interessen des Auftraggebers sollen in eine Vereinbarung über die wichtigsten System-Ziele und System-Einschränkungen münden.

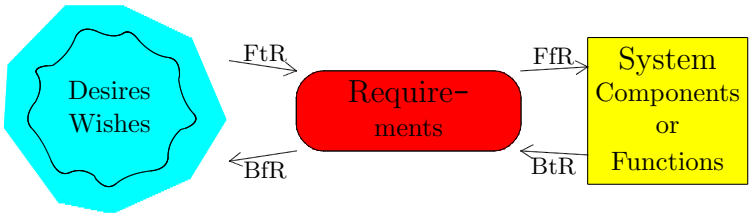
2. Hinreichende Übereinstimmung bezüglich der Realisierung des Systems (Funktionalität, nicht-funktionale Eigenschaften, beabsichtigte und unbeabsichtigte Seiteneffekte) soll erreicht werden.
3. Diese Vereinbarungen sollen in adäquater Form dokumentiert werden, damit sie sowohl für Menschen lesbar sind alsauch der rechnergestützten System-Entwicklung dienen.

Das Resultat ist eine Menge strukturierter, von Auftraggeber *und* Auftragnehmer anerkannter und verstandener, weiterentwickelbarer und gut dokumentierter Anforderungen.

Requirements-Tracebility ist definiert als die Fähigkeit, den Lebenszyklus einer Anforderung zu verfolgen in jeder Richtung während des ganzen System-Lebenszyklus. Es werden vier Pfade unterschieden:

1. ***Forward from requirements:*** Für jede Anforderung muss es möglich sein, die Systemkomponenten zu nennen, die zu ihrer Erfüllung beitragen. Welche Systemkomponenten resultieren aus einer gegebenen Anforderung?
2. ***Backward to requirements:*** Für jede Systemkomponente müssen Anforderungen benennbar sein, die durch sie befriedigt werden. Sogenannte ***gold-plating designs (creeping featurism)*** (Systemkomponenten, für die es keine Anforderung gibt) müssen vermieden werden. Welche Anforderungen werden durch eine gegebene Systemkomponente (partiell) erfüllt?
3. ***Forward to requirements:*** Für jedes "Bedürfnis" des Auftraggebers muss nachweisbar sein, in welchen Anforderungen dessen Befriedigung mündet. Welche Anforderungen resultieren aus einem gegebenen Bedürfnis des Auftraggebers?
4. ***Backward from requirements:*** Für jede Anforderung muss angegeben werden können, welche Bedürfnisse des Auftraggebers durch sie partiell befriedigt werden. Welche Bedürfnisse des Auftraggebers führen zu einer gegebenen Anforderung?

Eine leere Menge als Antwort zu einer dieser Fragen ist ein deutliches Zeichen für schlechtes Design.



6.4 Risikoanalyse

Die Prozesskomponente der Anforderungsanalyse sollte auch die Risiken berücksichtigen, die die Durchführung des Projekts mit sich bringt. Je größer die Risiken, desto stärker müssen sie berücksichtigt werden.

Risiko ist qualitativ das Produkt aus der Wahrscheinlichkeit dafür, dass ein Schadensfall eintritt, und den Kosten, die dieser Schadensfall nach sich zieht.

Risiken kann man in vier Kategorien einteilen:

1. Anforderungsrisiken: Die große Gefahr besteht darin, dass man das falsche System baut. Es leistet nicht, was der Kunde erwartet.
2. Technologische Risiken: Wie gut kann man mit geforderten Technologien umgehen? Stehen die eingeplanten Technologien beim Einsatz des Projekts zur Verfügung bzw. sind sie dann noch kompatibel mit dem entwickelten System? Sind eingesetzte Technologien demnächst obsolet?
3. Risiken der eigenen Fähigkeiten: Kann man das Projekt mit dem vorhandenen oder einzustellenden Personal durchführen?
4. Politische Risiken: Gibt es gesellschafts- oder firmenpolitische Gründe, die das Projekt gefährden können?

Kapitel 7

Analyse eines Software-Systems

Mit der Prozesskomponente **Analyse** ***(analysis)*** wird versucht, den **Problembereich** ***(problem domain)*** (reale Welt des zu entwickelnden Systems) möglichst vollständig zu verstehen.

Dazu gehört die Beschaffung nötigen Fachwissens. Während dieser ganzen Prozesskomponente besteht ein intensiver Kontakt zum Auftraggeber.

Der Schwerpunkt liegt auf dem "Was ist das System und was muss getan werden", nicht auf dem "Wie".

Bildung von Abstraktionen ist eine weitere Aufgabe der Analyse.

Bei objekt-orientiertem Vorgehen wird das Schwergewicht der Arbeit in die Analyse verlegt. Da der Auftraggeber hier noch sehr stark eingebunden ist, dient das der Stabilität und künftigen Erweiterbarkeit der Software.

Grundsätze

- Die Analyse muss unabhängig von der Implementierungsumgebung sein *(implementation environment)*.
- Die Analyse ist anwendungsorientiert.
- Die Analyse ist nicht zu formal.

Die Analyse eines Unternehmens und seiner Geschäftsprozesse ist schwierig. Man muss zu den verschiedensten Mitarbeitern des Unternehmens Kontakt aufnehmen, von den Geschäftsführern bis zu den Hausmeistern und Portiers. Von jedem Mitarbeiter oder Mitarbeitertyp muss man grundlegende Informationen erfragen:

- Was tut das Unternehmen wirklich, um erfolgreich zu sein?
- Was macht das Unternehmen falsch?
- Wie tragen Sie zum Erfolg des Unternehmens bei?
- Was tun Sie dabei?

Man muss vermeiden, sich von der Analyse paralysieren zu lassen *(analysis paralysis)*. Die Gefahr, sich in der Analyse zu verlieren, ist groß. Da man ja gemäß UP die Analyse-Komponente immer wieder aufnimmt, muss man beim ersten Mal nicht alles vollständig oder richtig erfassen.

Wichtigstes Merkmal objekt-orientierter Vorgehensweise ist, *nicht* in erster Linie die Anforderungen oder die konkreten Automatisierungsaufgaben, sondern die Objekte und die Struktur des Systems zu ermitteln. Das hat insgesamt eine wesentlich stabilere Software zur Folge, da die Struktur eines Systems stabiler ist als die konkreten Anforderungen, die sich oft schon während der Entwicklungsphase ändern.

Wenn dagegen der Schwerpunkt auf den Anforderungen liegt, dann entspricht das einem funktionalen Ansatz.

Es ist darauf zu achten, dass keine Design- und Implementierungs-Entscheidungen impliziert werden. Leider ist das häufig Praxis, indem die Verwendung bestimmter Computer oder Sprachen gefordert wird. Diese Anforderungen sollten dann aber explizit gemacht werden.

Werkzeuge und Ergebnisse der Analyse sind:

Brainstorming: Zur Vorbereitung der Analyse wird ein Brainstorming durchgeführt.

Systembeschreibung: Textuelle Beschreibung des Problem-Bereichs und des zu entwickelnden Systems.

Anwendungsfälle: Erfassung von Geschäftsprozessen. Ein Geschäftsprozess besteht oft aus mehreren Anwendungsfällen. Ein Anwendungsfall ist typischerweise innerhalb weniger Minuten abgeschlossen, während ein Geschäftsprozess sich über Wochen ziehen kann.

Aktivitätsdiagramme: Modellierung von Geschäftsprozessen.

Prototyping: Das Prototyping ist ein nützliches Hilfsmittel, um den Auftraggeber einzubinden.

Analyse von Entitäten

7.1 Brainstorming

Zur Vorbereitung der Analyse wird ein Brainstorming durchgeführt.

In einer Teambesprechung werden innerhalb einer festgelegten Zeitdauer (etwa 30 Minuten) sämtliche Ideen der Teammitglieder ohne Rücksicht auf Qualität, Durchführbarkeit, Kosten, Teamstatus usw. gesammelt.

7.2 Systembeschreibung

Die Systembeschreibung ist eine im wesentlichen **textuelle** Darstellung der Ergebnisse der Analyse. Sie beschreibt die Komponenten des Systems für den Kunden *und* den Systementwickler verständlich und möglichst präzise.

Die Systembeschreibung enthält Protokolle für die Interaktion mit externen Systemen. Es wird die **Umgebung** ***(environment)*** analysiert und eventuell definiert.

Die Systembeschreibung enthält auch Anforderungen. Diese sind wahrscheinlich zunächst mehrdeutig, unvollständig oder widersprüchlich oder sind Anforderungen, die nur mit unvertretbarem Aufwand zu erfüllen sind. Da diese Mängel häufig erst während der Entwicklung eines Systems erkannt werden, muss die Systembeschreibung immer wieder angepasst werden. Sie stellt *kein* abgeschlossenes Dokument dar.

In einem Anhang zur Systembeschreibung sind alle benötigten Fachkenntnisse über das zu entwickelnde System aufzuführen. Dieser Teil kann während der weiteren Entwicklung laufend ergänzt werden.

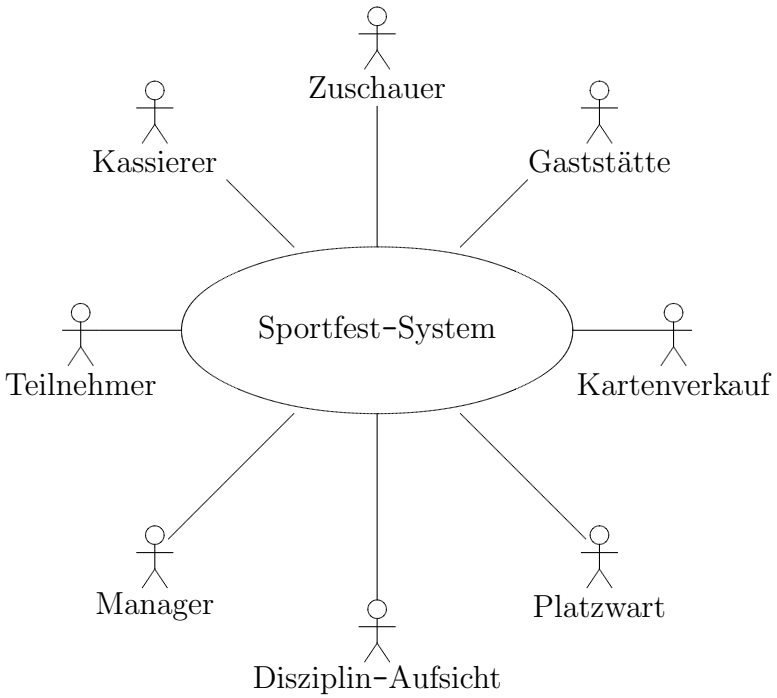
Eine große Gefahr dieser Phase ist, zu früh Detailfestlegungen oder Strukturen einzuführen. Die Systembeschreibung sollte so unvoreingenommen wie nur möglich sein. Flexibilität ist wichtig.

7.3 Anwendungsfälle

Use-Cases sollten keine Funktionsdiagramme von internen Funktionen sein.

Anwendungsfälle sollten etwas für den oder die Aktoren tun.

Ein System kann sehr viele Anwendungsfälle haben. Da kann es sinnvoll sein, sich ein Überblicks-Diagramm ***(overview business use case)*** zu erstellen.



7.4 Aktivitätsdiagramme

Mit Aktivitätsdiagrammen können Abläufe und parallele Prozesse gut dargestellt werden.

7.5 Prototyping

Das Prototyping ist ein nützliches Hilfsmittel, um den Auftraggeber einzubinden.

7.6 Analyse von Entitäten

- **Identifikation von Entitäten**
  Die das System konstituierenden Entitäten (Objekte und Klassen) werden an Hand der Systembeschreibung identifiziert. Substantive des Textes sind gute Kandidaten für Entitäten. Sie werden aufgelistet.

  Zunächst werden *alle* Substantive in eine Liste geschrieben. Dann wird diese Liste iterativ verbessert, indem sie geordnet wird und indem Synonyme, Homonyme, Tautologien zu einem möglichst qualifizierten Begriff konsolidiert werden.

  CRC-Karten ***(class responsibility collaboration card)*** sind ein praktisches Hilfsmittel. Jede Karte beschreibt eine Klasse (Vorderseite Klassenname, links Verantwortlichkeiten, rechts Kollaborateure, Rückseite Attribute und textuelle Beschreibung).

  Weitere Hilfsmittel sind die Analyse von Szenarien und natürlich das Fachwissen.

- **Identifikation von Attributen**
  Eigenschaften von Substantiven sind gute Kandidaten für Attribute.

- **Identifikation von Verhalten**
  Dienste und Verantwortlichkeiten, Verben des Textes sind gute Kandidaten für Verhalten.

  Verantwortlichkeiten sind die Botschaften, auf die ein Objekt reagiert.

- **Identifikation von Beziehungen und Kollaborationen**
  Identifiziere Kollaborationen: Kollaborateure sind andere Klassen, die mit einer Klasse interagieren. Es liegt eine Kollaboration vor, wenn eine Verantwortlichkeit andere Klassen benötigt.

  Folgende Beziehungen zwischen Entitäten sollten identifiziert werden:

  - Erweiterung, Vererbung (`is-a`, "Ist-ein", "Substituiert-ein", "Ist-Subtyp-von")
  - Kategorie, Rollen
  - Ähnlichkeit (`is-like-a`, "Ist-fast-ein")
  - Aggregation, Komposition (`has-a`, "Hat-ein", "Hat-als-Teil-ein-oder-viele")
  - Benutzung (`uses-a`, "Benutzt-ein", Client/Server – Beziehung)
  - andere Beziehung, Assoziation

- **Anwendungsfälle *(use-case)***: Verwende die CRC-Karten, um Anwendungsfälle durchzuspielen. Dabei ergeben sich weitere Kollaborationen und Verantwortlichkeiten. Einige werden sich auch als überflüssig erweisen. Die Anwendungsfälle können in Kollaborationsdiagrammen dargestellt werden.

  Verben und Tätigkeiten geben Hinweise auf Anwendungsfälle.

  Oft ist es nützlich, Anwendungsfälle schon nach der Identifikation der Objekte bzw. Klassen durchzusprechen, da dabei sich notwendigerweise Methoden und Datenelemente ergeben. Insgesamt sollte das Vorgehen natürlich iterativ sein.

Kapitel 8

Design eines Software-Systems

Das in der Analyse gesammelte Material wird nun zur Konstruktion des Systems verwendet. Dabei **müssen** Entscheidungen getroffen werden. Allerdings bedeutet jede Design-Entscheidung eine Einschränkung der Möglichkeiten. Je früher sie gefällt wird, desto einschneidender ist sie, desto schwerer ist sie rückgängig zu machen.

8.1 Allgemeine Design-Regeln

- Lokalisiere Botschaften-Austausch.
  "Beschäftige Dich nur mit dem Nachbarn, den Du gut kennst."
- Vermeide Vermischung von Problembereichen. Oft werden z.B. folgende drei Bereiche oder Layers unterschieden: Benutzerschnittstelle (UI), Datenspeicherung (DB) und Geschäftsprozesse (BP).
  "Divide et impera." (Teile und herrsche.)
- Isoliere Teile, die eine Tendenz zu Änderbarkeit zeigen. Trenne Standardverhalten von speziellen Anforderungen.
- Bilde einfache Schnittstellen.
- Balanciere zwischen Generalisierung und Spezialisierung. Für den Benutzer ist es häufig schwierig, ganz allgemeine Module zu benutzen, da er dort sehr viele Parameter einzustellen hat. Wenn es allerdings nur spezialisierte Klassen gibt, dann erstickt der Benutzer möglicherweise in einem unübersichtlichen Wust von Klassen.

  Beispiel: `Pizza (Groesse :int)` oder `Piccolo ()`, `Standard ()`, `Grande ()`?
- "Ziehe einem Wald den Baum vor." Strukturiere hierarchisch.
- Minimiere Kollaborationen zwischen Klassen, indem eventuell eine Klasse in mehrere Klassen aufgeteilt wird oder eine zusätzliche Klasse geschrieben wird.
  Durch die Einführung eines "Software-Busses" kann die Kollaboration zwischen Klassen eventuell drastisch reduziert werden.
- Vorsicht bei Einzelfällen. Sei auf multiple Fälle vorbereitet.

8.2 Design von Klassen

8.2.1 Fallen beim Klassenentwurf

- **Daten-Lager-Falle** ***(data warehouse trap)***: Eine Klasse ist kein Behälter beliebiger Daten, die dem ganzen Programm zur Verfügung stehen. Die Datenelemente einer Klasse sollten nur über die Objekte der Klasse etwas aussagen. Mit einem Datenlager wird man dieselben Probleme haben wie mit globalen Variablen.
- **Phantom-Objekt-Falle** ***(spectral object trap)***: Eine Klasse ist nicht nur eine Ansammlung von Methoden, die eventuell wenig mit Objekten der Klasse zu tun haben. Insbesondere wenn viele Methoden `static` sind, hat man einen prozeduralen Modul. Klassen ohne Daten sind "Phantome".
- **Mehrfache** oder **gespaltene** Persönlichkeitsstruktur der Klassen ***(multiple personality trap)***: Eine Klasse sollte nur **eine** Abstraktion modellieren. Das ist das Prinzip der **Kohäsion** ***(cohesion)***. Ein komplexer Name für die Klasse ist oft ein Hinweis auf ein Kohäsions-Problem.

8.2.2 Klassen-Entwicklung

Möglicherweise gehört dieser Abschnitt in die Implementierung.

1. Eine Abstraktion aus der Sicht des Kunden bzw. der BenuzerIn der Klasse ist ein guter Startpunkt. Daher sollte man mit dem Entwurf einer minimalen öffentlichen Schnittstelle beginnen. Man muss untersuchen, wie die Objekte der Klasse benutzt werden.
2. Für die Methoden sollten zunächst **Rümpfe (Stümpfe,** ***stub)*** geschrieben werden, d.h. leere Methoden, die eventuell nur Text – etwa eine textuelle Formulierung der durchzuführenden Schritte – ausgeben, die aber die korrekten Argumente und Rückgabetypen haben.
3. Überprüfung der Schnittstelle:
   - Repräsentiert die Klasse eine einzige Abstraktion bzw. Entität?
   - Repräsentiert jede Methode eine einzige Operation?
   - Tut die Klasse alles, was sie muss und nicht mehr? Man sollte die Schnittstelle nicht mit unbenutzten Methoden überladen.
   - Ist die Schnittstelle leicht zu benutzen?
     - Sind die Namen der Methoden und ihrer Argumente "sprechend" gewählt?
     - Lange Argumentlisten verlangen nach einer Vereinfachung, d.h. eventuell nach neuen Klassen.
   - Schreibe Methoden zur Fehlersuche, insbesondere Methoden, die den Zustand eines Objekts ausgeben.
4. Konstruktoren und Initialisatoren sollten so geschrieben sein, dass alle Felder initialisiert werden und Klasseninvarianten erhalten bleiben. Es dürfen nur legale Objekte erzeugt werden. Der **Arbeitskonstruktor** ***(working constructor)*** ist der Konstruktor, der den Zustand einer Instanz der Klasse i.a. vollständig definiert. Er sollte zuerst geschrieben werden. Andere Konstruktoren sollten diesen Konstruktor verwenden.

5. **Mutatoren** ***(mutator)*** sind Methoden, die Datenwerte eines Objekts, d.h. seinen Zustand ändern. Sie repräsentieren das interessante Verhalten des Objekts und sollten das Objekt in einem konsistenten Zustand hinterlassen.

6. **Akzessoren** ***(accessor)*** sind Methoden, die Informationen über das Objekt liefern. Sie dürfen den Zustand des Objekts nicht verändern.

7. In Java sollte die Methode `toString ()` überschrieben oder – besser noch – eine Methode `display ()` geschrieben werden. Dabei sollte auch `super.toString ()` bzw. `super.display ()` aufgerufen werden (bei Klasse `Object` schließlich `super.toString ()`). Die Methode sollte so geschrieben sein, dass sie den Zustand eines Objekts in lesbarer Form ausgibt.

8. **Fehlerbehandlungskonzepte:**

   - **Fehlercodes:** Alle Operationen, bei denen Fehler auftreten können, geben einen Fehlercode als Ergebnis zurück. Anwender der Operation müssen nach jedem Aufruf der Operation prüfen, ob die Operation mit Erfolg durchgeführt wurde.

     Vorteile:

     – Einfaches Konzept
     – Kann mit jeder Programmiersprache verwirklicht werden.
     – Funktioniert auch in verteilten Systemen.

     Nachteile:

     – Enge Verflechtung von normalem Code und Fehlerbehandlungscode, da die Fehlerüberprüfung nach jedem normalen Aufruf erfolgen muss.
     – Aufwendige Wartung: Überprüfungen können vergessen werden. Die Einführung neuer Fehlercodes führt zur Anpassung aller Überprüfungen.
     – Bei Fehlerbehandlung auf höherer Ebene müssen die Fehlercodes über viele Stufen durchgereicht werden.

   - **Ausnahmen:** Moderne Systeme haben ein sogenanntes Exception-Handling, wobei es im Ausnahme-Fall (= Fehler-Fall) zu einem Code-Abbruch und Werfen eines Ausnahme-Objekts kommt. Diese Objekte können gefangen werden und dadurch eine Fehlerbehandlung ermöglichen.

     Vorteile:

     – Normaler Code und Ausnahmebehandlung lassen sich leichter entkoppeln.
     – Man kann Ausnahmen nicht ignorieren. Sie müssen irgendwo behandelt werden.
     – Zwischenebenen können leicht übersprungen werden, wenn der Fehler nur auf einer höheren Ebene behandelt werden kann.
     – Neue Ausnahmen können durch Erweiterung der Ausnahmehierarchie eingeführt werden, ohne dass Behandlungsroutinen angepasst werden müssen.

     Nachteil: Ausnahme-Behandlung muss implementierbar sein.

     Verwendung von Ausnahmen:

     Das Werfen eines Ausnahme-Objekts ist eine sehr teure Operation. Daher sollte Ausnahme-Behandlung nicht bei Fehlern verwendet werden, die erwartet werden. (Z.B. Abbruch von Schleifenoperationen oder Division durch Null). Programmiere so, dass Ausnahmen eigentlich nie auftreten.

Der normale Gebrauch einer Schnittstelle sollte nicht zu einer Ausnahme führen, sondern nur, wenn gegen den "Vertrag" einer Schnittstelle verstoßen wird.

Ausnahmen werden in einer Erweiterungshierachie programmiert. Es ist in den seltensten Fällen sinnvoll, einen eigenen Hierarchiebaum aufzubauen, wobei dann eventuell vordefinierte Ausnahmen einer Programmierumgebung umgemapped werden. Stattdessen sollte man versuchen, die eigenen Ausnahmen in die Hierarchie der Programmierumgebung an geeigneter Stelle nach **Art** des Fehlers einzuhängen.

Komponenten und Bibliotheken sollten Ausnahmen nur melden, aber keine Strategie vorgeben, wie sie zu behandeln sind.

9. Fehlerbehandlung – Möglichkeiten:
   - Die Möglichkeit der fehlerhaften Verwendung der Klasse wird ignoriert.
   - Ausdruck einer Fehlernachricht und Stoppen des Programms.
   - Fehler wird korrigiert und Ausdruck einer Fehlernachricht. Programm läuft weiter.
   - Eine Exception wird geworfen. Damit wird der Benutzer gezwungen, den Fehler zu behandeln. Das dürfte in den meisten Fällen das Vernünftigste sein. Damit kann die oberste Ebene auf Fehler reagieren und z.B. versuchen, den Benutzer zu besänftigen.

   Man sollte das "Entdecken" und das "Behandeln" eines Fehlers unterscheiden. Ein Fehler sollte so früh wie möglich entdeckt werden.

   Wenn man etwas gegen einen Fehler tun kann, dann sollte man ihn lokal behandeln. Wenn das nicht möglich ist, sollte er gemeldet werden. Auf jeden Fall muss man dafür sorgen, dass beteiligte Objekte konsistent gehalten werden.

   Fehler, gegen die man nichts tun kann, sollten an *einer* zentralen Stelle behandelt werden.

   Man sollte nur die Ausnahmen beheben, die man gut kennt. Das sind i.a. nur Fehler des Benutzers. Diese sollten dann auch wirklich behoben werden.

   Faustregel: Das Programm sollte in der am wenigsten überraschenden Weise reagieren.

8.3 Design von Beziehungen

Alle Arten von **Beziehungen *(relationship)*** kann man in folgendem Schema sehen:

- ***is-a***, Erweiterung, Vererbung *(extension, inheritance)*
- Kategorie
- ***has-a*** Ganzes-Teil-Beziehung *(whole-part-relationship)*, Komponentengruppe-Komponenten-Beziehung
  - Komposition *(composition)*
  - Aggregation *(aggregation)*
- ***uses-a***, Benutzung *(usage)*
- Assoziation *(association)*
- Abhängigkeit (zeitlich begrenzte Beziehung, die nicht persistent ist.)

8.3.1 Erweiterung, Vererbung

Bei der Vererbung werden drei Aspekte unterschieden:

- **Erweiterung *(extension)***: Eine Subklasse erweitert eine Superklasse um neue Klassenelemente. Alle Klassenelemente der Superklasse stehen weiterhin zur Verfügung.
  - Es muss das Substitionsprinzip von Barbara Liskov gelten: Objekte der erweiterten Klasse müssen als Objekte der Basisklasse verwendbar sein.
  - Die Menge der Objekte der Subklasse ist eine Teilmenge der Menge der Objekte der Superklasse.
  - Ein Objekt der Subklasse **ist ein *(is a)*** Objekt der Superklasse.

  Eine strenge Anwendung des Begriffs Erweiterung (Schnittstelle *und* Implementierung werden **zwingend *(mandatory)*** geerbt.) würde bedeuten, dass die Methoden der Basisklasse nicht verändert werden dürfen und nur neue Methoden hinzugefügt werden dürfen. Zur Implementierung müsste man dazu alle Methoden der Basisklasse `final` (Java) deklarieren.
- **Spezifikation *(specification)***: Eine Superklasse oder eine Schnittstelle spezifiziert Verantwortlichkeiten (Methoden) ohne sie zu implementieren. Die Subklasse erbt nur die Spezifikation einer Methode und implementiert die Methode selbst.

  Spezifikation in der reinen Form bedeutet, dass die Basisklasse als `interface` deklariert wird.
- **Polymorphe Vererbung *(polymorphic inheritance)***: Normalerweise hat man es mit einer Kombination von Erweiterung und Spezifikation zu tun. Die Subklasse erbt die Spezifikation und eine Default Implementierung einer Methode. Die Subklasse kann die Implementierung der Methode überschreiben.

  Die polymorphe Vererbung ist der allgemeine Fall mit allen Möglichkeiten (Erweiterung, Spezifikation und Polymorphie):
  - `final` (Java) deklarierte Methoden *dürfen nicht* neu implementiert oder modifiziert werden (zwingende Schnittstelle und zwingende (d.h. unveränderliche) Implementierung).
  - `abstract` (Java) deklarierte Methoden der Basisklasse *müssen* implementiert werden (zwingende Schnittstelle, die implementiert werden muss).
  - Alle anderen Methoden *können* modifiziert oder überschrieben werden (zwingende Schnittstelle und optionale Implementierung).

Wie entdeckt man Erweiterungsbeziehungen:

- **Generalisierung *(generalization, bottom-up)***: Objekte haben gemeinsame Daten und/oder gemeinsames Verhalten. Entweder sind das dann Objekte einer Klasse oder ihre Klassen können von einer die Gemeinsamkeiten repräsentierenden Klasse erben.
- **Spezialisierung *(specialization, top-down)***: Die Implementierung einiger Methoden einer Klasse erfordert Fallunterscheidungen (`switch`-Statement oder `if-else if`-Konstrukte). Das ist ein Hinweis darauf, dass man spezialisierte Klassen anlegen sollte, die von der allgemeinen Klasse erben.

Besonders problematisch ist es, wenn die Fallunterscheidung auf einem Attribut einer anderen Klasse beruht. Das erfordert eine größere Design-Korrektur.

Vererbung ist richtig eingesetzt, wenn das Liskov'sche Substitutionsprinzip gilt. Ein häufig gemachter Fehler ist die **Kontraktion *(contraction)***, wobei man von einer Klasse A erbt, nur weil die Klasse A sehr große Ähnlichkeit hat. Dabei wird man die Klasse A häufig *kontrahieren*, indem man nicht benötigte Methoden leer überschreibt. Wenn das Liskov'sche Substitutionsprinzip nicht erfüllt ist, sollte man in solch einem Fall entweder eine gemeinsame Basisklasse finden oder eine Referenz auf ein Objekt der Klasse A anlegen und die von A verwendbaren Methoden emulieren.

8.3.2 Kategorie

Eine Klasse ist eine Kategorie, wenn sie aus einer Menge von Klassen *eine* erweitert bzw. erbt. Z.B. betrachten wir die Menge der Klassen { `Sachbearbeiter`, `Ingenieur`, `Hausmeister`, `Programmierer` usw }. Die Klasse `Chormitglied` wäre eine Kategorie, wenn ein Chormitglied entweder ein Sachbearbeiter, Ingenieur, Hausmeister oder Programmierer sein kann.

8.3.3 Aggregation und Komposition

Aggregation und Komposition stellen Ganzes-Teil-Beziehungen (Hat-ein, Ist-Teil-von, enthält, *whole-part, has-a, part-of, contains)* dar.

Der Unterschied zwischen Aggregation, Komposition einerseits und anderen Beziehungen wie Uses-a und Assoziation besteht darin, dass

- jedes Teil nur zu *höchstens einem* Ganzen gehört.
- Ferner gilt bei der Komposition, dass die Teile mit dem Ganzen leben und sterben.

Oder etwas genauer:

- Bei Komposition überlebt das Teil das Ganze nicht. Das Teil muss zu einem Ganzen gehören. Allerdings kann auch das Ganze ohne das Teil nicht leben bzw. ist ohne das Teil stark beeinträchtigt.
- Bei der Aggregation kann das Teil eine Existenz unabhängig von irgendeinem Ganzen haben. Das Ganze kann ohne das Teil nicht voll funktionsfähig sein.

  Das Teil wird nur von *höchstens einem* Ganzen benützt (Unterschied zu *uses-a*). Das Teil kann aber von einem Ganzen zu einem anderen Ganzen transferiert werden.

8.3.4 Benutzung

Das benutzte Objekt kann normalerweise von vielen benutzt werden. Das benutzte Objekt existiert unabhängig von den Benutzern.

8.3.5 Assoziation

Assoziationen sind alle anderen Beziehungen, die nicht eine der bisher genannten Beziehungen sind. Sie werden häufig als eigene Klassen implementiert. Die Lebensdauer der Objekte dieser Klassen hängt oft von der Existenz der Beziehungspartner ab. Diese Klassen sind sogenannte "schwache Entitäten".

8.3.6 Abhängigkeit

Eine Abhängigkeit ist eine zeitlich begrenzte Beziehung, die nicht persistent ist.

8.4 Design-Empfehlungen

Im folgenden listen wir Design-Empfehlungen oder -Regeln auf, die man beachten sollte, aber nicht muss. Die Erfüllung einer Regel hat immer ihren Preis, und manchmal mag sich das nicht auszahlen. Oft muss man sich eben entscheiden, ob man eine schnelle, praktische, statische, kurzfristige Lösung möchte, oder ob man eine Lösung will, die dynamisch an die sich ändernden Anforderungen angepasst werden kann und auf langfristigen Einsatz ausgelegt ist.

***Keep it DRY** (Don't repeat yourself!)*: Redundanz bei Code und Daten muss unbedingt vermieden werden.

Law of Demeter: Die Methode `m` einer Klasse `A` darf nur Botschaften (d.h. Methoden aufrufen von) an

- Instanzen von `A`
- Datenelemente von `A`
- Argumente von `m`

senden. Die Methode `m` sollte also nicht ein Datenelement einer referenzierten Klasse verwenden, um dafür etwa Methoden aufzurufen. Eine Methode kennt eigentlich nur die eigene Klasse. Eine Methode darf jede direkte Assoziation traversieren, aber keine indirekten Assoziationen. (Das sollte auch nicht mit `getObject ()`-Methoden umgangen werden.)

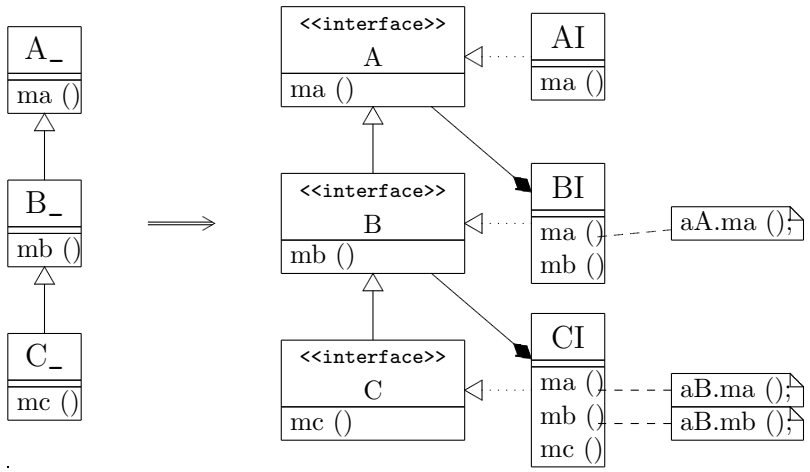
Bei Beachtung dieser Regel sind die Methoden dann weniger stark von Änderungen des Assoziations-Netzwerks betroffen.

Ferner sollte man auf das Resultat einer Methode nicht wieder eine Methode anwenden, es sei denn, dass das Resultat schon als Argument der Methode oder als Datenelement der eigenen Klasse bekannt ist, oder vom Typ einer low-level Bibliotheks-Klasse ist. Stattdessen sollte man eine neue, kombinierte Methode anbieten.

Ziehe Komposition der Vererbung vor: ***(Prefer composition to inheritance.)***

Komposition oder auch **Delegation** ist flexibler als Vererbung.

Übung: Diskutiere Polymorphismus bei Komposition. Wenn es da Probleme gibt, was muss man dann tun? Wie steht es mit Mehrfachvererbung?



Fluss *(flow)*: Objekte der Klasse A schicken Botschaften an Objekte der Klasse B, aber nicht umgekehrt.

Wenn sich wechselseitiger Botschaftenaustausch nicht vermeiden lässt, dann sollte man wenigstens prüfen, ob nicht für eine Richtung das Observer-Pattern einsetzbar ist.

Observer-Pattern: Wenn ein Objekt aktualisiert wird, dann soll es immer seine Beobachter benachrichtigen.

Logical Memory Leaks: In Java, `C#` und Go sorgt der Garbage Collector dafür, dass keine normalen Speicherlecks auftreten, d.h. nicht mehr benötigter Speicher wird freigegeben. Der GC räumt aber nur das auf, was nicht mehr referenziert wird. Es kann nun aber sein, dass man Objekte unabsichtlich unendlich lange referenziert, ohne sie noch zu benutzen. Das führt zu sogenannten **logischen** Speicherlecks. Das passiert leicht bei globalen Objekten, mit denen man daher in dieser Hinsicht vorsichtig umgehen muss.

Freigabe von Ressourcen: Nach Gebrauch sollten Ressourcen wieder freigegeben werden. Um das zu garantieren, empfiehlt sich hier in Java und `C#` ein `try-finally`-Konstrukt:

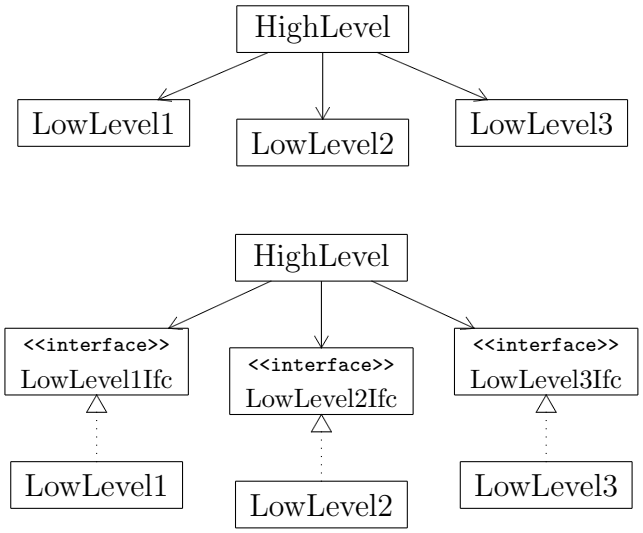
```
try
   {
   // Verwende Ressource R
   }
   finally
      {
      R.dispose ();
      }
```

In `C#` kann das über die Realisierung der Schnittstelle `IDisposable` mit einem `using`-Konstrukt abgekürzt werden.

Programmiere immer gegen Schnittstellen! ***Dependency Inversion Principle:*** *Depend upon abstractions. Do not depend upon concrete classes.*

"Inversion" betrifft den Fall, dass eine high-level Komponente auf dem Verhalten von low-level Komponenten beruht. Wenn jetzt eine Schnittstelle oder abstrakte Klasse zwischen

diese Komponenten gelegt wird, dann hängen nicht nur die high-level Komponente, sondern auch die low-level Komponenten von dieser Schnittstelle ab. In letzterem Fall hat sich also die Abhängigkeit umgekehrt, invertiert.



Für die drei low-level Klassen haben sich die Pfeilrichtungen (Abhängigkeiten) umgekehrt.

Die Entwurfsmuster Abstract Factory und Factory Method lösen dieses Problem, indem jeweils abstrakte Klassen oder Schnittstellen für die Kopplung zwischen high- und low-level Klassen verwendet werden.

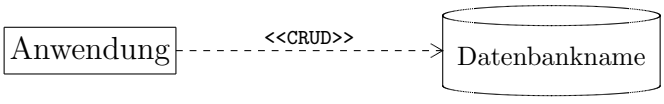
Konfigurieren statt Programmieren? ***(Configuration over Programming)***

Beziehungen, Zugehörigkeiten, Programmabläufe werden in einer Konfigurations-Datei spezifiziert.

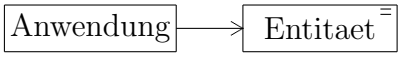
Konvention statt Konfiguration? ***(Convention over Configuration)***

Beziehungen und Zugehörigkeiten werden über Namenskonventionen hergestellt. Auch Programmabläufe können so gesteuert werden.

Notation Datenbankanbindung: Beinahe jede Software greift auf eine Datenbank zu. Daher ist es eigentlich unnötig, dies besonders zu modellieren. Im einfachsten Fall kann man einfach ein Datenbank-Symbol in das Diagramm stellen und eventuell mit einer oder mehreren Klassen über eine Abhängigkeit `<<CRUD>>` verbinden.



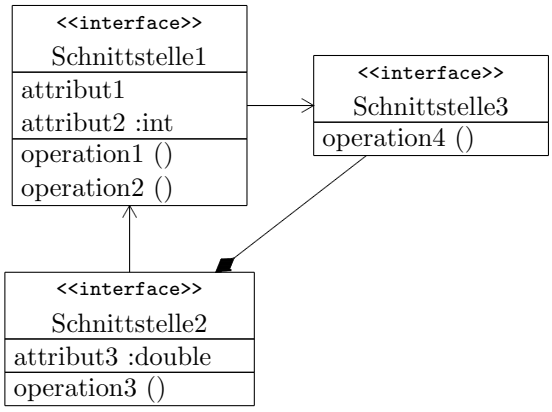
Um persistente von transienten Klassen zu unterscheiden, kann man für persistente Klassen einen besonderen Stereotyp kreieren:



Schnittstellen mit Attributen: Seit UML 2 dürfen Schnittstellen auch Attribute haben. Damit können sie auch aktiv Beziehungen zu anderen Klassen und Schnittstellen aufnehmen. Mit diesen Schnittstellen kann man zunächst ein sehr abstraktes Design erstellen, das alle Beziehungen enthält.

Insbesondere kann eine Klasse mehr als eine Schnittstelle mit Attributen implementieren, allerdings mit den Problemen der Mehrfachvererbung.

In Java ist die Implementierung einer solchen Schnittstelle auf elegante Weise nicht möglich (Diskussion siehe UML-Skriptum Kapitel Klassendiagramm – Schnittstelle).



8.5 Beispiel Konferenz-Terminplan-System [18]

8.5.1 Systembeschreibung

Das Konferenz-Terminplan-System bietet Mitarbeitern die Möglichkeit, ihre Terminpläne zu führen und Konferenzen mit anderen Mitarbeitern terminlich abzustimmen. Das System verwaltet auch Feiertage. Konferenzen finden an einem Ort statt.

Ein Mitarbeiter kann einen Standard-Terminplan eingeben, so dass der Plan für eine bestimmte Woche auf diesem Standard-Terminplan basiert, der an die wöchentlichen Gegebenheiten angepasst wird.

8.5.2 Identifikation von Objekten und Klassen

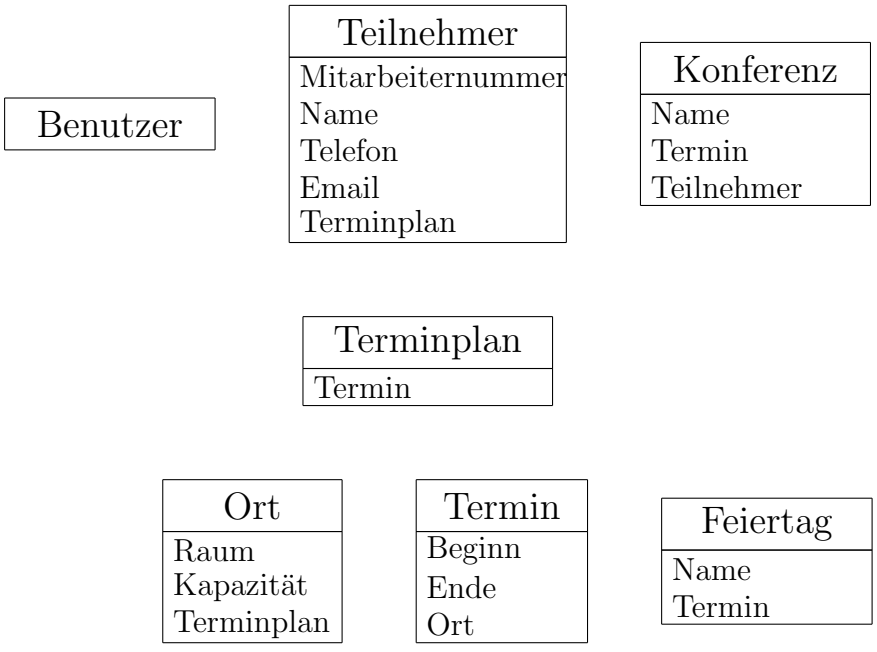
Aus der Problembeschreibung werden zunächst Substantive und Verben oder ähnliche Ausdrücken extrahiert.

Substantivische Ausdrücke	Tätigkeiten, Verben
Konferenz-Terminplan-System	bietet Möglichkeit
Mitarbeiter	Terminplan führen
Terminplan	
Konferenz	terminlich abstimmen
System	
Feiertag	verwaltet Feiertage
Standard-Terminplan	Standard-Terminplan eingeben
Gegebenheit	Standard-Terminplan anpassen
Ort	
Plan	
Woche	

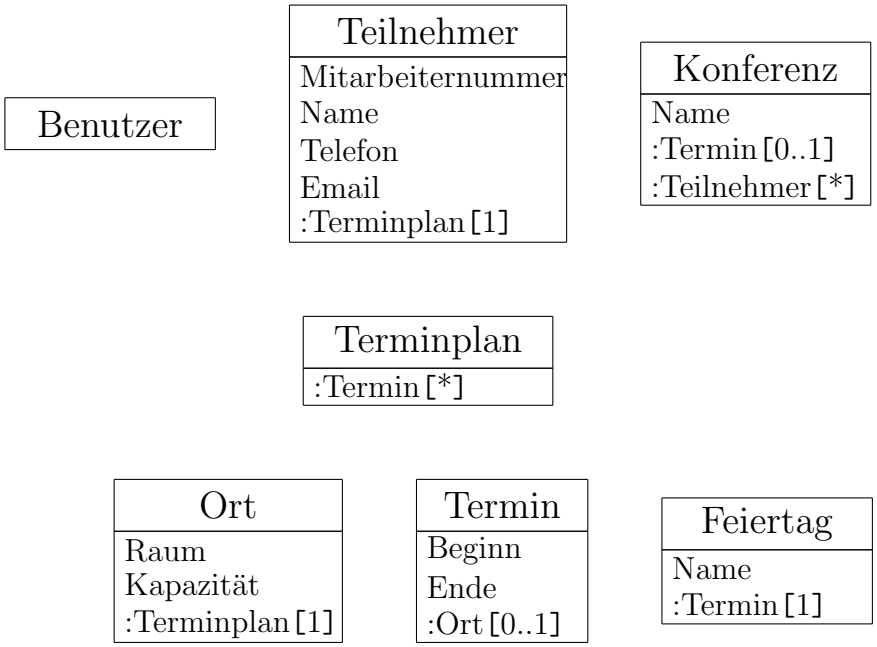
Von den Substantiven werden zunächst folgende als Klassen behalten:

- Benutzer (anstatt System)
- Teilnehmer (anstatt von Mitarbeiter)
- Konferenz
- Feiertag
- Termin
- Terminplan (Besteht aus einer Menge von Terminen)
- Ort

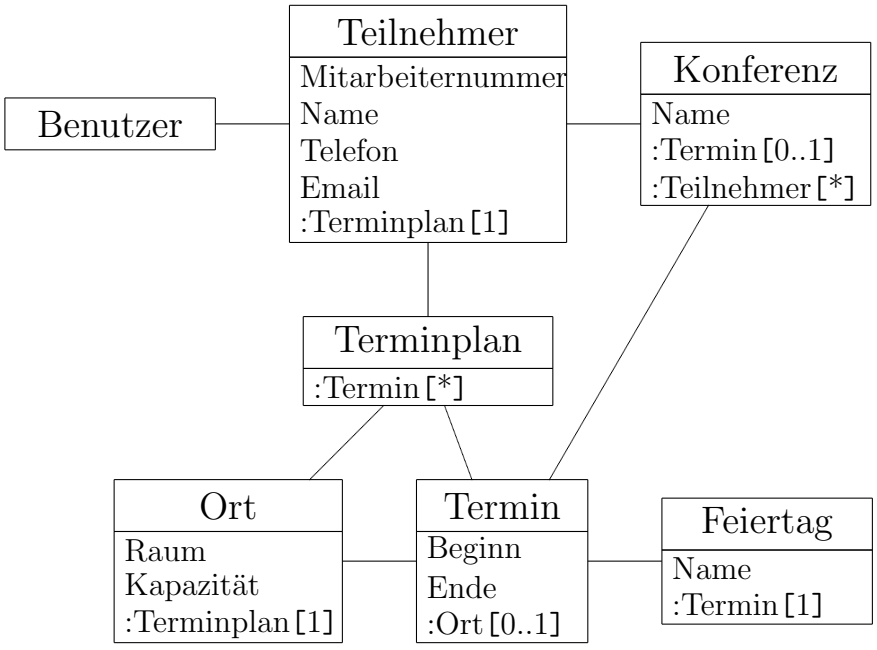
Jetzt werden die Attribute ermittelt und Klassen zugeordnet. Das ergibt:



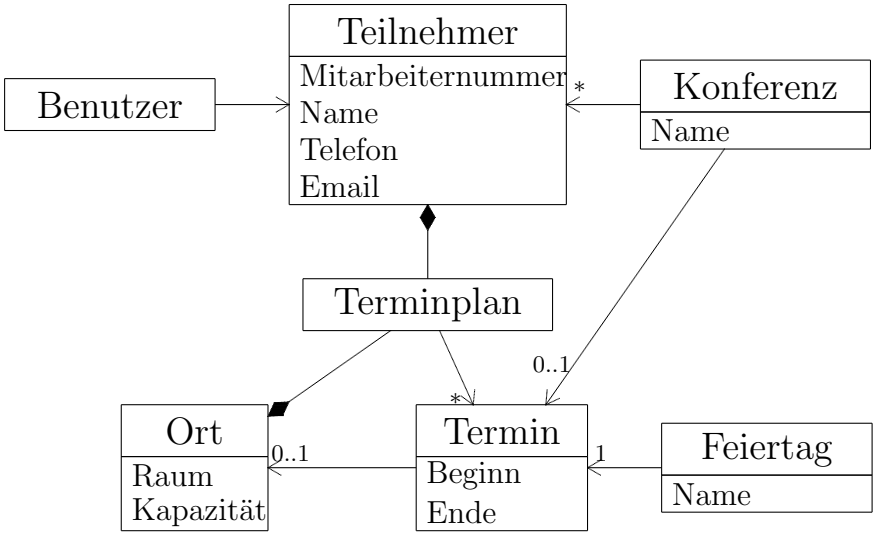
Iteration 1: Multiplizitäten der Attribute



Iteration 2: Kollaborationen



Iteration 3: Beziehungen



8.5.3 Identifikation von Verantwortlichkeiten

Dieser Punkt wird verschoben in der Hoffnung, dass die Anwendungsfälle Verantwortlichkeiten liefern.

8.5.4 Identifikation von Kollaborationen

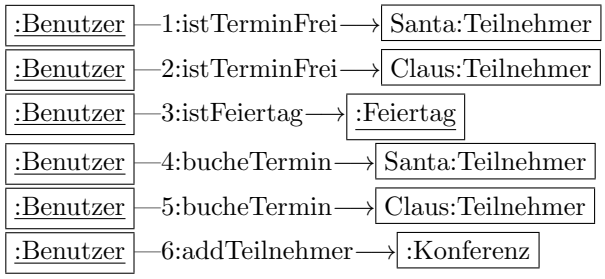
Dieser Punkt wird verschoben in der Hoffnung, dass die Anwendungsfälle Kollaborationen liefern.

8.5.5 Durchspielen von Anwendungsfällen

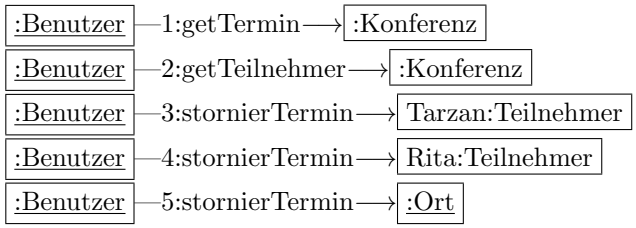
Erstelle von den Klassen CRC-Karten, um Anwendungsfälle durchzuspielen.

1. Fall: Tarzan möchte mit Rita einen Termin an einem bestimmten Ort vereinbaren.
   - :Benutzer —1:istTerminFrei⟶ Tarzan:Teilnehmer
   - :Benutzer —2:istTerminFrei⟶ Rita:Teilnehmer
   - :Benutzer —3:istTerminFrei⟶ :Ort
   - :Benutzer —4:bucheTermin⟶ Tarzan:Teilnehmer
   - :Benutzer —5:bucheTermin⟶ Rita:Teilnehmer
   - :Benutzer —6:bucheTermin⟶ :Ort
   - :Benutzer —7:addTeilnehmer⟶ :Konferenz

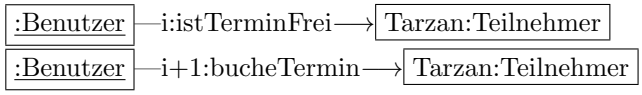
2. Fall: Herr Santa und Frau Claus wollen sich an einem bestimmten Feiertag treffen. Ort ist egal.



3. Fall: Tarzan möchte die Konferenz mit Rita absagen.



4. Fall: Tarzan möchte einen Standard-Terminplan eingeben.



Die CRC-Karten werden mit den Verantwortlichkeiten, die sich aus den Anwendungsfällen ergeben, ergänzt. Die CRC-Karten sehen dann folgendermaßen aus:

Class: Benutzer	
Verantwortlichkeiten:	Kollaborationen:
vereinbareKonferenz	Teilnehmer, Ort, Feiertag
sageKonferenzAb	Teilnehmer, Ort
standardTerminplan	Teilnehmer, Ort

Benutzer: Repräsentiert den Mitarbeiter der mit dem System arbeitet.

Attribute:

Class: Teilnehmer	
Verantwortlichkeiten:	Kollaborationen:
istTerminFrei	Terminplan
stornierTermin	Terminplan
bucheTermin	Terminplan

Teilnehmer: Repräsentiert den Mitarbeiter der an Konferenzen teilnimmt.

Attribute:

Mitarbeiternummer
Name
Telephon
Email
Terminplan

Class: Konferenz	
Verantwortlichkeiten:	Kollaborationen:
getTermin addTeilnehmer getTeilnehmer	

Konferenz: Repräsentiert eine Konferenz.
Attribute:
Name Termin Teilnehmer

Class: Feiertag	
Verantwortlichkeiten:	Kollaborationen:
istFeiertag	

Feiertag: Repräsentiert einen Feiertag.
Attribute:
Name Termin

Class: Termin	
Verantwortlichkeiten:	Kollaborationen:

Termin: Repräsentiert einen Termin.
Attribute:
Beginn Ende Ort

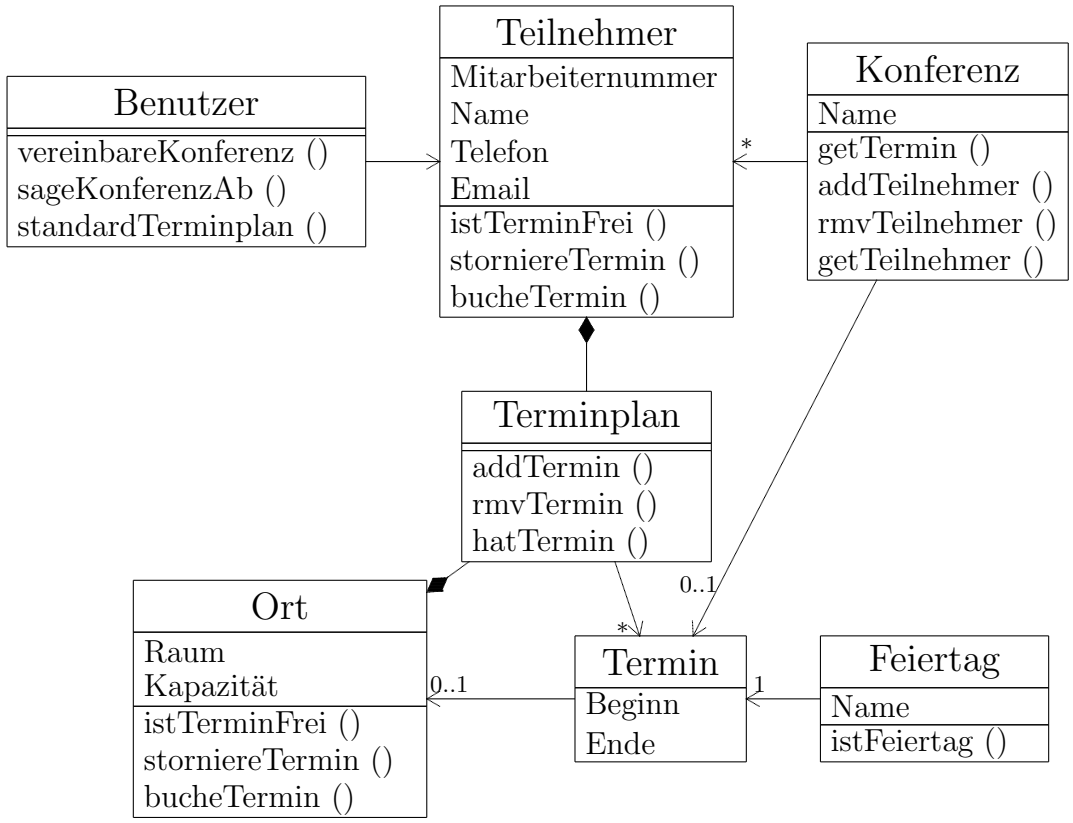
Class: Terminplan	
Verantwortlichkeiten:	Kollaborationen:
addTermin rmTermin hatTermin	

Terminplan: Repräsentiert einen Terminplan.
Attribute:
Menge von Terminen

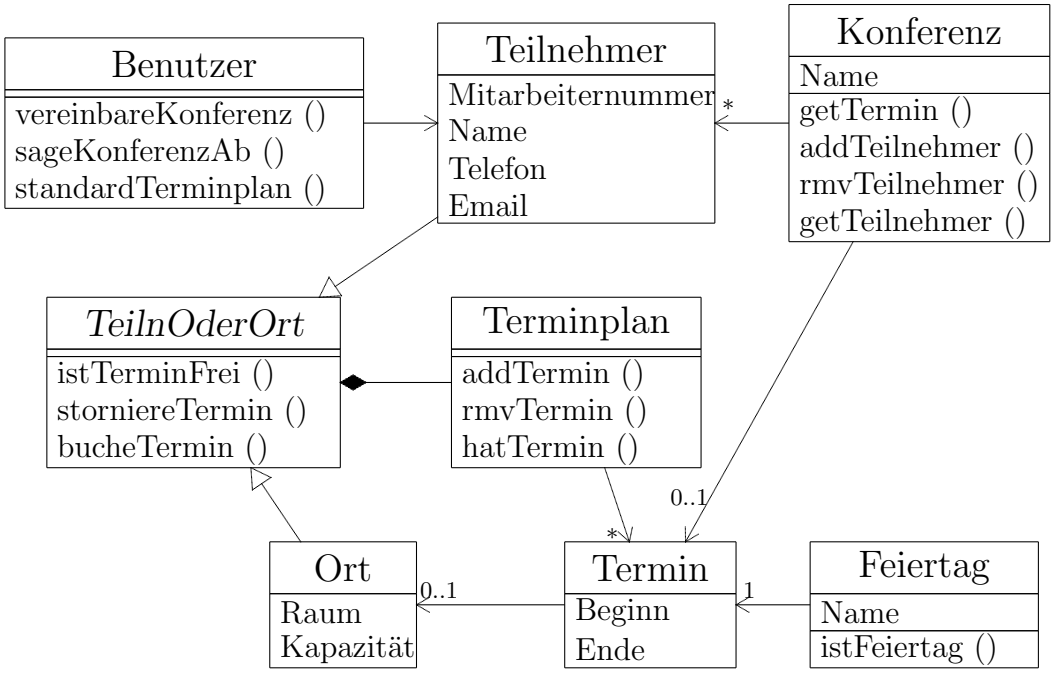
Class: Ort	
Verantwortlichkeiten:	Kollaborationen:
istTerminFrei stornierTermin bucheTermin	

Ort: Repräsentiert einen Konferenzort.
Attribute:
Raum Kapazität Terminplan

8.5.6 Klassendiagramm



Generalisierung:


Benutzer
vereinbareKonferenz ()
sageKonferenzAb ()
standardTerminplan ()
Teilnehmer
Mitarbeiternummer
Name
Telefon
Email
Konferenz
Name
getTermin ()
addTeilnehmer ()
rmvTeilnehmer ()
getTeilnehmer ()
*
TeilnOderOrt
istTerminFrei ()
storniereTermin ()
bucheTermin ()
Terminplan
addTermin ()
rmvTermin ()
hatTermin ()
0..1
*
Ort
Raum
Kapazität
0..1
Termin
Beginn
Ende
1
Feiertag
Name
istFeiertag ()

Kapitel 9

Implementierung eines Software-Systems

Implementierung bedeutet die eigentliche Erzeugung des Codes.

9.1 Bildung von Klassen

- Entitäts-Typen werden Klassen.
- Entitäts-Instanzen werden Objekte von Klassen.
- Attribute werden Datenelemente von Klassen.
- Das Verhalten einer Klasse wird zu einer Menge von Methoden der Klasse.

9.2 Beziehungen

9.2.1 Implementierung in C++

`B is-a A`	$\Longrightarrow$	`B` erbt `public` von `A`
`B is-like-a A`	$\Longrightarrow$	`B` erbt partiell `private` von `A`
`B has-a A`	$\Longrightarrow$	`B` hat Datenelement vom Typ `A` (oder erbt `private` von `A`)
`B uses-a A`	$\Longrightarrow$	`B` hat Zeiger vom Typ `A*` oder hat Referenz vom Typ `A&`
andere Beziehung	$\Longrightarrow$	Definition einer eigenen Klasse

9.2.2 Implementierung in Java

`B is-a A`	$\Longrightarrow$	`B` erweitert `A`
`B is-like-a A`	$\Longrightarrow$	Gemeinsamkeiten von `A` und `B` in eigene Klasse `G`, `A` und `B` erweitern `G`
`B has-a A`	$\Longrightarrow$	`B` hat Datenelement vom Typ `A` (Datenelement wird von `B` erzeugt oder `B` mitgegeben.)
`B uses-a A`	$\Longrightarrow$	`B` hat Datenelement vom Typ `A` (Datenelement wird an `B` übergeben.)
andere Beziehung	$\Longrightarrow$	Definition einer eigenen Klasse
`K Kategorie A, B ...`	$\Longrightarrow$	Gemeinsamkeiten von `A, B ...` in Interface `G`, `A, B, ...` implementieren `G` `K` hat Datenelement vom Typ `G`, das jenachdem mit `A, B ...` belegt wird.

Kapitel 10

Dokumentation

Unter **"Wissen"** verstehen wir die Gesamtheit der Informationen und Fähigkeiten, die zur Lösung von Software-Problemen benötigt werden. Das Wissen kann **implizit** – nur in den Köpfen der Mitarbeiter – oder **explizit** – dokumentiert – sein. Ziel der Dokumentation ist es, möglichst viel Wissen explizit zu machen, damit es kommunizierbar wird.

Problem: Erhaltung der Konsistenz einer Dokumentation.

Erfahrung: Erfolgreiche Projekte haben 5 bis 15 % Dokumentationsaufwand. Der Dokumentationsaufwand sollte möglichst gering gehalten werden.

Regel: Am Ende jeder Iteration sollte die Dokumentation aktuell sein.

Im Prinzip werden folgende Dokumente benötigt:

- Anforderungs-Dokument
- Software-Architektur-Dokument
- Analyse-Dokument
- Design-Dokument
- Code-Dokumentation

  Der Code sollte dokumentiert sein. Bewährt haben sich Kommentare, die ganze Klassen, Komponenten, Teilsysteme oder Pakete erklären.

  - Pakete eignen sich als Dokumentationseinheit.
  - Die externe Spezifikation beschreibt die Anwendung einer Klasse oder eines Pakets. Beschreibung von Schlüsselklassen.
  - Details kann man dem Code entnehmen. Der Code sollte "selbst sprechend" sein. Kommentare sollten eigentlich überflüssig sein.

    Dennoch sollten eventuell wichtige Methoden, wichtige Variable und wichtige Teile eines Algorithmus kommentiert werden. Ein Kommentar etwa alle zehn Anweisungen scheint eine vernünftige Kommentardichte zu sein.

- Test-Dokument
- Benutzerhandbücher
- Pläne, Statusreports, Besprechungsprotokolle

Kapitel 11

Entwurfsmuster *(patterns)*

Was ist ein Entwurfsmuster? Floyd Marinescu: *"A pattern is a best practice solution to a common recurring problem."* [35]

Es gibt Entwurfsprobleme, die immer wieder ähnlich auftreten. Für diese Probleme gibt es bewährte Lösungen – ***design patterns*** **(Entwurfsmuster)**, die zuerst von Gamma, Helm, Johnson und Vlissides[17] zu einem Katalog zusammengestellt wurden.

Der Nutzen von Entwurfsmustern ist:

- Sie sind eine Sprache, mit der auf wesentlich höherem Niveau über Design-Probleme gesprochen werden kann. Der Name eines Musters kann langwierige Erklärungen von Beziehungen zwischen Klassen ersetzen.
- Sie helfen, Software zu ordnen und schneller verständlich zu machen.
- Das Wissen (Erfahrungen, Vorteile, Nachteile u.a.) über ein Entwurfsmuster fließt unmittelbar in das spezielle Design ein, so dass man Problemen von vornherein aus dem Weg gehen kann.
- Die Entwurfsmuster bauen eventuell aufeinander auf und geben dadurch Hinweise für den Entwurf einer ganzen – oft wiederverwendbaren – Architektur.

Im Anhang B wird versucht, das Wichtigste über die verschiedenen Entwurfsmuster zusammenzustellen. Dieser Anhang kann und will nicht oben genanntes Buch von Gamma et al.[17] ersetzen. Dieses Buch sei hiermit als wichtigste Referenz empfohlen.

Für jedes Entwurfsmuster gibt es in Anhang B i.a. vier oder fünf Abschnitte:

1. Struktur: Die Struktur des Patterns in UML-Notation.
2. Anwendbarkeit: Hier werden die Symptome genannt, die auf eine mögliche Anwendbarkeit des Patterns hinweisen.
3. Bemerkungen

4. Abstraktes Code-Beispiel: Implementierung der Struktur als Java-Code-Beispiel, wobei die Struktur ohne Bezug auf ein konkretes Beispiel implementiert wird. Konkrete Beispiele sind oft zu kompliziert, da der ganze Problembereich erst verstanden werden muss, ehe man das Pattern anwenden kann.

5. Beispiel: Hier wird ein konkretes Beispiel vorgestellt.

Der Abschnitt "Anwendbarkeit" dient der Diagnose des Problems, wozu in "Struktur" dann eben die Lösung angeboten wird.

Die Entwurfsmuster werden folgendermaßen klassifiziert:

Creational Patterns **(Erzeugungsmuster):**
Singleton
Factory Method
Abstract Factory
Prototype
Builder
Structural Patterns **(Struktur-Muster):**
Composite
Bridge
Adapter
Decorator
Proxy
Facade
Flyweight (to do)
Model-View-Controller
Behavioral Patterns **(Verhaltensmuster):**
Chain of Responsibility (to do)
Command
Interpreter (to do)
Iterator (to do)
Mediator (to do)
Memento (to do)
Observer
State
Strategy
Template Method
Visitor

11.1 Beispiel Bier

"Bier" ist ein Beispiel, bei dem versucht wird, für möglichst viele Entwurfsmuster eine Anwendung zu finden. Dabei sind folgende Informationen interessant, die die Anwendung eines Patterns motivieren könnten:

- Es gibt in Deutschland 1388 Brauereien, die etwa 5500 Biermarken herstellen.

- Es gibt 40 Biersorten, z.B. Pils, Export, Weizen, Alt, Bock, Kölsch.
- Nach dem Reinheitsgebot von 1516 dürfen nur die natürlichen Rohstoffe
  - vermälztes Braugetreide (Malz) aus Braugerste
  - Hopfen
  - Wasser
  - Hefe

  verwendet werden.

Kapitel 12

Umstrukturierung *(Refactoring)*

12.1 Einleitung

Refactoring bedeutet die "Verbesserung" der internen Struktur eines Software-Systems, ohne dass sein externes Verhalten geändert wird. Das Design von Code wird verbessert, *nachdem* er geschrieben wurde. Verbesserung heißt bessere Lesbarkeit und – insbesondere – leichtere Modifizierbarkeit.

Als Erfinder des Refactorings gelten Kent Beck [28] [29], William Opdyke und Ralph Johnson. Das erste Lehrbuch, dem das Folgende weitgehend entnommen ist, hat Martin Fowler [16] geschrieben. Ferner sei noch auf Fields et al. [15] hingewiesen.

Die deutsche Übersetzung von Refactoring "Umstrukturierung" versteht niemand als informations-technischen Fachausdruck. Daher werden wir im folgenden Refactoring verwenden, gelegentlich das Verb "umstrukturieren".

Code, der eingesetzt wird, wird im Laufe der Zeit modifiziert. Dabei leidet seine ursprüngliche Struktur. Die Entropie des Codes, das Chaos im Code nimmt immer weiter zu. Refactoring versucht diese Tendenz umzukehren, indem der Code in kleinen Schritten so verändert wird, dass er einem vernünftigen Design entspricht. Dabei werden typischerweise Datenelemente von einer Klasse in eine andere verlegt, Codestücke werden aus einer – eventuell zu großen – Methode herausgenommen und in eigene Methoden gestellt, oder in einer Erweiterungshierarchie rauf- oder runterbewegt. Neue "Ebenen" werden eingezogen. Design-Patterns werden eingesetzt. Verständlichere, treffendere Namen werden verwendet. Gute Programme sind von Menschen lesbar.

Refactoring ist oft ein Eingriff in gut funktionierenden Code. Dies widerspricht zwar gewissen Engineering-Grundsätzen wie: "If it works, don't fix it" oder "it ain't broke, don't fix it". Aber es lohnt sich dennoch. "Embrace Change!" [30]!

12.2 Vorgehensweise

Refactoring ist ein methodischer Prozess.

- Was sind die Gründe für ein Refactoring? Man *muss* **umstrukturieren *(refactor)***, wenn man
  - eine neue Funktion einem Software-System hinzufügen will und wenn die Struktur des Systems so ist, dass das nicht leicht möglich ist,
  - einen Fehler beheben muss,
  - einen Code-Review durchführt und dabei bemerkt, dass man den Code nur schwer versteht.
- Bevor man mit dem Refactoring beginnt, muss man unbedingt Testcode schreiben. Diese Tests sind selbstprüfend. Sie sagen entweder "OK" oder sie geben die falschen Resultate aus.
- Refactoring ändert das Programm in *kleinen* Schritten (einzelne ***Refactorings)***, so dass Fehler leicht gefunden werden können. Nach jedem Schritt wird getestet. Das bedeutet, dass Tests automatisiert werden müssen.

12.3 Probleme beim Refactoring

12.3.1 Datenbanken

Da man beim Refactoring eventuell Datenelemente hinzufügt, eliminiert oder in andere Klassen verlegt, hat das natürlich ernste Auswirkungen auf persistente Objekte mit der Folge, dass eine ganze Datenbank umorganisiert werden muss.

Bei nicht-objektorientierten Datenbanken kann man sich eine Zwischenschicht einbauen, die zwischen dem objektorientierten Code und dem Datenbankschema vermittelt.

Man kann eventuell die Verschiebung von Datenelementen durch geeignete Zugriffsfunktionen maskieren.

Bei objektorientierten Datenbanken könnte man die Migration im Prinzip automatisieren.

Insgesamt ist das trotz der Werkzeuge, die ein modernes DBMS zur Verfügung stellt, ein offenes Problem, und ein ernstes, da die meisten Anwendungen mit Datenbanken zu tun haben.

12.3.2 Änderung von Schnittstellen

Wenn das Refactoring Schnittstellen ändert, dann hat das Auswirkungen auf jeden Code, der die Schnittstelle verwendet.

Fowler unterscheidet zwischen "öffentlichen *(public)*" und "veröffentlichten *(published)*" Schnittstellen. Einmal betrifft das die Sichtbarkeit der Schnittstelle, das andere Mal betrifft das die Freigabe der Schnittstelle zur Verwendung durch andere Programmierer. Die Änderung von veröffentlichten Schnittstellen ist natürlich gravierender.

Es gibt verschiedene Möglichkeiten, damit fertig zu werden:

- Bei nur öffentlichen Schnittstellen ist definitionsgemäß nur der eigene Code betroffen, den man dann mit Search-Edit-Zyklen aktualisiert.

- Bei veröffentlichten Schnittstellen ist das komplizierter:
  - Der erste Ratschlag ist einfach: Vorsicht mit der Veröffentlichung von Schnittstellen. Das ist insbesondere innerhalb von Teams oft unnötig. Organisationen mit expliziten ***Code-Ownership***-Regeln mögen da Probleme haben. Vielleicht sollte man überlegen, ob man diese Regeln ändert.
  - Man kann die alte Schnittstelle unverändert lassen und eine neue Schnittstelle definieren. Die alte Schnittstelle (Methode) wird dann mit Hilfe der neuen implementiert. In Java kann man dann die alte Schnittstelle mit einem `@deprecated`-Kommentar versehen.

12.3.3 Wann sollte man nicht umstrukturieren?

- Wenn Code überhaupt nicht funktioniert und nicht stabilisierbar ist, dann sollte man ihn neu entwerfen.
- Wenn man ganz nahe vor einem Abgabetermin steht, dann sollte man das Refactoring auf später verschieben.

12.3.4 Verschiedenes

Danger of "creeping elegance"

Durch Refactoring wird der Code immer "eleganter". Das ist eigentlich nicht schlecht. Aber Eleganz kann auch zu schwer verständlichem Code führen, denn "elegant" bedeutet oft, dass man mit einem einzigen Mechanismus gleich mehrere Probleme löst. Das ist aber eventuell schwer nachvollziehbar und führt zu überraschendem Verhalten.

12.4 Refactoring und Design

Wenn man beim **Vorabdesign** ***(upfront design)*** jede Art von Flexibilität berücksichtigt, dann wird das Vorabdesign sehr kompliziert. (Mit "Vorabdesign" ist hier einfach "Design" gemeint, das eben vor der Implementierung gemacht wird.) Oft wird nicht die ganze Flexibilität benötigt. Ferner kann durch ein Refactoring lokal Flexibilität in das Design gebracht werden.

Daher sollte man sich beim Vorabdesign auf die Arten der Flexibilität konzentrieren, die schwer durch ein Refactoring nachträglich zu machen sind. (Das betrifft insbesondere die Struktur persistenter Daten.) Ansonsten kann man sich durchaus zunächst auf eine einfache Lösung beschränken.

Ein Extremfall ist, dass man ganz auf das Vorabdesign verzichtet und sofort eine Implementierung erstellt, diese zum Laufen bringt und sie dann durch Refactoring verbessert (Extrem Programming [30]). Das scheint aber insgesamt nicht der effizienteste Weg zu sein.

Jedenfalls muss man kein schlechtes Gewissen haben, wenn man sich nicht besonders lang beim Vorabdesign aufhält. Man muss beim Design nicht *die* Lösung schlechthin finden.

Durch diesen Verzicht wird das Vorabdesign wesentlich einfacher, ohne dass man letztlich Flexibilität opfert.

12.5 Refactoring und Performanz

Die durch Refactoring gemachten Änderungen lassen das Programm i.a. langsamer laufen, da man wahrscheinlich mehr Indirektionen bzw. Ebenen eingeführt hat.

Performanz-Optimierung dagegen macht Code oft weniger verständlich.

Hier sollte man so vorgehen, dass man erst – durch Refactoring – verständlichen Code schreibt. Dann sollte man sich systematisch auf die Suche nach Performanz-Engpässen ***(performance hot spots)*** machen, indem man irgendeine Art von Profiling verwendet. Das sind oft wenige Stellen (10%), die dann – unter Opferung von Verständlichkeit – möglichst lokal optimiert werden. ***(First make your code right, and then make it fast.)***

12.6 Katalog von Refactorings

Fowler[16] hat einen Katalog von Refactorings zusammengestellt, von denen nur die wichtigsten hier vorgestellt werden.

12.6.1 *Extract Method*

Es gibt ein Codestück, das eine Einheit bildet.

Symptome:

- Codestück mit Kommentar.
- Codestück, das sich wiederholt.
- Codestück, das zu groß ist.

Therapie:

Definiere eine (oder mehrere) neue Methode, in die das Codestück ausgelagert wird:

- Der Name der Methode orientiert sich am Kommentar oder am Zweck des Codestücks. Ein guter Name ist sehr wichtig!
- Der Körper der Methode besteht aus dem Codestück.
- Temporäre Variable werden nach Möglichkeit in der Methode definiert.
- Andere lokale Variable werden zu Parametern der Methode.

Beispiele:

- Kommentiertes Codestück:

```
{
// ...
int   n;

// berechne Zufallszahl zwischen 50 und 100:
n = 50 + (int) (Math.random () * (101 - 50));
// ...
}
```

$\longrightarrow$

```
{
// ...
int   n;
n = zufallsZahlImIntervall (50, 100);
// ...
}

int   zufallsZahlImIntervall (int n1, int n2)
   {
   return n1 + (int) (Math.random () * (n2 + 1 - n1));
   }
```

  Hier bietet sich dann auch an, eine bessere Methode zur Erzeugung von Zufallszahlen einzusetzen.

- Wiederholtes Codestück:
  Wenn die Zufallszahl an verschiedenen Stellen im Code zu berechnen ist, dann haben wir hier ein Beispiel für redundanten Code, der in eine Methode ausgelagert werden sollte.

12.6.2 *Replace Temp with Query*

Es wird eine temporäre Variable verwendet, die das Resultat eines Ausdrucks enthält.

Symptome:

- Code wird nur deshalb lang, weil man eine temporäre Variable nicht aufgeben mag. (Man wollte vermeiden, diese Variable irgendwelchen Funktionen zu übergeben, die Codestücke ersetzen.)
- Eine temporäre Variable soll an vielen unterschiedlichen Stellen zugänglich sein.
- Eine temporäre Variable behindert die Anwendung des Refactorings *Extract Method*.

Therapie:

- Schreibe eine zunächst `private` Methode für den Ausdruck, der die temporäre Variable definiert – Query-Methode.
- Ersetze die temporäre Variable durch den Methodenaufruf.
- Die Query-Methode darf nicht den Zustand des Objekts verändern (daher auch der Name "Query").
- Eine temporäre Variable wird oft verwendet, um summarische Informationen zwischenzuspeichern. Ersatz durch eine Methode kann die Performanz senken. Darum sollt man sich aber zunächst nicht kümmern.

Beispiele:

-
```
{
// ...
double   sum = 0;
for (int i = 0; i < f.length; i++)
   {
   sum = sum + f[i];
   }
System.out.println ("Die Summe ist: " + sum);
// ...
double   mean = sum / f.length;
// ...
}
```

$\longrightarrow$

```
{
// ...
System.out.println ("Die Summe ist: " + sum (f));
// ...
double   mean = sum (f) / f.length;
// ...
}

private double sum (double[] f)
   {
   double   sum = 0;
   for (int i = 0; i < f.length; i++)
      {
      sum = sum + f[i];
      }
   return sum;
   }
```

Kapitel 13

Qualitätssicherung

Qualitätssicherung hat die Aufgabe sicherzustellen, dass die im Pflichtenheft festgehaltenen Anforderungen erfüllt werden. Dabei kann man grob eine externe und eine interne Sicht unterscheiden [25].

Externe Sicht: Die externe Qualität ist die Qualität, die der Kunde oder Benutzer des Systems sieht (Aspekt des Kunden).

- Funktionalität
- Zuverlässigkeit
- Effizienz
- Benutzbarkeit

Interne Sicht: Die interne Qualität ist die Qualität, die der Software-Entwickler sieht (Aspekt des Engineerings).

- Übertragbarkeit, Standardisierung
- Wartbarkeit, Änderbarkeit, Erweiterbarkeit
- Testbarkeit (manuell, automatisch)
- Transparenz, Verständlichkeit und Struktur des Codes

Oft wird wegen Terminvorgaben die interne Qualität geopfert in der Hoffnung, dass sich dies bei der externen Qualität nicht bemerkbar macht. Kurzfristig mag das funktionieren. Langfristig müssen die internen Qualitätsmängel unbedingt behoben werden.

Ein naives Verständnis von Qualität bringt diese in Verbindung mit Testen von Code – möglichst durch ein separates Test-Team. Dieses Modell erinnert an das Wasserfallmodell (Analyse, Design, Implementation, Testen). Allgemein anerkanntes Problem dabei ist, dass Fehler erst sehr spät im Prozess gefunden werden und die Beseitigung dieser Fehler wesentlich – hier werden Faktoren von 100 diskutiert – kostspieliger ist als bei moderneren Entwicklungs-Prozessen.

Unser Selbstwertgefühl hängt ab von der Qualität, die wir produzieren. Man muss dem Entwickler die Gelegenheit geben, qualitativ gut zu arbeiten. Man kann dann auch höhere Qualität fordern, als sie natürlicherweise erbracht würde.

Qualität bedeutet ein erhöhter Aufwand. Trotzdem lehrt die Erfahrung, dass Projekte früher erfolgreich beendet werden, wenn auf Qualität bestanden wird. Das Schreiben von Tests bringt mehr Verständnis und Vertrauen in den Code. Dadurch kann Code stressfreier und eben schneller geschrieben werden.

Methoden der Qualitätssicherung sind:

- konstruktive Maßnahmen, wie z.B.
  - Stilrichtlinien *(style guide)*
  - Kodierkonventionen *(coding conventions)*
  - Refactoring
- analytische Maßnahmen
  - Testen
  - Inspektionen
  - Reviews
  - Debugging
  - Prototypen

Ziele sind

- Zuverlässigkeit *(reliability)*
- Leistung *(performance)*
- Robustheit *(stress, robustness)*
- Benutzbarkeit *(usability)*
- Richtigkeit *(correctness)*
- Wartbarkeit *(maintainability)*

13.1 Testen

"It compiled? First screen came up? Ship it!"
– Bill Gates?

Auf Systemebene wird überprüft, ob die Anforderungsdefinition noch erfüllt wird ***(Validation: Are we doing the right job?)***. In den darunterliegenden Ebenen wird versucht, die Korrektheit nachzuweisen ***(Verifikation: Are we doing the job right?)***. Hierzu dient neben anderen Verfahren der Qualitätssicherung das **Testen**.

Mit **Testen** können nur Fehler-**Wirkungen**, d.h. die nach außen hin sichtbare Wirkung eines Fehlers gefunden werden. Das Finden der auslösenden Fehler-**Zustände** und das Beheben dieser gehört zum **Debugging**.

Testen ist sehr aufwendig und sollte daher automatisiert werden. Die Entwicklungsumgebungen bieten dazu Frameworks an:

Java: z.B. `JUnit`, `TestNG`
C++: z.B. `CppUnit`
`.NET`: z.B. `NUnit`

Bewährt hat sich ein "Test-First-Design", wobei man den Test *vor* der zu testenden Funktion schreibt. Das hat den Vorteil, dass man dabei schon sehr viel über das Verhalten der Funktion lernt. Ferner: Wenn ein Fehler auftritt, sollte man erst einen Test schreiben, der den Fehler produziert, und dann den Fehler beseitigen. Auf diese Weise wird dieser Fehler dann immer wieder mitgetestet.

Der Aufwand beim Erstellen des Test-Codes zahlt sich auf jeden Fall aus!

- Für jede Klasse sollte ein Testprogramm oder Test-Mechanismus eingebaut werden.

  In Java ist das typischerweise die `main`-Funktion einer Klasse.

  In C++ müsste etwa folgender Code geschrieben werden:

```
#ifdef Klassenname_Test
main ()
   // ---
#endif
```

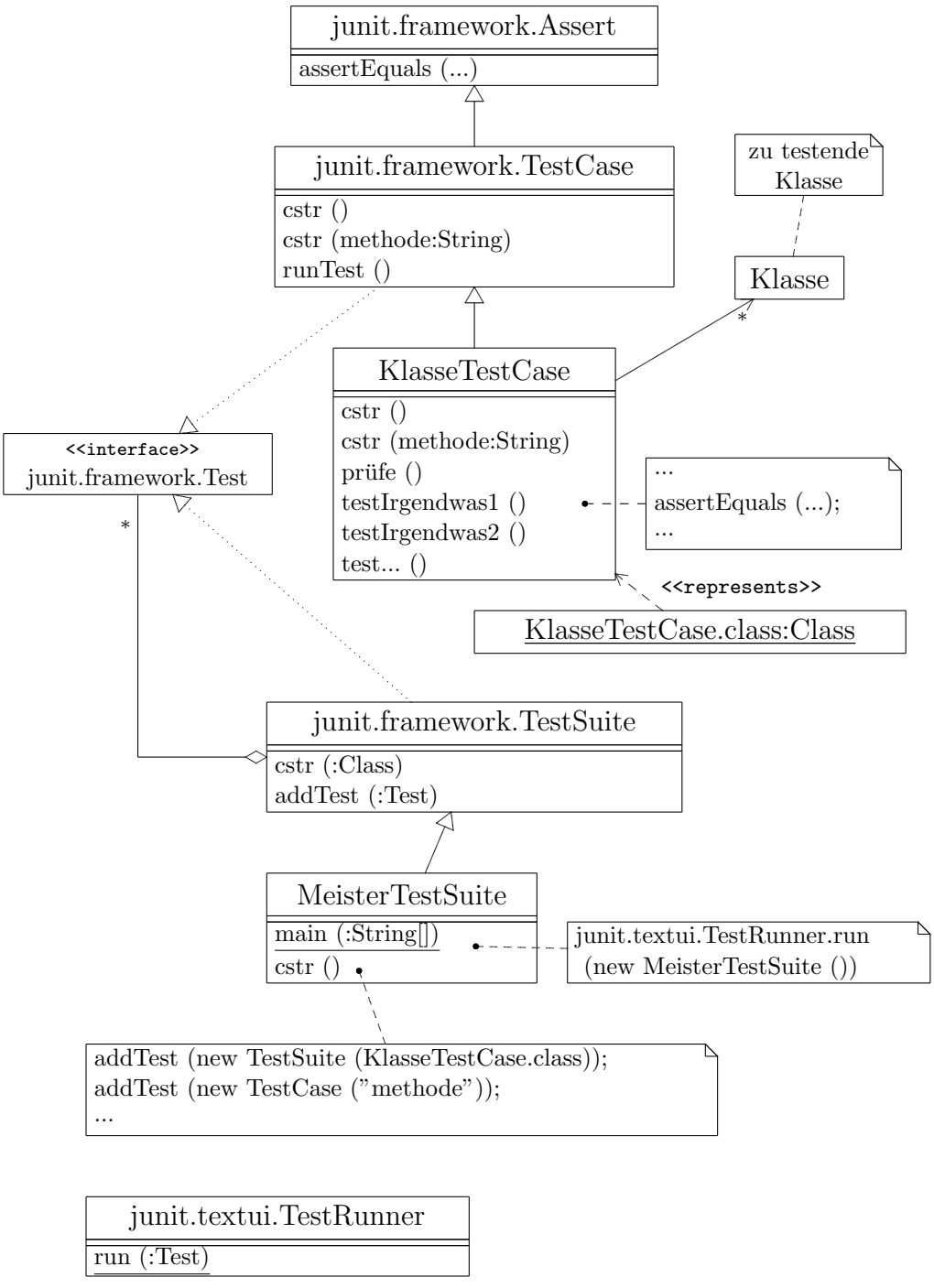
  und im Makefile:

```
Klassenname_Test:
$(COMPILER) -DKlassenname_Test ... Klassenname.C  ...
```

13.1.1 JUnit

JUnit ist ein Test-Framework für Java, das von Erich Gamma und Kent Beck entwickelt wurde.

Einen Überblick gibt folgendes Klassendiagramm:


junit.framework.Assert
assertEquals (...)
junit.framework.TestCase
cstr ()
cstr (methode:String)
runTest ()
zu testende Klasse
Klasse
*
KlasseTestCase
cstr ()
cstr (methode:String)
prüfe ()
testIrgendwas1 ()
testIrgendwas2 ()
test... ()
...
assertEquals (...);
...
<<interface>>
junit.framework.Test
*
<<represents>>
KlasseTestCase.class:Class
junit.framework.TestSuite
cstr (:Class)
addTest (:Test)
MeisterTestSuite
main (:String[])
cstr ()
junit.textui.TestRunner.run
(new MeisterTestSuite ())
addTest (new TestSuite (KlasseTestCase.class));
addTest (new TestCase ("methode"));
...
junit.textui.TestRunner
run (:Test)

Das Klassendiagramm zeigt unter anderem ein Composite-Pattern, bei dem ein `Test` entweder ein `TestCase` (Blatt) oder eine `TestSuite` (Composite) sein kann, die mehrere Tests enthalten kann.

Die Verwendung von JUnit zeigen wir an einem sehr einfachen Beipiel, nämlich einer Klasse `Zeichenkette`, die eine Zeichenkette repräsentiert und Methoden zur Manipulation von Zeichenketten zur Verfügung stellt. Insbesondere interessiert uns die Methode

```
dividiereDurch (Zeichenkette z),
```

die angibt, wie oft die Zeichenkette `z` in der `this`-Zeichenkette vorkommt.

Hier ist zunächst die Klasse `Zeichenkette`:

```
public class   Zeichenkette
   {
   private String s;

   public   Zeichenkette (String s)
      {
      this.s = s;
      }

   public int  dividiereDurch (Zeichenkette z)
      {
      int   n = 0;
      String   h = s;
      int   k;
      while (h.length () >= z.s.length ()
         && (k = h.indexOf (z.s)) >= 0)
         {
         n = n + 1;
         h = h.substring (k + z.s.length ());
         }
      return n;
      }

   public static void   main (String[] arg)
      // Klassischer Test mit main der Klasse
      {
      String   s1 = "abcdefabcwueirabc";
      String   s2 = "abc";
      int   n = new Zeichenkette (s1).
         dividiereDurch (new Zeichenkette (
            s2));
      System.out.println ("\"" + s1 + "\""
         + " dividiert durch \"" + s2
         + "\" ergibt " + n + ".\n");
      }
```

```
   }
```

Diese Klasse soll nun mit JUnit getestet werden. Dazu schreiben wir eine Klasse `ZeichenketteTestCase`, die `junit.TestCase` erweitert und eine Methode `testDividiereDurch ()` enthält. (Wenn die Testmethoden mit `test` beginnen, dann kann man bequemere JUnit-Mechanismen verwenden.) Im folgenden Beispiel ist auch ein Fall eingebaut, der absichtlich schief geht.

```
import   junit.framework.*;

public class   ZeichenketteTestCase
   extends TestCase
   {

   public void testDividiereDurch ()
      {
      Zeichenkette   a = new Zeichenkette (
         "abcdefghabcdefghaabcdefr");
      Zeichenkette   b = new Zeichenkette (
         "abc");
      int   n = a.dividiereDurch (b);
         // n muss 3 sein. Dies wird mit folgender Methode gefordert:
      assertEquals (3, n);
         //oder
      assertEquals (b + " kommt 3-mal vor",
         3, n);
      Zeichenkette   c = new Zeichenkette (
         "cba");
      int   m = a.dividiereDurch (c);
      assertEquals (c + " kommt 0-mal vor",
         0, m);
      int   k = a.dividiereDurch (a);
      assertEquals (
         "Zeichenkette kommt bei sich 1-mal vor",
         1, k);
      assertEquals (
         "Absichtlich falsch: Zeichenkette kommt bei sich 1-mal vor",
         0, k);
         // Das ist ein absichtlicher Fehler.
      }

   public static void   main (String[] arg)
      // Hiermit starten wir den Test.
      {
      junit.textui.TestRunner.run (
         new TestSuite (
            ZeichenketteTestCase.class));
```

```
        // Jede Methode, die mit test beginnt,
        // wird getestet.
    }
}
```

Der Test wird hier einfach mit

```
$ java ZeichenketteTestCase
```

gestartet.

In der `main`-Methode wird die Klasse `TestSuite` verwendet, deren Konstruktor aus jeder Methode der übergebenen Klasse, die mit `test` beginnt, ein `TestCase`-Objekt macht und dieses dann in einer Testfolge verwaltet.

Wir können natürlich auch mehrere Testfälle zu einer Testfolge zusammenbauen bzw eine ganze Hierarchie von Testfällen und Testfolgen aufbauen:

```
import   junit.framework.*;

public class   MeisterTestSuite extends TestSuite
   // Lassen von TestSuite erben, damit man
   // diesen MeisterTestSuite in anderen Testfolgen
   // verwenden kann.
   {
   public static void   main (String[] arg)
      {
      junit.textui.TestRunner.run (
         new MeisterTestSuite ());
      }

   public   MeisterTestSuite ()
      {
      addTest (
         new TestSuite (
            ZeichenketteTestCase.class));
      //addTest (new TestSuite (
      //    ????.class));
      // usw
      }
   }
```

Diese komplexere Testfolge wird mit

```
$ java MeisterTestSuite
```

gestartet.

13.1.2 TestNG

Wesentlich flexiblere Tests können mit dem Framework TestNG gebaut werden. Hier wird nur ein ganz kurzer Einblick gegeben. Eine umfassende Darstellung gibt das Buch von Beust und Suleiman[2].

- Flexible Test-Konfiguration durch JDK 5 Annotations
- Daten-getriebenes Testen (`@DataProvider`)
- Unit-, Funktions-, Integrations-, usw. Tests
- ...

13.2 *Code Review*

Code Review bedeutet, dass eine andere Person als die ProgrammiererIn den Code Zeile für Zeile durchgeht. Diese Inspektionen haben die höchste Fehlererkennungsrate von allen bekannten derartigen Methoden.

13.3 Korrektheit von Programmen

Korrektheit (Richtigkeit, ***correctness)*** sollte eine der wichtigsten Eigenschaften eines Programms sein. Mit mathematischen Verifikationsmethoden kann die Korrektheit eines Programms im Prinzip gewährleistet werden. Die praktische Anwendung dieser Methoden scheitert aber an dem benötigten Aufwand. Stattdessen schlägt man in der Praxis einen pragmatischen Weg ein, indem man bei der Programmierung gewisse Regeln einhält.

Folgende Maßnahmen oder Regeln dienen dazu, die Korrektheit von Programmen zu erhöhen.

- Schon in der Entwurfsphase sollten Kontrollen vorgesehen werden. Bei jeder Funktionalität sollten **Annahmen (Zusicherung**, ***assertion)*** definiert werden. Diese sind
  - **Vorbedingungen *(precondition)***, die eine Funktion als Voraussetzung fordert *(require)*, und
  - **Nachbedingungen *(postcondition)***, die eine Funktion nach Durchführung gewährleistet *(ensure)*.

  Das kann jeweils eine Folge von Ausdrücken sein, die als Ergebnis "wahr" oder "falsch" liefern.
- ***"Programming (Design) by Contract" [36]:*** Eine Funktion f bekommt von einer Funktion g einen Auftrag, d.h. wird von g aufgerufen. Für die Funktion f müssen gewisse
  - Vorbedingungen *(preconditions)*
  - Invarianten *(invariants)*
  - Nachbedingungen *(postconditions)*

erfüllt sein. Vorbedingungen sind Bedingungen, die der Client erfüllen muss, damit die Funktion f korrekt durchgeführt werden kann. Nachbedingungen sind Bedingungen, deren Erfüllung die Funktion nach ihrer Ausführung garantiert. Invarianten sind Eigenschaften die vor und nach der Ausführung der Funktion gleich sind. Sie können auch bei den Vor- und Nachbedingungen angegeben werden.

Wenn während der Abarbeitung von f ein Fehler auftritt, gibt es je nachdem, wo der Fehler passiert, verschiedene Reaktionsmöglichkeiten:

- Die Vorbedingung von f wurde von g nicht erfüllt: g bekommt die Antwort "Fehler" und muss sehen, wie es damit umgeht. Der vorgeschlagene Vertrag wurde nicht angenommen.
- Während der Abarbeitung von f tritt ein Fehler auf:
  - f kann den Fehler beheben, indem eventuell ein anderer Weg eingeschlagen wird **(Wiederaufnahme**, ***resumption)***.
  - f erkennt, dass die Aufgabe unlösbar ist und der Vertrag nicht erfüllbar ist. In diesem Fall ist es nur noch möglich, den Fehler zuzugeben und g als Antwort zurückzugeben ("organisierte Panik in Eiffel"). Es müssen zumindest wieder die nötigen Aufräumarbeiten durchgeführt werden, um wieder neue, andere Aufträge annehmen zu können.

- Trennung von Zustandsänderungen und Lesen eines Objekts: Eine Methode sollte nicht gleichzeitig den Zustand eines Objekts ändern und Informationen über das Objekt zurückgeben. Eine Methode, die den Zustand, d.h. Werte von Datenobjekten eines Objekts ändert, darf nichts außer Fehlercodes zurückgeben. Eine Methode, die Informationen über das Objekt liefern soll, darf den Zustand des Objekts nicht verändern.

13.4 Kodier-Konventionen

Kodierkonventionen dienen der Verständigung zwischen Entwicklern eines Teams. Sie sind ein wichtiges Mittel der Qualitätssicherung, und müssen daher von den einzelnen Teammitgliedern unbedingt beachtet werden.

Kodierkonventionen erhöhen die Lesbarkeit von Software. Andere Entwickler können sich schneller in eine Software einarbeiten.

Kodierkonventionen betreffen sowohl Äußerlichkeiten des Codes als auch Architektur-Entscheidungen.

Für Java seien folgende Kodier-Konventionen empfohlen, die man vollständig übernehmen oder als Grundlage für eigene Konventionen nehmen kann:

- "Java Code Conventions"
  `http://java.sun.com/docs/codeconv`
  (etwa 20 Seiten)

- `ftp://ftp.javasoft.com/docs/codeconv/CodeConventions.pdf`
  (etwa 20 Seiten)

- "Draft Java Coding Standard" von Doug Lea
  `http://g.oswego.edu/dl/html/javaCodingStd.html`
  (etwa 10 Seiten)
- "Writing Robust Java Code" von Scott W. Ambler
  `http://www.AmbySoft.com/javaCodingStandards.pdf`
  (etwa 60 Seiten)
- "Linux kernel coding style" von Linus Torvald
  (etwa 4 Seiten)

13.5 Beispiel Kodierkonventionen für `kj`-Projekte

Für `kj`-Projekte wird die Sprache Java verwendet.

13.5.1 Dateistruktur

Für jede Klasse wird eine Datei angelegt. Der Name der Klasse kann auch "Englisch" klingen. (Das war früher mal anders.)

Der Klassenname beginnt mit einem Großbuchstaben. Gutes Design resultiert oft in einem Schnittstelle-Klasse-Paar. Die Schnittstelle bekommt dann den Namen dieses Paars und die Klasse denselben Namen mit dem Postfix `R` (für "Realisierung"): Z.B. Schnittstelle `Regler` und implementierende Klasse `ReglerR`.

Diese Konvention wurde mehrfach geändert. Daher sind noch folgende Formen zu finden:

1. Die Schnittstelle bekommt den Namen dieses Paars und die Klasse denselben Namen mit dem Prefix `Default`: Z.B. Schnittstelle `Regler` und implementierende Klasse `DefaultRegler`.
2. Die Schnittstelle bekommt den Namen dieses Paars und die Klasse denselben Namen mit dem Postfix `I` (für "Implementierung"): Z.B. Schnittstelle `Regler` und implementierende Klasse `ReglerI`.

Die Datei beginnt mit einer ***modification history***. Jede Modifikation enthält eine Versionsnummer, die aus einer zweistelligen Zahl und einem Buchstaben besteht. Die Buchstaben werden hochgezählt, wenn eine kleine Änderung vorgenommen wurde, die Zahlen werden hochgezählt, wenn eine große Änderung erfolgte. Eine große Änderung ist etwa dadurch gekennzeichnet, dass sich das Benutzerinterface einer Funktion oder Klasse geändert hat. Die neueste Modifikation wird zuerst genannt.

Nach der Versionsnummer folgt das Datum der Änderung, ein Kürzel für den Programmierer und eine Kurzbeschreibung.

```
// modification history:
// 03a, 21apr14, kfg Änderung der Kodierkonvention
// 02a, 16jan94, kfg Änderung der Kodierkonvention
// 01b, 4jun93, kfg Ergänzung der Kodierkonvention
// 01a, 4feb93, kfg Ergänzung der Kodierkonvention
```

Nach der Modifikations-Historie kommt eine Leerzeile, dann eventuell eine `package`-Definition und nach einer weiteren Leerzeile eventuelle `import`-Statements. Die Schlüsselwörter `package` und `import` werden durch ein `<TAB>` vom Rest der Zeile getrennt.

Jede Klasse sollte ein `main`-Programm enthalten, mit dem die Klasse getestet werden kann.

13.5.2 Kodierstil

Zeilenstruktur

Keine Zeile darf mehr als 80 Zeichen enthalten. Werkzeug:

```
$ java kj.io.Zeilenlaenge80Reader <Dateiname>
```

Jedes Objekt oder jede Variable wird in einer **eigenen** Zeile deklariert bzw definiert. Zwischen Typ und Variablenname steht ein `<TAB>`. Dahinter kann – um ein weiteres `<TAB>` getrennt – ein Kommentar folgen. Das "Tabstop" soll auf **drei** Zeichen eingestellt sein.

```
int   eineGanzeZahl; // Definition eines Integers
```

Als String geschrieben lautet diese Zeile:

```
int\teineGanzeZahl;\t//\tDefinition eines Integers\n
```

Kommentare

`javadoc`-Kommentare haben folgende Form:

```
/**
*  Kommentarzeile
*  Kommentarzeile
*  Kommentarzeile
**/
```

Mit `javadoc`-Kommentaren werden Klassen, Schnittstellen und Klassenelemente kommentiert. Abweichend von der `javadoc`-Konvention werden diese Kommentare **nachgestellt**. D.h. sie erscheinen **hinter** dem jeweiligen Klassen-, Schnittstellen- oder Klassenelement-Kopf um ein `<TAB>` eingerückt.

`javadoc`-Kommentare werden in Englisch gegeben.

Klassen und Schnittstellen beginnen mit einer Angabe des Autors.

Beispiel:

```
// modification history:
// 01a, 16feb01, kfg Änderung der Javadocs

public class   Hello
   /**
   *  @author  2001 kfg
   *  hello-message handler
```

```
   **/
   {

   String   hello;
      /**
      *  Contains hello-message.
      **/

   public void grüßt ()
      /**
      *  Greets with hello-message.
      **/
      {
      System.out.println (hello);
      }

   }
```

`javadoc` kann die Nachstellung nicht verarbeiten. Daher gibt es das Werkzeug `kj.tool.Kj2Javadoc`, mit dem die Kommentare umgestellt werden können.

Kurze Kommentare C++-Style (`//`), längere Komentare auch C++-Style. Bei diesen Kommentaren ist die Sprache beliebig. Dabei soll direkt hinter den Kommentarzeichen ein Tab stehen. Kommentare gehen möglichst hinter die zu erklärende Zeile. Ansonsten gehen die Kommentare unter die zu erklärende Zeile und werden ein Level eingerückt.

```
if (a == 1)
   // Dies ist ein
   // langer
   // Kommentar
   {
   ...
   }
if (a == 1 && b == 2 || c == 0)

if (a == 1) // Dies ist ein kurzer Kommentar
   {
   ...
   }

if (a == 1 && b == 2 || c == 0)
   // Dies ist ein kurzer Kommentar
   {
   ...
   }
```

Einrückungen sind **ein** `<TAB>`.

Blanks werden

- vor und hinter binäre Operatoren,
- nach Komma bzw Semikolon,
- meist vor eine öffnende runde Klammer "("

gesetzt.

Vor der öffnenden Klammer "(" muss ein Blank stehen, es sei denn

> sie ist das erste sichtbare Zeichen in einer Zeile
> oder sie steht hinter einem "(", ")" oder ".".

Hinter der schließenden Klammer ")" muss ein Blank stehen, es sei denn

> sie ist das letzte sichtbare Zeichen in einer Zeile
> oder sie steht vor einer ")", "(" oder ".".

Beispiel:

> `methode ();` und nicht `methode();`
> `if ((a = d1) != 3)` und nicht `if ( (a = d1) != 3)`

Vor der öffnenden Klammer "[" steht **kein** Blank.

Geschweifte Klammern "{}"

Die öffnende Klammer "{" gehört – um ein `<TAB>` eingerückt – auf die zweite Zeile einer Kontrollstruktur. Der Code der Kontrollstruktur beginnt auf einer weiteren Zeile auf demselben Einrückniveau wie die Klammer. Die schließende Klammer "}" und die Kontrollstruktur selbst sind ein Level eingerückt. Beispiele:

```
int   func ()
   {
   Code der Funktion
   }

if (a == 1)
   {
   Code der Kontrollstruktur
   }

if (a == 1)
   {
   Code der Kontrollstruktur
   }
```

```
else if (a == 2)
   {
   Code der Kontrollstruktur
   }
else
   {
   Code der Kontrollstruktur
   }

if (a == 1) b = 2;

for (;;)
   {
   Code der Kontrollstruktur
   }

do {
   Code der Kontrollstruktur
   } while ();

switch (a)
   {
   case 1:
      Code der Kontrollstruktur
      break;
   case 2: Code einer kurzen Kontrollstruktur break;
   case 3: b =1; c = 2; break;
   case 4:
   case 5:
      Code der Kontrollstruktur
      break;
   default: ;
   }

public class   Meineklasse
   {
   }
```

Ausnahme: Wenn eine Funktion oder Kontrollstruktur komplett auf eine Zeile passt, dann kann (oder sogar sollte) sie auch auf eine Zeile geschrieben werden. Ferner, wenn der Block auf eine Zeile passt, kann er auch auf eine Zeile geschrieben werden. In diesen Fällen muss hinter "{" und vor "}" ein Blank stehen.

```
int   func (int a) { return a * a; }

double   methode (double xTerm, double yTerm)
   { return xTerm + yTerm; }
```

Funktionen

Funktionen mit Parametern, die kommentiert werden müssen, sollten folgendermaßen definiert werden:

Prototyp:

```
int   function    // Kommentar
   (
   int   arg1,    // Kommentar
   double   arg2  // Kommentar
   )
   {
   ...
   }
```

Alternativ können `javadoc`-Kommentare verwendet werden.

Identifikatoren

Klassennamen beginnen mit einem Großbuchstaben, Objekt- Variablen- und Methodennamen mit einem Kleinbuchstaben. Namensteile werden durch einen Großbuchstaben nicht durch ein Underline verdeutlicht.

```
KlassenName k;    // richtig
klassenName k;    // falsch
int   variablenName;    // richtig
int   VariablenName;    // falsch
int   variablenname;    // falsch
int   variablen_Name;   // falsch
int   variablen_name;   // falsch
int   Variablen_name;   // falsch
```

Konstanten

Konstanten werden ganz mit Großbuchstaben geschrieben, z.B.:

```
public static final int BUFSIZE = 256;
```

Hier ist ein Underline erlaubt, um Namensteile zu trennen.

13.5.3 Klassenstruktur

Die Klassen sollen der Java-Beans-Konvention genügen.

Kapitel 14

Prinzipien, Warnungen, Ratschläge und Trost

Riskiere große, radikale Änderungen. Es lohnt sich meistens.

Ein gutes Design verträgt auch große Änderungen oder es war nicht wert, konserviert zu werden.

Dave Winer:
Software is a process, it's never finished, it's always evolving.

Douglas McIlroy: (Erfinder der Unix-Pipe)

- Write programs that do one thing and do it well.
  Do one thing, do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

C. A. R. Hoare:
Premature optimization is the root of all evil.

Rob Pike:

Rule 1: You cannot tell where a program is going to spend its time. Bottlenecks occur in surprising places, so do not try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

Rule 2: Measure. Do not tune for speed until you have measured, and even then don't unless one part of the code overwhelms the rest.

Rule 3: Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.)

Rule 4: Fancy algorithms are buggier than simple ones, and they are much harder to implement. Use simple algorithms as well as simple data structures.

Rule 5: Data dominates. If you have chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming. ***Write stupid code, that uses smart data.***

Ken Thompson:
When in doubt, use brute force.

Mike Gancarz:

1. Small is beautiful.
2. Make each program do one thing well.
3. Build a prototype as soon as possible.
4. Choose portability over efficiency.
5. Store data in flat text files.
6. Use software leverage to your advantage.
7. Use shell scripts to increase leverage and portability.
8. Avoid captive user interfaces.
9. Make every program a filter.

Eric S. Raymond:
Keep it Simple and Stupid (KISS principle).

Rule of Modularity: Write simple parts connected by clean interfaces.

Rule of Clarity: Clarity is better than cleverness.

Rule of Composition: Design programs to be connected to other programs.

Rule of Separation: Separate policy from mechanism; separate interfaces from engines.

Rule of Simplicity: Design for simplicity; add complexity only where you must.

Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.

Rule of Transparency: Design for visibility to make inspection and debugging easier.

Rule of Robustness: Robustness is the child of tranparency and simplicity.

Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.

Rule of Least Surprise: In interface design, always do the least surprising thing.

Rule of Silence: When a program has nothing surprising to say, it should say nothing.

Rule of Repair: When you must fail, fail noisily and as soon as possible.

Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.

Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.

Rule of Optimization: Prototype before polishing. Get it working before you optimize it.

Rule of Diversity: Distrust all claims for "one true way".

Rule of Extensibility: Design for the future, because it will be here sooner than you think.

Gerald M. Weinberg:

Anyone who's not confused in today's world has to be out of touch with reality [46].

Most of the time, for most of the world, no matter how hard people work at it, nothing of any significance happens [46].

Claude Weets, Manager bei Renault:

Wer kriecht, kann nicht stolpern.

Winston Churchill:

Success is going from failure to failure without loosing enthusiasm.

Chinesisches Sprichwort:

Failure is the mother of success.

Chinesisches Sprichwort:

Those, who can, do.
Those, who can't, teach.
(Those, who can't teach, teach teaching.)

James Gosling:

In any design, you must strive for simplicity. If you don't, complexity will nail you.

Robert Eliot, Kardiologe:

Rule No. 1 is, don't sweat the small stuff.
Rule No. 2 is, it's all small stuff.
And if you can't fight and you can't flee, flow.

Karlotto Mangold:

Benutzerfreundlichkeit: "Trotz aller Probleme soll der Benutzer zu seinem System freundlich sein."

Josef Schmid, Politikwissenschaft, Uni Tübingen:

Wenn ich mit Theoretikern rede, dann muss ich in ausreichender Weise unverständlich bleiben, damit wir aneinander vorbeireden können.

Ratschläge und Einsichten von Fernando J. Corbató:

1. Failures in complex, ambitious systems are inevitable.
2. Put emphasis on the value of simplicity and elegance. Elegance is the achievement of a given functionality with a minimum of mechanism and a maximum of clarity.
3. Metaphors have the virtue of an expected behavior that is understood by all.
4. Use of constrained languages for design or synthesis is a powerful methodology.
5. Try to anticipate errors of human usage, hardware and software.
6. Assume repair and modification.
7. Try to cross-educate your team. Each team member should learn something about the work of his team colleges.

Ratschläge und Einsichten von Dennis Tsichritzis:

1. Be revolutionary in ideas
2. Have long range goals
3. Build a durable good image
4. Try to be evenly funded
5. Invest in infrastructure
6. Support your people emotionally
7. Help young researchers
8. Don't be unnecessarily aggressive
9. Avoid gimmicks and surface quality
10. Don't lock yourself in a clique, or position

Conway's Law:
The structure of a software system resembles the structure of the team that built it.

Edsger Dijkstra:
Testing can show only the presence of bugs, never their absence.

Frederick P. Brooks, Jr.:
Law: *Adding manpower to a late software project makes it later.*

This then is the demythologizing of the "man-month". The number of months of a project depends upon its sequential constraints. The maximum number of men depends upon the number of independent subtasks. From these two quantities one can derive schedules using fewer men and more months. (The only risk is product obsolescence.) One cannot, however, get workable schedules using more men and fewer months. More software projects have gone awry for lack of calender time than for all other causes combined.

Ralph Johnson:
Before software can be reusable it first has to be usable.

Peter Senge:

- Die Probleme von heute sind die Folge der "Lösungen" von gestern.
- Je stärker man drückt, umso mehr drückt das System zurück.
- Es wird kurzfristig besser, bevor es schlimmer wird.
- Der einfache Ausweg führt meistens wieder zurück.
- Die Behandlung kann schlimmer sein als die Krankheit.
- Schneller ist langsamer.
- Ursache und Wirkung liegen weder räumlich noch zeitlich eng beieinander.
- Kleine Änderungen können große Wirkung entfalten – aber die Bereiche mit dem höchsten Potenzial sind meistens die unscheinbarsten.
- Man kann den Kuchen haben und auch essen – aber nicht gleichzeitig.
- Wenn man einen Elefanten in zwei Teile teilt, bekommt man keine zwei kleine Elefanten.
- Man kann die Schuld nicht nach außen abschieben.

Poul Martin Møller:

Dänischer Philosoph 1794 bis 1838.

Fleiß ist eine Folge geistiger Armut.

Die Fleißigen haben es nötig, die Leere im Kopf mit Fremdwissen vollzustopfen.

Kapitel 15

Modifizierte Excerpte

15.1 Software-Entwicklungs-Prinzipien

Im folgenden wird ein Extrakt aus dem Buch von Alan M. Davis "201 Software-Entwicklungs-Prinzipien" vorgestellt [10].

Prinzip 3 – Produktivität und Qualität sind untrennbar:

Zwischen Produktivität (gemessen in LOCs *(lines of code)* oder FPs *(function points)* pro Mann-Monat) und Qualität besteht eine *umgekehrte* Proportionalität. Je größer das Verlangen nach Qualität ist, desto geringer wird die Produktivität. Je weniger Qualität verlangt wird, desto höher ist die Produktivität.

Prinzip 7 – Geben Sie Produkte früh zum Kunden:

Unabhängig davon, wie sehr Sie während der Ermittlung der Anforderungen versuchen, die Bedürfnissse der Anwender zu verstehen, ist der beste Weg die wahren Bedürfnisse zu ermitteln der, den Anwendern ein Produkt zu geben um sie damit spielen zu lassen.

Geben Sie daher dem Kunden zunächst einen "quick and dirty"-Prototyp, um von ihm Feedback zu erhalten, das dann die Basis für die Anforderungsdefinitionen bildet. Anschließend führen Sie das Projekt in der vollen Breite fort.

Bis zu dem Zeitpunkt, an dem der Kunde das Produkt zum ersten Mal ausprobieren kann, sind erst 5-20% der Entwicklungsressourcen verbraucht (gegenüber 99% beim "Wasserfall"-Modell), so daß der Rest der Ressourcen darauf verwendet werden kann, das "richtige" System zu erstellen.

Prinzip 8 – Kommunikation mit Kunden/Benutzern:

Man sollte niemals aus den Augen verlieren, warum Software entwickelt wird. Programme sollen reelle Bedürfnisse befriedigen bzw. existierende Probleme lösen.

Entwickelt man kommerzielle Software, so ist es wichtig, so oft wie möglich mit den Kunden bzw den Benutzern zu reden, um nicht an ihren Bedürfnissen vorbei zu entwickeln. Wenn man für den Massenmarkt programmiert, kann es schwierig sein, in Kontakt mit Kunden zu kommen. Es sollten ein paar Personen im Team oder im Unternehmen in die Rolle

der Kunden schlüpfen und ihre Ideen einbringen. Bei manchen Aufträgen (z.B. auch von Regierungen) wird man kaum mit den Benutzern in Kontakt treten können, sondern nur mit den Auftraggebern.

Prinzip 14 – Vergrößern Sie das System stückweise:
Starten Sie klein, mit einem lauffähigen System, welches nur wenige Funktionen enthält. Dann vergrößern Sie es, um mehr und mehr Teilbereiche der eigentlichen Funktionalität einzuschließen. Die Vorteile sind:

1. Geringe Risiken mit jeder Erweiterung und
2. wenn Benutzer schon eine Version des Produktes sehen, können sie sich besser weitere Funktionen vorstellen, die sie noch haben möchten.

Wenn allerdings schon früh eine ungeeignete Architektur gewählt wurde, wird möglicherweise ein komplettes Neudesign erforderlich. Reduzieren Sie dieses Risiko, indem Sie zunächst Prototypen entwickeln, bevor Sie anfangen "aufbauend zu Entwickeln".

Prinzip 19 – Jedes komplexe Problem hat eine Lösung:
"Zu jedem komplexen Problem gibt es eine einfache Lösung ... und die ist falsch!" (W. Turski). Mit Vorsicht sind Ratschläge zu geniessen wie: "Du brauchst nur diesen zehn einfachen Punkten zu folgen und deine SW-Qualitäts-Probleme werden verschwinden!"

Prinzip 22 – Zuerst die Technik, dann das Werkzeug *(Tool)***:**
Bevor man ein Werkzeug benutzt, sollte man Disziplin üben, das heißt, die Fähigkeit besitzen, ein angemessenes Verfahren anwenden zu können. Natürlich, muß man auch wissen, wie man das Werkzeug benutzt, aber das ist zweitrangig gegenüber der Disziplin. Ich empfehle eindringlich, eine Technik erst "von Hand" anzuwenden, und sich selbst und vor allem das Management davon zu überzeugen, daß diese Technik funktioniert, bevor man in geeignete Werkzeuge investiert. In den meisten Fällen ist es so, daß eine Technik, die ohne Automatisierung nicht funktioniert, auch mit Automatisierung keinen Erfolg hat.

Prinzip 36 – Erst entwickeln und dann austauschen funktioniert nicht:
Die Literatur ist voll von Berichten über zu *technische* oder *theoretische* Vorgehensweisen in SW-Entwicklungslabors. Gründe hierfür sind:

1. Im allgemeinen haben die SW-Entwickler nur wenig Erfahrung mit der Entwicklung von realen Systemen.
2. SW-Entwickler finden es einfacher, technische Probleme schnell zu lösen, ohne sich Gedanken darüber zu machen, ob ihre Entwicklung in die reale Welt paßt.
3. Entwickler und Auftraggeber haben ein unterschiedliches Vokabular, sodaß es schwer ist, zu kommunizieren.

Prinzip 37 – Übernehme Verantwortung:
In allen Ingenieurwissenschaften werden die Ingenieure zur Verantwortung gezogen, falls ein Design schiefgeht. Bei Softwarefehlern geschieht dies nur selten, und es werden Ausflüchte wie z.B. "Der Compiler hat einen Fehler gemacht" gesucht. Es ist eine Tatsache, daß auch die besten Methoden nicht verhindern, ein schlechtes Design zu produzieren. Es gibt keine Entschuldigungen. Wenn man Systementwickler ist, hat man die Verantwortung, es richtig zu machen. Mach es richtig, oder mach es gar nicht!

Prinzip 39 – Bestimme das Problem, bevor du die Anforderungen schreibst:
Bevor du ein Problem lösen willst, sei dir sicher, wer genau das Problem hat und was das Problem ist. Laß dich nicht von dem erst besten Lösungsvorschlag verblenden.

Prinzip 60 – Anforderungen sollen in Datenbanken gespeichert werden:
Anforderungen sind komplex und sehr änderungsanfällig. Daher ist es günstig sie elektronisch zu speichern, vorzugsweise in einer Datenbank. Dadurch sind Änderungen, Sicherung von Attributen spezieller Anforderungen und Folgen von Änderungen einfacher zu handhaben.

Dinge, die man in einer Datenbank ablegen sollte, sind zum Beispiel:

- einheitlich vorkommende Identifikatoren
- Text der Anforderung (Beschreibung)
- Beziehung zu anderen Anforderungen
- Wichtigkeit der Anforderung
- erwartete Veränderungen
- Hinweise auf die Quellen der Anforderung
- vergleichbare Produkt-Versionen

Prinzip 67 – Keep it simple:
Einfachheit gewährleistet hohe Wartbarkeit. Der normale Mensch kann nur 7 Objekte gleichzeitig erfassen. Zwei Arten des SW Designs: Entwerfe so, daß es offensichtlich keine Fehler gibt, oder so, daß es keine offensichtlichen Fehler gibt.

Prinzip 70 – Das Design wartbar halten:
Ein Design ist nur dann wartbar, wenn es verständlich erstellt und dokumentiert wurde. Hierbei ist es wichtig, daß das Design hierarchisch aufgebaut ist. Hierarchien ermöglichen es, das vollständige System abstrakt zu erfassen und sich dann Level für Level weiter nach unten durchzuarbeiten. Die einzelnen Komponenten in der Hierarchie sollten einfach zu verstehen sein.

Prinzip 71 – Behalte konzeptionelle Integrität:
Konzeptionelle Integrität ist ein Qualitätsmerkmal eines Designs. Wenn ein Design fertig ist, sollte es aussehen, als hätte es eine Person erstellt. Man soll nicht der Versuchung erliegen, von der vereinbarten Art abzuweichen, es sei denn, es läßt sich durch zusätzliche Integrität, Eleganz, Einfachheit oder Performanz des Systems rechtfertigen.

Prinzip 91 – Namen mit Bedeutung verwenden:
Einige Programmierer bestehen darauf, Variablennamen wie `N_FLT` oder schlimmer noch nur `F` zu verwenden. Ihr Argument ist, daß dies das Programmieren effektiver machen würde, da weniger Tastenbenutzungen notwendig sind. Gute Programmierer sollten jedoch nur einen kleinen Teil ihrer Zeit mit Tippen verbringen (ca. 10 - 15 Prozent) und den Rest der Zeit mit Denken. Wieviel Zeit wird also wirklich eingespart? Bessere Argumente gibt es dafür, daß zu sehr verkürzte Namen die Produktivität in Wirklichkeit sinken lassen.

1. Test und Wartung werden teurer, da die Leute Zeit verlieren, weil sie versuchen die Variablennamen zu dekodieren.
2. Es kann sein, daß man mehr Zeit mit Tippen zubringt, wenn man abgekürzte Namen verwendet, da es notwendig werden kann, zusätzliche Kommentare einzufügen, die mit ausführlichen Variablennamen nicht notwendig gewesen wären.

Prinzip 100 – Strukturierter Code ist nicht notwendigerweise guter Code:

- Ursprüngliches Anliegen der strukturierten Programmierung (vorgeschlagen von Edsger Dijkstra) war die Erleichterung des formalen Beweisens von Programmen.
- Die verwendeten Konstrukte (IF-THEN-ELSE, DO-WHILE, usw) sind inzwischen allgemein üblich (im Gegensatz zu ihrer Nutzung in Bezug auf das formale Beweisen von Programmen).
- Man kann fürchterlich obskure Programme schreiben, die immer noch strukturiert sind.

Struktur ist eine notwendige, aber keine hinreichende Bedingung für qualitativ hochwertige Software.

Prinzip 107 – Test der Anforderungen:
Es ist wichtig, das man weiß, welche Anforderungen man testen muß.

Es gibt zwei wichtige Punkte, die man beachten sollte:

1. Beim Erzeugen/Erstellen von Testszenarien, sollte man wissen, ob alle Anforderungen getestet werden.
2. Beim Durchführen der Tests ist es wichtig, daß man weiß, was man gerade testet.

Wenn im Laufe einer Entwicklung bestimmte Anforderungen wichtiger werden, dann haben auch die Tests für diese Anforderungen eine höhere Priorität.

Bei der Durchführung der Testszenarien sollte man sich eine Tabelle erstellen, die binäre Werte enthält. In den Spalten der Tabelle stehen alle Tests, in den Zeilen stehen alle Anforderungen, die verifiziert werden müssen. Eine "1" an einer Tabellenposition sagt aus, daß der Test zur Prüfung der Anforderung sinnvoll ist. Eine Spalte, die keine "1" enthält, sagt aus, daß der Test keine Auswirkung auf die Verifikation der Anfordeung hat. Eine Zeile, in der keine "1" steht, zeigt an, daß es für diese Anforderung noch keinen Test gibt.

Die erfolgreiche Erstellung einer solchen Tabelle basiert in erster Linie auf der Fähigkeit sich individuell auf jede Anforderung zu beziehen.

Prinzip 126 – Nehmen Sie Fehler nicht persönlich:
Um Software zu schreiben, ist ein Level an Detailkenntnis und Perfektion erforderlich, den kein Mensch erreichen kann. Streben Sie daher eine konstante Steigerung und keine Perfektion an. Wenn Sie oder andere einen Fehler in Ihrem Code finden, diskutieren Sie ihn offen und benutzen Sie diese Gelegenheit als Erfahrung für Sie selbst.

Prinzip 127 – Gutes Management ist wichtiger als Technologie:
Gutes Management motiviert die Mitarbeiter. Hochentwickelte Werkzeuge, wie z.B. CASE-Tools, kompensieren schlechtes Management nicht. Auch mit knapperen Ressourcen kann hervorragende Arbeit geleistet werden, wenn das Management stimmt. Der Manager hat große Verantwortung. Er muß seinen Verhandlungs-, Führungs- und Gesprächsstil immer der Situation anpassen. Newcomern gelingt es nur mit gutem Management im Softwaremarkt Fuß zu fassen. Die Technologie ist zweitrangig.

Prinzip 130 – Verstehe die Prioritäten des Kunden:
Wenn man sich mit Kunden unterhält, sollte man sich über die Prioritäten verständigen. Dabei ist es am wichtigsten, die Begriffe "unumgänglich, wünschenswert und optional" eindeutig zu definieren, damit später keine Mißverständnisse auftreten. Es ist durchaus

vorstellbar, daß ein Kunde in Kauf nimmt, 90% des Produkts verspätet zu bekommen, wenn 10% (die unumgänglichen Anforderungen) rechtzeitig zum vereinbarten Termin ausgeliefert werden.

Prinzip 135 – Erwarten Sie Brillianz:
Mitarbeiter arbeiten besser, wenn man viel von ihnen erwartet. Seien Sie selbst beispielhaft: hart arbeiten, stolz auf Resultate sein, keine Spiele! Belohnen Sie ihre Mitarbeiter, helfen Sie schlechteren sich zu verbessern. Wenn das nicht hilft, helfen Sie bei der Suche nach einem passenderen Job für denjenigen in oder ausserhalb ihrer Firma. Solche Leute dürfen nicht bei ihnen arbeiten, da sie die Moral und Motivation sehr leicht untergraben.

Prinzip 161 – Die 10 größten Risiken:
Ein Projektmanager sollte die größten Risiken bzw Probleme der Softwareentwicklung kennen, um Katastrophen verhindern zu können:

1. Personalknappheit (zu wenig Leute im Team)
2. Unrealistische Planung (z.B. Terminpläne)
3. Bedürfnisse des Kunden werden nicht verstanden.
4. Schlechtes oder unpassendes Benutzerinterface
5. Produkt wird schön geredet, wenn der Kunde nicht zufrieden ist.
6. Geänderte Anforderungen werden ignoriert.
7. Kaum wiederbenutzbare Komponenten (z.B. schlechte Schnittstellen)
8. Kaum Zeit für andere Aufgaben
9. Schlechte Antwortzeiten des Systems
10. Überforderung der bestehenden Hardware (auch Betriebssystem, usw)

Zusätzlich sollten noch Ueberlegungen angestellt werden, welche speziellen Risiken ein Projekt und die Umgebung (Team, Firma) bergen und wie sie minimiert werden können.

Prinzip 162 – Plane Risiken vor vornherein ein:
Es geht immer etwas schief!

Schreibe dir die größten Risiken auf. Eine Maßzahl für das Risiko ist Wahrscheinlichkeit des Eintretens multipliziert mit dem Ausmaß des Schadens.

Male dir einen Baum auf, der alle Entscheidungen skizziert, die Risiken verhindern.

Prinzip 164 – Die Methode wird dich nicht retten:
Die Welle der strukturierten Softwareentwicklung in den 70er und frühen 80er Jahren sowie die Einführung der Objektorientierung haben gezeigt, daß Firmen, die vorher schon erfolgreich gute Software entwickelt haben auch diese Methoden gut umgesetzt haben. Softwareentwickler, die jedoch zuvor schlechte Produkte entwickelt haben, haben auch weiterhin schlechte Ergebnisse nach Umsetzung der neuen Methoden. Es ist nichts falsch dabei, neue Methoden einzusetzen. Aber wenn die Softwareentwickler bereits vorher versagt haben (entweder in der Produktivität oder der Qualität), versuche die Ursachen für diese Mißlage zu finden, bevor eine neue Methode eingeführt wird.

Prinzip 172 – Mache ein Projekt-Postmortem:
"Diejenigen, die sich nicht an ihre Vergangenheit erinnern, sind dazu verdammt, diese wiederzuerleben." (G. Santayana, 1908)

In jedem Projekt gibt es technische und Management Probleme. Diese Probleme müssen dokumentiert und analysiert werden. Nur so ist es möglich, in folgenden Projekten die gleichen Probleme zu vermeiden. Deshalb sollten sich alle Beteiligten nach dem Projektabschluß noch für drei bis vier Tage zusammensetzen und die Probleme analysieren.

Principle 173 – Produktsicherheit ist kein Luxus:
Produktsicherheit beinhaltet Konfigurationsmanagement, Qualitätssicherung, Verifikation und Validation und Testen von Software. Die Sicherung dieser Disziplinen resultiert in einer signifikant höheren Wahrscheinlichkeit ein Produkt zu erstellen, daß den Kunden zufrieden stellt und das näher am Zeitplan ist. Der Schlüssel ist die Anteile der Produktsicherung dem Projekt in Größe, Form und Inhalt anzupassen.

Prinzip 186 – Die Entropie einer SW nimmt zu:
Jedes laufende SW-System wächst weiter. Dadurch steigt die Komplexität und die Unordnung. Unordnung verursacht Instabilität und reduziert die Zuverläßigkeit und Wartbarkeit.

Prinzip 190 – Prerelease Fehler erzeugen postrelease Fehler:
Komponenten mit einer hohen Rate von prerelease Fehlern haben auch eine hohe Rate von postrelease Fehlern, d.h. je mehr Fehler man in einer Komponente findet, desto mehr wird man später auch noch finden. Deshalb verwerfe und ersetze man diese Komponenten.

Prinzip 196 – Rückwirkendes Testen ***(Regression Test)*** **nach jeder Änderung:**
"Regression testing" meint das Testen aller bereits getesteten Features, nachdem eine Änderung gemacht wurde. Manch einer macht diese Tests nicht, da er die Änderungen nicht für so wichtig hält. Nach jeder Änderung eines Moduls, sei es das Beheben eines Fehlers, das Hinzufügen eines neuen Features oder das Erweitern der Funktionen dieses Moduls, muß dieses Modul neu getestet werden. Es muß geprüft werden, ob alles richtig läuft. Nach diesem Test ist man aber noch nicht fertig. Software verhält sich nämlich manchmal sehr seltsam. Man muß jetzt noch "Regression testing" durchführen, um sicher zu sein, daß alles was vorher schon einwandfrei lief auch jetzt nach der Änderung noch genauso gut läuft.

Anhang A

Dokumentations-Schablone für UP

A.1 Überblick

*(Wichtigste Merkmale der Problemstellung.
Zusammenfassung der Ergebnisse.)*

A.2 Management

(Diese Prozesskomponente enthält eventuell alles, was zum typischen Projektmanagment (Vorgehensweisen, Zeitpläne usw.) gehört.)

Arbeitstitel: **Projekttitel**

A.2.1 Technisches Management

(Wird eventuell ganz im Abschnitt "Architektur" abgehandelt.

Allerdings kann hier die Definition und Verwaltung der Prozessphasen untergebracht werden. Was wurde in den Phasen getan? Wie lange haben sie gedauert? Welche Probleme gab es? Die Darstellung eines Designs wäre hier fehl am Platz.)

Konzeption

Entwurf

Konstruktion

Auslieferung

A.2.2 Kundenmanagement

Auftraggeber

Anwender

A.2.3 Teammanagement

(Wer sind die Teammitglieder? Gibt es eine Hierarchie oder Aufgabenverteilung?)

A.2.4 Organisationsmanagement

(Wie ist das Projekt in das Unternehmen eingebettet? Gibt es Unternehmens-politische Randbedingungen? Gibt es Projekte, die konkurrieren oder mit denen zusammengearbeitet werden kann?)

A.2.5 Besprechungsprotokolle

A.3 Architektur

A.3.1 Entwicklungsumgebung

(Welche Entwicklungsumgebung wird verwendet? Programmiersprachen? Kodier-Konventionen? Editor? Hardware?)

A.3.2 Klassenbibliotheken

(Welche Klassenbibliotheken sollen verwendet werden? Werden die gekauft oder sind die open source? Werden Klassenbibliotheken im Projek† entwickelt?)

A.3.3 Komponenten

(Werden Komponenten besorgt? Werden Komponenten im Zuge des Projekts entwickelt?)

A.3.4 Frameworks

(Werden Frameworks besorgt? Werden Frameworks im Zuge des Projekts entwickelt?)

A.3.5 System/Subsystem-Struktur

A.3.6 Fehlerbehandlung

(Programmierfehler? Fehler des Benutzers? Systemabstürze?)

A.3.7 Persistenz

(Wird ein Datenbanksystem verwendet? Wenn ja, welches? Wann und wo wird auf das DBS zugegriffen? Behandlung von Transaktionen?)

A.3.8 Testen

(Wie wird getestet? Hier keine Testresultate.)

A.3.9 Verteilung von Objekten – Networking

(Resultiert das Projekt in einem verteilten System? Welche Software soll auf welchen Hosts laufen?)

A.3.10 Sicherheit

(Wie sollen eventuelle Sicherheitsprobleme behandelt werden?)

A.3.11 Benutzeroberfläche

A.3.12 Dokumentations

(Wo wird dokumentiert? Welche Dokumente werden erzeugt? Wo findet man was?)

A.3.13 Systemstart und -ende

(Wahrscheinlich gehört das in die Prozesskomponente "Einsatz". Aber eventuell gibt es allgemeinere, projektübergreifende Vorgehensweisen.)

A.4 Anforderungsanalyse

(Spezifische Anforderungen des Kunden werden hier gelistet.

Eventuell werden eigene Anforderungen an das System getellt.)

A.5 Analyse

(Wie sieht die Welt des Kunden aus? Erarbeitung des notwendigen Fachwissens. Was soll das System tun?)

A.5.1 Brainstorming

(Wird ganz früh im Projekt durchgeführt.)

A.5.2 Stand der Technik

(Z.B.: Wurde das Thema schon mal bearbeitet? Gibt es ähnliche Projekte? Worauf kann man aufbauen?

A.5.3 Systembeschreibung

(Textuelle Beschreibung: Wie sieht die Welt des Kunden aus? Wie soll das zu entwickelnde System aussehen?)

A.5.4 Anwendungsfälle

(Liste der dargestellten Anwendungsfälle. Erwähnung trivialer Anwendungsfälle, die nicht explizit dargestellt werden.)

Anwendungsfall Name

Name:

Kurzbeschreibung:

Ablaufbeschreibung:

1. x
2. x

Akteure:

- Primäre Akteure:
- Sekundäre Akteure:

Vorbedingungen:

Nachbedingungen:

Invarianten:

Regeln:

Nicht-funktionale Anforderungen:

Erweiterungspunkte:

Ausnahmen, Fehlersituationen:

Variationen:

Dienste:

Anmerkungen:

A.5.5 Fachwissen

(Erarbeitung und Darstellung des Fachwissens über die Welt des Kunden. Das sollte unbedingt vom Kunden geprüft werden.)

A.5.6 Analyse von Entitäten

(Eine Liste von Substantiven der Systembeschreibung, der Anwendungsfälle und sonstiger Texte wird iterativ bearbeitet, bis eine Liste von möglichen Klassen und eventuell auch Attributen resultiert. Eventuell CRCs.)

Datenstruktur

(Hier werden die Attribute der Klassen entwickelt.)

Verhalten

(Hier werden die Methoden der Klassen entwickelt. Welche Dienste soll eine Klasse zur Verfügung stellen?)

A.5.7 Datenflüsse

A.5.8 Ideensammlung

A.6 Design

A.6.1 Substantivlisten

A.6.2 CRC-Karten

A.6.3 Klassendiagramme

A.6.4 Beschreibung der Klassen

(Oft sind die Namen von Klassen so gewählt, dass nicht unmittelbar klar ist, was diese Klassen wozu und mit welchen anderen Klassen tun. Diese Informationen sollen hier gegeben werden. Es müssen daher nicht alle Klassen beschrieben werden. Ferner genügen nur die wichtigsten Methoden.)

A.6.5 Design-Patterns

(Wurden Design-Patterns verwendet? Wenn ja, wo?)

A.6.6 Verhaltens-Diagramme

A.7 Implementierung

(Interessante, kommentierte Code-Snipets wären hier ideal.)

A.8 Testen

A.8.1 Anwendungsfälle

A.9 Systemintegration

A.9.1 Teilsysteme

A.9.2 Komponenten

A.9.3 Bibliotheken

A.10 Einsatz

A.10.1 Benutzerdokumentation

A.10.2 Auslieferung, Installation

A.10.3 Einweisung und Schulung

A.10.4 Produktionseinsatz, Operation des Systems

A.10.5 Wartung

A.10.6 Erweiterung

A.11 Literatur

(Enthält insbesondere auch Literatur zum Fachwissen und zur Welt des Kunden.)

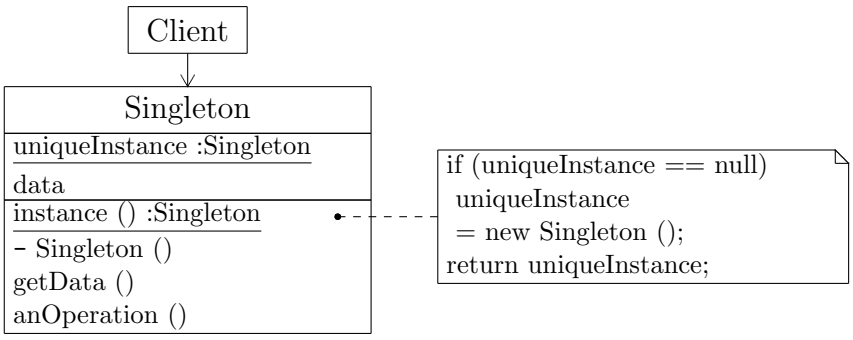
Anhang B

Entwurfsmuster

B.1 Singleton

Das Pattern ***Singleton*** stellt sicher, dass es nur *eine* Instanz einer Klasse gibt, und stellt einen globalen Zugriffspunkt dafür zur Verfügung.

Struktur



Anwendbarkeit

- Eine Klasse darf nur genau eine Instanz haben. Diese soll global zugänglich sein.
- Diese eine Instanz der Klasse sollte erweiterbar sein. Clients sollten eine erweiterte Version benutzen können ohne ihren Code zu modifizieren.
- Nur eine begrenzte Anzahl von Instanzen einer Klasse soll zur Verfügung gestellt werden.

Bemerkungen

1. Singleton erlaubt die Erweiterung von Methoden. Das geht bei einem `static` Modul (Klasse mit nur `static` Methoden) nicht.
2. Wenn das Singleton erweitert wird, dann kann es vernünftig sein, die Superklasse abstrakt zu machen.
3. Die Programmierung mit "globalen" Variablen, indem beliebig viele Singletons definiert werden, ist ein Missbrauch des Patterns Singleton
4. Anstatt mit der Methode `instance ()` könnte man auch auf das Datenelement `uniqueInstance` direkt zugreifen, das dann allerdings mit einem geeigneten Klassen-Initialisator erzeugt werden müsste.
5. Das Pattern lässt sich "leicht" so modifizieren, dass mehr als eine Instanz möglich ist. Man kann die Anzahl der Instanzen kontrollieren.

   Ferner könnte über eine parametrisierte oder weitere `instance`-Methoden auf Objekte erweiterter Klassen zugegriffen werden.
6. Unter Java kann das Singleton sinnvoll eigentlich nur durch `private static` innere Klassen erweitert werden.
7. Die Methode `instance ()` sollte in einer Multitasking-Umgebung unter gegenseitigem Ausschluss ausgeführt werden.

Abstraktes Code-Beispiel

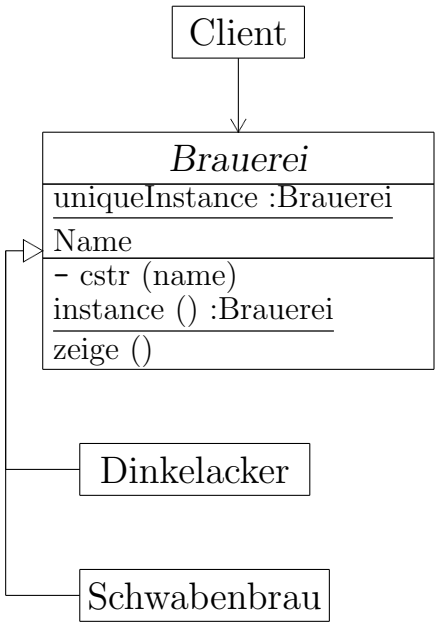
```
public class   Client
   {
   public static void   main (String[] arg)
      {
      Singleton.instance ().anOperation ();
      // ...
      Singleton.instance ().anOperation ();
      }
   }
```

```
public class   Singleton
   {
   private static Singleton   uniqueInstance;
   private String data = "irgendetwas";
   private  Singleton ()
      {
      }
   public static Singleton instance ()
      {
      if (uniqueInstance == null)
         uniqueInstance = new Singleton ();
      return uniqueInstance;
      }
   public String  getData ()
      {
      return data;
```

```
    }
public void anOperation ()
    {
    System.out.println ("anOperation mit "
       + "Objekt " + this);
    }
}
```

Beispiel Bier

Wir nehmen an, dass höchstens *eine* Brauerei in unserem System verwendet werden kann. Wir entscheiden zur Compilezeit, welche Brauerei das sein soll.



Übung

Beschränkte Anzahl von Singletons

Schreiben Sie eine Singleton-Klasse, die höchstens n Instanzen liefert. Wenn n geliefert sind, dann kann keine Instanz mehr ausgeliefert werden, bis der Garbage-Kollektor eine Instanz entfernt hat.

Hinweis: Das ist möglicherweise eine Anwendung von `finalize`, wo die Instanz vor der Zerstörung noch einmal zum Leben erweckt wird nur um sich abzumelden und dann endgültig zerstört zu werden.

Vorgehensweise: Halte n Instanzen in einem Feld. Wenn eine Instanz ausgeliefert wird, wird das Feldelement auf `null` gesetzt. Wenn alle Instanzen ausgeliefert sind, dann wird gewartet (auf einen Interrrupt), bis ein Feldelement wieder belegt ist, nämlich durch eine Instanz, die zerstört

werden soll, und die sich im letzten Moment wieder auf ihren Feldplatz setzt, den sie kennt. Sie löst auch den Interrupt aus. Dann wird sie endgültig dem GC übergeben und es kann eine neue Instanz erzeugt werden.

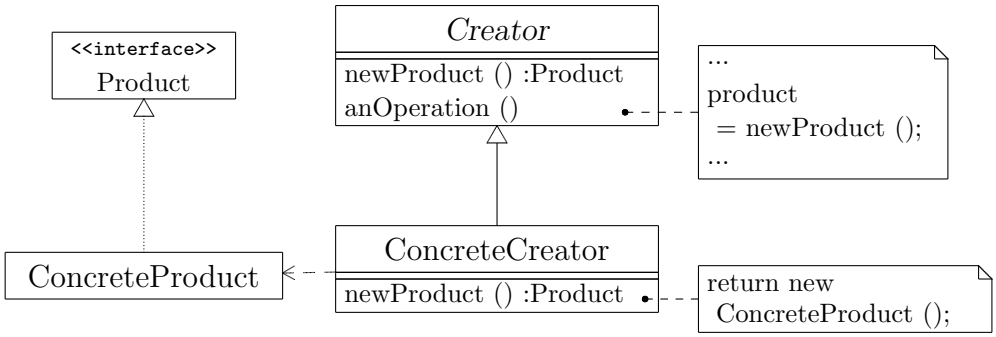
Beispiel: Datenbankverbindungen.

Schreiben Sie auch JUnit-Test-Code dazu. Das ist möglicherweise schwierig.

B.2 Factory Method

Das Pattern ***Factory Method*** oder auch ***Virtual Constructor*** erlaubt die Erzeugung von verwandten Objekten (Varianten), ohne dass zur Compilezeit die konkrete Klasse spezifiziert werden muss. (Im Gegensatz zum Pattern Abstract Factory (siehe unten) haben wir es hier nur mit **einem** Produkttyp, nicht mit einer ganzen Produktfamilie zu tun.)

Struktur



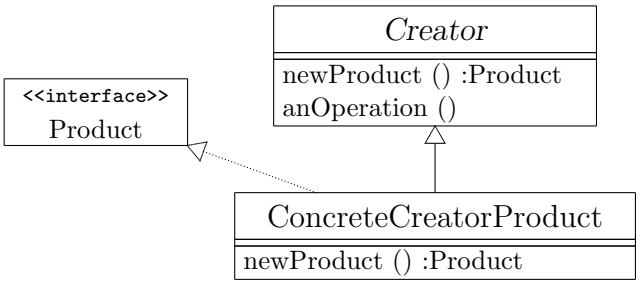
Anwendbarkeit

- Eine Klasse (der Creator) weiß nicht, welche Art von Objekten sie erzeugen muss.
- Subklassen einer Klasse sollen spezifizieren, welche Objekte erzeugt werden sollen.
- Kopplung paralleler Klassenhiearchien.

Bemerkungen

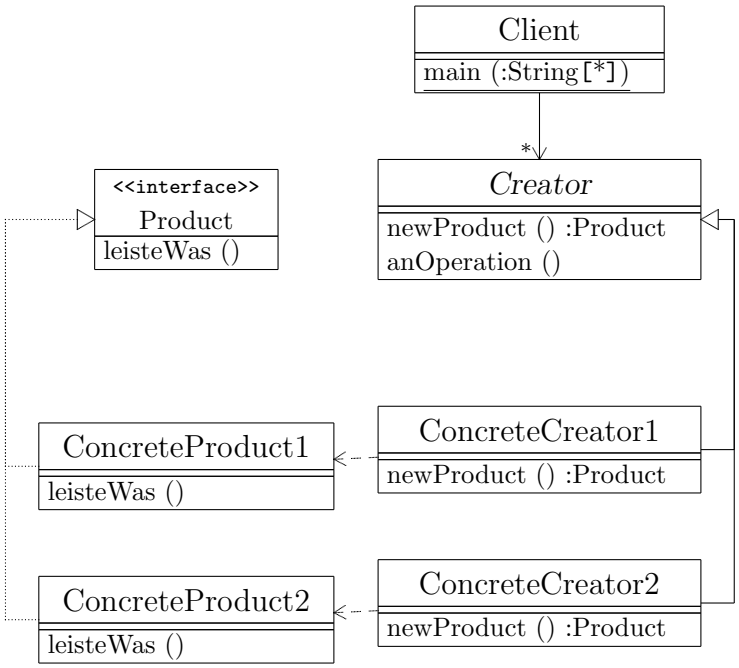
1. Factory-Methoden machen die Anwendung unabhängig von Anwendungs-spezifischen Klassen (Architekturunabhängigkeit).
2. Allerdings muss Creator jedesmal erweitert werden, wenn ein neues Produkt dazukommt.
3. Die Kopplung paralleler Klassenhierarchien ist damit möglich. Parallele Klassenhierarchien enstehen, wenn eine Klasse in einer Hierarchie Verantwortlichkeiten an eine andere Klasse delegiert, die dann meistens in einer dazu parallelen Hierarchie steht.
4. `Creator` muss nicht abstrakt sein.
5. Mit parametrisierten Factory-Methoden kann man die Vererbung umgehen.
6. Die `Product`-Hierarchie und die `Creator`-Hierarchie können oft zu einer Hierarchie zusammengefasst werden.

   Das sieht dann folgendermaßen aus:



7. Wir können überladene Methoden `newProdukt (...)` verwenden, um unterschiedliche Konstruktoren darzustellen.

Abstraktes Code-Beispiel




```
public class   Client
   {
   public static void   main (String[] aString)
      {
      new ConcreteCreator1 ().anOperation ();
      new ConcreteCreator2 ().anOperation ();
      }
   }
```

```
public interface  Product
   {
   void  leisteWas ();
   }
```

```
public class   ConcreteProduct1
   implements  Product
   {
   public void leisteWas ()
      {
      System.out.println ("Das ConcreteProduct1 "
         + "tut etwas für den");
      System.out.println ("   ConcreateCreator1.");
      }
   }
```

```
public class   ConcreteProduct2
   implements  Product
   {
   public void leisteWas ()
      {
      System.out.println ("Das ConcreteProduct2 "
         + "tut etwas für den");
      System.out.println ("   ConcreateCreator2.");
      }
   }
```

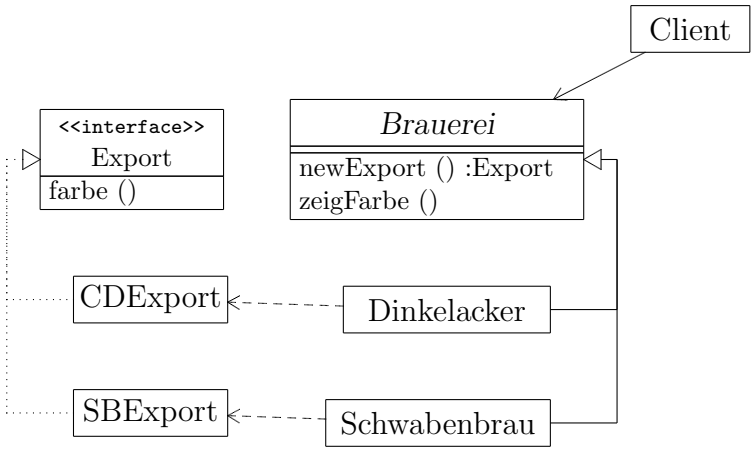
```
public abstract class   Creator
   {
   public abstract Product newProduct ();
   public void anOperation ()
      {
      Product  product = newProduct ();
      System.out.println ("Der " + getClass ().getName ()
         + " macht sich ein passendes");
      System.out.println ("   Product, nämlich "
         + product.getClass ().getName () + ", und");
      System.out.println ("   lässt dieses für sich arbeiten:");
      product.leisteWas ();
      }
   }
```

```
public class   ConcreteCreator1
   extends     Creator
   {
   public Product newProduct ()
      {
      return new ConcreteProduct1 ();
      }
   }
```

```
public class   ConcreteCreator2
   extends     Creator
   {
   public Product newProduct ()
      {
      return new ConcreteProduct2 ();
      }
   }
```

Beispiel Bier

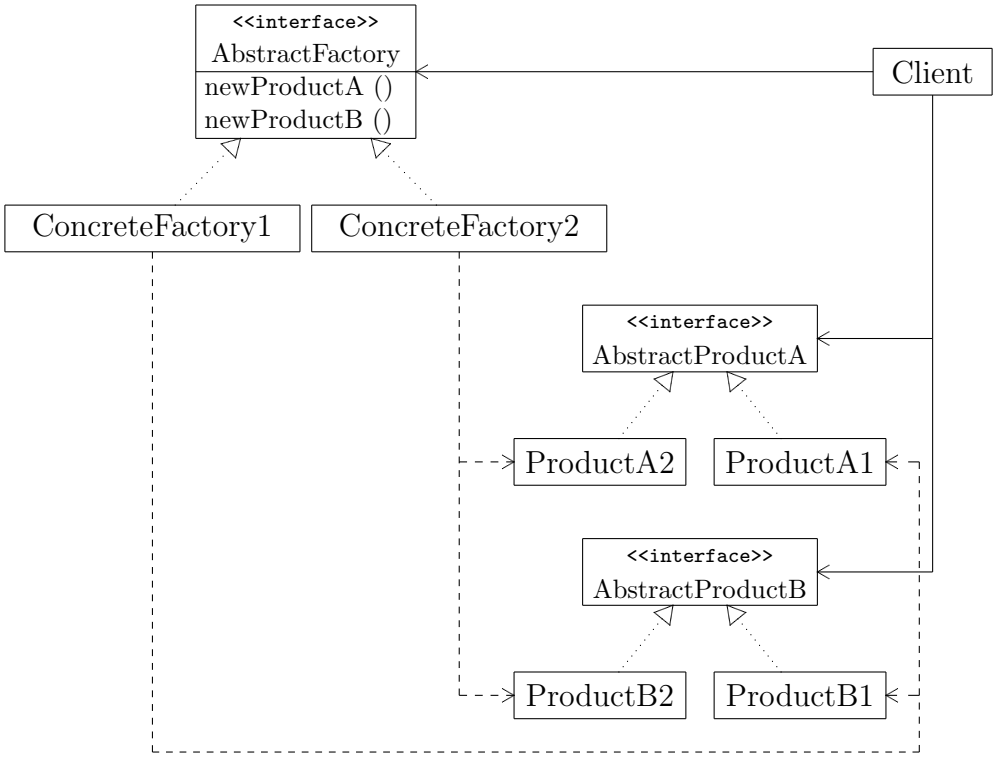
Als Creators haben wir mehrere Brauereien und als Produkt ein Export-Bier. Jede Brauerei stellt nur ihr Export-Bier her.



B.3 Abstract Factory

Das Pattern ***Abstract Factory*** oder kurz ***Factory*** oder ***Kit*** erlaubt die Erzeugung von verwandten Objekten (Varianten), ohne dass zur Compilezeit die konkrete Klasse spezifiziert werden muss. Die erzeugten Objekte heißen in diesem Zusammenhang "Produkte". Jede Fabrik kann eine "Produktfamilie" erzeugen. (Produkte, die irgendwie zusammengehören (z.B. Schraube und Mutter) bilden eine Familie. D.h. eine Fabrik macht Schrauben und Muttern aus Stahl, die andere aus Kunststoff.)

Struktur



Anwendbarkeit

- Ein System sollte mit einer von mehreren Produktfamilien konfigurierbar sein.
- Eine Familie von aufeinander bezogenen Objekten muss gemeinsam benutzt werden, und diese Bedingung soll automatisch erfüllt sein.
- Eine Klassenbibliothek von Produkten soll so geschrieben werden, dass nur ihre Schnittstellen, nicht die Realisierungen gezeigt werden.

Bemerkungen

1. Eine Factory kapselt die Erzeugung von Klassen und trennt Clients damit von Realisierungen. Clients manipulieren Objekte nur durch ihre Schnittstellen. Der Client-Code kennt keine Produktklassennamen. Architekturunabhängigkeit.

2. Eine konkrete Factory-Klasse erscheint typischerweise nur einmal in einer Anwendung, nämlich dort, wo sie instanziiert wird. Daher kann sie leicht ausgetauscht werden.

3. Die Konsistenz zwischen Produkten wird gefördert.

4. Erweiterung auf ein neues Produkt (`AbstractProductC`) ist schwierig, weil dazu die Schnittstelle `AbstractFactory` und alle realisierenden Factory-Klassen angepasst werden müssen.

5. Factories sind oft Singletons.

Abstraktes Code-Beispiel

```
public class   Client
   {
   public static void   main (String[] aString)
      {
      AbstractFactory[] factories = new AbstractFactory[]
         { new ConcreteFactory1 (), new ConcreteFactory2 () };
      for (AbstractFactory f :factories)
         {
         AbstractProductA  a = f.newProductA ();
         AbstractProductB  b = f.newProductB ();
         System.out.println (f.getClass ().getName () + " macht Produkte:");
         a.leisteWasWieA ();
         b.leisteWasWieB ();
         }
      }
   }
```

```
public interface  AbstractFactory
   {
   AbstractProductA  newProductA ();
   AbstractProductB  newProductB ();
   }
```

```
public class   ConcreteFactory1
   implements  AbstractFactory
   {
   public AbstractProductA newProductA ()
      {
      return new ProductA1 ();
      }
   public AbstractProductB newProductB ()
      {
      return new ProductB1 ();
      }
```

```
   }
```

```
public class   ConcreteFactory2
   implements  AbstractFactory
   {
   public AbstractProductA newProductA ()
      {
      return new ProductA2 ();
      }
   public AbstractProductB newProductB ()
      {
      return new ProductB2 ();
      }
   }
```

```
public interface  AbstractProductA
   {
   void  leisteWasWieA ();
   }
```

```
public class   ProductA1
   implements  AbstractProductA
   {
   public void leisteWasWieA ()
      {
      System.out.println ("Das ProductA "
         + "der Produktfamilie 1 tut etwas.");
      }
   }
```

```
public class   ProductA2
   implements  AbstractProductA
   {
   public void leisteWasWieA ()
      {
      System.out.println ("Das ProductA "
         + "der Produktfamilie 2 tut etwas.");
      }
   }
```

```
public interface  AbstractProductB
   {
   void  leisteWasWieB ();
   }
```

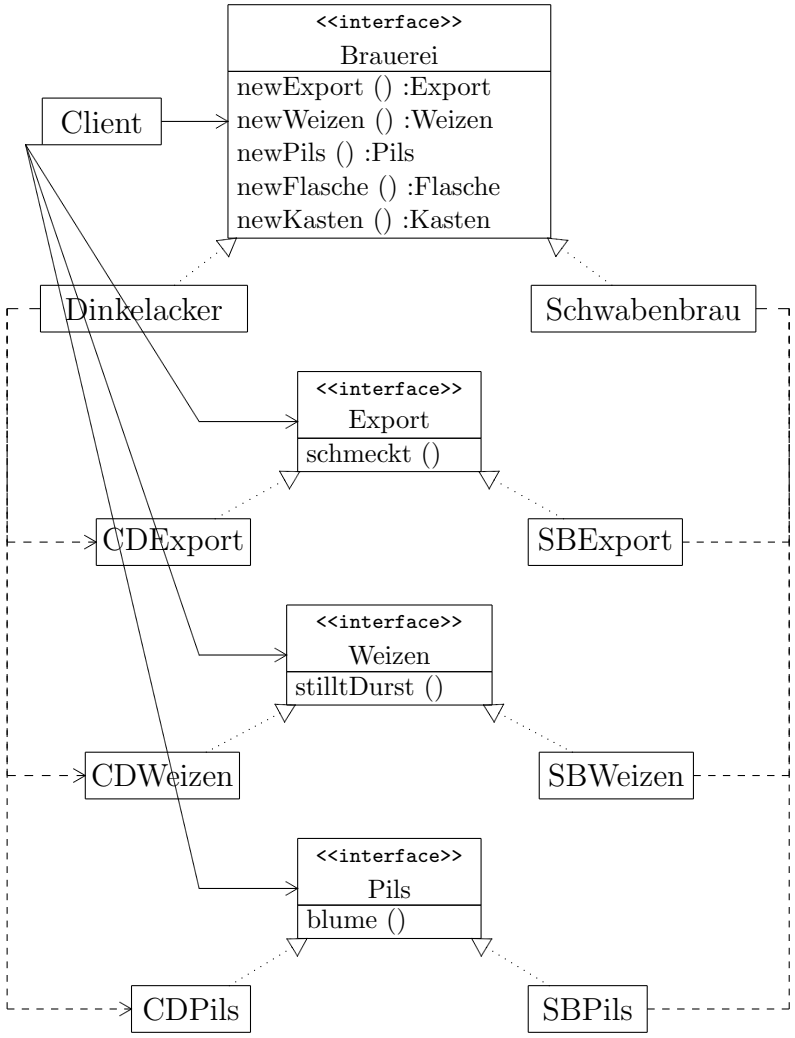
```
public class   ProductB1
   implements  AbstractProductB
   {
```

```
   public void leisteWasWieB ()
      {
      System.out.println ("Das ProductB "
         + "der Produktfamilie 1 tut etwas.");
      }
   }
```

```
public class   ProductB2
   implements  AbstractProductB
   {
   public void leisteWasWieB ()
      {
      System.out.println ("Das ProductB "
         + "der Produktfamilie 2 tut etwas.");
      }
   }
```

Beispiel Bier

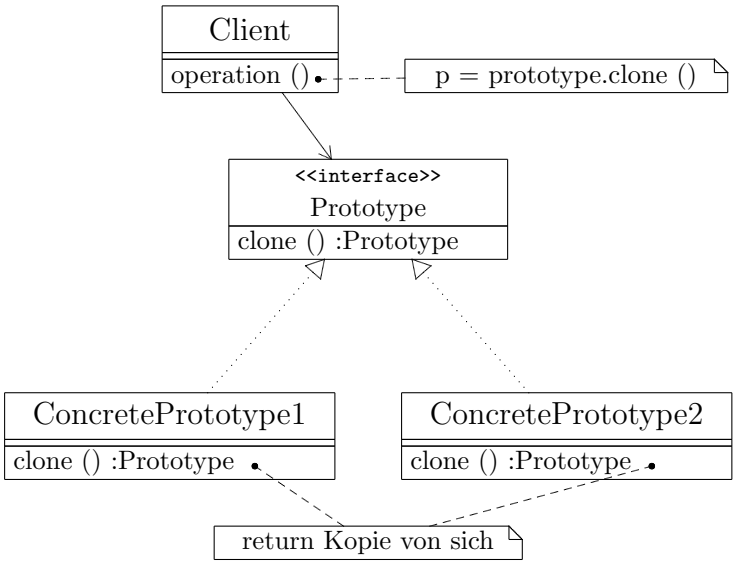
Wir haben hier die Produktfamilie bestehend aus Export, Weizen, Pils, Flasche und Kasten. `Brauerei` ist die Abstract Factory, die von den verschiedenen Brauereien realisiert wird.


<<interface>>
Brauerei
newExport () :Export
newWeizen () :Weizen
newPils () :Pils
newFlasche () :Flasche
newKasten () :Kasten
Client
Dinkelacker
Schwabenbrau
<<interface>>
Export
schmeckt ()
CDExport
SBExport
<<interface>>
Weizen
stilltDurst ()
CDWeizen
SBWeizen
<<interface>>
Pils
blume ()
CDPils
SBPils

B.4 Prototype

Ein prototypisches Objekt wird verwendet, um die Art der benötigten Objekte zu spezifizieren. Diese Objekte werden durch Kopie des Prototypen erzeugt.

Struktur



Anwendbarkeit

Das Entwurfsmuster ***Prototype*** wird verwendet, wenn eine Anwendung **viele** gleichartige Objekte benötigt, die zur Laufzeit erzeugt werden, deren Zustand oder auch Typ zur Compilezeit nicht bekannt ist. Also:

- Klasse und Zustand von zu erzeugenden Objekten werden zur Laufzeit spezifiziert (Dynamisches Laden von Objekten).
- Man möchte den Bau einer zu den Produkten parallelen Factory-Hierarchie vermeiden.
- Die Objekte einer Klasse können nur wenige unterschiedliche Zustände annehmen. Wenn es umständlich ist, diese Zustände genau zu beschreiben, etwa durch einen komplizierten Konstuktor-Aufruf, dann mag es bequemer sein, für diese wenigen Zustände Prototypen zu verwenden, die dann kopiert werden.

Bemerkungen

Die Vorgehensweise ist immer so, dass ein Prototyp-Objekt bereitgestellt wird. Dieser Prototyp kann

- voreingestellt sein oder
- von einem Prototyp-Manager geliefert werden oder
- vom Benutzer zur Verfügung gestellt werden.

In Java gibt es für das "Klonen" verschiedene Möglichkeiten. Eine korrekte Überschreibung der Methode `clone ()` hat den Vorteil, dass eine flache Kopie angelegt wird, d.h. alle Referenzen des Prototyps auf Objekte werden mitübergeben. Außerdem werden Objekte mit `clone ()` wahrscheinlich am schnellsten erzeugt.

Beispiele

- Sportfest, bei dem die Art der durchgeführten Disziplinen erst zur Laufzeit feststeht.
- Schachspielserver, der verschiedene Spielertypen anbietet.
- Ein Netzwerkserver, der in Abhängigkeit von Laufzeitparametern unterschiedliche Dienste anbietet.
- Artificial Life: Wir schreiben Code für eine Welt von Lebewesen. Diese Lebewesen werden zur Laufzeit mit unterschiedlichen Lebewesensarten instanziiert.
- Ein Regelkreis-Testprogramm, das verschiedene Regler-Typen und Strecken-Typen ausprobieren soll. Regler-Typen sollten leicht austauschbar sein.

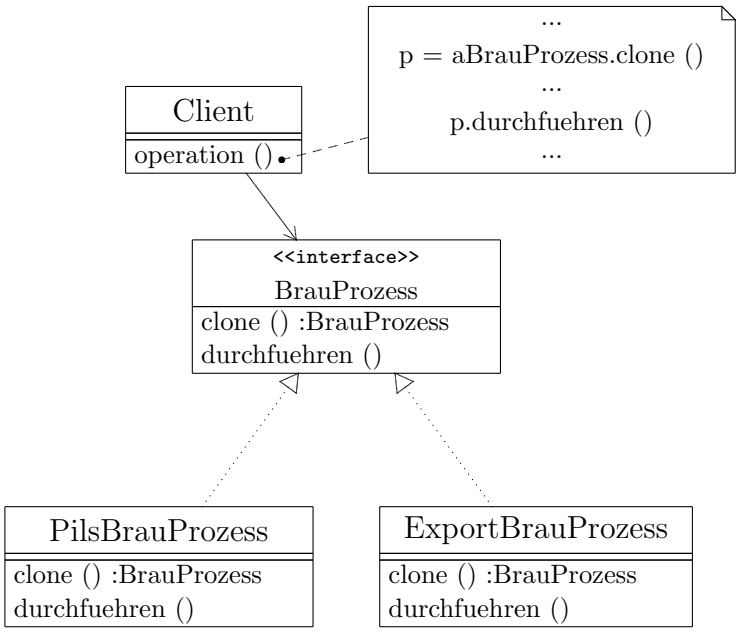
Beispiel Bier

Beim Brauprozess können sehr viele Parameter wie Temperaturen, Zeitenparameter, Konzentrationen usw. eingestellt werden. In einer modernen Brauerei werden all diese Parameter erfasst und könnten etwa in einem Objekt vom Typ `BrauProzess` verwaltet werden. Diese Objekte könnten auch verwendet werden, um aktuelle Brauprozesse durchzuführen.

Wenn eine Charge, d.h. ein Brauprozess besonders gut gelungen ist, dann möchte man diesen natürlich gern wiederholen. Hier bietet sich an, den besonders erfolgreichen Brauprozess zu **klonen** und als Prototypen für den nächsten Brauprozess zu verwenden.

Bei vielleicht 100 Einstellparametern wäre ein entsprechender Konstruktor zu kompliziert wie auch die nachträgliche Einstellung.

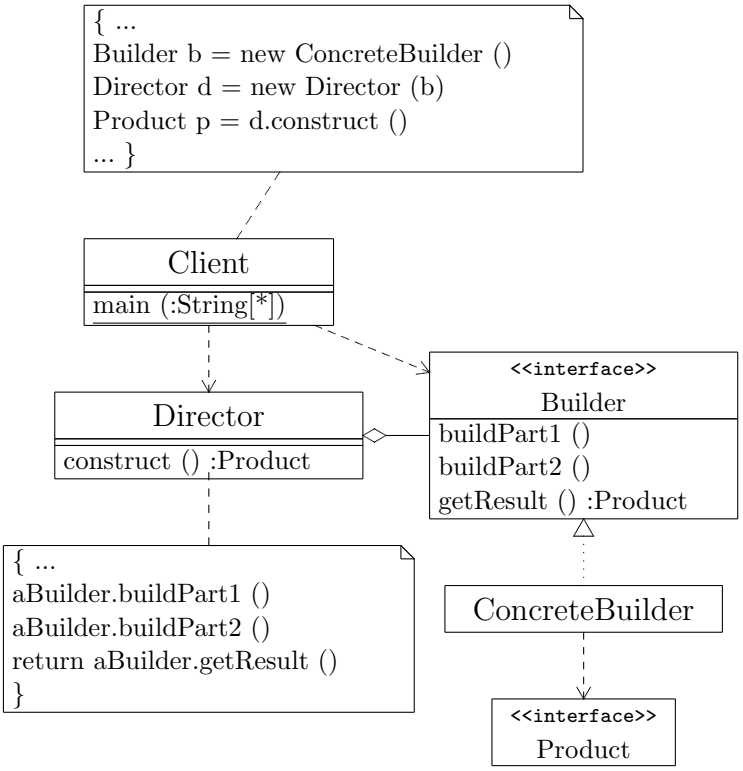
Ferner gibt es für die verschiedenen Biersorten Pils, Export usw. unterschiedliche Brauprozesse und auch Prototypen.


...
p = aBrauProzess.clone ()
...
p.durchfuehren ()
...
Client
operation ()
<<interface>>
BrauProzess
clone () :BrauProzess
durchfuehren ()
PilsBrauProzess
clone () :BrauProzess
durchfuehren ()
ExportBrauProzess
clone () :BrauProzess
durchfuehren ()

B.5 Builder

Das Entwurfsmuster ***Builder*** separiert den Erzeugungsprozess eines komplexen Objekts von der Erzeugung der Teile des komplexen Objekts.

Struktur



Anwendbarkeit

- Der Algorithmus zur Erzeugung eines komplexen Objekts soll unabhängig von den konkreten Teilen des Objekts sein. Die Teile sollen konsistent erzeugt werden.
- Der Erzeugungs-Prozess soll unterschiedliche Repräsentationen des Objekts erlauben.

Bemerkungen

1. Dieses Muster ist vergleichbar mit ***Template Methode***. Bei ***Builder*** liegt der Schwerpunkt auf der variablen Erzeugung von Objekten, dort auf der variablen Durchführung eines Algorithmus.

2. Der Client bestimmt, welcher Builder dem Director zur Verfügung gestellt wird.
3. Die interne Repräsentation eines Produkts kann unabhängig vom Erzeugungsprozess variiert werden.
4. Objekt-Konstruktions-Code und Objekt-Repräsentations-Code werden isoliert.
5. Der Konstruktions-Code kann genauer gesteuert werden.

Abstraktes Code-Beispiel

```
public class   Client
   {
   public static void   main (String[] aString)
      {
      System.out.println ("Client:   Macht sich einen Builder.");
      Builder  b = new ConcreteBuilder ();
      System.out.println ("Client:   Macht sich einen Director.");
      Director d = new Director (b);
      System.out.println ("Client:   Und lässt ihn das Produkt bauen.");
      Product  p = d.construct ();
      }
   }
```

```
public class   Director
   {
   private Builder   aBuilder;
   public final Builder getBuilder () {return aBuilder;}
   private final void   setBuilder (Builder b) {aBuilder = b;}
   public   Director (Builder aBuilder) {setBuilder (aBuilder);}
   public Product construct ()
      {
      System.out.println ("Director:   Lässt von Builder Produkt bauen.");
      getBuilder ().buildPart1 ();
      getBuilder ().buildPart2 ();
      Product  p = getBuilder ().getResult ();
      System.out.println ("Director:   Gibt das Produkt zurück.");
      return p;
      }
   }
```

```
public interface  Builder
   {
   void  buildPart1 ();
   void  buildPart2 ();
   Product  getResult ();
   }
```

```
public class   ConcreteBuilder
   implements  Builder
   {
   public void buildPart1 ()
      {
```

```
      System.out.println ("ConcreteBuilder:   Erstelle Teil 1");
      }
   public void buildPart2 ()
      {
      System.out.println ("ConcreteBuilder:   Erstelle Teil 2");
      }
   public Product getResult ()
      {
      System.out.println ("ConcreteBuilder:   Baut Produkt aus den Teilen");
      System.out.println ("                     und gibt es zurück.");
      return null;   // new XProduct ()
      }
   }
```

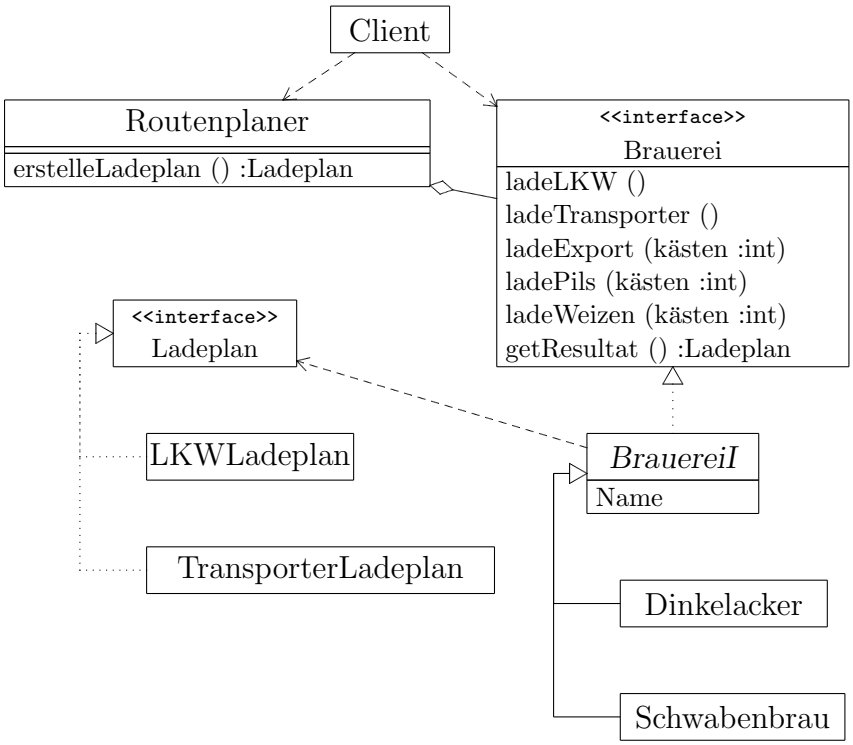
```
public interface  Product
   {
   }
```

Beispiel Bier

Lastwagen sollen für die Lieferung von Bier so beladen werden, dass sie ohne Probleme beim Kunden entladen werden können.

Der `Director` ist hier der `Routenplaner`, der ermitteln kann, in welcher Reihenfolge die Kunden angefahren werden müssen und wieviel Kästen Bier jeder Kunde geliefert bekommt.

Der `Builder` ist die `Brauerei`. Sie weiß, wie man einen Lastwagen belädt bei einer vorgegebenen Reihenfolge von Bierkästen, damit die Kästen in der angegebenen Reihenfolge auch wieder ausgeladen werden können. Das macht eventuell jede Brauerei anders.


Client
Routenplaner
erstelleLadeplan () :Ladeplan
<<interface>>
Brauerei
ladeLKW ()
ladeTransporter ()
ladeExport (kästen :int)
ladePils (kästen :int)
ladeWeizen (kästen :int)
getResultat () :Ladeplan
<<interface>>
Ladeplan
LKWLadeplan
TransporterLadeplan
BrauereiI
Name
Dinkelacker
Schwabenbrau

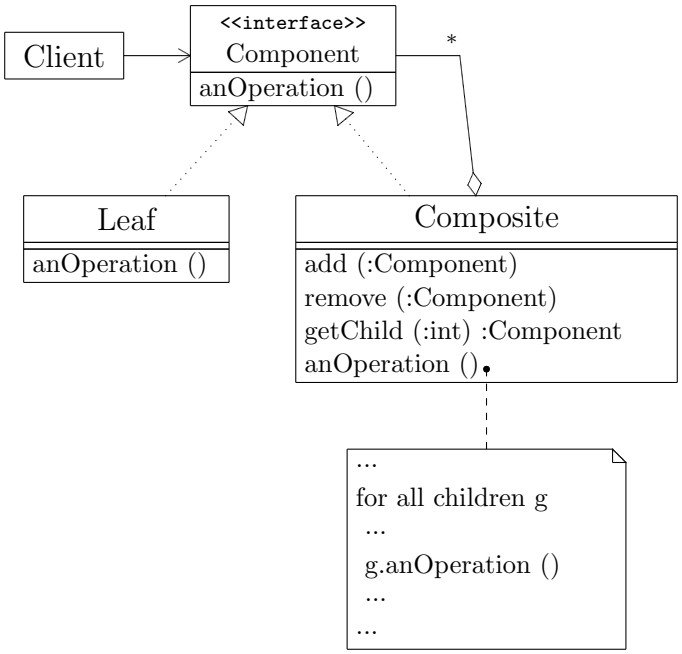
B.6 Composite

Das Entwurfsmuster ***Composite*** wird verwendet, wenn Ganzes-Teil-Hierarchien als Baumstrukturen repräsentiert werden sollen und Knoten und Blätter einheitlich behandelt werden sollen.

Dieses Muster gehört zu den strukturellen Mustern und ist sehr verbreitet.

Wir präsentieren dieses Muster in einer Form, die sicherer als die von Gamma et al.[17], dafür aber weniger transparent ist.

Struktur



Anwendbarkeit

- Repräsentation von Teil-Ganzes-Hierarchien ***(whole-part-hierarchy)***
- Client will Aggregate (Verbünde) und elementare Objekte gleichartig behandeln.

Teilnehmer

Component: Definiert die Schnittstelle der Objekte in der Composite-Struktur und kann daher in den meisten Fällen unter Java ein `interface` sein. In unserer Variante hat `Component` keine Methoden oder Schnittstellen zur Manipulation von Kind-Komponenten.

Erweiterung: Häufig ist es notwendig, eine Referenz für den Zugriff auf Elter-Komponenten zu definieren.

Leaf: Repräsentiert Blatt-Objekte (Elementar-Objekte) und realisiert die Schnittstelle `Component`.

Composite: Ist ein Aggregat (Verbund-Objekt), enthält Kind-Komponenten und definiert Manipulations-Methoden (Addieren, Entfernen) für Kind-Komponenten. `Composite` realisiert die Schnittstelle `Component` typischerweise so, dass die Methoden von `Component` für alle Kinder ausgeführt werden.

Oft ist es so, dass `Composite Leaf` erweitert. In dem Fall würden `Component` und `Leaf` wahrscheinlich zu einer Klasse verschmelzen.

Client: Manipuliert Komponenten-Objekte (`Leaf` oder `Composite`) durch die Schnittstelle `Component`. Wenn Kinder einer Komponente manipuliert werden, muss der Client allerdings wissen, dass es sich um ein `Composite` handelt. Hier nehmen wir bewusst einen Mangel an Transparenz in Kauf um unsinnige Operationen auf Blättern zu verhindern.

Bemerkungen

1. Die Verwaltung einer Referenz auf den Elter (übergeordneter Verbund) ist oft sehr nützlich.
2. Verwende das State-Pattern um eventuell ein `Leaf` in ein `Composite` zu verwandeln.
3. Normalerweise kann ein `Component` höchstens in *einem* `Composite` sein. Daher wurde auch die Aggregation im UML-Diagramm gewählt. Allerdings ist natürlich auch eine Struktur denkbar, bei der ein `Component` in mehr als einem `Composite` ist. Man muss dann dafür sorgen, dass es bei der Traversierung des Baums nicht zu unendlichen Rekursionen kommt.
4. Sicher ist es sinnvoll `Composite` als Schnittstelle oder abstrakte Klasse zu definieren, die `Component` erweitert bzw. realisiert. `Composite` wird dann von konkreten Composites realisiert oder erweitert.

Abstraktes Code-Beispiel

Da wir im folgenden Code-Beispiel für `Leaf` und `Composite` einen Namen verwalten wollen, haben wir anstatt der Schnittstelle `Component` eine abstrakte Klasse `Component` genommen.

```
public class   Client
   {
   public static void   main (String[] arg)
      {
      Component   b1 = new Leaf ("Punkt P1");
      Component   b2 = new Leaf ("Punkt P2");
      Component   b3 = new Leaf ("Punkt P3");
      Component   b4 = new Leaf ("Punkt P4");
      Component   b5 = new Leaf ("Punkt P5");
      Component   k1 = new Composite ("Gerade G1");
      ((Composite) k1).add (b1);
      ((Composite) k1).add (b2);
      Component   k2 = new Composite ("Rechteck R1");
      ((Composite) k2).add (b3);
      ((Composite) k2).add (b4);
      ((Composite) k2).add (k1);
      Component   w = new Composite ("Wurzel");
      ((Composite) w).add (k2);
      ((Composite) w).add (b5);
```

```
      w.anOperation (0);
      }
   }
```

```
public abstract class   Component
   {
   protected String  name;
   public   Component (String name)
      {
      this.name = name;
      }
   public void anOperation (int stufe)
      {
      for (int i = 0; i < stufe; i++) System.out.print ("   ");
      System.out.println (name);
      }
   }
```

```
public class   Leaf extends Component
   {
   public   Leaf (String name) { super (name); }
   public void anOperation (int stufe)
      {
      System.out.print ("Blatt:    ");
      super.anOperation (stufe);
      }
   }
```

```
import   java.util.*;
public class   Composite extends Component
   {
   private Vector<Component>  v = new Vector<Component> ();
   public   Composite (String name) { super (name); }
   public void anOperation (int stufe)
      {
      System.out.print ("Knoten:   ");
      super.anOperation (stufe);
      for (int i = 0; i < v.size (); i++)
         getChild (i).anOperation (stufe + 1);
      }
   public void add (Component b) { v.add (b); }
   public void remove (Component b) { v.remove (b); }
   public Component  getChild (int i) { return v.get (i); }
   }
```

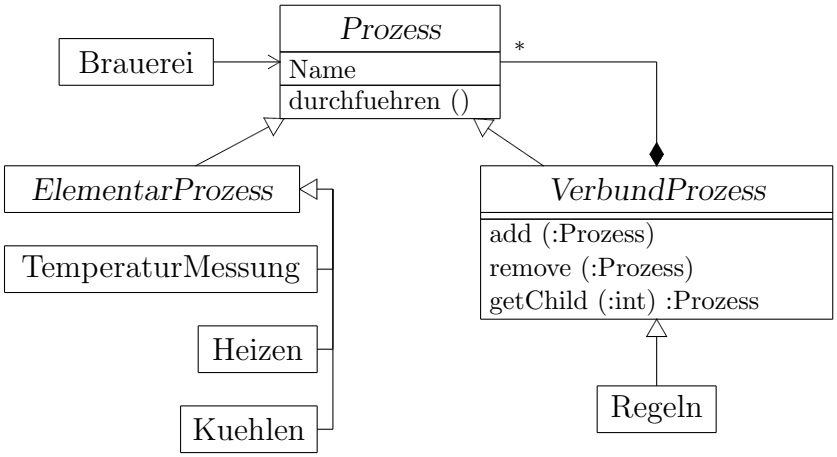
Beispiele

Hierarchisches Filesystem
Prozesse und Teilprozesse
Koordinatentransformationen

Graphische Komponenten, die sich aus Teilkomponenten zusammensetzen

Beispiel Bier

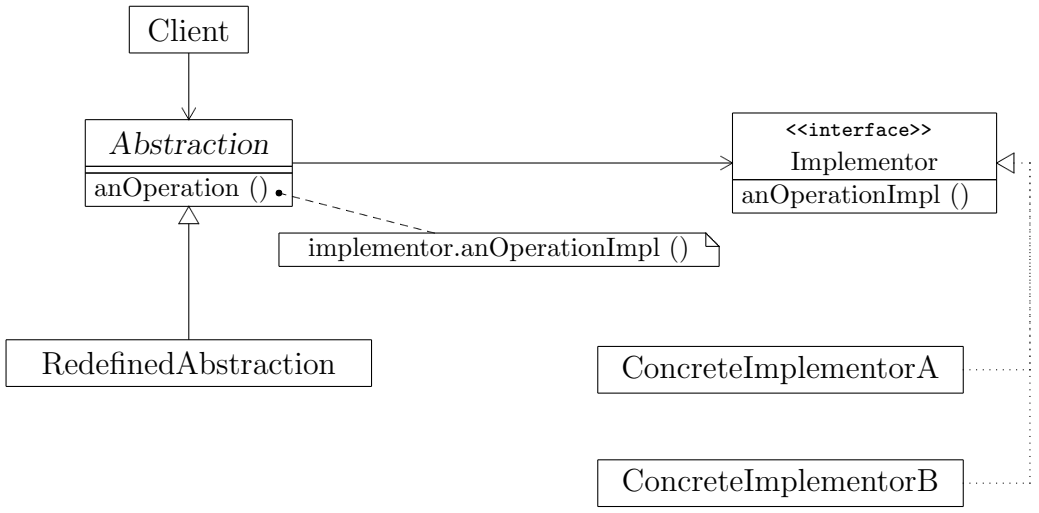
Der Brauprozess setzt sich hierarchisch zusammen aus Teilprozessen, die sich schließlich aus Elementarprozessen (Temperaturmessung, Rühren, Heizen, Kühlen usw.) zusammensetzen. Sie alle realisieren die Methode `durchfuehren ()`.



B.7 Bridge

Das Entwurfsmuster ***Bridge*** oder auch ***Handle/Body*** entkoppelt eine Abstraktion von ihrer Implementierung, so dass beide unabhängig variieren können.

Struktur



Anwendbarkeit

- Eine permanente Bindung zwischen Abstraktion und Implementierung ist unerwünscht, insbesondere wenn die Implementierung zur Laufzeit wählbar bzw. austauschbar sein soll.
- Abstraktion und Implementierung sollten durch eine Vererbungshierarchie erweiterbar sein. Die verschiedenen Abstraktionen und Implementierungen sollten frei miteinander kombinierbar sein.
- Änderungen in der Implementierung sollten nicht zur Recompilierung der Abstraktion führen.
- Die Implementierung einer Abstraktion soll versteckt werden.
- Wir haben eine Klassenexplosion *("nested generalizations")*, weil wir zwei oder mehr Aspekte eines Objekts variieren.
- Wir möchten eventuell aus einer Menge von Klassen mit gemeinsamem Basistyp variabel und/oder zur Laufzeit festlegbar eine Klasse erben. Wir verwenden das Bridgepattern und realisieren Vererbung durch Komposition.

Bemerkungen

1. `Implementor` kann auch eine (abstrakte) Klasse sein.
2. Bei je 10 Klassen jeweils auf der Abstraktions- und der Implementierungsseite müssten wir ohne dieses Pattern 100 Klassen anbieten, um alle Kombinationen abzudecken.

Abstraktes Code-Beispiel

```
public class   Client
   {
   public static void   main (String[] argument)
      {
      Abstraction abstraction
         = new RedefinedAbstraction (
            new ConcreteImplementorA ());
      abstraction.anOperation ();
      abstraction.implementor
         = new ConcreteImplementorB ();
      abstraction.anOperation ();
      }
   }
```

```
public abstract class   Abstraction
   {
   public Implementor   implementor;
   public   Abstraction (Implementor implementor)
      {
      this.implementor = implementor;
      }
   public void anOperation ()
      {
      implementor.anOperationImpl ();
      }
   }
```

```
public class   RedefinedAbstraction
   extends     Abstraction
   {
   public   RedefinedAbstraction (
      Implementor implementor)
      {
      super (implementor);
      }
   public void anOperation ()
      {
      System.out.println ("RedefinedAbstraction"
         + " verwendet");
      System.out.print ("   ");
      super.anOperation ();
      }
   }
```

```
public interface  Implementor
   {
```

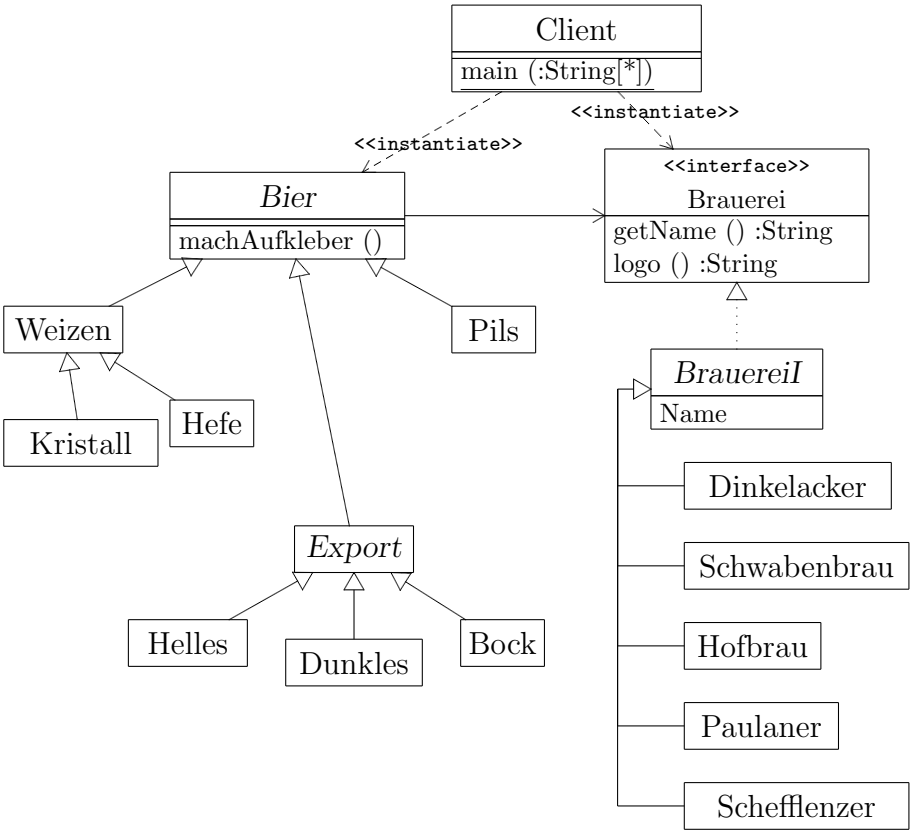
```
   void  anOperationImpl ();
   }
```

```
public class   ConcreteImplementorA
   implements  Implementor
   {
   public void anOperationImpl ()
      {
      System.out.println ("ConcreteImplementorA");
      }
   }
```

```
public class   ConcreteImplementorB
   implements  Implementor
   {
   public void anOperationImpl ()
      {
      System.out.println ("ConcreteImplementorB");
      }
   }
```

Beispiel Bier

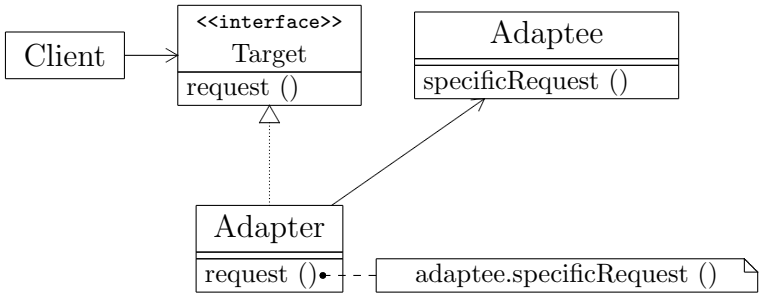
Für verschiedene Biersorten (Pils, Weizen, Export usw.) wollen wir für verschiedene Biermarken Flaschenaufkleber drucken. Bei 7 Biersorten und 5 Biermarken müssten wir 35 Klassen schreiben. Mit dem Bridgepattern sind nur 12 Klassen notwendig.


Client
main (:String[*])
<<instantiate>>
<<instantiate>>
<<interface>>
Brauerei
getName () :String
logo () :String
Bier
machAufkleber ()
Weizen
Pils
Kristall
Hefe
Export
Helles
Dunkles
Bock
BrauereiI
Name
Dinkelacker
Schwabenbrau
Hofbrau
Paulaner
Schefflenzer

B.8 Adapter

Das Entwurfsmuster ***Adapter*** oder auch ***Wrapper, Emulation*** konvertiert die Schnittstelle einer Klasse ***(Adaptee)*** in eine andere Schnittstelle ***(Target)***, nämlich eine Schnittstelle, die der Client erwartet.

Struktur

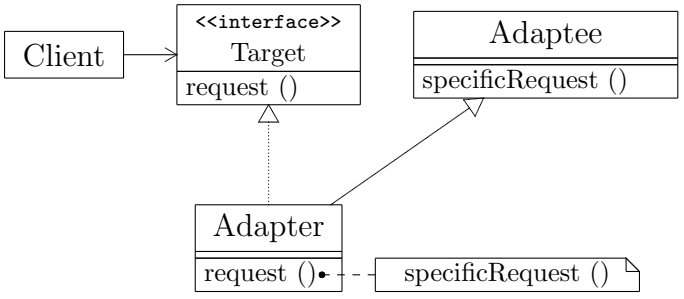


Anwendbarkeit

- Wir wollen eine vorhandene Klasse oder Klassenhierarchie verwenden, deren Schnittstelle nicht passt.

Bemerkungen

1. Wir haben hier als allgemeine Struktur den sogenannten ***object adapter*** gezeigt, da er wesentlich allgemeiner einsetzbar ist als der sogenannte ***class adapter***, bei dem der Adapter von Adaptee erbt.



2. Wenn das Target sehr viele Methoden hat, aber nur wenige adaptiert werden müssen (d.h. Target und Adaptee haben viele Methoden gemeinsam.), dann kann ein Class-Adapter bequemer sein.

3. Adaptee kann natürlich auch eine Schnittstelle sein.

4. Adaptee kann eine Vererbungshierarchie im Gefolge haben (beim *object adapter*).

5. Der Client verwendet Adapter-Instanzen.

6. Beim *class adapter* kann die zu adaptierende Methode vom Adapter überschrieben werden.

7. Ein Adapter kann im Prinzip wenig bis sehr viel tun. Er kann nur die Schnittstelle konvertieren. Er kann aber auch das ganze Verhalten ändern.

Abstraktes Code-Beispiel

```
public class   Client
   {
   public static void   main (String[] argument)
      {
      Target   target = new Adapter (new Adaptee ());
      System.out.println ("target.request ():");
      System.out.print ("   ");
      target.request ();
      }
   }
```

```
public interface  Target
   {
   void  request ();
   }
```

```
public class   Adaptee
   {
   public void specificRequest ()
      {
      System.out.println (
         "specificRequest of Adaptee (" + this + ")");
      }
   }
```

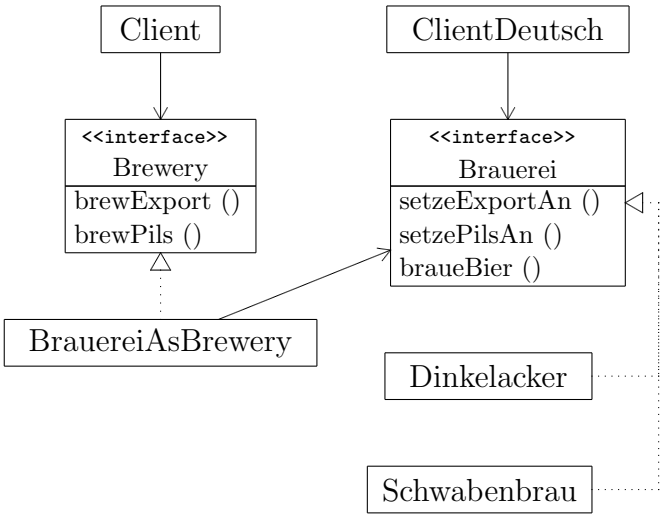
```
public class   Adapter
   implements  Target
   {
   private Adaptee   adaptee;
   public   Adapter (Adaptee adaptee)
      {
      this.adaptee = adaptee;
      }
   public void request ()
      {
      adaptee.specificRequest ();
      }
   }
```

Beispiele

Die Implementoren des Bridge-Patterns haben oft sehr unterschiedliche Schnittstellen, z.B. wenn es sich um verschiedene Grafik-Implementierungen handelt. Das Bridge-Pattern braucht aber eine einheitliche Implementor-Schnittstelle. Daher kann es hier notwendig sein, dass man Adapter einsetzt, die eine einheitliche Implementor-Schnittstelle als Target bereitstellen.

Beispiel Bier

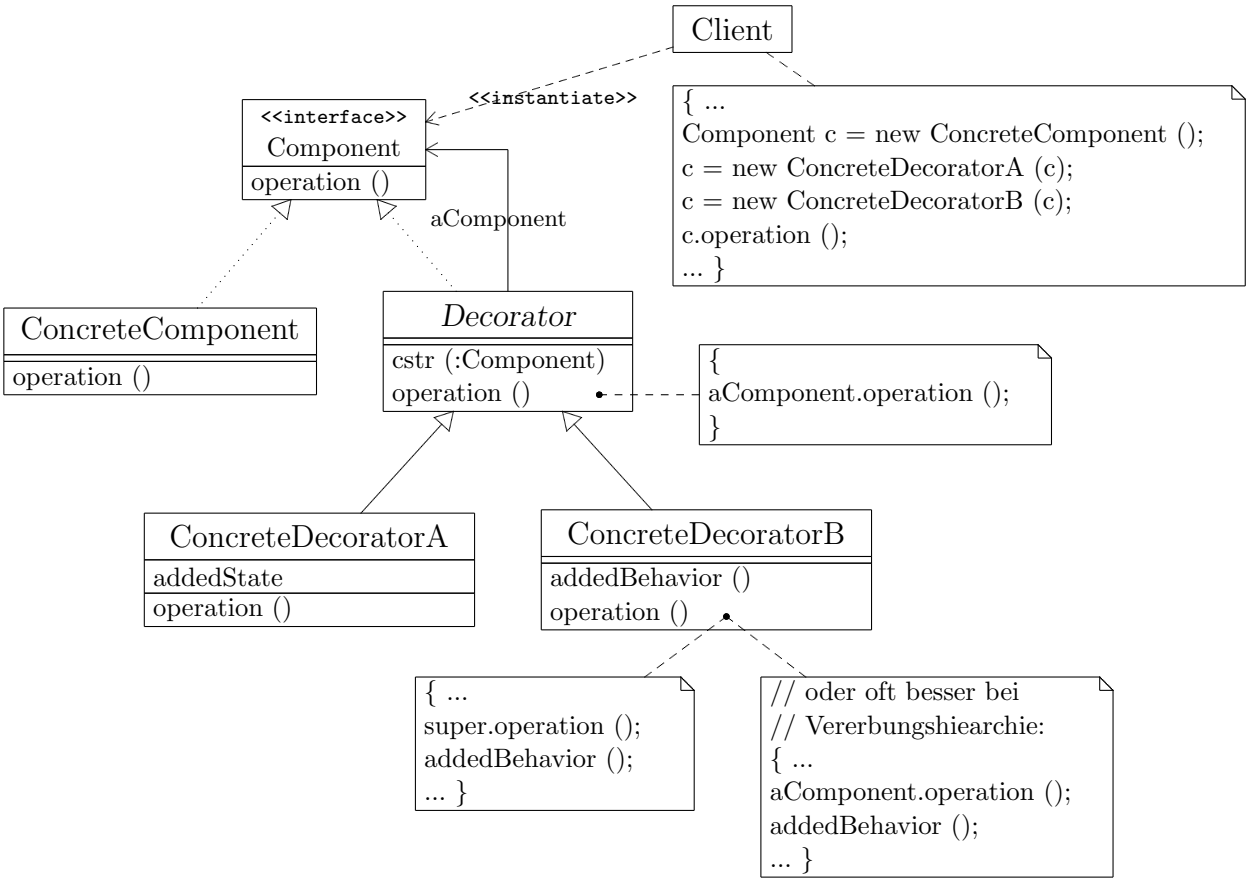
Wir haben schon eine Klassenhierachie für Brauereien, die alle eine Schnittstelle `Brauerei` mit deutschen Methodennamen implementieren. Wir wollen diese Brauereien nun auch in einer Software verwenden, die die Brauereien unter einer anderen Schnittstelle `Brewery` (englische Namen und auch etwas andere Methoden) erwartet. Dafür schreiben wir einen `BrauereiAdapter`.



B.9 Decorator

Mit dem Entwurfsmuster ***Decorator*** oder auch ***Wrapper*** können einem Objekt zur Laufzeit zusätzliche Verantwortlichkeiten, d.h. Methoden gegeben werden. Wenn ein Objekt erweitert werden soll, dann ist das eine flexible Alternative zur Vererbung.

Struktur



Anwendbarkeit

- Spezialisierung und Erweiterung individueller Objekte zur Laufzeit, ohne dass andere Objekte desselben Typs berührt werden.
- Verantwortlichkeiten, die hinzugefügt und wieder weggenommen werden können.

- Klassenexplosion wegen zu vieler Varianten oder Optionen, d.h. wenn jede Optionskombination zu einer neuen Klasse führt. Die Optionen werden dann zu Decorator-Klassen.
- Eine Klasse kann aus verschiedenen Gründen nicht vererbt werden, soll aber trotzdem erweitert werden.

  Der Decorator wird dann zum vererbbaren Wrapper.

Bemerkungen

1. Im Strukturdiagramm haben wir eine `uses-a`-Beziehung zwischen `Decorator` und `Component` verwendet, um deutlich zu machen, dass die Beziehung relativ locker ist, und dass ein `Component` prinzipiell mehrfach "dekoriert" werden kann. Wenn man allerdings den Aspekt der Objekt-Vererbung im Auge hat, dann wäre eine Kompositionsbeziehung vorzuziehen.
2. Gamma et al.[17] empfehlen zur Implementierung von `operation ()` bei den konkreten Dekoratoren, mit `super.operation ()` schließlich auf die konkrete Komponente zuzugreifen. Das geht aber nicht, wenn die konkreten Dekoratoren in einer Vererbungshiearchie stehen. In dem Fall muss man `aComponent` direkt verwenden. Daher sei dieses Vorgehen empfohlen. (Das gilt natürlich auch für die "addedState"-Seite.)
3. Die Fälle `addedState` oder `addedBehavior` lassen sich oft nicht unterscheiden. Diese Unterscheidung ist auch unwichtig.
4. Das Decorator-Pattern ist flexibler als statische Vererbung.
5. Erweiterungen können leicht zwei- oder mehrfach gemacht werden.
6. Man kann zunächst relativ einfache Klassen schreiben und später Funktionalität nach Bedarf hinzufügen.
7. Aber: Der Decorator und seine Komponente sind nicht identische Objekte.
8. Man hat eventuell viele Objekte vom gleichen Typ, die sich alle unterschiedlich verhalten.
9. Eine Dekoration kann zusätzliches Verhalten mitbringen.
10. Wenn sich das Verhalten der dekorierten Komponente in **einfacher** Weise (d.h. als Summe oder Produkt oder einfacher sequentieller Ausführung) aus dem Verhalten der Dekoratoren ergibt, dann ist eine Kompositions-Struktur einfacher zu handhaben.

Abstraktes Code-Beispiel

```
public class   Client
   {
   public static void   main (String[] aString)
      {
      Component   c = new ConcreteComponent ();
      System.out.println ("1.) Nur ConcreteComponent:");
      c.operation ();
      System.out.println ();
      Component   ca = new ConcreteDecoratorA (c, "Einstein");
```

```
      System.out.println ("2.) ConcreteComponent und ConcreteDecoratorA:");
      ca.operation ();
      System.out.println ();
      Component   cb = new ConcreteDecoratorB (c);
      System.out.println ("3.) ConcreteComponent und ConcreteDecoratorB:");
      cb.operation ();
      System.out.println ();
      Component   cab = new ConcreteDecoratorB (ca);
      System.out.println ("4.) ConcreteComponent,"
         + " ConcreteDecoratorA und ConcreteDecoratorB:");
      cab.operation ();
      System.out.println ();
      Component   cba = new ConcreteDecoratorA (cb, "Pauli");
      System.out.println ("5.) ConcreteComponent,"
         + " ConcreteDecoratorB und ConcreteDecoratorA:");
      cba.operation ();
      System.out.println ();
      Component   cbaaa = new ConcreteDecoratorA (cb, "Podolski");
      cbaaa = new ConcreteDecoratorA (cbaaa, "Rosen");
      cbaaa = new ConcreteDecoratorA (cbaaa, "Einstein");
      System.out.println ("6.) ConcreteComponent,"
         + " ConcreteDecoratorB, und drei mal ConcreteDecoratorA:");
      cbaaa.operation ();
      System.out.println ();
      }
   }
```

```
public interface  Component
   {
   void  operation ();
   }
```

```
public class   ConcreteComponent
   implements  Component
   {
   public void operation ()
      {
      System.out.println ("   Hier schreiben wir einen");
      System.out.println ("wissenschaftlichen Text");
      System.out.println ("über physikalische Paradoxa.");
      }
   }
```

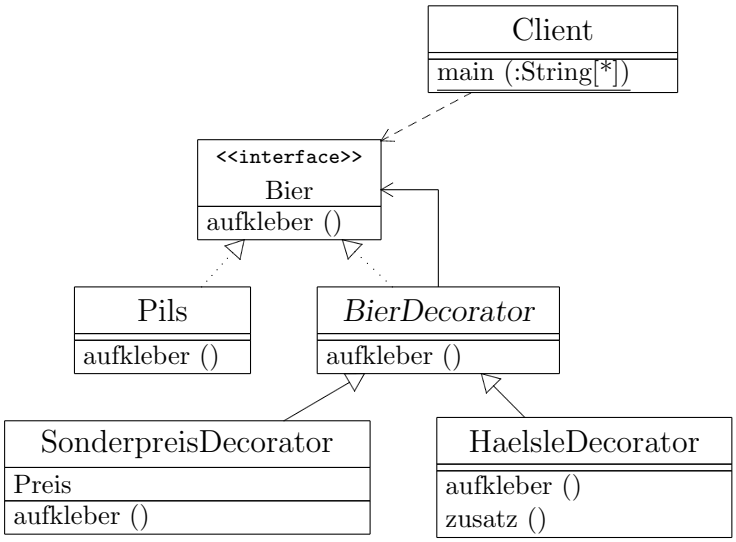
```
public abstract class   Decorator
   implements  Component
   {
   private Component aComponent;
   public final Component  getComponent ()
      {
      return aComponent;
      }
   private final void   setComponent (Component aComponent)
      {
      this.aComponent = aComponent;
      }
   public   Decorator (Component aComponent)
      {
      setComponent (aComponent);
      }
```

```
   public void operation ()
      {
      getComponent ().operation ();
      }
   }
```

```
public class   ConcreteDecoratorA
   extends     Decorator
   {
   private String addedState;
   public final String  getAddedState ()
      {
      return addedState;
      }
   private final void   setAddedState (String addedState)
      {
      this.addedState = addedState;
      }
   public   ConcreteDecoratorA
      (
      Component   aComponent,
      String   addedState
      )
      {
      super (aComponent);
      setAddedState (addedState);
      }
   public void operation ()
      {
      System.out.println ("Einer der Vortragenden heute ist "
         + addedState + "!");
      System.out.println ();
      super.operation ();
      }
   }
```

```
public class   ConcreteDecoratorB
   extends     Decorator
   {
   public   ConcreteDecoratorB (Component aComponent)
      {
      super (aComponent);
      }
   public void operation ()
      {
      super.operation ();
      System.out.println ();
      addedBehavior ();
      }
   public void addedBehavior ()
      {
      System.out.println ("   Hier haben wir noch ein paar");
      System.out.println ("Erläuterungen zum wissenschftlichen Text.");
      }
   }
```

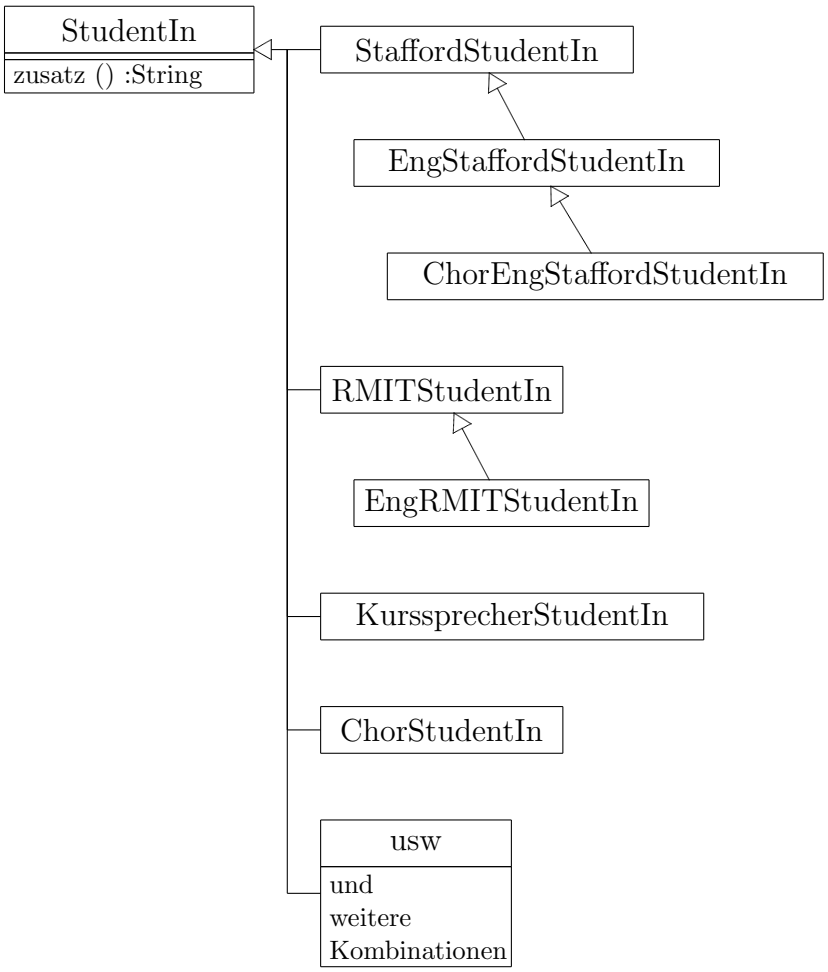
Beispiel Bier



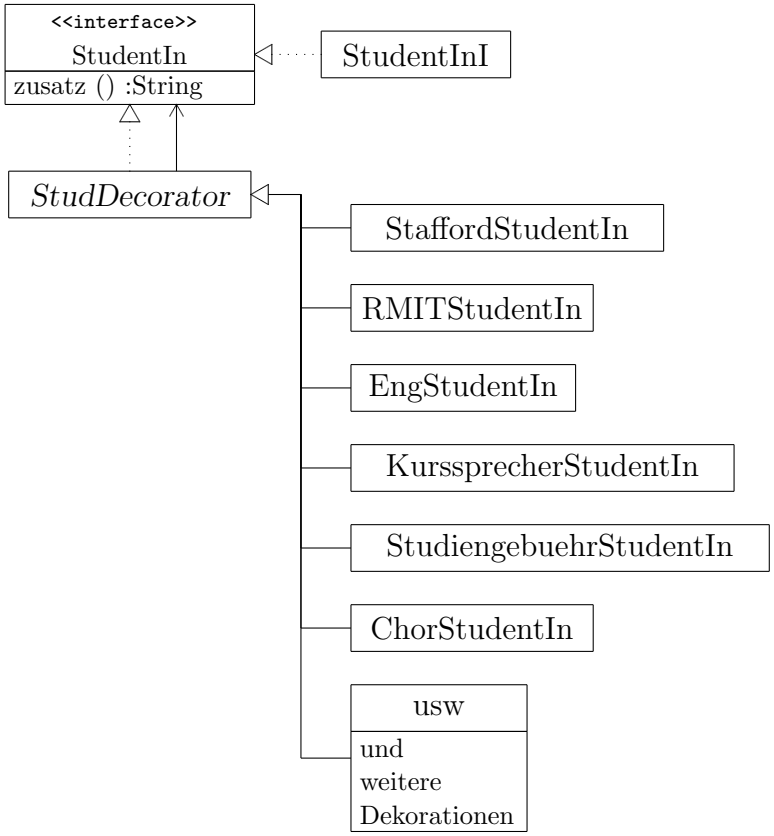
Beispiel Zeugnis-Supplement

Zum Bachelorzeugnis gehört ein Supplement-Dokument, das unter anderem über zusätzliche Leistungen oder Eigenschaften einer StudentIn Auskunft gibt. Also, ob gewisse Module im Ausland erbracht wurden und wie diese angerechnet werden oder ob jemand Kurssprecher war oder an sonstigen Veranstaltungen der DHBW teilgenommen hat. Das Programm, das dieses Dokument erstellt, möchte für jeden Studenten nur eine Methode `zusatz ()` aufrufen.

Wenn man das durch Vererbung realsiert, wird es schnell zu einer Klassenexplosion kommen.



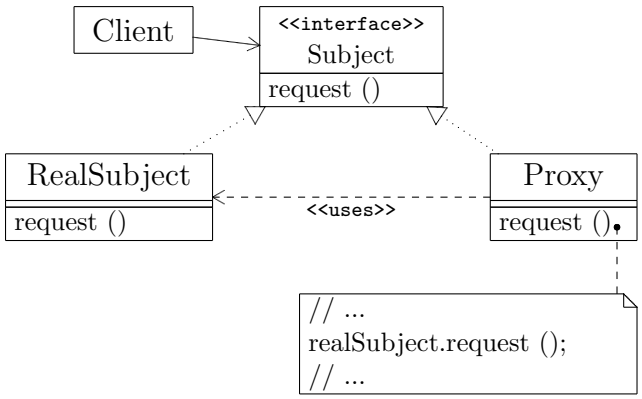
Mit dem Decorator-Pattern werden diese einzelnen Zusätze zu Dekorationen der StudentIn.


<<interface>>
StudentIn
zusatz () :String
StudentInI
StudDecorator
StaffordStudentIn
RMITStudentIn
EngStudentIn
KurssprecherStudentIn
StudiengebuehrStudentIn
ChorStudentIn
usw
und
weitere
Dekorationen

B.10 Proxy

Das Entwurfsmuster ***Proxy*** oder auch ***Surrogate, Ambassador, Placeholder*** wird als Platzhalter für ein anderes Objekt verwendet und kontrolliert den Zugriff auf dieses Objekt.

Struktur



Anwendbarkeit

Das Entwurfsmuster Proxy erlaubt einen intelligenteren, flexibleren Zugriff auf ein Objekt (hier des Typs `Subject`).

- ***remote proxy:*** Das Objekt befindet sich in einem anderen Adressraum.
- ***virtual proxy:*** Ein teures Objekt wird nur teilweise (nach Bedarf, *on demand*) geladen (z.B. Bilder).
- ***protection proxy:*** Ein Objekt wird mit anderen, typischerweise eingeschränkten Zugriffsrechten verwendet.
- ***smart reference:***
  - ***reference counting***
  - ***loading from DB:*** Objekt wird von einer Datenbank geladen.
  - ***locking:*** Das Objekt wird bei Benutzung gesperrt.
  - ***copy-on-write:*** Wenn der Zustand des Objekts verändert wird, dann wird eine Kopie angelegt.

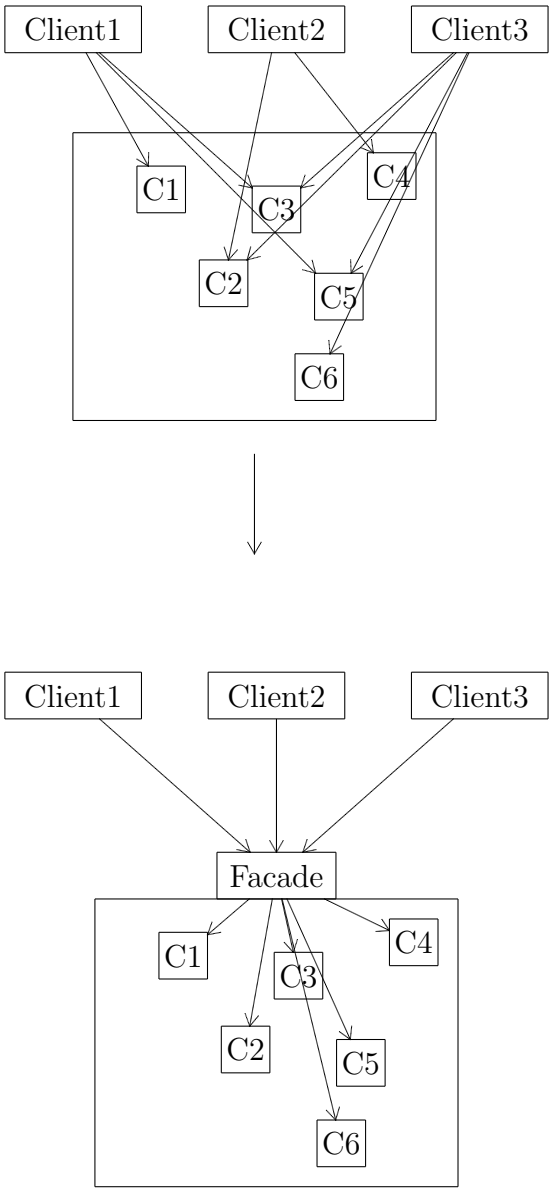
Bemerkungen

1. Das Proxy kann sehr kompliziert sein; kann eventuell aus mehreren Klassen in verschiedenen Adressräumen bestehen. Insbesondere wird es die Verbindung zu einer Datenbank oder zu einem anderen Adressraum verwalten.

B.11 Facade

Das Entwurfsmuster ***Facade*** bietet eine einheitliche Schnittstelle zu vielen Schnittstellen in einem Teilsystem. Es definiert eine ***high-level*** Schnittstelle, damit ein Teilsystem leichter benutzt werden kann.

Struktur



Anwendbarkeit

- Unabhängigkeit von Teilsystemen
- Komplexe Teilsysteme
- Schichtung von Teilsystemen

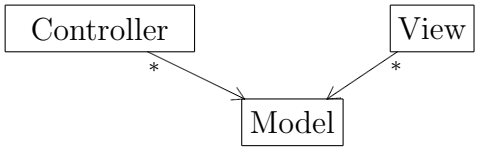
B.12 Model-View-Controller

Das ***Model-View-Controller***-Muster trennt

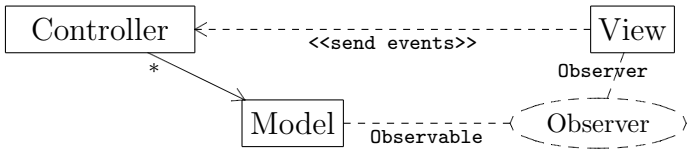
> **Modell, Repräsentation** *(model)*,
> **Präsentation, Darstellung** *(view)* und
> **Steuerung, Manipulation** *(controller)*

eines Objekts, so dass man sie unabhängig verändern oder ersetzen kann.

Struktur



Zwischen Model und View wird häufig ein Observer-Pattern eingesetzt. Oft sind Controller und View in einer GUI integriert, oder der Controller lebt von Ereignissen die ein View sendet. Das ergibt daher folgende Struktur:



Anwendbarkeit

- Eine Klasse ist zu komplex, weil sie alles tut (Eingabe, Ausgabe, Verarbeitung).
- Man möchte eine zusätzliche Darstellung der Objekte einer Klasse verwenden. Die Integration der zusätzlichen Darstellung führt zu unübersichtlichem Code.

Bemerkungen

1. Normalerweise wird dieses Pattern als Architekturmuster behandelt (siehe Kapitel "Architektur"), wobei die Komponenten des Musters Teilsysteme sind. Hier betrachten wir es als Entwurfsmuster (auf der Ebene einzelner Klassen) und rechnen es zu den Strukturmustern.
2. Das Pattern wurde als Architekturmuster zuerst von Trygve Reenskaug 1979 vorgestellt.
3. Das Modell ist typischerweise eine "Bean"-Klasse mit Persistenz.
4. Der Controller kann u.U auch direkt auf die Views einwirken, insbesondere, wenn das Modell nicht geändert wird.

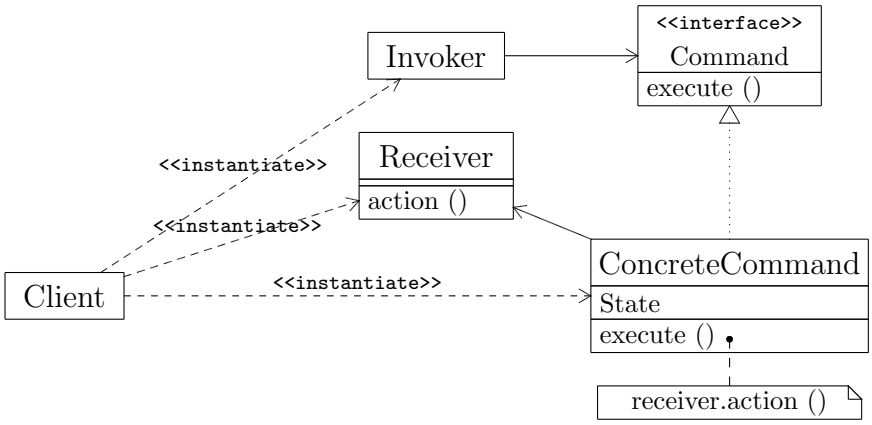
5. Ein MVC-Muster repräsentiert oft einen Anwendungsfall. Die Ablauflogik des Anwendungsfalls kann im Controller oder im Modell untergebracht werden.

6. Ein Modell kann u.U. in mehreren MVCs Modell sein.

B.13 Command

Das Entwurfsmuster ***Command*** oder auch ***Action***, ***Transaction*** kapselt eine Anforderung ***(request)*** als ein Objekt. Dadurch werden Anforderungen austauschbar. Operationen können rückgängig gemacht werden.

Struktur



Anwendbarkeit

- Objekte sollen bezüglich einer auszuführenden Aktion parametrisierbar sein. In prozeduralen Sprachen wird das mit sogenannten ***Callback-***Funktionen gemacht. Das Command-Pattern ist ein objekt-orientierter Ersatz dafür.

- Ein Command-Objekt kann eine Lebenszeit haben, die unabhängig von der ursprünglichen Anforderung ist. Damit lassen sich Anforderungen in Warteschlangen stellen. Sie können zu unterschiedlichen Zeiten befriedigt werden.

- "Rückgängig machen" soll unterstützt werden. Ein Command-Objekt kann Zustandsinformation speichern, die verwendet werden kann, um Aktionen rückgängig zu machen. Dazu muss Command eine Operation `unexecute ()` haben, die das schließlich bewirkt. Mit der Verwaltung von ***history lists*** kann man beliebig weit zurückgehen.

- Logging von Anforderungen soll unterstützt werden, damit man im Fall eines Systemabsturzes die Anforderungen wiederholen kann. Dazu muß die Command-Schnittstelle mit Speicher- und Lade-Operationen erweitert werden *(persistent commands)*.

- Ein System soll *high-level* Operationen verwenden, die auf *low-level* Operationen aufbauen. **Transaktionen** bestehen oft aus vielen Datenänderungen, die durch ein Command gekapselt werden können.

Bemerkungen

1. Das Command-Pattern entkoppelt das Objekt, das eine Operation aufruft, von dem Objekt, das die Operation ausführt.
2. Command-Klassen können sehr große Klassen sein. Insbesondere können sie erweitert werden. Commands können beliebig intelligent bzw. komplex sein.
3. Für Command-Klassen kann oft das Composite-Pattern verwendet werden.
4. Man kann leicht neue Commands hinzufügen.
5. Invokers können nach Priorität behandelt werden.

Abstraktes Code-Beispiel

```
public class   Client
   {
   public static void   main (String[] arg)
      {
      Receiver r = new Receiver ();
      Command  c = new ConcreteCommand (r);
      Invoker  i = new Invoker (c);
      i.invoke ();
      }
   }
```

```
public class   Receiver
   {
   public void action ()
      {
      System.out.println ("      "
         + "Receiver führt "
         + "Anforderung aus.");
      }
   }
```

```
public class   Invoker
   {
   private Command   command;
   public   Invoker (Command command)
      {this.command = command;}
   public void invoke ()
      {
      System.out.println ("Invoker bittet Command,"
         + " die Anforderung auszuführen:");
      command.execute ();
      }
   }
```

```
public interface  Command
   {
   void  execute ();
```

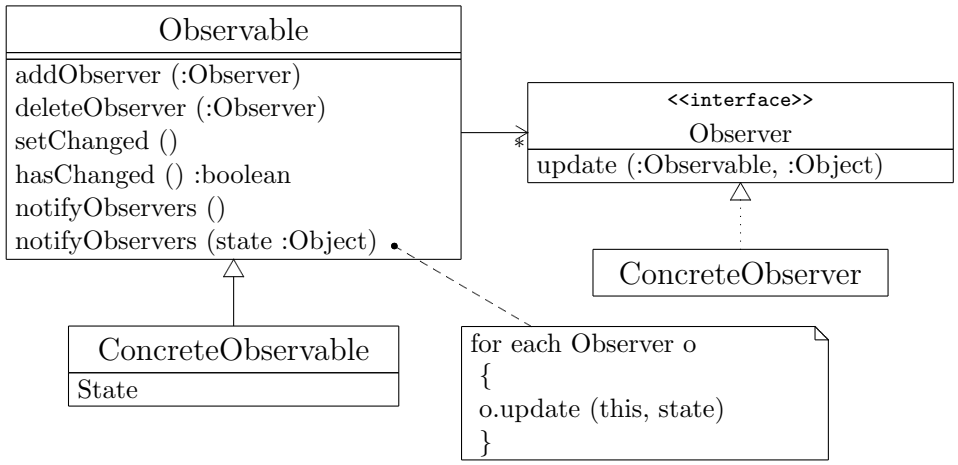
```
   }
```

```
public class   ConcreteCommand
   implements  Command
   {
   private Receiver  receiver;
   public   ConcreteCommand (Receiver  receiver)
      { this.receiver = receiver;}
   public void execute ()
      {
      System.out.println ("   ConcreteCommand leitet "
         + "Anforderung an Receiver weiter:");
      receiver.action ();
      }
   }
```

B.14 Observer

Das Entwurfsmuster ***Observer*** oder auch ***Publish-Subscribe*** und ***Dependents*** definiert eine Abhängigkeit mehrerer Objekte *(observer)* vom Zustand eines Objekts *(observable)*. Die abhängigen Objekte werden bei einer Zustandsänderung des beobachteten Objekts automatisch benachrichtigt.

Struktur



Anwendbarkeit

- Eine Abstraktion hat mehrere Aspekte, wobei ein Aspekt vom anderen abhängt. Verwaltung dieser Aspekte in unterschiedlichen Klassen erlaubt unabhängige Änderungen und Wiederverwendbarkeit.
- Die Zustandsänderung eines Objekts hat die Änderung anderer Objekte zur Folge.
- Die Kopplung zwischen abhängigen Objekten soll sehr lose sein.

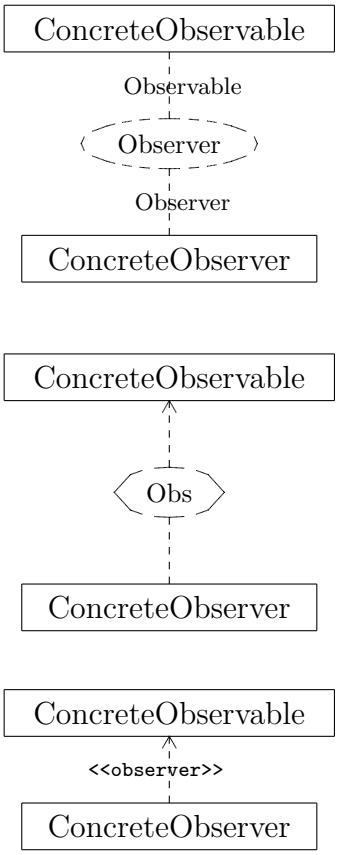
Bemerkungen

1. Das beobachtete Objekt hat eine Liste von Beobachtern und kennt deren Methode `update (...)`. Ansonsten weiß es nichts über die Beobachter. Die Kopplung ist daher minimal.
2. Ein Observer kann natürlich eine Nachricht ignorieren.
3. Unbedeutende Zustandsänderungen können zu Update-Kaskaden führen.
4. Zyklen können auftreten, wenn Objekte sich gegenseitig – eventuell über mehrere Stufen hinweg – beobachten.

5. `notifyObservers (...)` kann
   - entweder vom Observable bei jeder Zustandsänderung automatisch aufgerufen werden
   - oder von einem Client des Observable, wenn dieser es für angebracht hält.

   Der Automatismus im ersten Fall ist zwar bequem und eventuell sicher, führt aber möglicherweise zu überflüssig vielen Updates.
6. ***Push- and Pull-Model:*** Beim Push-Model schickt das Observable mit einem `state`-Objekt detailierte Informationen über die Zustandsänderung an den Observer, ob der das will oder nicht. Beim Pull-Model wird das `state`-Objekt nicht verwendet. Stattdessen holt der Observer bei Bedarf beim Observable nähere Informationen ein.
7. Komplexe Update-Semantiken erfordern einen sogenannten **Change-Manager**, der z.B. die Observer erst dann benachrichtigt, nachdem mehrere Observable eine Zustandsänderung erfahren haben.
8. Eine der häufigsten Anwendungen des Observer-Patterns ist innerhalb des Model-View-Controller-Patterns die Realisierung der Beziehung zwischen Modell (als Observable) und verschiedenen Views (als Observer).

Vergekürzte grafische Notationen:



Abstraktes Code-Beispiel

```
public class   Client
   {
   public static void   main (String[] aString)
      {
      System.out.println ("Zustand wird initialisiert mit \"Grüß Gott!\":");
      ConcreteObservable   beobachtet = new ConcreteObservable ("Grüß Gott");
      ConcreteObserver  beobachter = new ConcreteObserver ();
      System.out.println ("Zustand wird geändert zu \"Guten Tag!\":");
      beobachtet.setState ("Guten Tag!");
      System.out.println ("Ein Beobachter wird addiert.");
      beobachtet.addObserver (beobachter);
      System.out.println ("Zustand wird geändert zu \"Hello World!\":");
      beobachtet.setState ("Hello World!");
      }
   }  // end Client
```

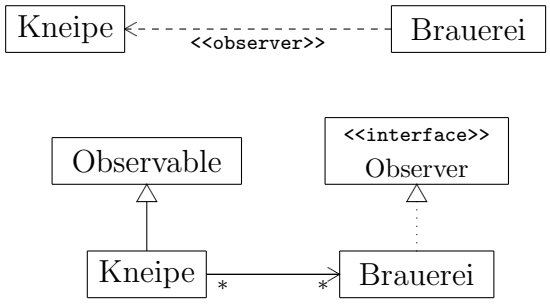
```
public class   ConcreteObservable extends java.util.Observable
   {
   private String state;
   public final String  getState () {return state;}
   public void setState (String aString)
      {
      state = aString;
      System.out.println ("   Beobachtetes Objekt "
         + this + " hat seinen Zustand geändert.");
      setChanged ();
      notifyObservers (getState ());
      }
   public   ConcreteObservable (String state) {setState (state);}
   }  // end ConcreteObservable
```

```
public class   ConcreteObserver implements java.util.Observer
   {
   public void update
      (
      java.util.Observable aObservable,
      Object   aObject
      )
      {
      System.out.println ("Beobachter " + this + " hat bemerkt,");
      System.out.println ("   dass Beobachteter "
         + aObservable + " seinen Zustand zu \"" + aObject
         + "\" geändert hat.");
      }
   }  // end ConcreteObserver
```

Beispiel Bier

Brauereien beobachten Kneipen, ob dort ein Fass leerläuft oder Bierkästen drohen leergetrunken zu werden, um als Update mit entsprechenden Lieferungen automatisch zu reagieren.

Das Beispiel ist insofern nicht typisch, als wir hier viele Observable (Kneipen) und einen oder wenige Observer (Brauereien) haben.

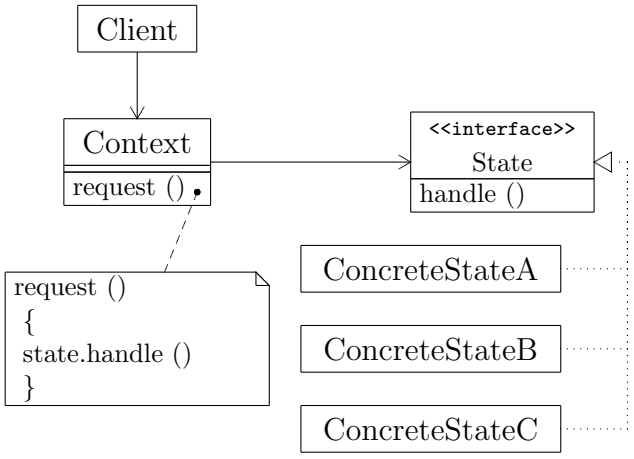

Kneipe
<<observer>>
Brauerei
Observable
<<interface>>
Observer
Kneipe
*
*
Brauerei

B.15 State

Das Entwurfsmuster ***State*** ermöglicht einem Objekt sein Verhalten zu ändern, wenn sich sein Zustand ändert. Das Objekt scheint dabei seine Klasse zu ändern.

In C++, C# und Java kann die Klasse eines Objekts nicht geändert werden. Wenn wir z.B. eine Angestellten-Hierarchie haben, dann wäre es nicht möglich, einen Angestellten zu befördern. Aber durch die Implementation der Angestellten-Hierarchie als eine State-Pattern-Hierarchie wird eine – wenn auch scheinbare – Klassenänderung ermöglicht.

Struktur



Anwendbarkeit

- Das Verhalten eines Objekts hängt von seinem Zustand ab und es muss sein Verhalten zur Laufzeit ändern.
- Eine oder mehrere Methoden haben große, bedingte Anweisungen, die vom Zustand des Objekts abhängen.

Bemerkungen

1. Zustandsspezifisches Verhalten wird lokalisiert.
2. Zustandsübergänge werden explizit gemacht.
3. Wenn innere Klassen transparent auf die Datenrepräsentation der äußeren Klasse zugreifen können, dann können die Zustandsklassen eines Objekts elegant als innere Klassen implementiert werden.

Abstraktes Code-Beispiel

```
public class   Client
   {
   public static void   main (String[] arg)
      {
      Context  c = new Context (new ConcreteStateA ());
      c.request ();
      c.setState (new ConcreteStateB ());
      c.request ();
      c.setState (new ConcreteStateC ());
      c.request ();
      }
   }
```

```
public class   Context
   {
   private State  state;
   public final State   getState () {return state;}
   public final void setState (State state) {this.state = state;}
   public   Context (State state) { setState (state); }
   public void request ()
      {
      getState ().handle ();
      }
   }
```

```
public interface  State
   {
   void  handle ();
   }
```

```
public class   ConcreteStateA
   implements  State
   {
   public void handle ()
      {
      System.out.println (
         "Bin im Zustand A.");
      }
   }
```

Übung Kofferband

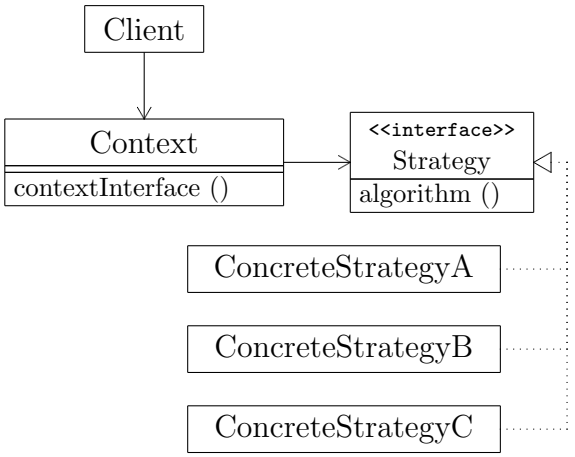
- Erstellen Sie einen Zustandsgraphen.
- Erstellen Sie eine Lösung mit Switch-Case-Kontrukt.
- Erstellen Sie eine Lösung mit dem State-Pattern.
- Diskutieren und/oder realisieren Sie das Template-Method Pattern für die Zustandsmanager-Methode.

- Diskutieren Sie den Unterschied zwischen State-Pattern und Strategy-Pattern.

B.16 Strategy

Das Entwurfsmuster ***Strategy*** oder auch ***Policy*** ermöglicht die für den Klienten transparente Austauschbarkeit von Algorithmen.

Struktur



Anwendbarkeit

- Es gibt viele Klassen, die sich nur in wenigen Methoden unterscheiden. Diese Klassen können durch eine Klasse ersetzt werden, die dann mit verschiedenen Strategien konfiguriert werden kann.
- Wir haben viele Varianten eines Algorithmus, die ausprobiert oder unter bestimmten Bedingungen zum Einsatz kommen sollen.
- Ein Algorithmus benutzt (komplexe) Datenstrukturen, die gekapselt werden sollen.
- Eine Klasse definiert viele Verhaltensweisen, die als konditionale Anweisungen erscheinen. Konditionale Verzweigungen können in eigene Strategie-Klassen ausgelagert werden.

Bemerkungen

1. Das Muster hat große Ähnlichkeit mit ***State***. Bei Strategy geht es um Austauschbarkeit einer Methode durch eine andere Methode, die ähnliches oder sogar dasselbe leistet, aber eventuell mehr oder weniger effizient. D.h. das allgemeine Verhalten eines Objekts wird nicht geändert. Bei State wird das Verhalten eines Objekts insgesamt verändert, so dass es unter Umständen völlig anders reagiert. Änderungen des Zustands eines Objekts gehören zum Lebenszyklus eines Objekts, während Strategien i.a. einmal konfiguriert werden.

   Mit Strategy kann man für dassselbe Verhalten unterschiedliche Implementationen anbieten.

2. Strategy ist eine Alternative zu Vererbung. Mit Strategy unterscheiden wir klarer zwischen Kontext und Algorithmus, die dann unabhängig voneinander variiert werden können.

Abstraktes Code-Beispiel

```
public class   Client
   {
   public static void   main (String[] argument)
      {
      Context  a = new Context (new ConcreteStrategyA ());
      a.contextInterface ();
      Context  b = new Context (new ConcreteStrategyB ());
      b.contextInterface ();
      Context  c = new Context (new ConcreteStrategyC ());
      c.contextInterface ();
      }
   }
```

```
public class   Context
   {
   private Strategy  strategy;
   public   Context (Strategy strategy)
      {
      this.strategy = strategy;
      }
   public void contextInterface ()
      {
      strategy.algorithm ();
      }
   }
```

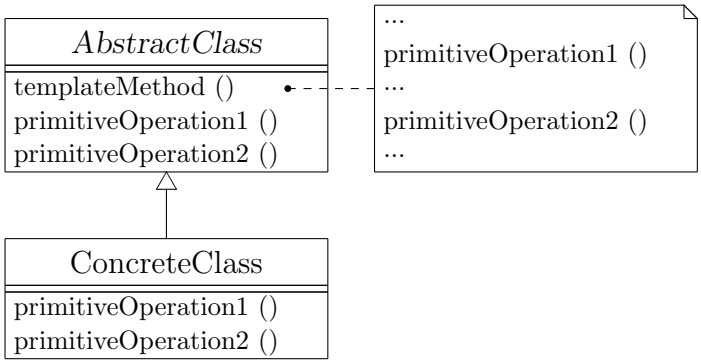
```
public interface  Strategy
   {
   void  algorithm ();
   }
```

```
public class   ConcreteStrategyA
   implements  Strategy
   {
   public void algorithm ()
      {
      System.out.println (
         "Context führt Algorithmus A aus.");
      }
   }
```

B.17 Template Method

Mit dem Entwurfsmuster ***Template Method*** definiert man in einer Operation einen Algorithmus, wobei einige Schritte durch Subklassen überschrieben werden können. Die Struktur des Algorithmus bleibt erhalten.

Struktur



Anwendbarkeit

- Invariante Teile eines Algorithmus sollen nur einmal implementiert werden. Teile, die variieren, werden durch Unterklassen implementiert.
- Gemeinsames Verhalten in Unterklassen soll lokalisiert und ausgelagert werden *("refactoring to generalize").*
- Überschriebene Methoden haben große Ähnlichkeiten, die zu Code-Wiederholungen führen.
- Ein Algorithmus kann sogenannte ***Hook Operations*** verwenden, die zwar eine Implementierung haben, aber von Unterklassen überschrieben werden können.

Bemerkungen

1. Template-Methoden sind eine fundamentale Technik, um Code wiederzuverwenden. Gemeinsamkeiten werden "herausfaktorisiert".

Abstraktes Code-Beispiel

```
public abstract class   AbstractClass
   {
   public void templateMethod ()
      {
      System.out.println (
         "Invariantes Template Codestück A");
```

```
      primitiveOperation1 ();
      System.out.println (
         "Invariantes Template Codestück B");
      primitiveOperation2 ();
      System.out.println (
         "Invariantes Template Codestück C");
      }
   public abstract void primitiveOperation1 ();
   public abstract void primitiveOperation2 ();
   }
```

```
public class   ConcreteClass
   extends     AbstractClass
   {
   public void primitiveOperation1 ()
      {
      System.out.println (
         "Variante primitive Operation 1");
      }
   public void primitiveOperation2 ()
      {
      System.out.println (
         "Variante primitive Operation 2");
      }
   public static void   main (String[] arg)
      {
      new ConcreteClass ().templateMethod ();
      }
   }
```

Beispiel Bier

Nimm aus Bridge-Pattern das Bier-Beispiel und wende Template Method Pattern auf die Methode `machAufkleber ()` an.

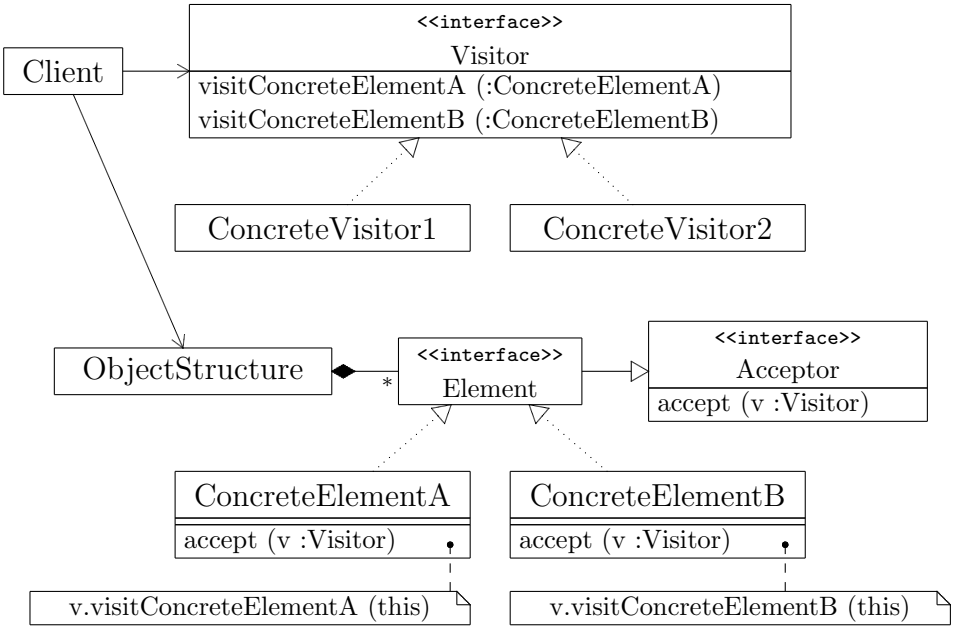
B.17.1 Übungen

1. Vergleichen Sie das Entwurfsmuster *Template Method* mit dem Entwurfsmuster *Bridge*.

B.18 Visitor

Das Entwurfsmuster ***Visitor*** repräsentiert eine Operation, die auf Objekte von unterschiedlichen Klassen angewendet werden kann. Voraussetzung ist, dass diese Klassen eine Schnittstelle `Acceptor` realisieren. Typischerweise werden die Objekte als Elemente einer Objekt-Struktur verwendet. Das Entwurfsmuster erlaubt es, eine neue Operation zu definieren, ohne die Klassen der Objektstruktur zu ändern.

Struktur



Anwendbarkeit

- Eine Objekt-Struktur enthält viele Klassen mit unterschiedlichen Schnittstellen und wir wollen Operationen auf diesen Objekten ausführen, die von den konkreten Klassen der Objekte abhängen.

- Viele verschiedene Operationen sollen auf Objekten einer Objektstruktur durchgeführt werden. Wir wollen aber die Klassen dieser Objekte nicht mit diesen Operationen "versauen". Mit Visitor können wir zusammengehörige Operationen in einer neuen Klasse zusammenfassen.

- Die Klassen einer Objektstruktur ändern sich selten. Aber oft will man neue Operationen für die ganze Struktur definieren.

Bemerkungen

1. Man kann die `accept`-Methode in einer schon vorhandenen Schnittstelle für die Objektstruktur (hier `Element`) deklarieren.
2. Für Elemente von `ObjectStructure` kann `accept (v)` direkt aufgerufen werden. Oder der Aufruf von `accept (v)` kann in einer Operation `anOperation ()` erfolgen.
3. Die Methoden der Schnittstelle `Visitor` können alle den gleichen Namen haben, also etwa `visit (...)`. Sie unterscheiden sich nur durch den Typ des Arguments.
4. Für jede Art von Operationen (z.B. alle graphischen Darstellungs-Operationen) muss eine neue Klasse geschrieben werden, die `Visitor` realisiert. In dieser Klasse ist jede Klasse der Objektstruktur durch eine Methode repräsentiert.
5. Mit Visitor können leicht neue Operationen hinzugefügt werden, indem ein neuer Visitor definiert wird. Eine neue Funktionalität wird auf eine Klasse lokalisiert, obwohl sie eine ganze Klassenstruktur betrifft.
6. Zusammengehöriges Verhalten findet sich in *einer* Klasse.
7. Die Erweiterung der Objekt-Struktur ist umständlich, da jede Visitor-Klasse berührt werden muss.
8. Ein Visitor kann Zustandsinformation verwalten, die sonst als zusätzliches Argument übergeben werden müsste.
9. Die Schnittstelle der Objekt-Struktur muss groß genug sein, damit der Visitor seine Arbeit tun kann.
10. Vergleich mit Bridge-Pattern: Beim Bridge-Pattern bleibt die Semantik der abstrakten Operation ("linke Seite") unabhängig von den Implementatoren konstant. Beim Visitor-Pattern ist die Semantik der abstrakten Operation (`accept (...)`) völlig offen und hängt ab vom Visitor.
11. Kann man ganz auf die Schnittstelle `Acceptor` verzichten? D.h. könnte man anstatt `o.accept (v)` einfach nur `v.visit (o)` schreiben? Ja, wenn man nicht auf polymorphes Verhalten angewiesen ist. Denn die `visit`-Methoden sind nur **überladen**, so dass in einer Vererbungsstruktur nicht die richtige `visit`-Methode über eine Superklassen- oder Schnittstellen-Referenz angezogen wird. Die `accept`-Methoden dagegen werden in einer Vererbungsstruktur **überschrieben** und verhalten sich polymorph.

Abstraktes Code-Beispiel

```
public class   Client
   {
   public static void   main (String[] argument)
      {
      ObjectStructure   os = new ObjectStructure ();
      Visitor  v = new ConcreteVisitor1 ();
      System.out.println ("Client: Wendet ConcreteVisitor1"
         + " auf alle Elemente einer Objektstruktur an.");
      os.traverse (v);
      v = new ConcreteVisitor2 ();
```

```
      System.out.println ("Client: Wendet ConcreteVisitor2"
         + " auf alle Elemente derselben Objektstruktur an.");
      os.traverse (v);
      }
   }
```

```
public class   ObjectStructure
   {
   private Element[] element
      = new Element[]
         {
         new ConcreteElementA (),
         new ConcreteElementB ()
         };
   public void traverse (Visitor v)
      {
      for (int i = 0; i < element.length; i++)
         {
         element[i].accept (v);
         }
      }
   }
```

```
public interface  Visitor
   {
   void  visitConcreteElementA (
      ConcreteElementA aConcreteElementA);
   void  visitConcreteElementB (
      ConcreteElementB aConcreteElementB);
   }
```

```
public class   ConcreteVisitor1
   implements  Visitor
   {
   public void visitConcreteElementA (ConcreteElementA cea)
      {
      System.out.println ("   Visitor1 besucht Element A:");
      System.out.println ("      " + cea.satz);
      cea.anOperation ();
      }
   public void visitConcreteElementB (ConcreteElementB ceb)
      {
      System.out.println ("   Visitor1 besucht Element B:");
      System.out.println ("      " + ceb.satz);
      ceb.anOperation ();
      }
   }
```

```
public class   ConcreteVisitor2
   implements  Visitor
   {
   public void visitConcreteElementA (ConcreteElementA cea)
      {
      System.out.println ("   Visitor2 besucht Element A:");
      System.out.println ("      " + cea.satz.toUpperCase ());
      }
```

```
   public void visitConcreteElementB (ConcreteElementB ceb)
      {
      System.out.println ("   Visitor2 besucht Element B:");
      System.out.println ("      " + ceb.satz.toLowerCase ());
      }
   }
```

```
public interface  Acceptor
   {
   void  accept (Visitor v);
   }
```

```
public interface  Element
   extends  Acceptor
   {
   void  anOperation ();
   }
```

```
public class   ConcreteElementA
   implements  Element
   {
   public String  satz = "Ich bin A.";
   public void accept (Visitor v)
      {
      v.visitConcreteElementA (this);
      }
   public void anOperation ()
      {
      System.out.println ("      anOperation von A");
      }
   }
```

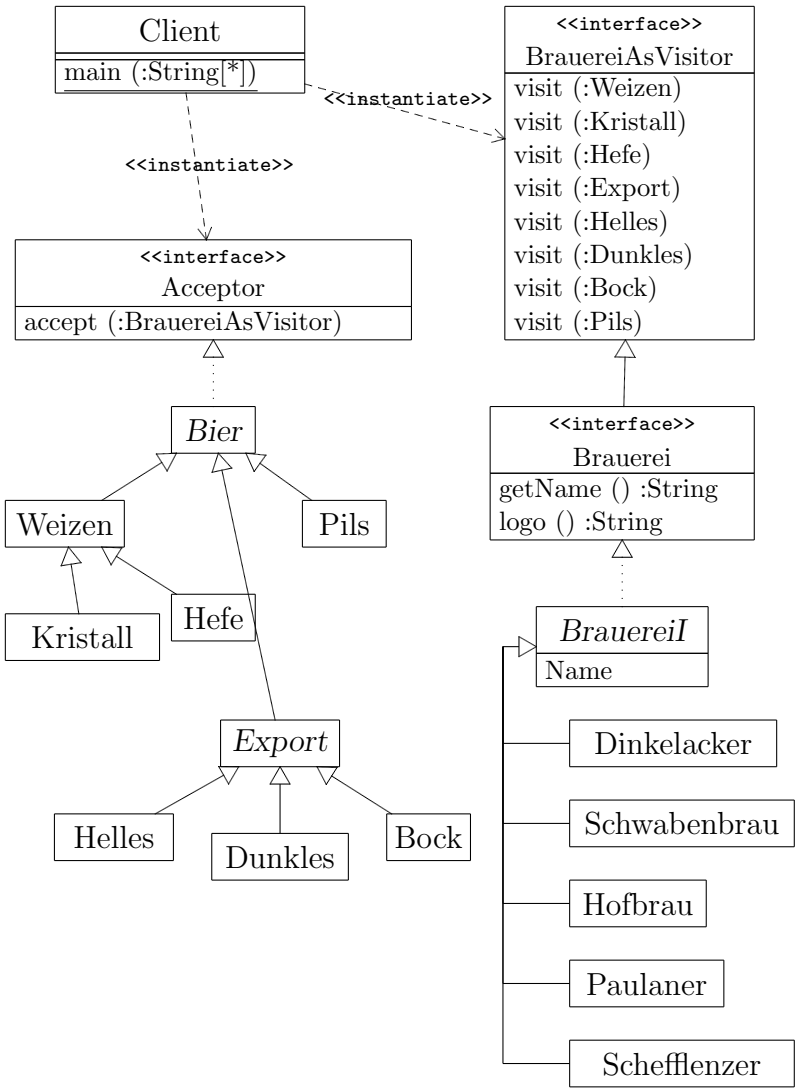
```
public class   ConcreteElementB
   implements  Element
   {
   public String  satz = "Ich wäre B.";
   public void accept (Visitor v)
      {
      v.visitConcreteElementB (this);
      }
   public void anOperation ()
      {
      System.out.println ("      anOperation von B");
      }
   }
```

Beispiel Bier

Für verschiedene Biersorten (Pils, Weizen, Export usw.) wollen wir für verschiedene Biermarken Flaschenaufkleber drucken. Bei 7 Biersorten und 5 Biermarken müssten wir 35 Klassen schreiben. Dieses Beispiel hatten wir schon mit dem Bridge-Pattern behandelt. Hier wird eine Lösung mit

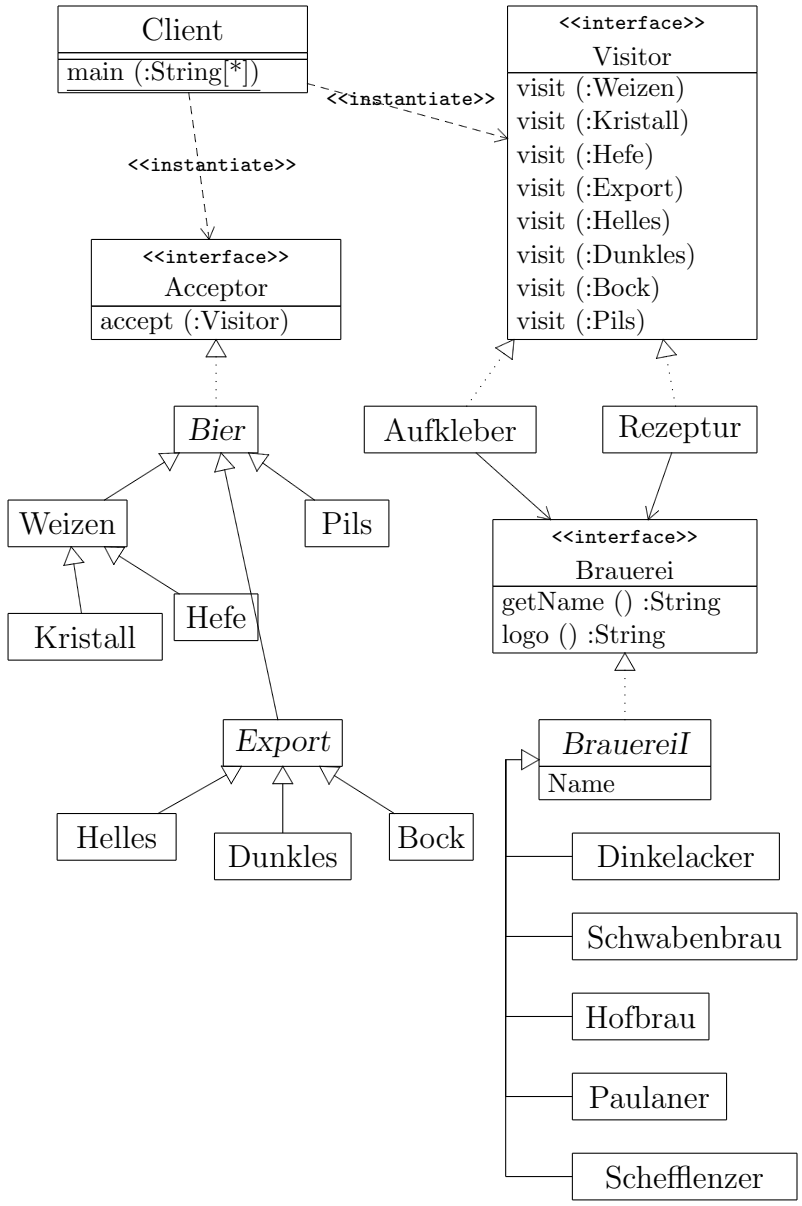
dem Visitor-Pattern vorgestellt, bei dem die Anzahl der Klassen ebenfalls wesentlich reduziert wird.

Wenn wir einen Visitor konstruieren, der Aufkleber für die Flaschen macht, dann müsste jede Brauerei die Methode `visit (...)` realisieren. Der Vorteil gegenüber dem Bridge-Pattern ist, dass die Aufkleber für jede Brauerei ganz individuell gestaltet werden können. Außerdem können beliebig viele andere Visitors erstellt werden, z.B. einer, der die Kronenkorken macht, oder die Rezeptur erstellt.



Die folgende Struktur zeigt einen Aufkleber-Visitor und einen Rezeptur-Visitor. Wenn der Aufkleber nur Name und Logo der Brauerei zur Verfügung hat, dann werden die Aufkleber nicht sehr

individuell sein. Das Ergebnis ist ähnlich wie beim Bridge-Pattern, außer dass wir die Klassen unserer Objektstruktur nicht mit Aufkleber-Code "versaut" haben.



Literaturverzeichnis

[1] Len Bass, Paul Clements und Rick Kazman, "Software Architecture in Practice", Addison-Wesley 2003

[2] Cédric Beust und Hani Suleiman, "Next Generation Java Testing – TestNG and Advanced Concepts", Addison-Wesley

[3] Grady Booch, "Object-Oriented Analysis and Design", The Benjamin/Cummings Publishing Company 1994

[4] Frederick P. Brooks Jr., "The Mythical Man-Month", Addison-Wesley (1995)

[5] Nathaniel S. Borenstein, "Programming as if People Mattered – Friendly Programs, Software Engineering, and Other Noble Delusions", Princeton University Press

[6] Rainer Burkhardt, "UML – Unified Modeling Language", Addison-Wesley 1997

[7] Peter Pin-Shan Chen, "The Entity-Relationship Model: Toward a Unified View of Data", ACM Transactions on Database Systems **1**, 9-36 (1976)

[8] Peter Coad und Edward Yourdon, "Object-Oriented Analysis", Prentice-Hall 1991

[9] Peter Coad und Edward Yourdon, "Object-Oriented Design", Prentice-Hall 1991

[10] Alan M. Davis, "201 Principles of Software Development", McGraw-Hill 1995

[11] T. DeMarco, "Structured Analysis and System Specification", Yourdon Press 1979

[12] Bruce Powel Douglass, "Real-Time Design Patterns" Addison-Wesley

[13] Bruce Powel Douglass, "Real Time UML" Addison-Wesley

[14] Anton Eliëns, "Object-Oriented Software Development", Addison-Wesley 1994

[15] Jay Fields, Shane Harvie und Martin Fowler mit Kent Beck, "Refactoring – Ruby Edition", Addison-Wesley

[16] Martin Fowler, "Refactoring – Improving the Design of Existing Code", Addison-Wesley 2000

[17] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, "Design Patterns", Addison-Wesley 1994

[18] Stephan Gilbert und Bill McCarty, "Object-Oriented Design in Java", Mitchell Waite 1998

[19] Robert Glass, Communications of the ACM **41**, 10 (1998)

[20] Brian Goetz, "Java Concurrency in Practice", Addison-Wesley Longman

[21] Hassan Gomaa, "Designing Software Product Lines with UML", Addison-Wesley

[22] Hassan Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison-Wesley

[23] David Harel, "Statecharts: a visual formalism for complex systems", Science of Computer Programming **8** (1987), 231 – 274

[24] Brian Henderson-Sellers, "A Book of Object-Oriented Knowledge", Prentice-Hall 1991

[25] D. W. Hoffmann, "Software-Qualität", Springer Vieweg 2013

[26] John E. Hopcroft, Rajeev Motwani und J. D. Ullman, "Introduction to automata theory, languages, and computation", Addison-Wesley 2001

[27] Ivar Jacobson, Magnus Christerson, Patrik Jonsson und Gunnar ̇Övergaard, "Object-Oriented Software Engineering A Use Case Driven Approach", Addison-Wesley 1995

[28] Kent Beck, "Smalltalk Best Practice Patterns", Prentice Hall 1997

[29] Kent Beck, "Make it Run, Make it Right: Design Through Refactoring", The Smalltalk Report, **6**, 19-24 (1997)

[30] Kent Beck, "eXtreme Programming eXplained: Embrace Change", Addison-Wesley 2000

[31] Ph. Kruchten, "The Rational Unified Process – An Introduction", Addison-Wesley 2000

[32] Doug Lea, "Concurrent Programming in Java: Design Principles and Patterns", Addison-Wesley

[33] Craig Larman, "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", Pearson Education (2005)

[34] Peter Liggesmeyer und Dieter Rombach (Hrsg.), "Software Engineering eingebetteter Systeme", Elsevier Spektrum Akademischer Verlag

[35] Floyd Marinescu, "EJB Design Patterns", Wiley 2002

[36] Bertrand Meyer, "Object-oriented Software Construction", Prentice Hall 1988

[37] Bernd Oestereich, "Objekt-orientierte Softwareentwicklung", Oldenbourg 1998

[38] Shari Lawrence Pfleeger und Joanne M. Atlee, "Software Engineering", Pearson 2010

[39] Roger S. Pressman, "Software Engineering: A Practitioner's Approach", McGraw-Hill

[40] Dave Radin, "Building a Successful Software Business", O'Reilly & Associates 1994

[41] Shangping Ren und Gul A. Agha, "A Modular Approach for Programming Embedded Systems", Lecture Notes in Computer Science, 1996

[42] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy und William Lorensen, "Object-Oriented Modeling and Design", Prentice Hall (Deutsch bei Hanser)

[43] Chris Rupp und Stefan Queins, "UML 2 glasklar", Hanser

[44] Ben Shneiderman und Catherine Plaisant, "Designing the User Interface", Addison Wesley

[45] Software Program Manager's Network, `http://www.spmn.com`

[46] Gerald M. Weinberg, "The Secrets of Consulting", Dorset Kouse Publishing (1985)

[47] Ann L. Winblad, Samuel D. Edwards und David R. King, "Object-Oriented Software", Addison-Wesley 1990

[48] R. Wirfs-Brock, B. Wilkerson und L. Wiener, "Designing Object-Oriented Software", Prentice-Hall 1990