**Zusammenfassung**

# Software Engineering 1

Michael Wieland

Hochschule für Technik Rapperswil

26. August 2017

**Mitmachen**
Falls du an diesem Dokument mitarbeiten möchtest, kannst du es auf GitHub unter `https://github.com/michiwieland/hsr-zusammenfassungen` forken.

# Inhaltsverzeichnis

# 1. OOA: Objektorientierte Analyse

OOA = Statisches Modell + Dynamisches Modell + Contracts

## 1.1. Domainanalyse

Unter Domainanalyse versteht man das Verstehen eines noch unbekannten Problembereichs. Es geht also um eine methodische Analyse des Problems. Des Weiteren betrachtet man das Problem aus einer Black-, und White-Box Sicht. Aus der Domain Analyse resultieren zwei Modelle:

- Statisches Modell (Domainmodell)
- Dynamisches Modell (Interaktionsdiagramme)

## 1.2. Probleme im Software Engineering

Viele Software Projekte laufen nicht zu letzt wegen den folgenden Punkten schief:

- Komplexität wird nicht beherrscht
- Termine werden überschritten
- Kosten werden überschritten
- Das Ziel wird verfehlt
- Es mangelt an Qualität (evtl. auch auf Grund von fehlender Zeit)

## 1.3. Statisches Modell

Das statische Modell ist das **Domainmodell** mit all seinen Entitäten (Konzepten). Es zerlegt die Problem Domain in verständliche Teile und verzichtet dabei auf Implementierungsdetails wie Fremdschlüssel und Methoden.

### 1.3.1. Terminologie

#### Domain Modell

- Das Domain Modell zeigt wesentliche konzeptionelle Klassen und ihre Zusammenhänge, also **auch Dinge, die nicht persistent werden**.
- Im Domain Modell spricht man immer von einem **Konzept** anstatt von einer Klasse. Der Klassenname wird jedoch oft vom Konzept abgeleitet.
- Im Domain Modell sind Assoziationen **bidirektionale** Beziehungen zwischen Konzepten der realen Welt
- Das Domain Modell kann eine Inspiration für das Design Modell sein.

#### Design Modell, Klassenmodell

- Das Design Modell zeigt persistente Entitäten und ihre Zusammenhänge für eine Realisation.
- Das Design Model verfügt **zusätzlich über Methoden**
- Im Design Modell sind Assoziationen **unidirektionale** Navigationspfade von einem Software Objekt zu einem anderen.

#### Daten Modell

- Das Daten Modell zeigt persistente Entitäten in der Datenbank (ERD)

### 1.3.2. Bestandteile

#### Nach Larman

- Klassenkonzepte mit Attributen und Beziehungen untereinander, aber ohne Operationen

#### Klassisch

- Klassenkonzepte mit Attributen, Operationen und Beziehungen untereinander

### 1.3.3. Konzepte

Konzepte werden im Design Modell zu Klassen. Abstrakte Klassen werden in *Italics* geschrieben

**Konzepte ohne Attribute** Wenn z.B bei Vererbungen bestimmte Childklassen keine Attribute besitzen, muss die Existenzberechtigung der Childklasse überdacht werden.

**Vererbung in relationalen DB**
Es gibt zwei Möglichkeiten, eine Vererbungshierarchie in einer relationalen DB abzubilden. Wenn man es mit einem gut entworfenen Vererbungshierarchie zu tun hat, sollte der Table per Hierarchy Ansatz verwendet werden. (die leeren Felder halten sich in Grenzen). Da die Vererbung so oder so relativ viel Schwierigkeiten bereitet, fährt man mit Decoration etc. so oder so besser.

**Table per Class** Hier wird pro modellierte Klasse eine Tabelle abgelegt. Beim Abruf der Daten für eine abgeleitete Klasse muss ein Join gemacht werden. Dafür gibt es keine Platzverschwendung

**Table per Hierarchy** Hier macht man für die ganze Vererbungs-Hierarchie nur eine Tabelle. Dabei werden nicht benutze Felder leer gelassen.

**Aggregation und Komposition**
Aggregation und Komposition sind Teil-Ganzes Beziehungen. Die Diamanten haben in der Regel eine 1-Multiplizität (implizit).

**Aggregation** Unabhängies Überleben der Teile (weisser Diamant)

**Komposition** Beim Löschen des Ganzen, existieren die Teile auch nicht mehr (schwarzer Diamant)

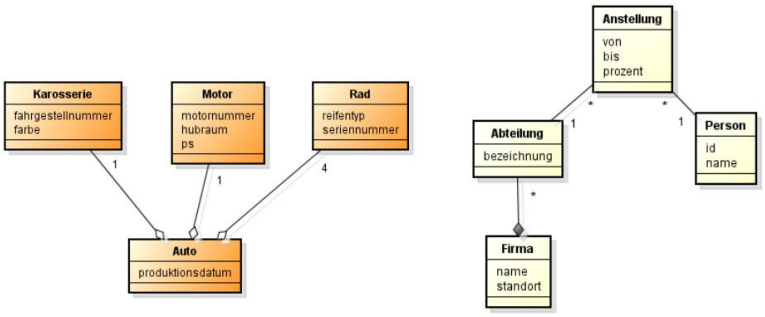


Abbildung 1: Aggreagation und Komposition

### 1.3.4. Attribute

Attribute sind logische Datenelemente eines Objekts.

**Konzept oder Attribut?** Ist ein Attribut ein komplexes Objekt, erstellt man für das Attribut besser ein neues Konzept, da man damit flexibler wird. Zum Beispiel würde man das Attribut "`destination`" des Konzepts "`Flight`", besser als eigenes Konzept "`Airport`" mit einem Attribut "`name`" auslagern.

**Auslagern von redundaten Informationen (Beschreibungen)** Attribute, in welche bei jedem Anlegen der gleiche Wert eingefügt wird, sollten ausgelagert werden. z.B "`description`" bei dem Konzept "`Kurs`". In diesem Fall führt man ein neues Konzept "`Kursbeschreibung`" mit dem Attribut "`description`" ein. Das neue Konzept "`Kursbeschreibung`" gilt dann für mehrere "`Kursdurchfuehrungen`".

### 1.3.5. Typen

Der Typ des Attributs ist im Domain Modell **optional**. Falls der Typ angegeben wird, muss einer der folgenden verwendet werden.

- Boolean
- Date
- Number
- String / Text
- Time

### 1.3.6. Assoziationen

Eine Assoziation ist eine Beziehung zwischen Konzepten, welche bedeutungsvolle und interessierende Verbindungen (zwischen ihren Instanzen) darstellen. Assoziationen werden immer in die Richtung des Pfeiles gelesen.

#### 1:1 Beziehungen

- 1:1 Beziehungen sind im Domainmodell Ok, im Datenmodell sollten sie jedoch aufgelöst werden
- 1:1 Beziehungen machen nur Sinn, wenn die beiden Teile verschiedenen Domänen entstammen oder zu verschiedenen Zeiten erstellt werden.
- 1:0..1 Beziehungen sind in beiden Modellen Ok.

**n:m Beziehungen** Diese Art von Assoziationen sind zulässig, sehr oft aber nicht richtig, da die Assoziation eigenen Attribute enthält, und deshalb mit einem Zwischenkonzept aufgelöst werden muss. m:n Beziehungen können auf zwei Arten aufgelöst werden:

1. Mit Zwischenklasse: Dies ist der Standardfall
2. Mit Assoziationsklasse: Hier darf jedes Beziehungspaar nur genau einaml vorkommen. (Kombination der zwei Foreign Keys muss unique sein)

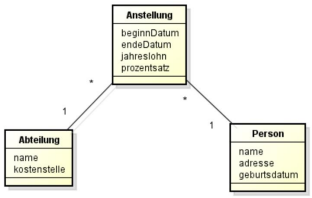


Abbildung 2: Zwischenklasse für n:m Beziehungen

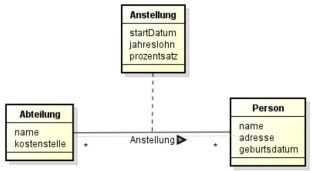


Abbildung 3: Assoziativklasse für m:n Beziehungen

**Richtung von Beziehungen in Datenmodell**
Die folgenden Regeln helfen, dass Domainmodelle einfacher zu lesen sind. Sie sind jedoch nur eine Empfehlung.

1. Eins-zu-n Beziehungen: eins unten, n darüber
2. Eins-zu-eins Beziehungen: auf gleicher Höhe
3. n-zu-m Beziehungen: auf gleicher Höhe
4. Ableitungen: Basisklasse unten, abgeleitete darüber

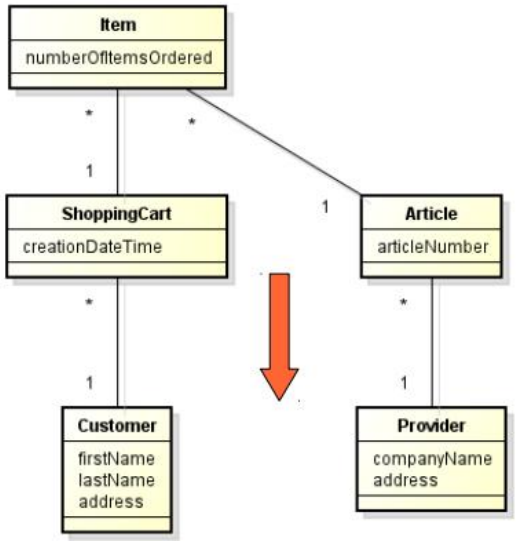


Abbildung 4: Richtungen von UML Beziehungen

**Rollen**
UML bietet die Möglichkeit, eine Beziehung an drei Orten anzuschreiben. Man sollte eine Assoziation immer anschreiben, wenn es mehr als eine Beziehung zwischen zwei Klassen gibt und der Sinn nicht sofort klar ist.

- In der Mitte kann man die Art der Beziehung zusammen mit einem Pfeil für die Leserichtung anschreiben.
- An jedem Ende der Assoziation kann (je nahe bei einer Klasse) die Rolle, welche die Klasse in der Beziehung spielt, spezifiziert werden.
- Ist die Beziehung ein simples (hat ein) dann muss es nicht angeschrieben werden.

### 1.3.7. Multiplizität

Die Multiplizität definiert wie viele Instanzen des Types A mit einer Instanz des Types B verbunden sein können.

| Multiplizität | Beschreibung |
|---|---|
| 1 | Genau ein |
| 1..* | Ein oder mehrere |
| * | Keine oder mehrere |
| n | Genau n |
| 1..n | Ein bis n |
| x,y,z | x Mal oder y Mal oder z Mal |

Tabelle 1: Multiplizitäten

### 1.3.8. Vorgehen

1. 1x grob durchlesen und Schlüsselwörter notieren
2. 2-3x genauer durchlesen und Konzepte, Attribute & Assoziationen identifizieren
3. Verfeinern: redundantente und falsche Assoziationen löschen
4. Überprüfen ob alles beachtet wurde
5. (Evtl. Fragenkatalog und Annahmen bei Unklarheiten)

## 1.4. Dynamisches Modell

Das dynamische Modell bietet eine **Überprüfung zwischen dem Domain Modell und den effektiven Use Cases**.

### 1.4.1. Bestandteile

#### Nach Larman

- Black-box Interaktionsdiagramm für Use Case Szenarien
- Operation Contracts
- Evtl. Zustandsdiagramme für das System oder einzelne Klassen
- Aktivitätsdiagramme für einzelne Use Cases

#### Klassisch

- White-box Interaktionsdiagramme für Use Case Szenarien oder Systemoperationen
- Evtl: Zustandsdiagramme für System oder einezlne konzeptionelle Klassen

# 2. Anforderungsanalyse

In der Anforderungsanalyse oder Requirements Analysis werden die **Anforderungen an die Software** festgelegt. Man beschreibt was zu realisieren ist (Use Cases) und wer in dem Projekt involviert ist. Es ist empfehlenswert, iterativ vorzugehen, da sich Anforderungen schnell ändern können. Nach der Anforderungsanalyse resultieren folgende Dinge:

- Vision
- Use Cases und User Stories inkl. Use Case Diagramme
- Anforderungsspezifikation (Software Requirements Specification (SRS))
- Glossar

## 2.1. SRS: Software Requirements Specification

SRS beschreibt verständlich und umfänglich, was die Absicht und die Anforderungen einer Software sind.

| Document | Captures | Goal |
|---|---|---|
| Vision | Big Ideas; Executive summary | Introduction; Basis for management decisions |
| Use Cases | Functional Requirements | Basis for development |
| Supplementary Specification | Requirements not captured by use cases (FURPS+) | |
| Glossary | Terms and Definitions | |
| Business Rules | Rules and policies that transcend the current application | |

Abbildung 5: SRS: Software Requirements Specification

## 2.2. Klassifikation

**Funktionale Anforderungen**
Funktionale Anfoderungen fragen nach dem Was? Welche Funktionen mit welchen Daten? Welche Funktionen, mit welchen Daten

**Nichtfunktionale Anforderungen**
Nichtfunktionale Anforderungen sind Qualitätsanforderungen und fragen nach dem Wie? Es geht um die Leistung, Menge, Qualitätsmerkmale (Benutzbarkeit, Wartbarkeit), Randbedingungen

**Prozessanforderungen** Kostenschätzung

## 2.3. Bestandteile

**User Stories** Informale kurze Geschichten aus Benutzersicht (Agile SE)

**Use Cases** Formale Geschichte aus Benutzersicht auf der Ebene elementarer Geschäftsprozesse.

**Actors** Benutzerrollen in Use Cases

**Personas** Fiktive Personen die den typische Nutzer beschreiben

## 2.4. Funktionale Anforderungen

Funktionale Anforderungen werden mit **Use Cases** beschreiben. Eine Use Case ist eine textuelle Ablaufbeschreibung mit Ziel und Zweck. (Im Stil: "Actor macht A, dann macht das System B") Uses Cases gibt es in drei Formen

**Brief**
: Kurze Beschreibung (2-3 Sätze; gibt einem eine Idee). Zusammenfassung in einem Abschnitt, beschreibt nur erfolgreiches Szenario

**Casual**
: Mittle Beschreibung (ca. 1/4 Seite). Mehrere Abschnitte, informale Beschreibung mehrerer Szenarien

**Fully dressed**
: Ausführliche Beschreibung. (ca. 2 Seiten) Alle Szenarien im Detail beschrieben, mit Raster für ergänzende Abschnitte, wie Vorbedingungen und Nachbedingungen.

Listing 1: Fully Dressed Use Case Beispiel

```
Use Case Name: [..]
Primary Actor: [..]
Stakeholder and Interests:
   - Actor A: wants [..]
   - Actor B: want [..]
Preconditions: Actor A is [..]
Postconditions (Success Guarantee) [..]
Main Sucess Scenario (or Basic Flow):
   1. Actor A starts [..]
   2. Actors B enters id [..]
   3. System presents item [..]
   4. [..]
Extensions (or Alternative Flows):
   2.a invalid id [..]
   3.a item is not [..]
   3.b [..]
Special Requirements:
   - text must be visible on ipad
   - [..]
Technology and Data Variations List:
   2.a bar code reader enters id
   4.a [..]
Frequency of Occurence: [..]
Open Issues:
   - what about the law
   - [..]
```

### 2.4.1. Aktoren

**Actor/Stakeholder**
Sind immer ausserhalb des zu entwickelnde Systems. Stakeholder sind Anspruchsgruppen/-Interessensgruppen.

**Primary Actor**
Der Primary Actor löst den Use Case aus. Services des Systems erfüllen Ziele der primären Aktoren. Sie helfen Ziele der Benutzer zu identifizieren/finden.

**Supporting Actor**
Stellt einen Service für das System zur Verfügung. Sie helfen die Schnittstellen des Systems zu klären.

**Offstage Actors**
Haben Interesse an Use Case, sind aber nicht direkt involviert

### 2.4.2. EBP: Elementary Business Process

Ein Elementary Business Process (EBP) ist eine elementarer Prozess in einem Betrieb. Use Cases sollten alle EBP beschreiben.

## 2.5. Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen werden oft vernachlässigt, sind aber wichtig, das sie einen grossen Einfluss auf die Architektur und Benutzerzufriedenheit haben. Es geht insbesondere um folgende Punkte

- Leistung. **Wie schnell?** Antwortzeiten, Durchsatzraten (Ist von der Hardwareumgebung und Systembenutzung abhängig → Randbedingung)
- Mengen: **Wie viel?** Anzahl Kundendatensätze, Anzahl gleichzeitiger Benutzer
- Qualitätsmerkmale: **Wie gut?** (Benutzbarkeit, Warbarkeit) ISO9126
- Randbedingungen: **Wie?** Programmiersprache, Framework, Datenbankserver

### 2.5.1. FURPS+ / ISO 9126

Nicht Funktionale Anforderungen sind Qualitätsmerkmale nach ISO 9126. Ein weiteres Mass an nichtfunktionalen Charakteristiken ist FURPS+. (Functionality, Usability, Reliability, Performance/Portability, Supportability, +Others (FURPS+))

- **Functionality**: Angemessenheit, Richtigkeit, Sicherheit, Interoperabilität
- **Usability**: Bedienbarkeit, Verständlichkeit, Erlernbarkeit, Farben
- **Reliability**: Zuverlässigkeit, Fehlertoleranz, Wiederherstellbarkeit
- **Performance**: Effizient, Wie schnell wird die Seite angezeigt
- **Supportability**: Wartbarkeit
- **+ Others**: Hardware, Software Constraints, Lizenzen

# 3. Diagramm Modelierung

Da UML Diagramme wenig abstrahieren, haben sie keinen bedeutenden Vorteil gegenüber dem direkten Programmieren. Sie vereinfachen die Kommunikation mit dem Kunden daher kaum. In welchen Fällen lohnt es sich also, ein Diagramm zu zeichnen? Grundsätzlich gilt, wenn man im Code viele Klassen anschauen müsste, ist ein Diagramm die viel übersichtlichere Darstellungsform.

| *Name* | *Nah am Kunden* | *Nützliche Details* | *Einsatz-Möglichkeiten* |
|---|---|---|---|
| Use Case Diagramm | ja | - | Übersicht, Scope/Funktionsumfang; zeigt auch (indirekt) Rollen und Berechtigungen |
| Domainmodell | ja | (ja) | Fast immer sinnvoll, zeigt Zusammenhänge; Basis für Glossar (ist oft > 80% des Inhalts) |
| Datenmodell | (ja) | ja | Fast immer sinnvoll, zeigt Zusammenarbeit; DDL kann generiert werden; meist sehr nahe am Domainmodell |
| Design-Klassenmodell | - | (ja) | Meist nicht sinnvoll, da 1:1 wie Code |
| Sequenz-Diagramm | - | ja | Manchmal sinnvoll, bei kooperierenden Teams und Klassen, z.B. komplexe Startup Sequenz; kann aus Code generiert werden |
| Zustands-Diagramm | ja | ja | Manchmal sinnvoll für zentrale Objekte (z.B. Product Life Cycle in Webshop) oder für Masch.-Steuerungen |
| Aktivitäts-Diagramm | ja | - | Manchmal sinnvolle Ergänzung für komplexe Use Cases |
| Package-Diagramm | - | - | reine Übersicht, zeigt Schichten, immer sinnvoll in Software Architecture Documentation (SAD) |
| Deployment-Diagramm | (ja) | (ja) | zeigt Verteilung auf Tiers, oft in mehreren Variationen; wichtig für Betrieb |
| Komponenten-Diagramm | - | (ja) | Kann – falls das System so entworfen wurde – die Übersicht zeigen und Schnittstellen definieren |

Abbildung 6: Übersicht Diagramme

## 3.1. Use Case Diagramm

Bei Use Case Diagramm muss auf folgende Dinge geachtet werden:

- Der Kontext wird als grosses leeres Reckteck abgebildet (Systemkontext)
- Akteure werden als Strichmännchen dargestellt
- Die Use Cases werden in Ellipsen abgebildet
- Include-Beziehungen: Werden mittels `<<include>>` gekennzeichnet und mit einer gestrichelten Linie. Sie wird verwendet, wenn ein Use Case einen anderen enthält.
- Extend-Beziehungen: Werden mittels `<<extend>>` gekennzeichnet und mit einer gestrichelten Linie. Sie wird verwendet, wenn ein Use Case einen anderen erweitert
- Externe Systeme werden als Klasse vom Typ `<<actor>>` gekennzeichnet
- Änderungen an Kundendatensätzen können mit CRUD abgekürzt werden.

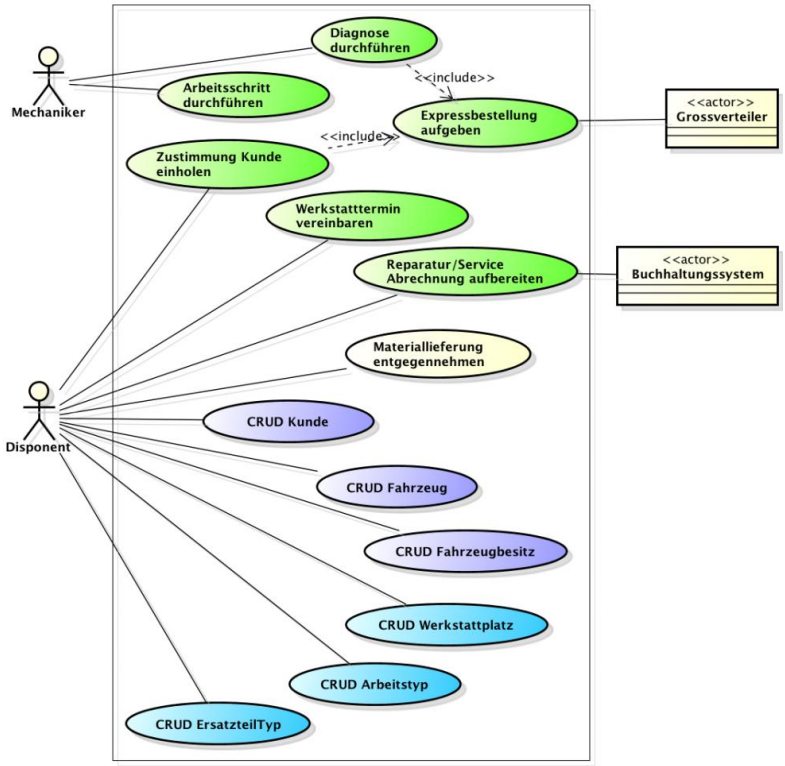


Abbildung 7: Use Case Diagramm

## 3.2. Zustandsdiagramm

Zustandsdiagramme (State Machine Diagrams) zeigen zustandsabhängiges Verhalten von Systemen. Man kann ein Zustandsdiagramm für einen bestimmten Use Case oder für ein ganzes System erstellen (wird bei komplexen System unübersichtlich). Zustandsdiagramme zeigen die zulässige Reihenfolge aller Systemereignisse.

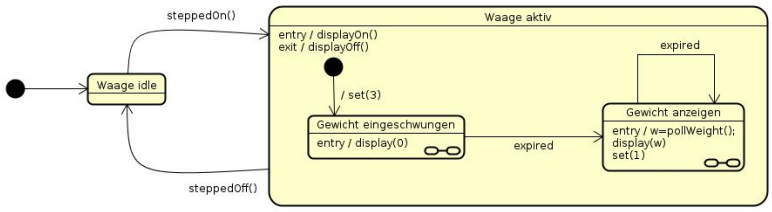


Abbildung 8: Zustandsdiagramm

**Startzustand** Der Startzustand wird mit einem schwarzen vollen Punkt gekennzeichnet

**Endzustand** Der Endzustand ist ebenfalls ein voller Punkt, jedoch mit einem Kreis herum.

**State** Zustand eines Objektes. Ein Zustand hat mehrere Events als Eigenschaft. Hat Start- und End-Zustand

- Entry: Aktion die beim Eintritt ausgeführt wird
- Do: Wird während dem State ausgeführt
- Exit: Wird beim Verlassen des States ausgeführt

**Nested State** Ist eine genauere Darstellung eines States. Enthält wiederum mehrere States. Hat meistens keinen Endzustand

**Event** Passiert zu einem Zeitpunkt und löst Aktivität aus

**Activity** Das Resultat was ein System macht aufgrund eines Events. (optional)

**Guard** Voraussetzung (optional)

**Transition** Beim Zustandübergang können Activities ausgeführt werden. Er wird nach folgendem Schema beschriftet: `event [guard] / activity`. Eine Transition benötigt immer **mindestens einen Guard oder Event!**

### 3.2.1. Vorgehen

1. Relevante Zustände identifizieren.
2. Relevante Ereignisse (Events) identifizieren.
3. Transitionen erfassen.
4. Guards identifizieren und am richtigen Ort erfassen.
5. Aktivitäten identifizieren und am richtigen Ort erfassen.
6. Diagramm evtl. mittels Nesting vereinfachen

## 3.3. Activity Diagramme

Ein Activity Diagramm zeigt sequentielle und parallele Aktivitäten. Es ist eine Erweiterung des Flussdiagramms. Es wird insbesondere für Geschäftsprozesse und Use Cases mit parallelen Aktivitäten verwendet. Ein Activity Diagramm kann auch in verschiedene Partitionen unterteilt werden (z.B welche Abteilung erledigt welche Arbeit)

### 3.3.1. Nodes

**Initial Node** Startpunkt

**Fork Node** Paraleller Ablauf (zwei Token). Ein Fork Node wird wie der Join Node mit einem schwarzen Strich dargestellt.

**Decision Node** Der Token nimmt genau diesen Weg, auf welchen die Bedingung zutrifft

**Merge Node** Fügt eine Decision wieder zusammen

**Join Node** Wartet auf alle Tokens und fügt diese zusammen (Synchronisation)

**Object Node** Das resultierende Objekt einer Action

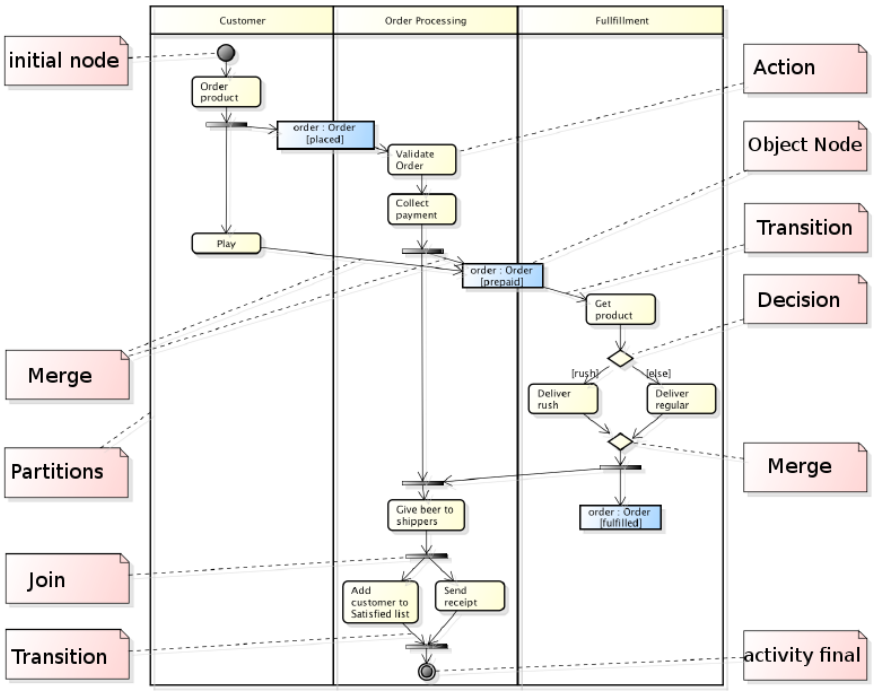


Abbildung 9: Aktivitätsdiagramm

## 3.4. Sequenzdiagramme

- Ein Sequenzdiagram zeigt detailiert das Zusammenspiel von mehreren Klassen, sowie deren Abhängigkeiten. Speziell komplexe Sequenzen können gut erklärt und diskutiert werden.
- Man programmiert ohne eine einzige Zeile Code zu schreiben
- Bestimmte Enwicklungsumgebungen können SD's zur Laufzeit generieren, was für die Dokumentation recht praktisch ist.
- Wie bei allen Diagrammen muss man immer darauf achten, dass die Diagramme aktuell bleiben. Dies ist insbesondere der Fall wenn das Diagramm nahe beim Code ist.
- Die Zeit verläuft von oben nach unten. Die dicken Blöcke entsprechen der Zeitdauer, die eine Klasse arbeitet.

### 3.4.1. Komponenten

- sd: ist das Sequenzdiagramm
- alt: Alternative Ablaufe mit Bedingungen. Ist ein `if-then-else`
- opt: "Optionaler" Ablauf mit einer Bedingung (`if`)
- ref: ist ein Unterprogramm
- loop: Wiederhole bis Abbruchbedingung eintritt. Abbruchbedingung wird direkt beim Loop definiert

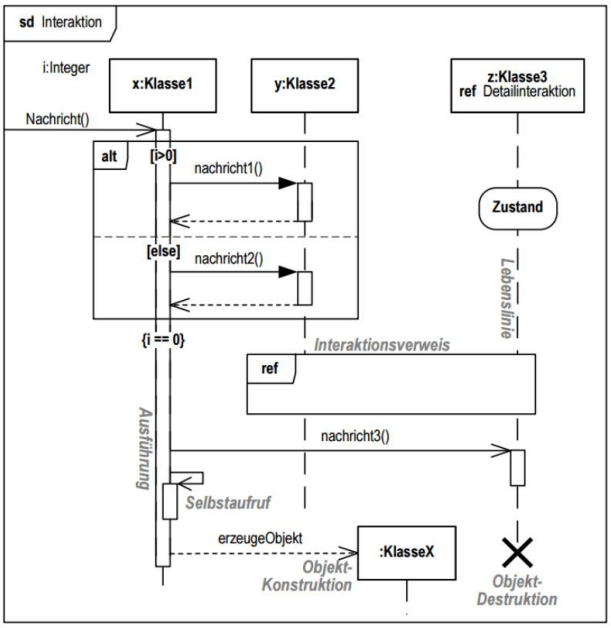


Abbildung 10: Sequence Diagramm

```
HouseBlend coffee = new HouseBlend();
Soy soy = new Soy(coffee);
Mocha mocha = new Mocha(soy);
Whip whip = new Whip(mocha);
// call
whip.cost();
```

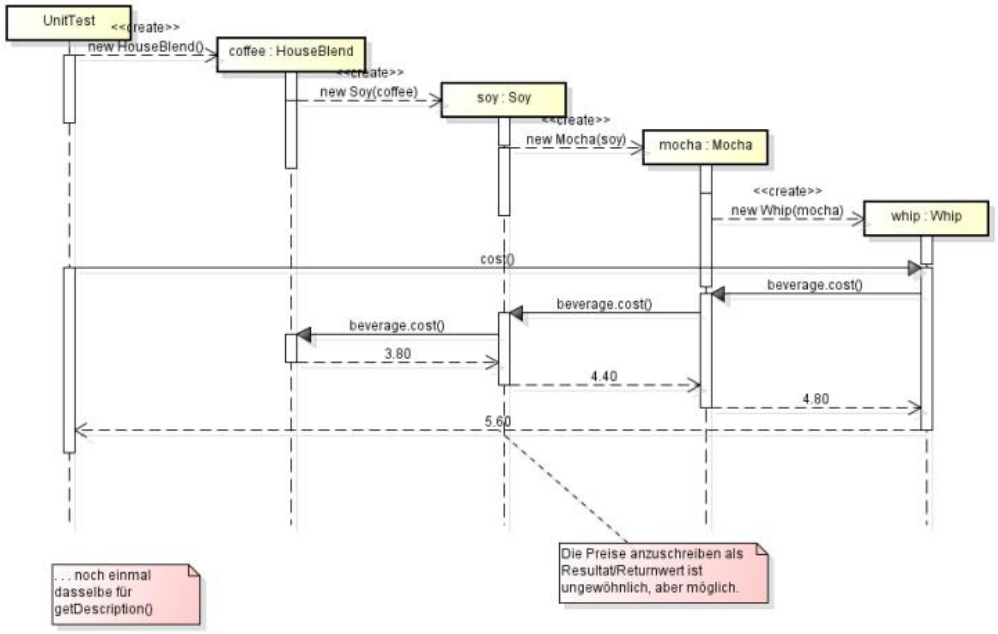


Abbildung 11: Sequenz Diagramm Decorator

## 3.5. SSD: System Sequenzdiagramm

- Es gibt immer nur einen Aktor und ein System. Das System ist immer eine Blackbox.
- Das SSD modeliert oft ein Use Case Szenario → Verweise auf das Main Success Szenario (**UC-Nummer angeben**)
- Dass SSD agiert auf einer höheren Abstraktionsbene, wobei der Wille des Benutzers beschrieben wird.
- Das SSD wird in der Enwicklung sehr früh erstellt, da es eine vereinfachte Sicht darstellt.

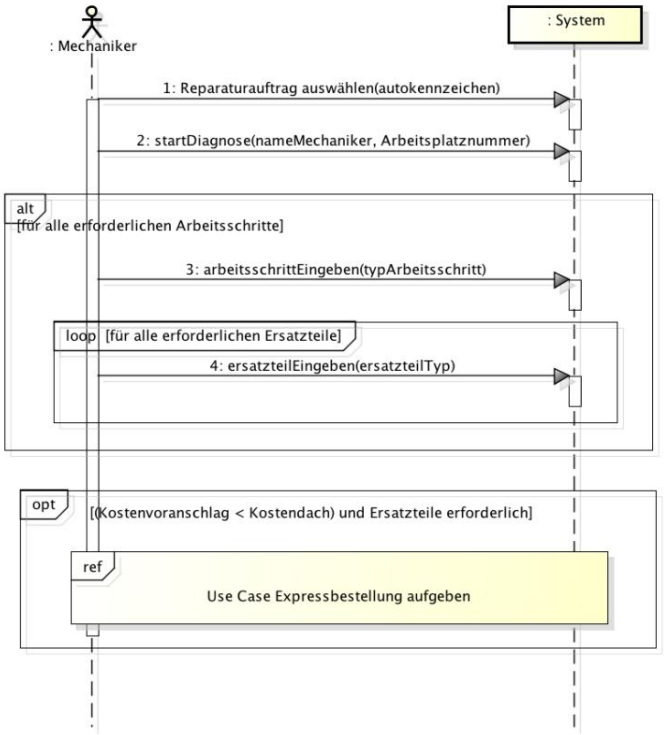


Abbildung 12: System Sequenzdiagramm

### 3.5.1. Operation Contract

Ein Operation Contract beschreibt eine Änderung im System, wenn eine Operation durchgeführt wird. Die Operation **bezieht sich immer auf eine einzige Operation im System Sequence Diagram (SSD)**. Wenn man einen Operation Contract schreibt, sollte man sich den Systemstatus **vor und nach einer Operation** vorstellen (Snapshots). Diese beiden Zustände müssen beschrieben werden. Es muss nicht aufgezeigt werden, wie die Operation durchgeführt wird, sondern wie das Resultat aussieht.

- Name: [Operation Name]
- Responsibilities: [Perform a function / Beschreibung Aufgaben dieser Operation]
- Cross References: [System functions and Use Cases]
- Exceptions: [..]
- Preconditions: [Wie sieht das System **vor** der Änderung aus?]
- Postconditions: [Wie sieht das System **nach** der Änderung aus?]

## 3.6. Sequenzdiagramm vs SSD

| | SSD: System Sequenzdiagramm | Sequenzdiagramm |
|---|---|---|
| Was | System Interaktion | Zusammenspiel von mehreren Klassen |
| Akteure | 2 Dialogpartner | sinnvoll mit mehr als 2 Dialogpartner |
| Ansicht | Black Box | White Box |
| Zeitpunkt | Teil von Requirements Analysis | Teil von OOD bzw. nachträgliche Doku |
| | Näher beim User/Kunden | Nah beim Code/Entwickler |

Tabelle 2: Multiplitäten

## 3.7. Deployment Diagramme

- Deploymentdiagramme zeigen, welche Software Komponenten auf welchen Knoten (Hardware) laufen.
- Oft macht man mehrere Diagramme die verschiedene Szenarien visualisieren (Performance, Security)
- Nested Execution Environments: Ausführungs Umgebungen werden typischerweise ineinander geschachtelt
- Assoziationen: Verbindung zum Protokoll (evtl. Multiplizitäten)
- Die Deployment Diagramme werden vom Server Manager geschrieben und sollten ebenfalls möglichst am Anfang definiert werden.

### 3.7.1. Bestandteile

**Device Node** Zeigt ein Gerät (Server oder Client) an

**Execution Environment Node** Zeigt eine Laufzeitumgebung an. Ist immer in einem Device Node gepackt.

**Artefact** Beschreibt ein Softwareproduct

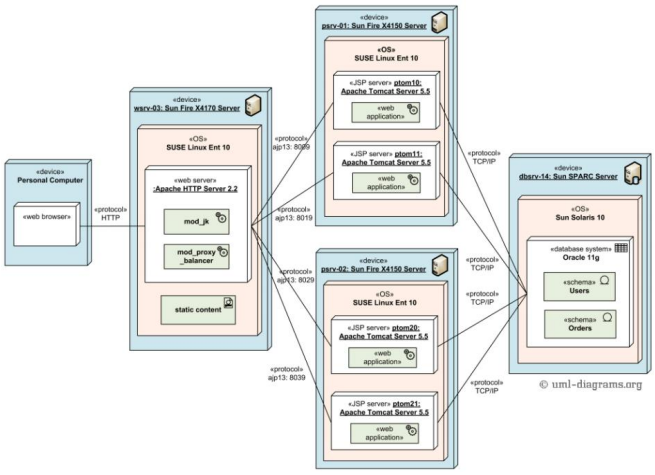


Abbildung 13: Deployment Diagram

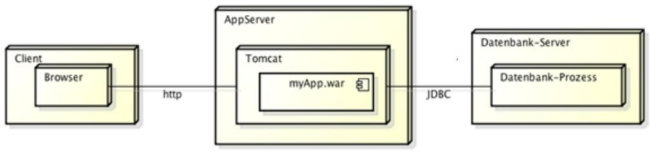


Abbildung 14: Vereinfachtes Deployment Diagram

## 3.8. Packagediagramm

Das Paketdiagramm ist ein Strukturdiagramm. Es zeigt eine bestimmte Sicht auf die Struktur des modellierten Systems

### 3.8.1. Bestandteile

**Package** Stellt eine Software Schicht dar

**Dependency** Zeigt die Abhängigkeiten der Packages.

**Merge** Mit dem Keyword `<<merge>>` wird angezeigt, dass ein Paket aus mehreren anderen Paketen kombiniert wird.

**Import** Mit dem Keyword `<<import>>` wird eine gerichtete Beziehung zwischen zwei Paketen visualiert.

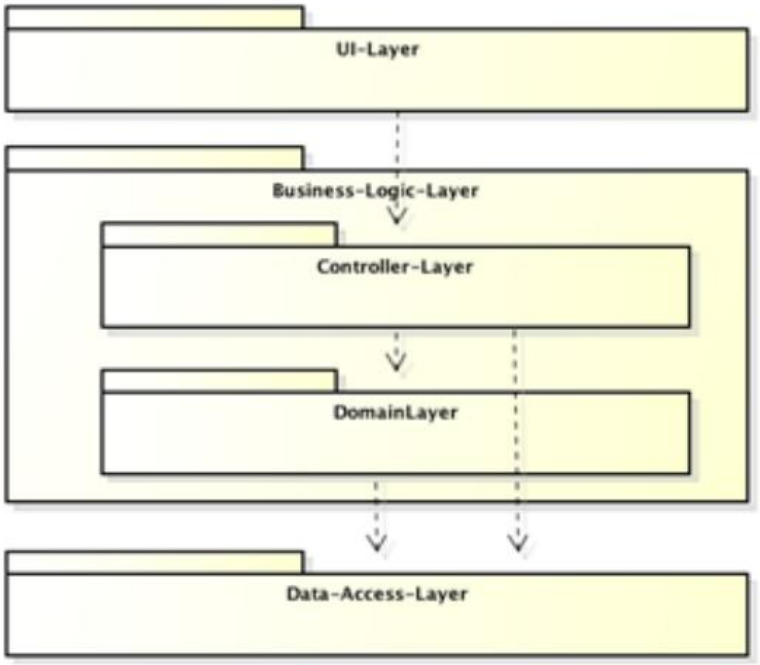


Abbildung 15: Package Diagram

# 4. Konfigurationsmanagement

Software Konfigurations Managment ist die Disziplin zur Verfolgung und Steuerung der Evolution von Software. Zum Konfigurationsmanagement gehören vier Teile die miteinander interagieren:

## 4.1. Versionskontrolle

- Verwaltet die Änderungen am Sourcecode
- Wer hat Wann, Was und Warum so gemacht
- Für die Versionskontrolle kommt oft Git oder SVN, CVS zum Einsatz

## 4.2. Buildmanagement

- Das Buildmanagment beschreibt den Vorgang der Erzeugung von Artefakten aus Software Elementen
- Ein Artefakt ist ein Produkt, das als Zwischen- oder Endergebnis in der Softwareentwicklung entsteht

## 4.3. Releasemanagement

- Das Releasemanagment befasst sich mit dem Veröffentlichen der erzeugten Artefakte
- Ebenfalls geht es hier um die Dokumentation und Definition des Installationsvorganges

## 4.4. Changemanagement

- Das Changemanagement definiert einen geordneten Umgang mit Änderungswüschen und Fehler

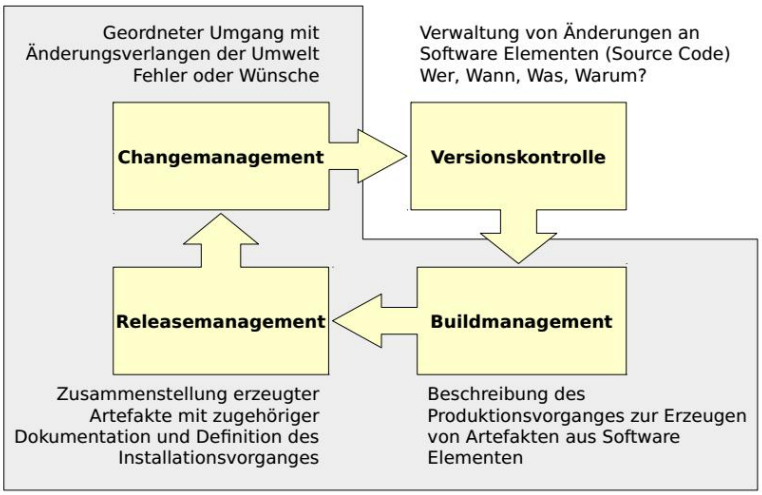


Abbildung 16: Konfigurationsmanagement

# 5. Versionskontrolle

Versionskontrollsysteme (VCS) protokollieren Änderungen an einer oder mehreren Dateien über die Zeit hinweg, so dass man zu jedem Zeitpunkt auf Versionen und Änderungen zugreifen kann. Bekannte Vertreter sind Git, Mercurial, Bazaar, CVS und SVN. Die ersten drei sind **lokal und dezentral** organisiert und die letzten beiden liegen zentral auf einem Server.

**Changeset** Inhaltsänderung an einer oder mehreren Dateien

**Commit** Abspeichern eines Changeset im Repository

**Version** Eindeutige Identifizierung eines Commits

## 5.1. Merkmale

- Jederzeit ist bekannt, wer, was, warum geändert hat.
- Alle Zustände einer Komponente kann nachverfolgt werden
- Unbeabsichtigte Änderungen sind ausgeschlossen, da diese explizit durch einen Commit bestätigt werden müssen.

## 5.2. Commits

Man sollte oft und früh commiten, sofern folgende Bedingungen erfüllt sind: (**Commit Early, Commit Often**)

- Feature funktioniert
- Feature hat Tests
- Feature ist dokumentiert
- Feature wurde gereviewed

## 5.3. Branches

Es empfiehlt sich, wenn immer möglich, mit lokalen Branches zu arbeiten und nur fertige Features in den `master`-Branch zu mergen.

## 5.4. Definition of Done

1. Code übersetzt ohne Fehler und ohne Warnungen
2. Unit Tests laufen fehlerfrei
3. Test-Abdeckungsgrad wie definiert (in sep. Doku)
4. Kein unaufgeräumter (z.B. auskommentierter) Code
5. Alles Unfertige ist markiert mit `TODO:`
6. Code ist wo nötig, und richtig mit Kommentaren versehen
7. metrics, findbugs, ReSharper, structure101, STAN und andere eingesetzte Analysetools geben grünes Licht
8. Vier-Augen-Prinzip: entweder Pair Programming oder gereviewt und Review dokumentiert

## 5.5. Git

Git ist eine Versionsverwaltung. Erzeugte Dateien wie z.B *.pdf oder *.exe sollte nie in einem Repo abgelegt werden.

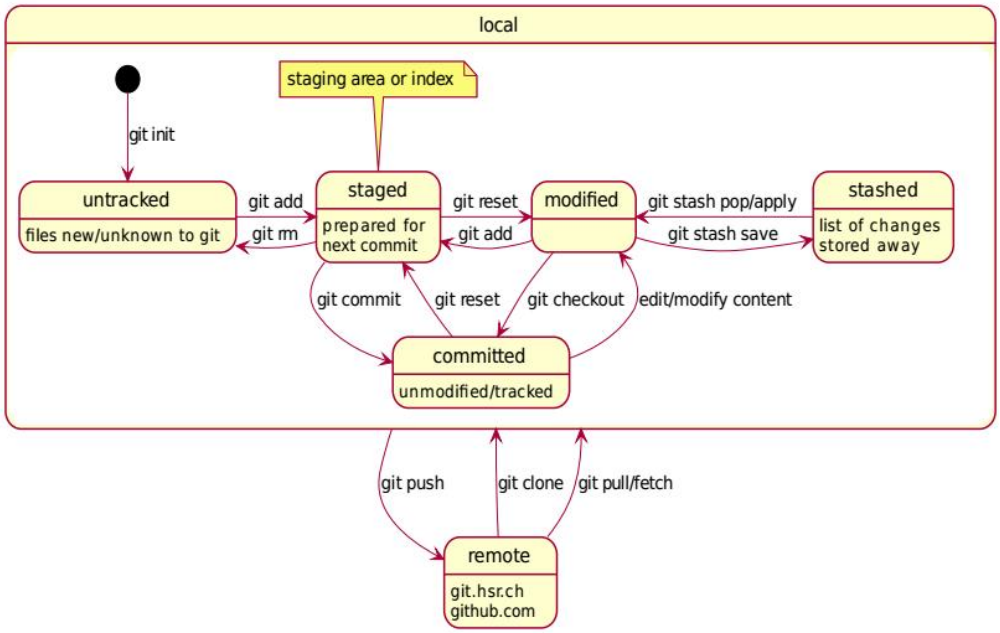


Abbildung 17: Git Übersicht

### 5.5.1. Struktur

**Master** Hauptstamm

**Branch** Abzweigung vom Stamm

**HEAD** Zeiger auf den aktuellsten Commit des aktuellen Branches. Kann mit `checkout` versetzt werden.

**Tag/Label** Eindeutiger Name für bestimmten Commit

### 5.5.2. Branches

Es empfiehlt sich, wenn immer möglich, mit lokalen Branches zu arbeiten.

1. Feature Branch erstellen
2. Änderung durchführen
3. Feature in Branch überführen (mittels Merge oder Rebase). Rebasing ist dem Merging vorzuziehen, da man damit eine saubere History erreicht. ACHTUNG: Falls sonst jemand auf dem gleichen Branch arbeitet, sollte kein Rebase gemacht werden.
4. Branch löschen

### 5.5.3. Operationen

**Einrichten**

**git config –global user.name ""** Username setzen

**git config –global user.email ""** Email setzen

**Basisoperationen**

**git clone https://github.com/xxx/yyy.git** Entferntes Repository kopieren

**git init** Erstelle ein lokales Git Repository

**git add file.txt** Füge file.txt in die Versionkontrolle (staged). Mit dem Flag -p kann selektiv hinzugefügt werden.

**git reset file.txt** Entfernen aus dem staged Bereich

**git rm –cached file.txt** Entfernen aus dem tracked Bereich. Mit dem Flag -p kann selektiv entfernt werden. Das –cached bewirkt, dass das File nicht im Filesystem gelöst wird.

**git commit -m "Message"** Commit staged changes

**Remotes**

**git remote add origin https://github.com/xxx/yyy.git** Remote hinzufügen

**git remote show -v** Remotes anzeigen

**git checkout origin/<Commit Hash|Branch|HEAD>** Entfernten Branch auschecken

**git fetch origin [[remote branch]:[local branch]]** Änderungen des entfernten Repositoy abholen.

**git pull origin [[remote branch]:[local branch]]** Pull from remote. Entspricht fetch & merge

**git push origin [[local branch]:[remote bra nch]]** Push to remote. Mit dem Parameter -u wird der verwendete Remote gespeichert.

**Abfragen**

**git status** Beschreibung

**git version file.txt** Beschreibung

**git diff HEAD** Vergleiche aktuelle Änderungen mit dem letzten Commit

**git diff -b -w –staged** Zeige was sich gerade geändert hat. (Ignore blanks and whitespaces)

**git log** Zeit Commit Hashes an

**Auschecken**

**git checkout [Commit Hash|Branch|HEAD]** Commit, Branch oder HEAD auschecken

**git checkout <Commit Hash|Branch|HEAD>^** Gehe eine Commit zurück

**git checkout <Commit Hash|Branch|HEAD>~<Anzahl>** Gehe eine bestimmte Anzahls an Commits zurück

**Mergen**

**git checkout master** Branch auschecken, in welchen gemerged werden soll

**git merge mybranch** Merge mybranch in den aktuellen Branch

**Branches**

**git checkout -b branch** Branch erstellen und zu ihm wechseln

**git checkout mybranch** Zu existierendem Branch wechseln

**git branch -d mybranch** Branch löschen

**git branch -f <Commit Hash|Branch|HEAD> <Commit Hash|Branch|HEAD>** Kopiere den einen Branch in einen anderen

**git reset HEAD~1** Branch auf den letzen Commit zurücksetzen

**git revert HEAD** Remote Branch zurücksetzen

**Rebase**

**git rebase master mybranch** Packe Branch vor den Master

**Cherry Pick**

**git cherry-pick C1 C2 C3** Füge einzelne Commit vor den HEAD ein

**git rebase -i Branch** Es öffnet sich ein interaktives GUI in welchem die Commits ausgewählt werden können.

**Tags**

**git tag descr branch** Erstelle Tag für Branch mit Beschreibung

**Stashes**

**git stash** Aktueller Stand in Zwischenablage kopieren

**git stash list** Stashes anzeigen

**git stash apply stash@1** Bestimmte Stash von der Zwischenablage zurückkopieren. `git stash pop` nimmt einfach den letzten Stash vom Stack

# 6. Software Testing

## 6.1. Terminologie

**SUT: System under Test** Das zu testende System

**Validierung** Machen wir das Richtige (Anforderungen spezifizieren)

**Verfikation** Machen wir es richtig (Systemtests, Integrationstests, Unit Tests)

## 6.2. Ablauf

1. **Testplan** Wer, was, wann, wie?
2. **Testspezifikation** Beschreibt was getestet werden muss
3. **Testprotokoll** Beschreibt das Resultat der Tests

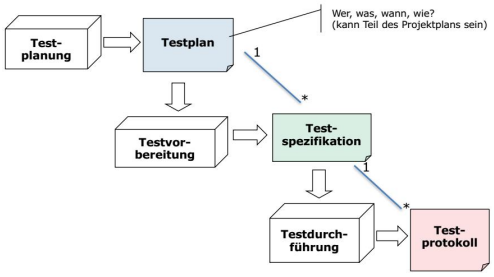


Abbildung 18: Ablauf einer Tests

## 6.3. Testfälle

Gute Testfälle sind, `null`, negative Werte, leere Strings, ungültige Eingaben

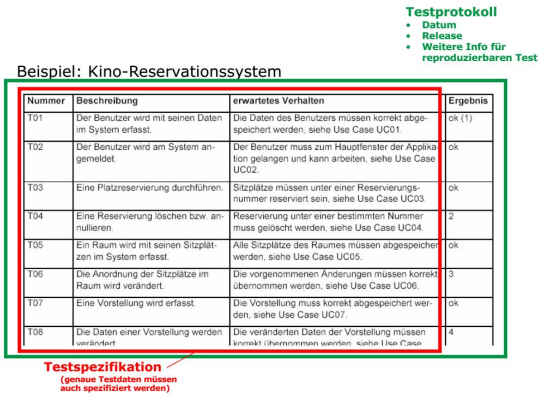


Abbildung 19: Testspezifikation und Protokoll

## 6.4. Vorgehen

1. Früh testen
2. Häufig testen
3. Systematisch testen
4. Automatisiert testen:
5. Test anything that might break: Keine Tests für funktionslosen Code
6. Test everything that does break: Test schreiben, der den Fehler reproduziert, danach korrigieren (Regressionstests)
7. New Code is guilty until proven innocent
8. Tests ausführen, vor dem einchecken (am besten automatisiert)

## 6.5. Varianten

**Modul-, Unit und Microtests** testen die korrekte Arbeitsweise einer einzelnen Klasse/Methode

**Integrationstests** testen das korrekte Zusammenspiel von mehreren Einheiten

**Systemtests** testen das Gesamtsystem nach der Erfüllung der geforderten Anforderungen

**Abnahmentests** Der Kunde testet das Gesamtsystem

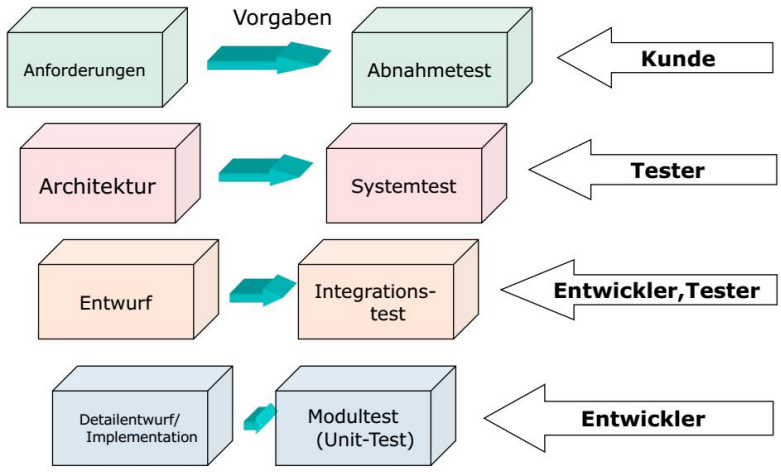


Abbildung 20: Testvarianten

## 6.6. Funktionale Tests

**Black Box Tests** Testfälle ohne Kenntnis der inneren Struktur. Die Menge der zu testenden Objekte sollte in Äquivalenzklassen unterteilt werden. Danach testet man am besten an den **Grenzen der Äquivalenzklassen**. (Grenzwertanalyse)

**White Box Tests** Testfälle mit Kenntnis der inneren Struktur (Unit Tests)

## 6.7. Nichtfunktionale Tests

Nichtfunktionale Anforderungen werden mit folgenden Tests überprüft.

- Usability Tests
- Performance Messungen

## 6.8. Automatisierte Tests

- Der grosse Vorteil von automatisierten Tests ist, dass sie wiederholbar sind
- Der Nachteil ist natürlich, dass man mehr Code schreiben muss. Es lohnt sich aber.

## 6.9. Gute Tests

Gute Tests folgen dem A-TRIP-Prinzip (Automatic, Thorough, Repeatable, Independent, Professional). Ein guter Test...

- Ist systematisch, wiederholbar und automatisiert
- Tested genau einen Execution Path
- Tested relevant Dinge (keine Getter und Setter)
- Abhängigkeiten werden durch Mocks ersetzt (Low Coupling)
- Folgt dem Arrange, Act, Assert, Teardown Prinzipt

## 6.10. Äquivalenzklassen

Äquivalenzklassen sind Wertebereiche für Eingabeparameter eines Tests, für welche ein gleichartiges Verhalten des Prüflings erwartet wird.

## 6.11. Failure und Errors

**Failure** Beschreibt einen Assert, der fehlschlägt

**Error** Beschreibt eine nicht behandelte Exception im zu prüfenden Code

## 6.12. Regressionstests

Unter einem Regressionstest versteht man in der Softwaretechnik die Wiederholung von Testfällen, um sicherzustellen, dass Modifikationen in bereits getesteten Teilen der Software keine neuen Fehler verursachen. Man schreibt dazu, für jeden gefundenen Bug einen Test, der den Fehler reproduziert.

## 6.13. Microtest

Microtests sind kurze, einfache, automatisierte Tests die **genau ein Verhalten** eines Objekts überprüfen. Dabei sollte so viele "Execution Paths" wie möglich abgedeckt werden. Microtests werden in drei Dimensionen gemessen. Bevor man mit Test Driven Developement anfängt, sollte man sich mit kleiner Komplexität, geringen Abhängigkeiten und After-Testing beschäftigen.

**Complexity, Komplexität:** Die Komplexitätsachse geht von einfach bis hart

**Collaboration, Abhängigkeiten:** Haben die Klassen grosse oder kleine Abhängigkeiten untereinander

**Timing, Zeitpunkt:** Schreiben wir die Tests vor oder nachdem wir den Code geschrieben haben

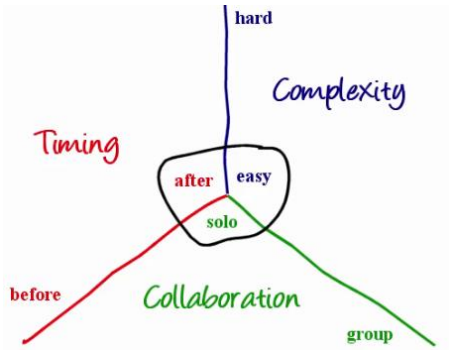


Abbildung 21: Microtest Dimensionen

### 6.13.1. Ablauf

Ein Microtest besteht immer aus folgender Abfolge:

1. Arrange: Die zu testenden Objekte instanzieren
2. Act: Die zu testende Funktionalität aufrufen
3. Assert: Definiere das erwartete Resultat und vergleiche den Rückgabewert mit diesem
4. Teardown: Aufräumen, sofern dies nicht vom Garbage Collector erledigt wird.

## 6.14. jUnit

JUnit Tests sollten im gleichen Package wie der Production Code sein, jedoch in separierten Order.

- myProject/src (ch.hsr.mypackage)
- myProject/tests (ch.hsr.mypackage)

```
import static org.junit.Assert.*;

public class MyTestClass {
   private List<String> myList;

   @BeforeClass
   public static void openSession() {
      openDBConnection(); // rather time consuming, so do it just once
   }

   @Before
   public void setUp() {
      list = new ArrayList<>();
   }

   @Test
   public void myTest() {
      myList.add("test");
      assertEquals("test", myList.get(0))
   }

   @After
   public void tearTown() {
      myList.clear();
   }

   @AfterClass
   public static void closeSession() {
      closeDBConnection();
   }
}

// Asserts
assertEquals(expected, actual);
assertEquals(expected, actual, precision); // float, double
assertSame(anObject, sameObject);
assertNull(object);
assertNotNull(object);

// Exceptions: Expected Annotation
@Test(expected=NumberFormatException.class)
public void parseIntThrowsException() { ... }
// Exceptions: Fail explicit
public void testParseIntThrowsException() {
   try {
      Integer.parseInt("trying to parse letters instead of numbers");
      fail("Should have thrown NumberFormatException"); // should not happen
   } catch (NumberFormatException expectedException) {
      // if we get here it means the test has passed
   }
}
```

# 7. Clean Code

## 7.1. Regeln

### Gutes Naming

- Aussprechbar (besser `powersOfTwo` anstatt `pwrsOf2`), konsistent, mit klarer Bedeutung (nicht text1, text2, text3)
- Methoden sind immer Verben
- boolean starten mit ”is”
- Kurze Namen nur bei kleinen Scope/Gültigkeitsbereich einer Variable/Methode
- Nicht abkürzen wenn man nur 3 Buchstaben spart (z.B groupID, grpID oder nameLength, namln)
- keine Präfixe
- Keine Zahlen, Umlaute verwendent
- Keine anonymen Konstanten (Ausnahme 0 und 1). z.B `for(int i = 1; i < 27; i++;)` → `const int MAX_OPEN_FILES = 27;`

### Nützliche Kommentare

- Der Methodenname sollte sagen, was gemacht wird
- Die Statements im Code zeigen, wie es gemacht wird
- Ein (eventueller) Kommentar sollte erklären warum es genau so gemacht wird.
- Ein Kommentar sollte nie das offensichtliche kommentieren
- Kommentare wirklich nur wenn nötig schreiben
- JavaDoc sollte nur auf den unteren Schichten geschrieben werden, damit die überliegenden Layer klar verständlich sind, ohne den Code zu lesen.

### Programmiere für den Menschen, nicht für den Compiler

- Code wird häufiger gelesen als geschrieben
- Der Code sollte so geschrieben werde, wie wir ihn selber gerne antreffen würden.
- Ein Review in der Gruppe zeigt, ob der Code gut lesbar ist.
- Weiter gehen als ”Hurra es läuft” oder ”Works for me”

### DRY: Don’t Repeat Yourself

- Kein duplizierter Code
- Bei viel Copy Paste, wird der Code aufgebläht und es entstehen mehr Fehler, Arbeit bei Änderungen
- Auch bekannt als DIE: Duplicate is Evil

#### Single Responsibility Principle

- Eine Methode ist genau für eine Sache zuständig.
- Dieses Paradigma ist auch bei Klassen sinnvoll, jedoch schwerer umzusetzen.
- Wird die Single Responsibility verletzt, hat dies Folgen bei Git, Unit Tests und Kopplung steigt.

#### KISS: Keep It Simple Stupid

- Es ist schwerer, den Programmcode einfach zu halten, anstatt ihn komplex zu programmieren.
- Beim Testen wird schnell klar, ob ein Code einfach ist.

#### YAGNI: You Ain't Gonna Need it

- Es sollte nichts entwickelt werden, ohne dass eine explizite Anforderung dafür besteht.
- Keine Implementierung im Sinne "In der Zukunft könnte man das brauchen".

#### Programm to the Interface, not Implementation

- `List<T>` anstatt `ArrayList<T>`

#### Isolte What Changes

- Alles was sich nicht ändert, soll in einem Interface modelliert werden

#### Aufgeräumter Coder

- Einrückung, Formatierung ist für die Übersicht wichtig
- Auskommentierter Code sollte weggelassen werden. Dafür gibt es Version Control
- TODO's implementieren
- Geschweifte Klammern immer setzen (insbesondere bei Verschachtelungen)

#### Positive Bedingungen

- Im `if` sollten immer die positiven Bedingen deklariert werden
- Ausnahme sind die Guards. Ein Guard lässt eine Methode schnell terminieren, falls bestimmte Bedinungen nicht gegeben sind.
- Ein schneller Ausstieg ist auch, wenn z.B eine Iteration bei einem Match frühzeitig abgebrochen werden

#### Coding Guidlines

- Man sollte so programmieren, wie man dies im Team abgemacht hat. Die Konvention kann von der Programmiersprache und der Präferenz des Mehrheit abhängen.

### Handhabbare Grössen

- Design Diagramme: max A3
- Code Breite: max 120 Zeichen
- Verschachtelungen: max. 5 Stufen
- Methoden: max 30 Zeilen
- Klassen: max 300 Zeilen
- Tiers: max. 4 Tier

### No Errors, No Warnings

- Warnungen sollten immer behoben werden, da sich diese ansonsten schnell ansammeln und nie mehr gefixt werden.
- Ausnahme: Serializeable UID → Compiler so einstellen, dass es die Warnungen ignoriert
- Gilt auch für Addons wie Checkstyle, Metrics, FindBugs etc.

### Kopplung und Kohäsion

In der Software Enwicklung strebt man immer nach **High Cohesion UND Low Coupling**.

- Kopplung: Wie stark ist eine Klasse Abhängig von vielen anderen Klassen. Aufrufe von einer Klasse zu anderen, von einem Package zum anderen
- Kohäsion: Zusammenhalt innerhalb einer Klasse. Kohäsion soll gegeben sein, sonst gehören die Dinge nicht ein eine Klasse/Package.

### Smart Data Structures

- Es ist viel wichtiger die richtige Datenstruktur zu verwenden, als wie die Datenstruktur verwendet wird.
- z.B gilt bei einer Schweizer Grossbank die Faustregel: `2:5:20` (ca. 2 Jahre hält das GUI, 5 Jahre hält die Geschäftslogik und 20 Jahre halten die Daten). Daher sollte die Datenstruktur gut gewählt werden.

# 8. Patterns

Die meisten der folgenden Pattern wurden von der Gang of Four (Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides) entworfen. Pattern sind **Best Practice Beispiele**, welche die Kommunikation unter Software Entwickler erleichert. Mit dem gemeinsamen Pattern Vokabular lässt sich **mit wenig, mehr sagen**. Charakteristik, Qualität und Einschränkungen sind auf anhin klar. Man unterscheidet zwischen drei Varianten von Pattern:

### Creational Pattern

- Abstract Factory 8.1
- Factory Method 8.2 (√)
- Singleton 8.3 (√)

### Structural Pattern

- Adapter 8.4 (√)
- Facade 8.5 (√)
- Use Case Controller 8.6 (√)
- Composite 8.7 (√)
- Decorator 8.8 (√)
- Proxy 8.9

### Behavioral Pattern

- Command 8.10 (√)
- Null Object 8.11 (√)
- Observer 8.12 (√)
- State 8.13 (√)
- Strategy 8.14 (√)
- Template Method 8.15 (√)
- Iterator 8.16

### Combined Pattern

- MVC 8.17 (√)

## 8.1. Abstract Factory

Das Abstract Factory Pattern bietet eine Schnittstelle zum Erstellen von Familien verwandter oder zusammenhängender Objekte an, ohne konkrete Klassen anzugeben.

- Im Gegensatz zum Factory Method Pattern sollte die Abstract Factory verwendet werden, wenn der Client eine ganze Familie zusammengehörenden Produkten verwenden muss.
- Die Abstract Factoy stützt sich auf Objekt Komposition. Die Objekt-Erstellung ist in Methoden implementiert, die in der Fabrik Schnittstelle vorgegeben werden muss.

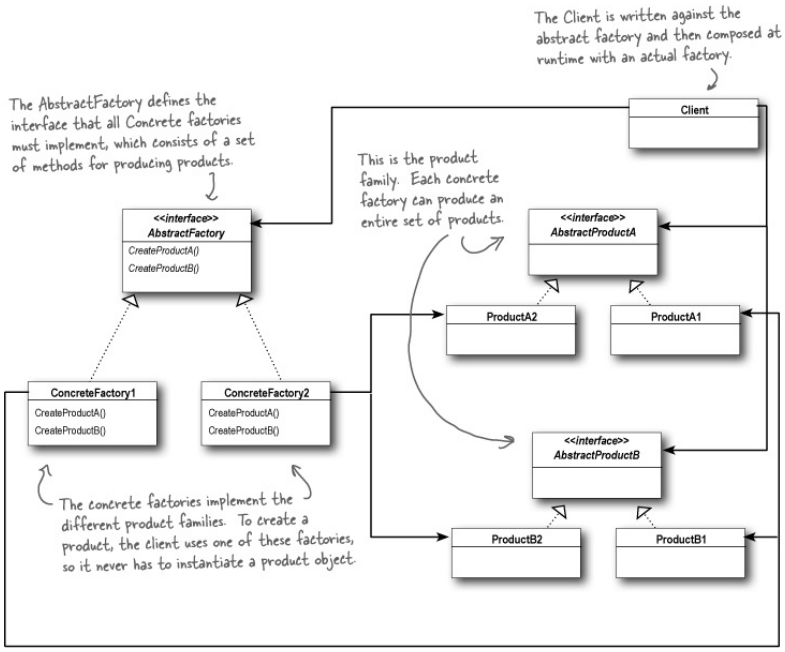


Abbildung 22: Abstract Factory

### 8.1.1. Implementierung

```java
// abstract product A
public interface Shape {
   void draw();
}

// concrete product A
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}

// concrete product A
public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}

// abstract product B
public interface Color {
   void fill();
}
// concrete product B
public class Red implements Color {
   @Override
   public void fill() {
      System.out.println("Inside Red::fill() method.");
   }
}
// concrete product B
public class Green implements Color {
   @Override
   public void fill() {
      System.out.println("Inside Green::fill() method.");
   }
}

// abstract factory
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Shape getShape(String shape) ;
}

// concrete factory A
public class ShapeFactory extends AbstractFactory {

   @Override
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if (shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
```

```
60          return null;
61      }
62
63      @Override
64      Color getColor(String color) {
65          return null;
66      }
67  }
68
69  // concrete factory B
70  public class ColorFactory extends AbstractFactory {
71      @Override
72      public Shape getShape(String shapeType){
73          return null;
74      }
75
76      @Override
77      Color getColor(String color) {
78          if(color == null){
79              return null;
80          }
81          if(color.equalsIgnoreCase("RED")){
82              return new Red();
83          } else if(color.equalsIgnoreCase("GREEN")){
84              return new Green();
85          }
86          return null;
87      }
88  }
89
90  // producer for the client
91  public class FactoryProducer {
92      public static AbstractFactory getFactory(String choice){
93          if(choice.equalsIgnoreCase("SHAPE")){
94              return new ShapeFactory();
95          } else if(choice.equalsIgnoreCase("COLOR")){
96              return new ColorFactory();
97          }
98          return null;
99      }
100 }
101
102 // client
103 AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");
104 Shape shape1 = shapeFactory.getShape("CIRCLE");
```

## 8.2. Factory Method

Das Factory Method Pattern definiert eine Schnittstelle zur Erstellung eines Objekts, lässt aber die Unterklassen entscheiden, welche Klassen instantiiert werden. Factory Method ermöglicht einer Klasse, die Instantiierung in Unterklassen zu verlagern.

- Das Factory Method Pattern wird verwendet, wenn man eine **komplexe Erzeugungslogik** hat. Die Logik der Objekterstellung liegt dabei in der Childklasse.
- Die Erzeugungslogik wird in dieser Methode gekapselt. Anhang von Parameter wird entschieden wie eine konkrete Klasse erzeugt wird
- Typische Methodennamen sind `createX()`, `makeX()` oder `newX()`
- Die Faktory Methode darf **nicht statisch** sein, da die Child die Methode überschreiben müssen. Statische Methoden können auch nicht `abstract` sein

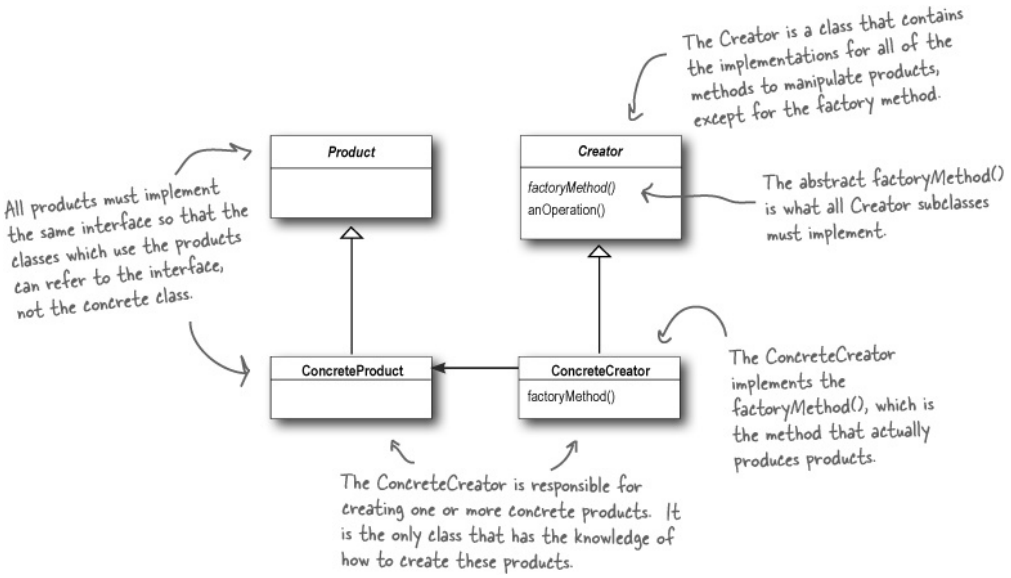


Abbildung 23: Factory Method

### 8.2.1. Implementierung

```
public abstract class Creator {

   public Product getProduct() {

      // create a pizza and execute actions like prepare, bake, cut, ..
      Product product = factoryMethod(type);
      product.x();
      product.y();

      return product;
   }

   // has to be overwritten in child
   protected abstract Product factoryMethod();
}

public class ConcreteCreatorA extends Creator {

   protected Product factoryMethod() {
      return new ConcreateProductA();
   }
}

public class ConcreteCreatorB extends Creator {

   protected Product factoryMethod() {
      return new ConcreateProductB();
   }
}

public class ConcreteCreatorC extends Creator {

   protected Product factoryMethod() {
      return new ConcreateProductC();
   }
}

// Client
Creator myCreator = ConcreteCreatorA();
Product product = myCreator.getProduct();
```

## 8.3. Singleton

Das Singleton Pattern stellt sicher, dass es **nur eine Instanz** einer Klasse gibt, und bietet einen globale Zugriffspunkt zu dieser Intanz

- Verglichen mit einer statischen Klasse (gibt es in Java nur bedingt) können Singletons, Interfaces oder nützliche Basisklasse implementieren. Ebenfalls sind Singletons im Heap und statische Klasse auf dem Stack. Ebenfalls werden Singletons erst bei Gebrauch initialisert. Statische Klassen hingegen bereits zur Compilezeit.
- Bei Multithreading Anwendungen muss die Singleton Implementierung noch entsprechen angepasst werden.
- Statische Klassen sind in Java als `final` deklariert, haben den Konstruktor `private` und haben alle Member `static`. Das `static` Keyword kann nicht auf der Ebene Klasse angewendet werden.

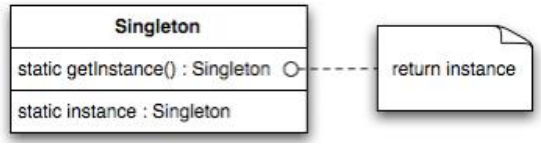


Abbildung 24: Singleton

### 8.3.1. Vorgehen

1. `private` Default Constructor
2. `static` `getInstance()` Methode, welche die interne Instanz zurückliefert.
3. `static` Instanzvariable vom Typ der Singleton Klasse

### 8.3.2. Implementierung

```
1  public class Singleton {
2     private volatile static Singleton uniqueInstance;
3
4     private Singleton() {}
5
6     public static Singleton getInstance() {
7        if (uniqueInstance == null) {
8
9           synchronized (Singleton.class) {
10             // check twice. could be set in the meantime
11             if (uniqueInstance == null) {
12                // allocate space only if needed
13                uniqueInstance = new Singleton();
14             }
15          }
16       }
17
18       return uniqueInstance;
19    }
20 }
```

## 8.4. Adapter

Das Adapter Pattern konvertiert die Schnittstelle einer Klasse in die Schnittstelle, die der Client erwartet. Adapter ermöglichen die Zusammenarbeit von Klassen, die ohne nicht zusammenarbeiten könnten, weil sie **inkompatible Schnittstellen** haben.

- Der Adapter ändert die Schnittstelle von einer existierende Klasse, damit sie dem Client entspricht.
- Der Adapter **kapselt** eine Klasse.

### 8.4.1. Anwendungsfälle

- Das Entwurfsmuster findet in erster Linie Anwendung, wenn eine existierende Klasse verwendet werden soll, deren Funktion nicht ganz der geforderten enspricht.

### 8.4.2. Varianten

Es gibt zwei Varianten von Adaptern:

**Objektadapter** : Verwendet Komposition

**Klassenadapter** : Verwendet Mehrfachvererbung

### 8.4.3. Objektadapter

- Auch als Wrapper bekannt
- Verwendet Komposition, welche zur Laufzeit hinzugefügt wird. Die Adapter Klasse besitzt also eine Instanzvariable vom Typ des Adaptee
- **Kapselt besser**, da die Methoden des Adaptee nicht sichtbar sind
- Alle Methoden die der Adapter zur Verfügung soll, müssen im Target Interface implementiert sein.
- Wird verwendet wenn man dem Client nur Methoden mit abgeänderter Adaptee Funktionalität zur Verfügung stellen möchte. Die restliche Adaptee Funktionalität soll aber versteckt bleiben.

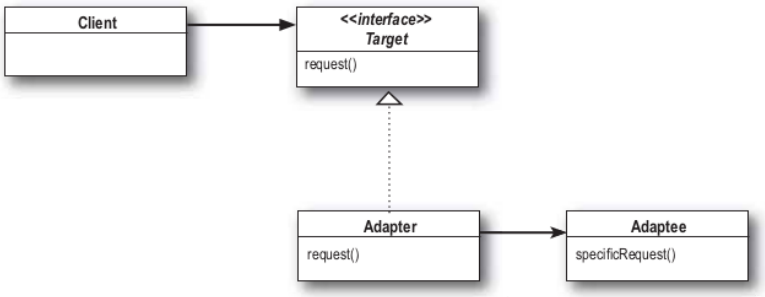


Abbildung 25: Objektadapter

**Implementierung** Der Objektadapter wird wie folgt verwendet

```
1  public class Adapter implements Target {
2     private Adaptee adaptee;
3
4     public void request() {
5        // convert, setup, etc.
6        adaptee.specificRequest();
7     }
8  }
9
10 public class Client {
11    // Adapter hat eine Instanzvariable des Adaptee
12    private Target myAdapter = new Adapter();
13
14    public Client() {
15       myAdapter.request();
16    }
17 }
```

### 8.4.4. Klassenadapter

- Verwendet Mehrfachvererbung welche bereits zur Compilezeit fixiert ist. Unter Java muss ein Interface und einer abstrakte Klasse verwendet werden, da Java keine Mehrfachvererbung unterstützt.
- Trägt die Methoden des Adaptees nach aussen
- Wird verwendet wenn eine zusätzliche Methode implementiert werden soll und man aber immer noch auf die Methoden des Adaptee zugreifen soll.

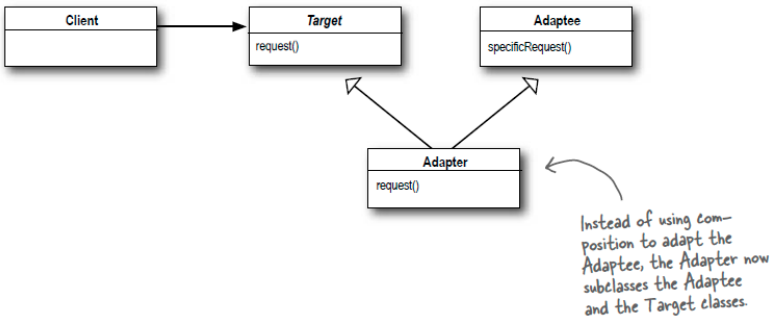


Abbildung 26: Objektadapter

**Implementierung** Der Klassenadapter wird wie folgt verwendet:

```
1  public class Adapter extends Adaptee implements Target {
2     public void request() {
3        super.specificRequest();
4     }
5  }
6
7
8  public class Client {
9     private Target myAdapter = new Adapter();
10
11    public Client() {
12       myAdapter.request();
13       // Alle Adaptee Methoden sind wegen der Vererbung ebenfalls sichtbar
14       myAdapter.anotherAdapteeMethod();
15    }
16 }
```

## 8.5. Facade

Das Facade Pattern biete eine **vereinheitlichte Schnittstelle für einen Satz von Schnitstellen** eines Basissystems. Die Facade definiert eine hochstufigere Schnittstelle, die die Verwendung des Basissystems vereinfacht.

- Die Facade **vereinfacht die Schnittstelle** einer Gruppe von Klassen.
- Die Facade abstrahiert die komplexe Verwendung mehrere Klassen für den Client weg.
- Die Facade kann im Gegensatz zum Adapter viele Klassen representieren
- Das Facade Pattern ist ähnlich wie der Adapter. Die Facade sollte immer dann verwendet werden, wenn eine grosse oder komplexe Schnittstelle vereinfacht werden muss. Der Adapter wird verwendet, wenn die gegebene Schnittstelle nicht der geforderten entspricht.
- Eine Facade entkoppelt einen Client von einem komplexeren Basissystem.

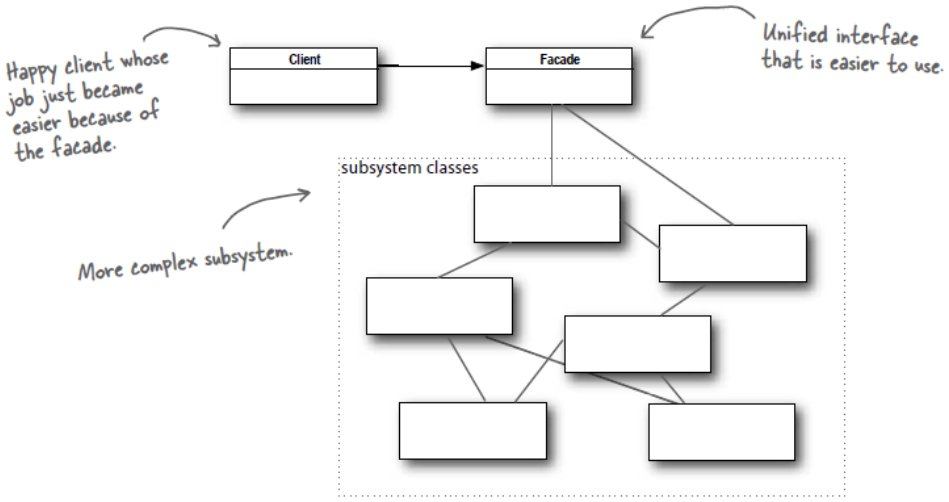


Abbildung 27: Fascade

### 8.5.1. Implementierung

```
// Client
Facade facade = new Facade();
facade.doAction();

// Facade
public class Facade {
   private ISystem SystemComponentA a;
   private ISystem SystemComponentB b;
   private ISystem SystemComponentC c;

   public Facade() {
      a = new SystemComponentA();
      b = new SystemComponentB();
      C = new SystemComponentC();
   }

   public void doAction(){
      a.doAction();
      b.doAction();
      c.doAction();
   }
}


// 'Compelex' system
public interface ISystem {
   void doAction();
}

public class SystemComponentA implements ISystem {
   @Override
   public void doAction() {
      System.out.println("System Component A created");
   }
}

public class SystemComponentB implements ISystem {
   @Override
   public void doAction() {
      System.out.println("System Component B created");
   }
}

public class SystemComponentC implements ISystem {
   @Override
   public void doAction() {
      System.out.println("System Component C created");
   }
}
```

## 8.6. Use Case Controller

- Der Use Case Controller representiert ein oder mehrere Use Cases.
- Der Use Case Controller bietet alle nötigen Operationen und Status Informationen um einen Workflow abzubilden.
- Die Daten hinter dem Use Case Controller können nur über ihn geändert werden
- Es können mehre Use Case Controller für einen Use Case existieren.

### 8.6.1. Anwendungsfälle

- Wird klassisch in einem MVC Umfeld eingesetzt, wobei er mit einer Model Facade kommuniziert.

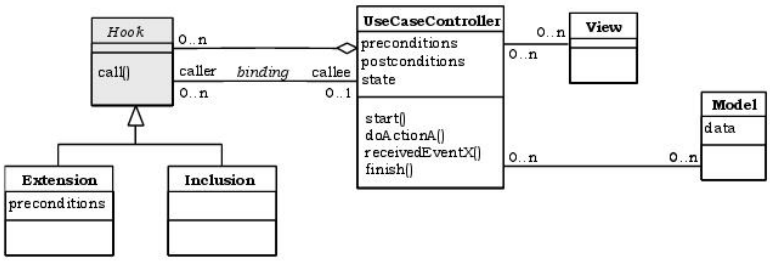


Abbildung 28: Use Case Controller

```
1 public class PlaceOrderUseCase extends UseCaseController {
2    public void start() { .. }
3    public void finish() { .. }
4    public void displayForm() { .. }
5    public void newItem() { .. }
6    public void saveItem() { .. }
7    public void submitOrder() { .. }
8    public void cancelOrder() { .. }
9 }
```

## 8.7. Composite

Das Composite Pattern ermöglicht es, Objekte zu einer **Baumstruktur** zusammenzusetzen, um **Teil/Ganzes** Hierarchien auszudrücken. Das Composite Pattern erlaubt den Clients, individuelle Objekte und Zusammensetzungen von Objekten auf gleiche Weise zu behandeln.

- Ein Composit enthält mehrere Komponenten, Diese sind entweder Composites oder Blatt Knoten.
- Eine Komponente ist eine Abstrakte Klasse, damit eine default Implementierung erstellt werden kann (seit neueren Java Versionen auch mit den Interfaces möglich)
- Die Leaf Knoten überschreiben nur diese Operationen die Sinn ergeben. Bei allen anderen Operationen wird die Default Implementierung verwendet.
- Die Composite Knoten überschreiben die Default Implementierung der abstrakten Klasse.
- Die Composite Knoten können selber nichts ausgeben. Sie delegieren die Operation nur an die Leafs.
- Die Default Implementierungen müssen nicht zwingend eine Exception werfen, sondern können auch sinnvollen Code enthalten.
- Bei teuren Durchquerungen des Trees, macht es evtl. Sinn die Zwischenergebnisse zu cachen.
- Der grösste Vorteil des Composit Pattern ist es, das ein Methodenaufruf einmalig getriggert und in der vollständigen Struktur aufgerufen wird.
- Clients behandeln Sammlungen von Objekten und einzelne Objekte auf die gleiche Weise

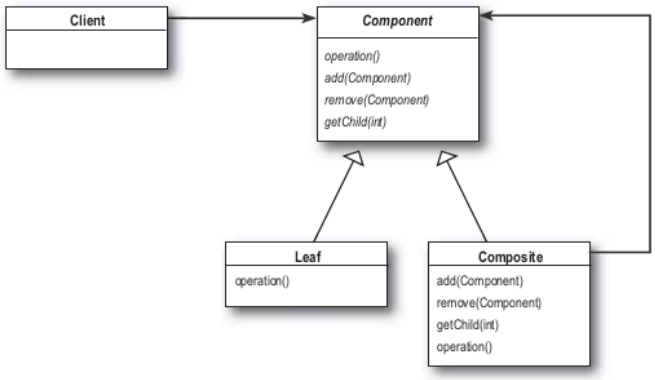


Abbildung 29: Composite

### 8.7.1. Anwendungsfälle

- GUI: Ein GUI besteht aus vielen Teilen. Bei der Anzeige, wird es jedoch als Ganzes betrachet. Man sagt der Top Komponenten, dass es sichtbar sein soll und dieses delegiert es an all seine Subkomponenten.

### 8.7.2. Implementierung

```
1  public abstract class Component {
2     public void add(Component c) {
3        throw new UnsupportedOperationException();
4     }
5
6     public void remove(Component c) {
7        throw new UnsupportedOperationException();
8     }
9
10    public Component getChild(int i) {
11       throw new UnsupportedOperationException();
12    }
13
14    public String operation() {
15       throw new UnsupportedOperationException();
16    }
17
18    public void print() {
19       throw new UnsupportedOperationException();
20    }
21 }
22
23 public class Leaf extends Component {
24    String name;
25
26    public Leaf(String name) {
27       this.name = name;
28    }
29
30    @Override
31    public String operation() {
32       return name;
33    }
34
35    @Override
36    public void print() {
37       System.out.println(getName());
38    }
39 }
40
41 public class Composite extends Component {
42    private List<Componet> componets = new ArrayList<>();
43    private name;
44
45    public Composite(String name) {
46       this.name = name;
47    }
48
49    public void add(Component c) {
50       componets.add(c);
51    }
52
53    public void remove(Component c) {
54       components.remove(c);
55    }
56
57    public Component getChild(int i) {
58       return (Component) components.get(i);
59    }
```

```
60
61     public String operation() {
62        return name;
63     }
64
65     public void print() {
66        Iterator it = components.iterator();
67        while(it.hasNext()) {
68           Component c = it.next();
69           component.print();
70        }
71     }
72
73  }
74
75  // Client
76  // Creation could be done with a factoy
77  Component group = new Composite("all");
78
79  Component part1 = new Composite("part1");
80  Component part2 = new Composite("part2");
81
82  group.add(part1);
83  group.add(part2);
84
85  Component sub1 = new Leaf("sub1");
86  Component sub2 = new Leaf("sub2");
87
88  part1.add(sub1);
89  part2.add(sub2);
90
91  // prints all components
92  group.print();
```

## 8.8. Decorator

Ein Decorator Muster fügt einem Objekt **dynamisch zusätzliche Verantwortlichkeiten** hinzu. Dekorierer bieten eine **flexible Alternative zur Ableitung** von Unterklassen zum Zweck der Erweiterung der Funktionalität.

- Ein Decorator hat den gleichen Supertyp (Interface oder Abstrakte Klasse) wie die Objekte, die er dekoriert
- Komponenten können beliebig oft, dynamisch zur Laufzeit, dekoriert werden (verschachtelt)
- Jeder Decorator hat genau eine Instanzvariable, die eine **Referenz auf das Komponenten-Objekt** hält
- Decorator werden oft durch Factories oder Builder erstellt.
- Der Client muss nicht Wissen, dass er es mit einem Decorator zu tun hat.

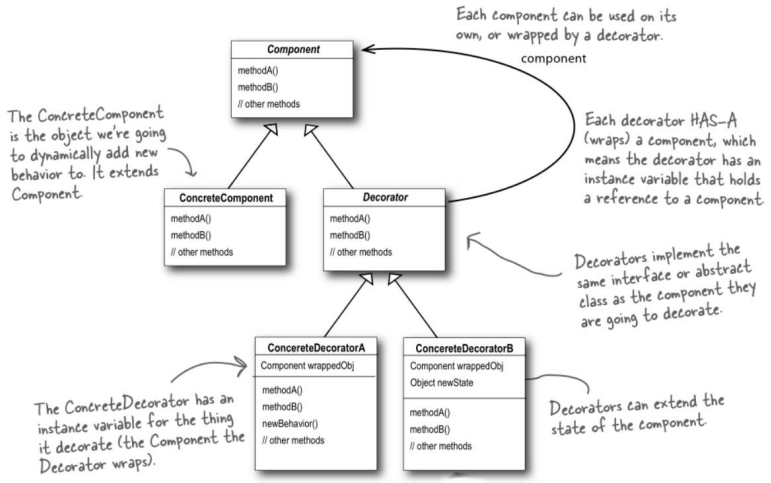


Abbildung 30: Decorator

### 8.8.1. Vorgehen

1. Gegeben ist eine Klasse `Gertaenk` (Interface oder abstrakte Klasse)
2. Erstelle eine Decorator Klasse (Interface oder abstrakte Klasse) und erbe von `Gertaenk`
3. Beliebige Subtypen von `Gertaenk` erstellen
4. Beliebig viele konkrete Klassen erstellen, die von Dekorator erben und eine Instanzvariable von `Gertaenk` besitzen. Die Instanzvariable ist eine Referenz auf das eingepackte Objekt. (z.B Mit Zitrone, Mit Alkohol)
5. Die einzelnen Objekte werden dem Konstruktor übergeben und so ineinander verschachtelt

### 8.8.2. Einschränkungen

- Decorator können nicht verwendet werden, wenn auf den Typ der konkrenten Komponenten geprüft werden muss.
- Ein Decorator kennt nie alle eingepackten Objekte, sondern nur das Nächste.
- Das Decorator Pattern verursacht viele kleine Klassen, weshalb man für die Erstellung der Decorator eine Factory Methode oder Builder nutzen sollte.

### 8.8.3. Implementierung

```
1  // Abstract Component
2  public abstract class Component {
3
4     public double getPrice() {
5        // call abstract method
6        return baseCost();
7     }
8
9     // abstract method
10    protected abstract double baseCost();
11 }
12
13
14 // Multiple concrete Components
15 public class ConcreteComponent extends Component {
16    @Override
17    public double baseCost() {
18       return 21;
19    }
20 }
21
22
23 // Decorator extends Component
24 public abstract class Dekorationen extends Component {
25    protected Component component;
26 }
27
28
29 // Concrete Decorator
30 public class ConcreteDecorator extends Decorator {
31
32    // Wrap now inner component
33    public ConcreteDecorator (Component component) {
34       // pass to parent
35       super.component = component;
36    }
37
38    @Override
39    public double baseCost() {
40       return component.basisKosten() + 4.5;
41    }
42 }
43
44 // Client
45 Component c = new ConcreteComponent();
46 c = new ConcreteDecoratorA(c); // wrap component in decorator
47 c = new ConcreteDecoratorB(c); // wrap component in decorator
48 c.getPrice();
```

## 8.9. Proxy

Das Proxy Pattern **kontrolliert den Zugriff** auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.

- Das Echte Subjekt und der Proxy implementieren die selbe Schnittstelle, damit den Stellvertreter am gleichen Ort wie das echte Objekt einsetzen kann.
- Der Proxy hält eine Referenz auf das echte Subjekt, dass die eigentliche Arbeit erledigt.

### 8.9.1. Varianten

Es gibt viele Proxy Varianten, wobei die drei wichtigsten die folgenden sind:

**Remote Proxy** Kontrolliert den Zugriff auf ein entferntes Objekt

**Virtual Proxy** Kontrolliert den Zugriff auf eine Ressource, deren Erzeugung aufwändig ist

**Protection Proxy** Kontrolliert den Zugriff auf eine Ressource mit genau festgelegten Zugriffsrechten

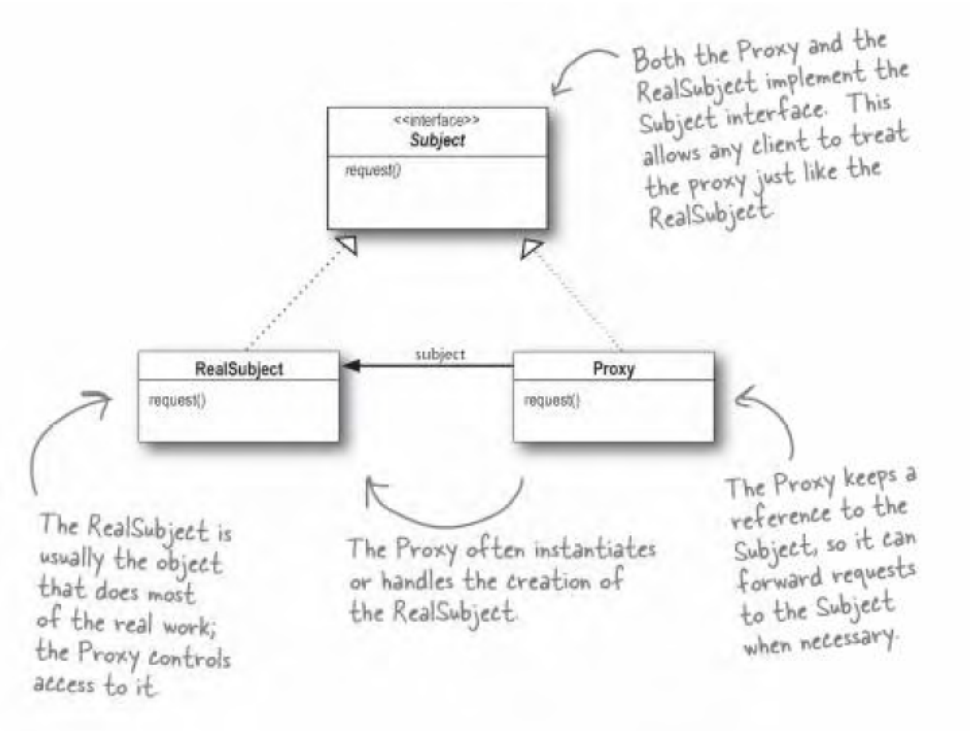


Abbildung 31: Proxy

### 8.9.2. Implementierung

```java
// Subject
public interface Image {
   void display();
}

// Realsubject
public class RealImage implements Image {
   private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }

   // this gets called only once! (when realImage is null in proxy)
   // after that, image is loaded and can be displayed multiple times
   private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
}

// Proxy
public class ProxyImage implements Image{
   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}

// Client
Image image = new ProxyImage("test_10mb.jpg");
image.display();
```

## 8.10. Command

Das Command Pattern kapselt einen Auftrag als ein Objekt und ermöglicht es so, andere Objekte mit verschiedenen Aufträgen zu paramatrisieren, Aufträge in **Warteschlangen** einzureihen oder zu protokollieren oder das **Rückgängigmachen** von Operationen zu unterstützen. Commands sind der **objektorientierte Ansatz von Callbacks**.

- Kapsle die Aktion als Command Objekt und führe Sie zu einem beliebigen Zeitpunkt aus.
- Gibt allen Command Objekten eine gemeinsame Schnittstelle (oft `execute()`)
- Statt `action()` direkt aufzurufen, erzeuge das entsprechende Command Objekt, auf dem dann die Aktion ausgeführt wird.
- Es können mehrere Commands ausgeführt werden (Makro-Commands)
- Commands können Rückgängig gemacht werden (besser Memento verwenden)
- Commands können gequeued und geloggt werden
- Oft wird der effektive Befehl im Receiver direkt im Command implementiert. Somit fällt der Receiver weg. Dieses vorgehen ist in der Praxis relativ gängig.

### 8.10.1. Anwendungsfälle

- Transaktionssysteme (Während die Commands ausgeführt werden, werden sie auf gespeichert und können im Fehlerfall wiederhergestellt werden.)

### 8.10.2. Vorgehen

1. Der Client ist verantworlich ein Command Objekt, ein Invoker und ein Receiver zu erstellen
2. Der Client ruft `setCommand()` auf, um den Command im Invoker zu speichern
3. Später fodert der Client den Invoker auf, den Befehl auszuführen.
4. Der Invoker kann mehrer Commands als Instanzvariablen haben. Bei gegebenen Event werden alle diese Commands ausgeführt.

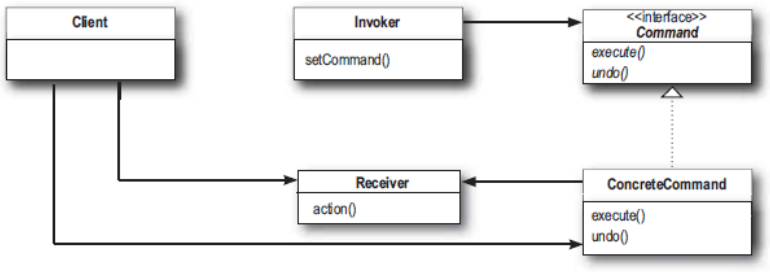


Abbildung 32: Command

### 8.10.3. Implementierung

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

// Receiver
public class Light() {
   public void on() {
      // magic happens here
   }
}

// Concrete command
public class LightCommand implements Command {

   // Holds a receiver
   Light light;

   public LightCommand(Light light) {
      this.light = light;
   }

   public void execute() {
      // action might be directly implemented in command class
      light.on();
   }

   public void undo() {
      light.off();
   }
}

// Invoker
public class Remote() {
   // could also be single command
   private List<Command> commands = new ArrayList<Command>();
   private List<Command> undoCommands = new ArrayList<Command>();

   public void setCommand(Comman command) {
      this.commands.add(command);
   }

   // event
   public void buttonPressed() {
      for (Command command : commands) {
         command.execute();
         undoCommands.add(command);
      }
   }

   // undo event
   public void undoCommand() {
      for (Command undoCommand : undoCommands) {
         undoCommand.undo();
      }
      undoCommands.clear();
   }
}

```

```
60 // Client
61 Remote remote = new Remote(); // invoker
62 Light light = new Light(); // receiver
63 Command command = new LightCommand(light); // command
64 remote.setCommand(command);
65
66 remote.buttonPressed(); // trigger event
```

## 8.11. Null Object

- Das Null Objekt wird immer dann verwendet, wenn man den Client davon befreien möchte, mit `null` umzugehen.
- Das Null Objekt ist eine **normale Klasse ohne Funktionilität**, die jedoch eine Schnittstelle implementiert.
- Das Null Objekt wird in vielen Pattern verwendet

### 8.11.1. Implementation

```
1  // Null Objekt im Command Pattern
2  public class NullCommand implements Command {
3     public void execute() { /* do nothing */ }
4     public void undo() { /* do nothing */ }
5  }
6
7  // Null Object beim Iterator
8  public class NullIterator implements Iterator {
9     public Object next() {
10       return null;
11    }
12
13    public boolean hasNext() {
14       return false;
15    }
16
17    public void remove() {
18       throw new UnsupportedOperationException();
19    }
20 }
```

## 8.12. Observer

Das Observer Pattern definiert eine **Eins-zu-viele-Abhängigkeit** zwischen Objekten in der Art, dass alle abhängingen Objekte benachrichtigt werden, wenn sich der Zustand eines Objekts ändert.

- Das Subjekt hält die Daten und gibt Bescheid, wenn diese ändern. (Die Java Variante nennt das Subject Observable)
- Die Observer melden sich beim Subjekt an oder ab
- Beim Observer Pattern sind Subjekt und Observer locker gebunden. Das Subjekt arbeitet einzig mit dem Observer Interface.
- Bei der Java Variante kann man sich nicht auf die Reihenfolge der Benachrichtigung verlassen. Ebenfalls muss `setChanged()` gesetzt werden.
- Der Observer ermöglicht es, eine Gruppe von Objekten zu benachrichtigen, wenn sich irgendein Zustand ändert.

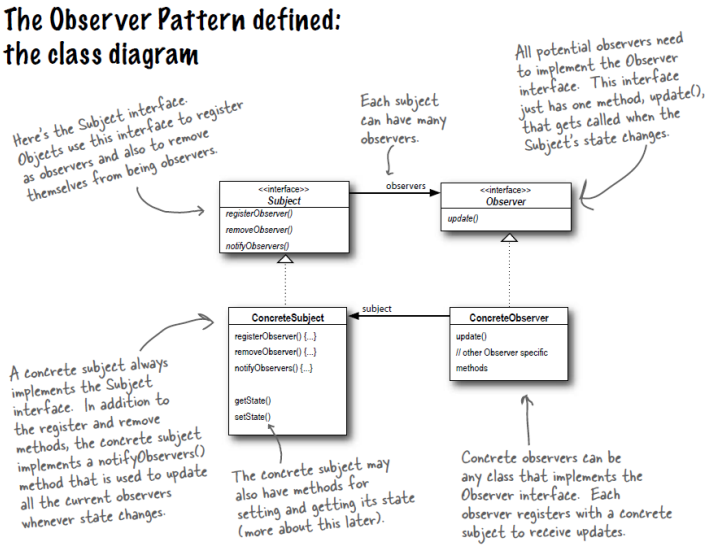


Abbildung 33: Observer

### 8.12.1. Vorgehen

1. Erstelle ein Subjekt mit den Methoden `register(Observer o)`, `remove(Observer o)` und `notify()`. Man erstellt immer ein Interface sowie ein konkretes Subjekt. Das konkrete Subject hält eine Liste von Observer, sowie die Werte die verteilt werden.
2. Die Observer müssen das Interface `Observer` implementieren. Das Observer Interface besitzt nur die Methode `update(x,y,z)`, wobei die Parameter so gewählt sind, dass die Daten vom Subject übergeben werden können. Auch hier wird ein Interface und ein konkreter Observer erstellt. Jeder Observer registriert sich bei einem konkreten Subjekt.

### 8.12.2. Implementierung

Ein Observer kann eigenhändigt erstellt werden oder man kann die vorhanden Klassen aus `java.util` verwenden. Das konkrete Objekt würde in diesem Fall von `java.util.Observable` erben und die konkreten Observer `java.util.Observer` implmentieren. Damit die Observer benachrichtig werden muss zunächst die `setChanged()` Methode augerufen und anschliessend einer der beiden Notify Methoden aufgerufen werden. (`notifyObservers()` oder `notifyObservers(Object arg)`). Beim konkreten Observer muss zusätzlich die Methode `update(Observable o, Object arg)` überschrieben werden.

Listing 2: Eigener Observer

```
1  // subject
2  public interface Subject {
3     void registerObserver(Observer o);
4     void removeObserver(Observer o);
5     void notifyObservers();
6  }
7
8  // Concrete Subject
9  public class MySubject implements Subject {
10    private List<Observer> observers;
11    private float myVal;
12
13    public MySubject() {
14       this.observers = new ArrayList<>();
15    }
16
17    public float getVal() {
18       return myVal;
19    }
20
21    public void setVal(float newVal) {
22       this.myVal = newVal;
23
24       // value has changed, notify
25       notifyObservers();
26    }
27
28    public void registerObserver(Observer o) {
29       observers.add(o);
30    }
31
32    public void removeObserver(Observer o) {
33       int i = observers.indexOf(o);
34       if (i >= 0) {
35          observers.remove(i);
36       }
37    }
38
39    public void notifyObservers() {
40       for (Observer o : observers) {
41          observer.update(myVal);
42       }
43    }
44 }
45
46
47
48
49
```

```
50 // Observer
51 public interface Observer {
52    protected Subject subject;
53
54    void update();
55 }
56
57 // Concrete Observer
58 public class MyObserver implements Observer {
59    private float myVal;
60
61    public MyObserver(Subject subject) {
62       this.subject = subject;
63       // register in subject
64       this.subject.registerObserver(this);
65    }
66
67    public void update(float val) {
68       // value in observable has changed
69       this.myVal = val;
70    }
71 }
72
73 // Client
74 Subject subject = new MySubject();
75
76 new MyObserverA(subject);
77 new MyObserverB(subject);
78 new MyObserverC(subject);
79
80 subject.setVal(55); // observers get notified
```

## 8.13. State

Das State Pattern ermöglicht einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es scheint dann so, als hätte das Objekt seine Klasse gewechselt.

- Mit dem State Pattern kann man zustandsabhängiges Verhalten umsetzen, ohne überall Fallunterscheidungen implementieren zu müssen
- Der Client kennt die Zustände oft nicht (nur der Context: Es ist Sache des Kontext seinen Zustand zu kontrollieren)
- Der Context delegiert sein Verhalten an die States, aus welchen er zusammengesetzt ist.
- Das State Pattern ist dem Strategy Pattern sehr ähnlich (**gleiches Klassendiagram, andere Absichten**), mit dem Unterschied, dass der Client kaum etwas über die States weiss und der Context die volle Kontrolle über das austauschen der States übernimmt. Beim Strategy Pattern sind die konkreten Instanzen hingegen meist fix programiert. (obschon es die Möglichkeit bietet, diese zur Laufzeit zu ändern)

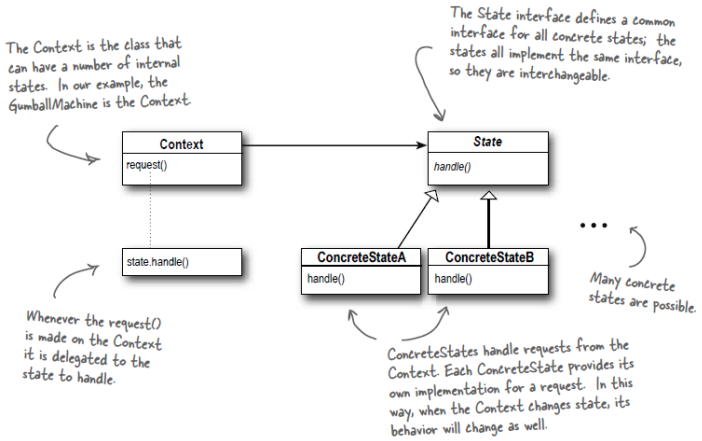


Abbildung 34: State

### 8.13.1. Implementierung

- Sind die Zustandsübergänge **genau festgelegt**, ist die Logik dafür meist im Kontext abgelegt
- Sind die Zustandsübergänge **dynamischer**, ist die Logik meist in den Zuständen selber hinterlegt. (Mit dem Nachteil von Abhängikeiten zwischen den Zuständen)

### 8.13.2. Vorgehen

1. Implementiere für jeden Zustand eine eignen Klasse
2. Alle Zustandsklassen haben eine gemeinsame Schnittstelle
3. Die Ausgangsklasse delegiert die zustandsabhängigen Methoden an diese Schnittstelle
4. Jeder Zustandswechsel tauscht das aktuelle Zustandsobjekt aus

```
1  // state
2  public interface State {
3     public void doAction(Context context);
4  }
5
6  // Concrete States
7  public class StartState implements State {
8     public void doAction(Context context) {
9        System.out.println("Start the game");
10       context.setState(this);
11    }
12    public String toString() {
13       return "Game is running";
14    }
15 }
16
17 public class StopState implements State {
18    public void doAction(Context context) {
19       System.out.println("Stop the game");
20       context.setState(this);
21    }
22    public String toString() {
23       return "Game has stopped";
24    }
25 }
26
27 // Context
28 public class Context {
29    private State state;
30
31    public Context() {
32       state = null;
33    }
34
35    public void setState(State state) {
36       this.state = state;
37    }
38
39    public State getState() {
40       return state;
41    }
42 }
43
44 // Client
45 Context context = new Context(); // context can change its state
46 State startState = new StartState();
47 startState.doAction(context);
48
49 State stopState = new StopState();
50 stopState.doAction(context);
```

## 8.14. Strategy

Das Strategy Muster definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy Muster ermöglicht es, den Algorithmus unabhängig von Clients die ihn einsetzen, variieren zu lassen.

- Das Strategy Pattern ist die flexible Alternative zur Vererbung
- Programmiere auf eine Schnittstelle und nicht auf eine Implementierung. Man nutzt dabei Polymorphismus, da der tatsächliche Objekttyp erst zur Laufzeit festgelegt wird.
- Identifiziere die Aspekte die sich ändern können und trenne sie von denen ab, die konstant bleiben.
- Komposition sollte der Vererbung vorgezogen werden, damit das Verhalten zur Laufzeit angepasst werden kann.
- Gelten viele Methoden in der Parent Klasse nur für einzelnene Child Klassen, sollte spezifisches Verhalten besser in Interfaces ausgelagert werden. Danach erstellt man mehrere Verhaltensklassen und zieht das Interface wieder in den Parent.
- Das Strategy Pattern ermöglicht einem Objekt, sein Verhalten zu ändern, wenn sich irgendein Zustand ändert

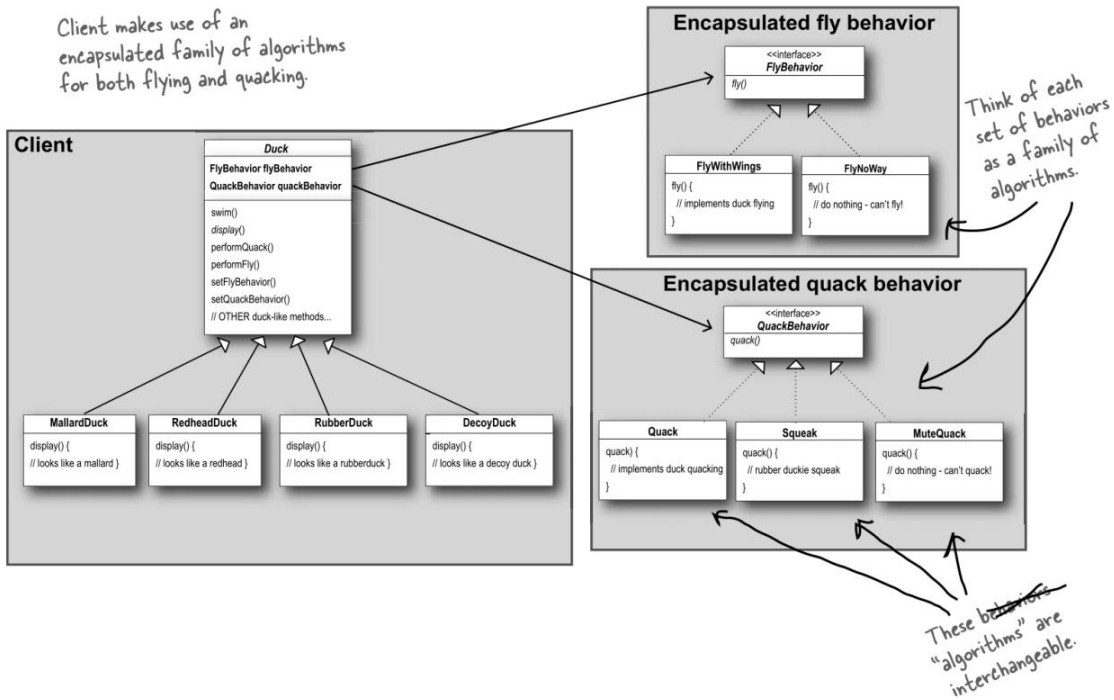


Abbildung 35: Strategy Pattern

### 8.14.1. Implementierung

```
// Strategy
public interface Strategy {
   public int doOperation(int one, int another);
}

// Concrete strategies
public class OperationAdd implements Strategy {
   public int doOperation(int one, int another) {
      return one + another;
   }
}

public class OperationSubstract implements Strategy {
   public int doOperation(int one, int another) {
      return one - another;
   }
}

public class OperationMultiply implements Strategy {
   public int doOperation(int one, int another) {
      return one * another;
   }
}

// Client hold a behaviour (context)
public abstract class Calculator {
   private Strategy strategy;
   public Calculator(Strategy strategy) {
      this.strategy = strategy;
   }

   public int getResult(int num1, int num2) {
      return strategy.doOperation(one, another);
   }
}

public class Multiplier extends Calculator {
   public Multiplier() {
      super(new OperationMultiply());
   }
}

public class Adder extends Calculator {
   public Adder() {
      super(new OperationAdd());
   }
}

// Client
Calculator c1 = new Multiplier();
c1.getResult(2,3) // 6

c1 = new Adder();
c1.getResult(2,3); // 5
```

## 8.15. Template Method

Das Template Method Pattern definiert in einer Methode das **Gerüst eines Algorithmus** und überlässt einige Schritte den Unterklassen. Template Method erlaubt Unterklassen, bestimmte Schritte des Algorithmus neu zu definieren, ohne die Struktur des Algorithmus zu ändern.

- **Hook Methoden** sind Methoden, die in der abstrakten Klasse nicht tun oder nur ein Default Verhalten anbieten, aber in den Unterklassen überschrieben werden können.
- Um zu verhindern, dass Unterklassen den Algorithmus in der Template Methode ändern, wird die Template Methode als `final` deklariert.
- Eine generelle Methode im Parent ruft dann alle Hoocks auf. Die gemeinsamen Teile werden in einer abstrakten Klasse implementiert und in den Subklassen bei Bedarf verfeinert.
- Die abstrakte Klasse besteht aus unveränderlichen Methoden welche zusätzlich abstrakte Methoden (Hooks) aufruft, welche in der Subklasse überschrieben werden müssen.
- Das Pattern verwendet das **Hollywood Prinzip**: Dont call us, we call you (Inversion of Control)
- Das Template Method Pattern erlaubt es viel Code wieder zu verwenden.
- Das Factory Method Pattern ist eine Spezialisierung des Template Method Pattern.

### 8.15.1. Vorgehen

1. Definiere die Struktur des Algorithmus (Skelett) in der Basisklasse. Dieses Template Method ruft andere Methoden für die variierenden Details auf
2. Diese anderen Methoden (Hook method)werden in den Unterklassen entsprechend überschrieben.

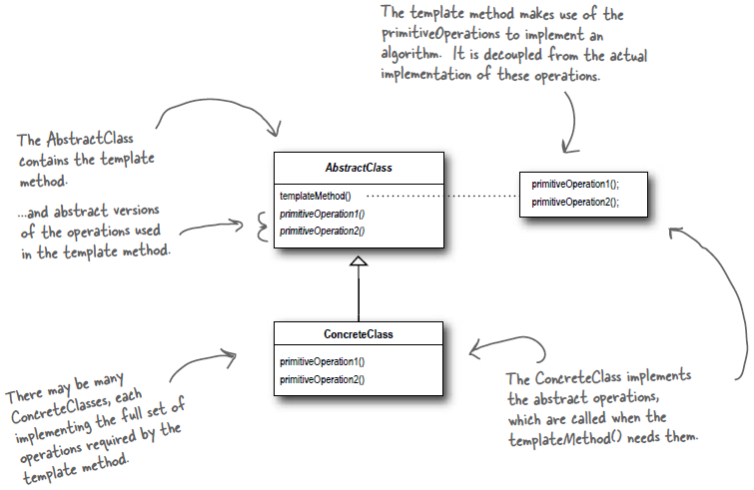


Abbildung 36: Template Method

### 8.15.2. Implementierung

```java
public abstract class Game {
   // Hooks
   abstract void initialize();
   abstract void startPlay();
   abstract void endPlay();

   //template method
   public final void play(){

      //initialize the game
      initialize();

      //start game
      startPlay();

      //end game
      endPlay();
   }
}

public class Football extends Game {
   @Override
   void endPlay() {
      System.out.println("Football Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Football Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
   System.out.println("Football Game Started. Enjoy the game!");
   }
}
```

## 8.16. Iterator

Das Iterator Pattern bietet eine Möglichkeit, auf die Elemente in einem Aggregat-Object (Container) sequenziell zuzugreifen, ohne die zu Grunde liegende Implementierung zu offenbaren.

- Iteratoren sind niemals sortiert, da die zugrundeliegende Struktur ein `Set<T>` sein könnte.
- Der Iteraotor bietet eine Möglichkeit, eine Sammlung von Objekten zu durchqueren, ohne die Implementierung der Sammlung zu offenbaren.

### 8.16.1. Variaten

Man unterscheidet zwischen externen und internen Iteratoren.

**Externer Iterator** Ein externer Iterator wird vom Client mit `next()` gesteuert, und erlaubt somit einen viel grösseren Grad an Flexibilität.

**Interner Iterator** Ein interner Iterator wird vom Iterator selbst gestuert und ist somit weniger flexibler, dafür einfacher zu gebrauchen

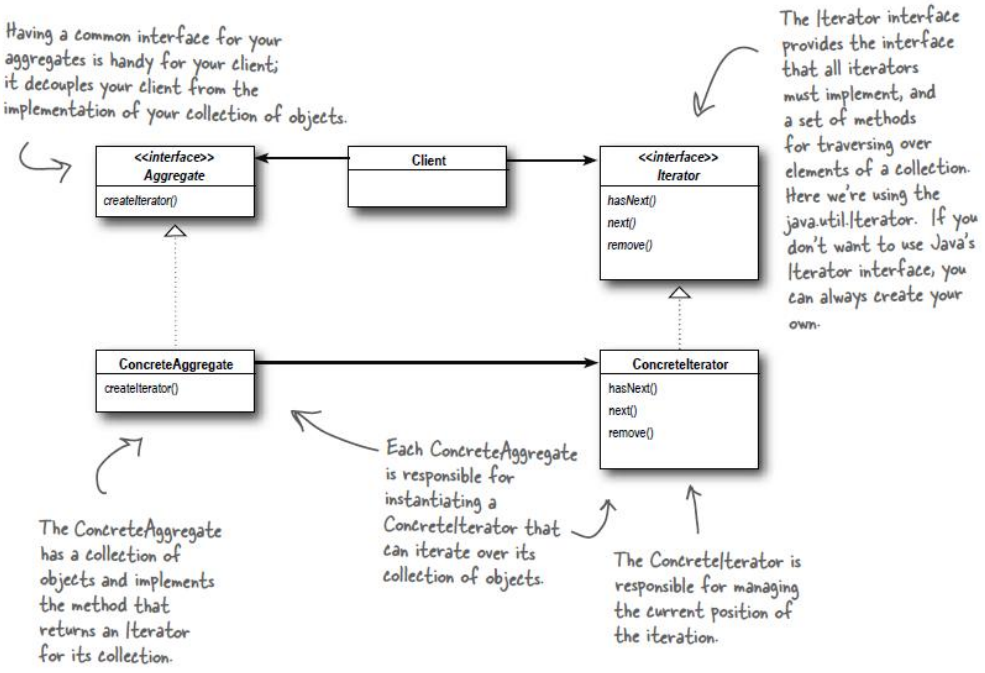


Abbildung 37: Iterator

### 8.16.2. Implementierung

Listing 3: Iterator

```
// Iterator
public interface Iterator {
   public boolean hasNext();
   public Object next();
}

// Aggregate
public interface Container {
   public Iterator getIterator();
}

public class NameRepository implements Container {
   public String names[] = {"Joe", "Lue", "Dave"};

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }

   // Innerclass
   private class NameIterator implements Interator {
      int index;

      @Override
      public boolean hasNext() {
         if (index < names.length) {
            return true;
         }
         return false,
      }

      @Override
      public Object next() {
         if (this.hasNext()) {
            return names[index++];
         }
         return null;
      }

   }

}
```

## 8.17. MVC: Model View Controler

Das MVC Pattern ist ein zusammengesetztes Muster, bestehend aus Observer, Strategy und Composite.

**Model** Die Daten und die Operationen auf den Daten. Es implementiert das Observer Pattern, um interessierte Objekte (Views, Controller) über Zustandsänderungen zu benachrichtigen

**View** Darstellung der Daten. Implementiert das Composite Pattern. Wenn der Controller eine Aktualisierung der Views haben möchte, muss er es einfach der obersten View Komponenten sagen.

**Controller** Nimmt Benutzereingaben entgegen und delegiert diese an das Model. Verwendete das Strategy Pattern damit die Controller in der View austauschbar sind.

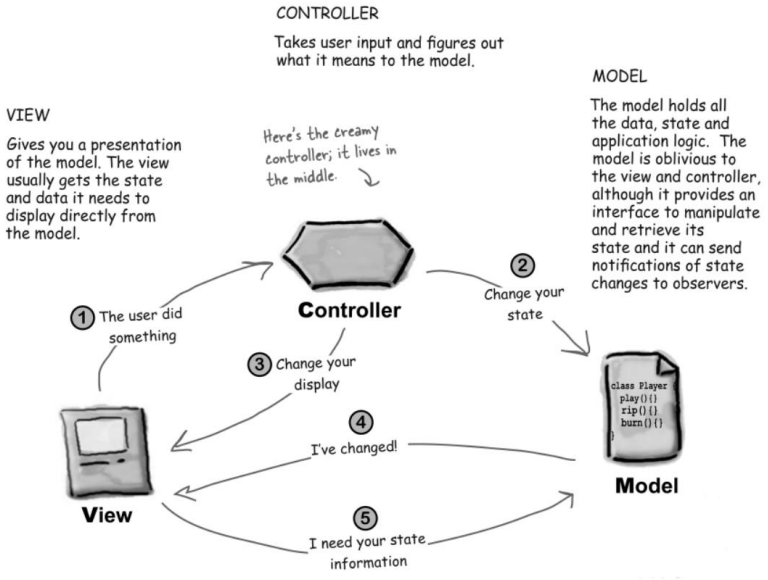


Abbildung 38: MVC: Model View Controller

### 8.17.1. Implementierung

```java
// Model
public class Student {
   private String rollNo;
   private String name;

   public String getRollNo() {
      return rollNo;
   }

   public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}

// View
public class StudentView {
   public void printStudentDetails(String studentName, String studentRollNo){
      System.out.println("Student: ");
      System.out.println("Name: " + studentName);
      System.out.println("Roll No: " + studentRollNo);
   }
}

// Controller
public class StudentController {
   private Student model;
   private StudentView view;

   public StudentController(Student model, StudentView view){
      this.model = model;
      this.view = view;
   }

   public void setStudentName(String name){
      model.setName(name);
   }

   public String getStudentName(){
      return model.getName();
   }

   public void setStudentRollNo(String rollNo){
      model.setRollNo(rollNo);
   }

   public String getStudentRollNo(){
      return model.getRollNo();
   }

   public void updateView(){
      view.printStudentDetails(model.getName(), model.getRollNo());
```

```
60      }
61  }
62
63  // Client
64  Student model = retriveStudentFromDatabase();
65  StudentView view = new StudentView();
66  StudentController controller = new StudentController(model, view);
67
68  controller.updateView();
69
70  //update model data
71  controller.setStudentName("John");
72  controller.updateView();
```

# 9. GRASP

Die General Responsibility Assignment Software Patterns (GRASP) Pattern beschreiben eine Menge von Entwurfsmuster, welche die Zuständigkeiten einer Klasse definieren. Die GRASP Pattern entstammen von Craig Larman.

## 9.1. Information Expert

Der Information Expert ist jene Klasse, welche die notwendigen Informationen besitzt, um eine Aufgabe zu erledigen. Eine Klasse muss alle Operationen bereitstellen, die mit ihr gemacht werden können. Diese Funktionsweise ist auch als **"Do it Myself"** Strategie oder **Kapselung** bekannt. Sie führt zu hoher Kohäsion und geringer Kopplung.

- Gutes Beispiel: Klasse `Kreis`, mit dem Property `radius` und der Methode `berechneFlaeche()`
- Schlechtes Beispiel: Klasse `BerechneFlaeche` mit einer Methode die eine geometrische Form entgegen nimmt.

## 9.2. Creator

Das Creator Prinzip legt fest, wer eine Instanz einer Klasse erstellt. Neue Objekte von der Klasse B sollten von A erzeugt werde, wenn:

- A eine Aggregation von B ist (schwache Teil-Ganzes Beziehung)
- A, B-Instanzen enthält
- A, B-Objekte erfasst
- A, B-Objekte mit starker Kopplung verwendet
- A die Initialisierungsdaten für B hat

## 9.3. Controller

Der Controller definiert, wer bestimmte Systemereignisse verarbeitet. Man unterscheidet zwischen Use-Case Controller und Facade Controller:

**Use Case Controller** Behandelt alle Ereignisse für einen Use Case

**Facade Controller** Nimmt alle Meldungen entgegen und leitet sie an den richtigen Service weiter. (MessageHandler)

## 9.4. High Cohesion

Ziel von hoher Kohäsion ist es, den inneren Zusammenhalt einer Klasse hoch zu halten. Dass heisst, dass **eine Klasse für genau eine Aufgabe zuständig** ist.

## 9.5. Low Coupling

Ziel von geringer Kopplung ist es, so wenig Abhängigkeiten wie möglich, zu einer anderen Klasse zu haben. Dies erlaubt es, Anpassungen leichter durchzuführen, die Klassenfunktion leichter zu verstehen, einfacher zu testen und die Klasse besser Wiederverwenden zu könnnen.

Unter Kopplung versteht man folgende Dinge:

- Vererbung
- Implementierung von Interfaces
- Halten von Instanzvariablen
- Benutzen von externen Methoden

## 9.6. Polymorphismus

Polymorphismus erlaubt es, das Verhalten abhängig vom Typ zu ändern. Somit können Fallunterscheidungen vermieden werden. (Siehe 8.14 Strategy Pattern)

## 9.7. Pure Fabrication

Eine Pure Fabrication Klasse (Reine Erfindung) gehört nicht zur Problem Domäne. Sie stellt oft Technologie Methoden zur Verfügung. Sie implemtentiert reines Verhalten und hat somit keinen Zustand.

## 9.8. Indirection

Indirection kann verwendet werden um geringe Kopplung zu erreichen. Bei der Indirection baut man einen Vermittler zwischen zwei Komponenten ein. (z.B Client / Server) Beispiel dafür ist der Controller, der zwischen Model und View vermittelt.

## 9.9. Protected Variations

Interfaces sollen immer verschiedene konkrete Implementierungen verstecken. Dadurch wird das System vor den Auswirkung eines Wechsels der Implementierung geschützt.

# 10. Software Architektur

Grosse Software Projekte werden in Schichten und Tiers unterteilt, damit Zuständigkeiten klarer werden, zusammengehörende Objekte gekapselt werden und Funktionalität explizit wiederverwendet wird. (Single Responsibility)

## 10.1. Grundlegend

- Der Kunde ist oben der Service ist unten!
- Man greift immer von den oberen Schichten auf die Unteren zu. Das bedeutet dass die **unteren Schichten einen höheren Grad an Wiederverwendbarkeit** aufweisen. Die oberen Schichten sind von den unteren Abhängig.
- Oben ist man mehr Applikationsspezifischer
- Bei der Vererbung, sollte die Parent Klassen ebenfalls unten gezeichnet werden, da man somit konsistenter mit der Schichten Architektur (unten genereller) ist. Gleiches gilt für Klassendiagramme (n-Klasse oben bei 1:n Beziehung)
- In einem Klassen/Schichten Diagramm sollte nur die wichtigsten Klassen (meist das Interface gegen die oberen Schichten) gezeichnet werden
- Ein Architekturdiagramm nützt nur etwas wenn es kommunizierbar ist. Das Diagramm sollte deshalb maximal auf einem A3 Blatt platz finden.
- Die Namen der Schichten sind nicht standardisiert und können variieren.
- Verbreitet sind 3 und 6 Schichten Modelle.

## 10.2. Motivation

- Schichten erlauben **bessere Wartbarkeit, Testbarkeit und Arbeitsaufteilung**
- Hat man saubere Schichten, können die einzelnen Schichten ohne weiteres ausgetauscht werden.
- Um eine hohe Kohäsion zu erreichen, sollten zusammengehörige Klassen in Packages gruppiert werden. Zusätzlich sollte eine Klasse nur für eine Aufgabe zuständig sein. (Single Responsibility)

## 10.3. Einflüsse

Die Architektur wird von verschiedenen Punkten beeinflusst. Den grössten Einfluss haben die **nicht-funktionalen Anforderungen**!

- Deployment: Web? Mobile?
- Schwerpunkt: Analytisch? Grafik? Datenhaltung? Datentransfer? Game?
- Performance
- Security
- Erweiterbarkeit
- Usability (enge Benutzerführung)

## 10.4. Schichten

Horizontale Schichten ("Wie der Code strukturiert ist"). Sie zeigen auf, wie die hierarchischen Abhängigkeiten sind. Die Aufrufe zwischen den Schichten sind synchron.

**UI** Zuoberst ist der **Presentation Layer**. Er zeigt die Daten an. (HTML, Reports, Windows)

**Application Logic** Das GUI greift auf die **Use Case Controller** im Application Layer zu. Die Use Case Controller nehmen die GUI Anfragen entgegen und leiten sie an die unteren Schichten weiter. (Workflow)

**DataAccess** Stellen funktionen für den Datenbank Zugriff zur Verfügung.

**Business Domain** Hier sind die Services, welche die Request aus dem Application Layer verarbeiten.

**Business Services** Beinhaltet Low Level Services für die Business Domain. Diese Klassen werden von mehreren Business Klassen verwendet. (z.B CurrencyConverter)

**Libraries / Foundation** Zuunterst liegen die **Util Klassen** für Security, I/O, DB, Math Libraries, etc.

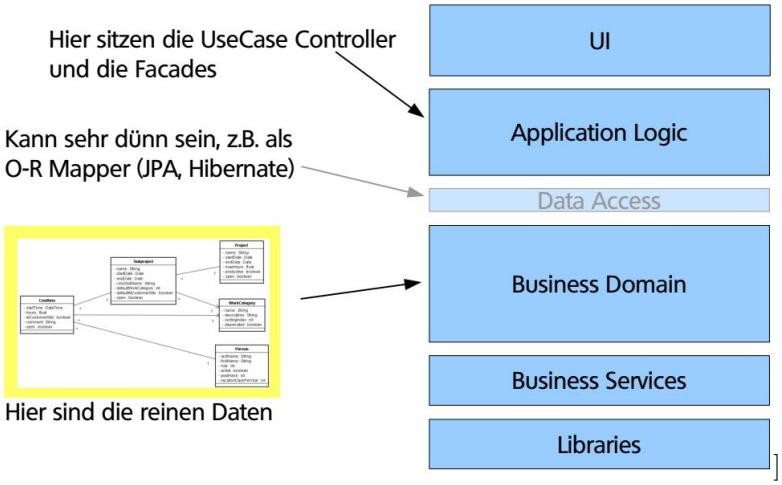


Abbildung 39: Sechs-Schichten-Modell

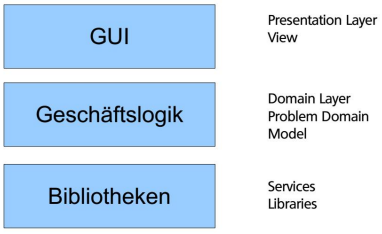


Abbildung 40: Drei-Schichten-Modell

### 10.4.1. Isolieren

Gemeinsam verwendeter Code sollte in einer tieferen Schicht isoliert werden. Wird ein Code Fragment sehr oft aufgerufen, ist man unter Umständen gezwungen, die Aufrufe in die tiefere Schicht weiterzuleiten.

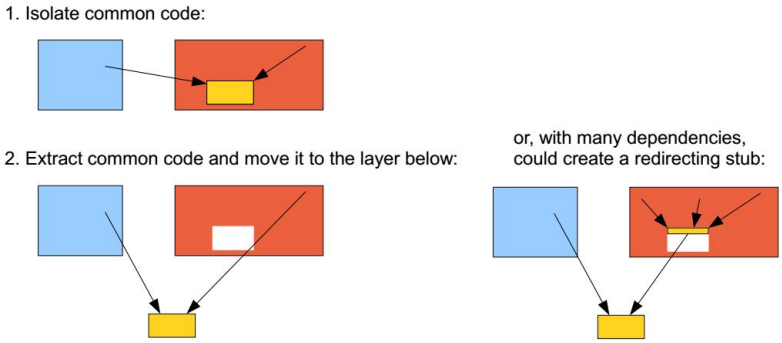


Abbildung 41: Isolierung von gemeinsamen Code

### 10.4.2. Regeln

1. Aufrufe von oben auf die nächste darunterliegende Schicht sind immer OK
2. Aufrufe nach unten, die eine Schicht überhüpfen sind manchmal auch OK
3. Aufrufe innerhalb einer Schicht und Partition sind OK, sollten aber minimiert werden
4. Aufrufe in einer Schicht quer zu einer anderen Partition sollten dringend vermieden werden
5. Aufrufe NIE von unten nach oben, ausser callbacks (z.B. Observer pattern)

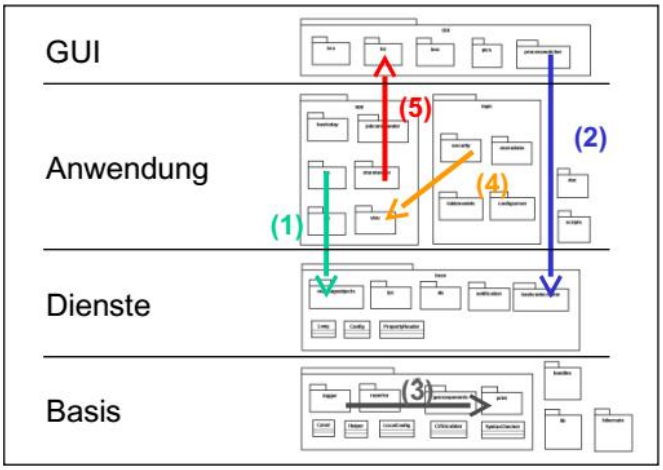


Abbildung 42: Regeln für Abhängigkeiten

## 10.5. Partitionen

Eine Partition ist eine vertikale Unterteilung innerhalb einer Schicht. Die Kopplung zwischen den Partitionen sollte möglichst klein gehalten sein. Abhängigkeiten zwischen Klassen in zwei Partition derselben Schicht sollten vermieden werden. Falls die Abhängigkeit unumgänglich ist, sollte gemeinsamer Code isoliert werden und in eine tiefere Schicht verschoben werden.

## 10.6. Tiers

Vertikale Ebenen von gleichberechtigten Laufzeitumgebungen ("Wo was läuft"). (User → Server → DB). Die Interfaces zwischen den Tiers sind asnychron. (d.h potentiell langsamer und somit teurer). In einem System werden n Schichten üblicherweise auf $n - x$ Tiers abgebildet (selten $n + x$).

## 10.7. Testing

Das Unit Testing geht in der untersten Schicht am besten, da keine Abhängigkeiten bestehen. Je weiter hoch man geht, desto mehr Abhängigkeiten müssen weggemockt werden, um eine Schicht isoliert zu testen.

## 10.8. Kohäsion und Kopplung

Bei den Schichten sollte eine hohe Kohäsion (Guter Zusammenhalt innerhalb der Klasse) und tiefen Kopplung(Minimierte Abhängigkeiten von andere Klassen) angestrebt werden. Mit einer tiefen Kopplung können die Komponenten einfacher ausgetauscht werden.

- **Kopplung:** Wie stark ist eine Klasse Abhängig von vielen anderen Klassen. Aufrufe von einer Klasse zu anderen, von einem Package zum anderen
- **Kohäsion:** Zusammenhalt innerhalb einer Klasse. Ähnliche Dinge sollten in einer Klasse gruppiert werden. Kohäsion soll gegeben sein, sonst gehören die Dinge nicht ein eine Klasse/Package.

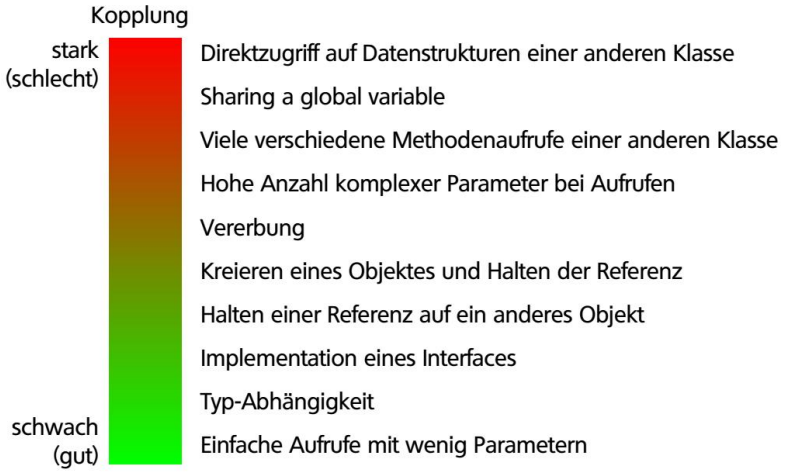


Abbildung 43: Arten von Kopplung

# 11. Unified Process

- Beim veralteten Wasserfallmodell (Hermes) war das Problem, dass die Fehler von vorherigen Schritt mitgenommen wurden.
- UP ist nur zu Beginn eine Art Wasserfall (aber auch da iterativ), nach End of Elaboration kann der Porzess voll agil sein.
- Der Unified Process ist für **kleinere und mittlere Projekte nicht geeignet**, da er 150 Aktivitäten, 40 Rollen und 80 Major Products definiert.
- Analogie zum "Weitersagen-Spiel". Bei so vielen Rollen, kommt die Nachricht des Kunden vollkommen verzerrt beim Programmierer an. $p^{(n-1)}$ (p= angenommene Übertragungsqualität in Prozent)
- Jede Iteration hat ein lauffähiges Produkt zum Ziel. Immer ein Stück abliefern: Demo → Freude → wichtiges Feedback
- Der **wichtigste Meilenstein ist End of Elaboration**
- Die Arbeit kann erst über mehrere Teams aufgeteilt werden, wenn der Kunden verstanden wurde (Requirements, Keine Fragezeichen), die Architektur steht (Prototypen), UI steht (Wireframes, Prototypen), Zeitschätzung steht (Detailplanung), und die Tool Chain (IDE, Versionskontrolle, Build Server, Deployment, Unit Testing, Workflow Tools, Wiki) steht. (Zum Zeitpunkt End of Elaboration)

## 11.1. Phasen

Unified Processes werden in 4 Phasen eingeteilt. Die einzelnen Tätigkeiten können zwischen den Phasen fliessend übergehen und überlappen.

1. **Inception**: Business Modelling, Requirements, Vision, Eckwerte (< 5%)
2. **Elaboration**: Analysis und Design (20-30%). Ab diesem Punkt ist ein Abbruch noch relativ günstig möglich.
   - Technische Risiken beseitigt
   - Prototyp erstellt
   - Die Anforderung vollständig definiert (100% Brief Use Cases und ca. 80% fully dressed)
   - Detailierte Planung und Kostenschätzung
3. **Construction**: Implementation, Test, Deployment (viel Personal)
4. **Transition**: Data Migration

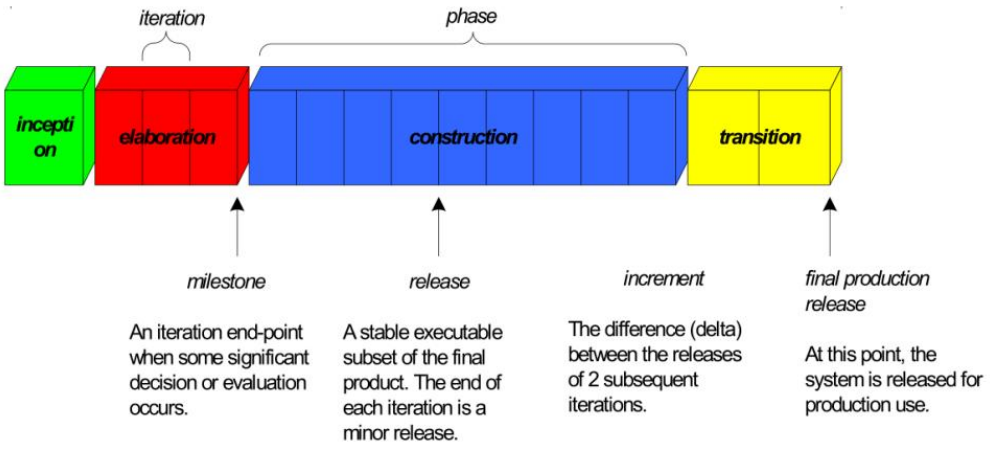


Abbildung 44: Unified Process

# 12. SCRUM

**Merke 12.1: Goldene Regel**

Enwickler schätzen, Kunde priorisiert


Abbildung 45: SCRUM

## 12.1. Terminologie

**User Stories**

Jede Story ist entweder Feature oder Bug. Eine User Story besteht aus vier Teilen:

1. Story Nummer
2. Aufgabenbeschreibung: As a [user/role] I want to [goal] because [reason/motivation]
3. Akzeptanzkriterien
4. Aufwandschätzung (Story Points)
5. Priorität

**Epic** Epics sind Aufgaben auf Feature-Ebene, die viele User Storys umfassen.

**Teamspeed/Velocity**

Der Teamspeed passt sich von Iteration zu Iteration an. Er pendelt sich aber meist nach 3-4 Iterationen ein. Der Aufwand kann spielerisch mit Schätzpoker geschätzt werden.

**Story Points**

Story Points sind eine Einheit, die den Aufwand für das Umsetzen einer User Story beschreiben. Es geht dabei nicht nur um die Zeit sondern auch um die Komplexität.

**Product Backlog** Nach Priorität sortierte Liste aus User Stories.

**Backlog Board**

Das Backlog Board ist in zwei Bereiche unterteilt. Die Story Area beinhaltet die User Stories und die Constraint Area die Vorgaben. Oft werden Post Its (Story Cards) in Gelb (Anforderung) und Rot (Bug) verwendet.

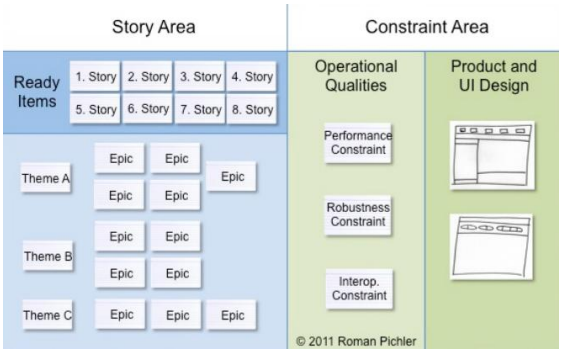


Abbildung 46: Backlog Board

**Sprint**

Sprints sind iterative Aufteilungen des Workloads in fixe Zeitintervalle. Ein Sprint dauert traditionell 2-3 Wochen. Alle Sprints dauern gleich lange. Das Intervall ist fix. Wird eine Story im Sprint nicht fertig, kommt es in den nächsten Sprint. (auch 98% fertige Stories)

**Sprint Backlog**

So viel Arbeit wie das Team in einem Spring bewältigen kann, kommt in den Sprint Backlog. Pro Sprint wird nur das bearbeitet, was im Sprint Backlog liegt. (Niemals weitere Anforderungen aus dem Product Backlog).

**Scrum Task Board**

Das Task Board visualisiert den Status der einzelnen Subtasks einer User Story im Sprint Backlog. (TODO, In Process, To Verify, Done)

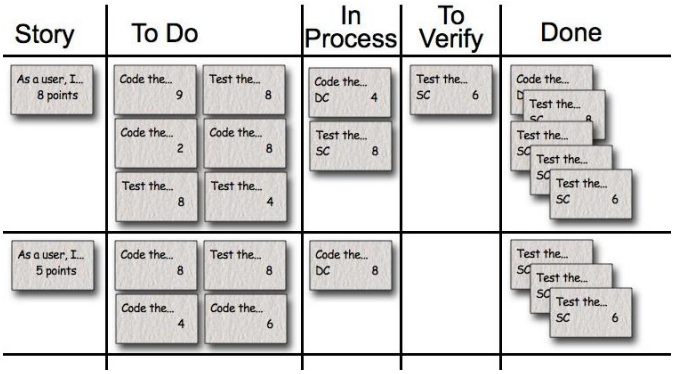


Abbildung 47: SCRUM Task Board

## 12.2. Ereignisse

**Sprint Planning**
Beim Sprint Planning wird so viel Arbeit, wie ein Team bewältigen kann in den Sprint Backlog verschoben. Dies geschieht immer am Anfang eines Sprints.

**Daily Standup Meeting** Beim Daily Standup Meeting bekommt jeder Enwickler max. 2min Zeit. Es wird gesagt, an was man gerade arbeitet und wo eventuelle Probleme liegen. Es geht darum, zu wissen, was die anderen machen.

**Sprint Review**
Am Schluss jedes Sprint gibt es eine Demo für den Kunden/Product Owner. Der Product Owner nimmt alle Features ab und es gibt eine Diskusion, was wirklich abgeschlossen ist. Der neue Speed wird hier definiert.

**Sprint Retrospective** Regelmässig am Schluss eines Sprints gibt es ein kurzes Meeting, um das Erledigte zu reflektieren. Was ist gut gelaufen? Was hat Probleme bereitet? Was kann man verbessern?

## 12.3. Rollen

**Product Owner** Ist verantwortlich, dass das Richtige gemacht wird. Er ist oft Teil der Firma des Auftraggebers. Oft sind das Produktmanager der Firma. Er kennt das Business. Ideal ist es, wenn der Product Owner im gleichen Büro wie das Enwicklungs-Team ist und ständig ansprechbar ist.

**Scrum Master** Es gibt einen SCRUM Master pro ca. 5 SCRUM Teams. Der SCRUM Master kennt sich sehr gut mit SCRUM aus und hilft bei allfälligen Schwierigekeiten.

**Enwickler Team** Die Entwickler setzen die Aufgaben gemäss ihren Kenntnissen und Fähigkeiten um. (Programmieren, Testen, User Stories Schreiben, User Expreience, UI Design, Architektur und Design, Netzwerk und Server)

## 12.4. Vorgehen

### Requirements Analysis

1. Product Backlog erstellen
2. Backlog Grooming: Das Team und der Product Owner diskutieren zusammen, bis alle Anforderungen verstanden sind
3. Das Team definiert die Kosten pro User Story (Story Points). Geht eine Story länger wie ein Sprint, muss sie aufgeteilt werden.
4. Der Product Owner priorisiert den Product Backlog nach seinen Prioritäten
5. Teamspeed definieren: Speed = Anz. fertig gestellter Story Points pro Sprint - Reserveren (Krankheit, Militär, Feiertage)

### Sprint Planen

1. Sprints einplanen: Der Kunde wünscht oft ein Datum, wann das Projekt abgeschlossen ist. Da SCRUM ein iterativer Prozess ist und das Kundenbudget nicht unendlich ist, ist dies schwierig zu definieren. Grundsätzlich kann man aber die geschätze Zeitdauer aller User Stories durch den Teamspeed teilen.
2. Die am höchsten priorisierten User Stories in den Sprint Backlog verschieben.
3. Nach dem Sprint soll immer eine lauffähige Version zur Verfügung stehen.
4. Stories die nach einem Sprint nicht abgenommen oder abgeschlossen sind, werden in den nächsten Sprint verschoben. Ebenfalls muss der Zeitaufwand neu geschätzt werden.
5. Erledigte User Stories werden in den Product Backlog zurück verschoben und als erledigt markiert.
6. Danach wird der nächste Sprint geplant. Die Prioritäten werden unter Umständen neu gesetzt und nicht abgeschlossene/abgenommene User Stories kommen in den neuen Sprint. Bei nicht abgeschlossenen User Stories muss der Zeitaufwand neu geschätzt werden. Ebenfalls wird der Aufwand dem neuen Teamspeed angepasst. Es ist dem Team auch frei, den Teamspeed selbständig hochzusetzen.
7. Die vorgehenden Schritte werden nun iterativ wiederholt.

# 13. Projektmanagement

## 13.1. Anforderungen

### Die vier Projekt-Variablen

1. Kosten / Aufwand / Personen (einfach zu verstehen/messen)
2. Zeit
3. Funktionalität (schwieriger zu verstehen/messen)
4. Qualität (sehr schwierig zu verstehen/messen)

### Kunde verstehen

Im Idealfall ist der Kunde im Team und bekommt die Änderungen direkt mit. Worte können missverstanden oder anders interpretiert werden. Hilfreich sind verständlich geschriebene Anforderungen, Use Cases, nicht funktionale Anforderungen oder Prototype und grafische Entwürfe. Es hilft auch, mit dem Team vor Ort den Businessprozess anzuschauen.

### So früh wie möglich so formal wie möglich

Je früher man formal wird, desto mehr Zeit spart man sich später. (UML, halb formale Text: Use Cases, Risikolisten, Tests). Bilder können Projekte retten.

### Die hohe Kunst: Sichtbar machen

Software ist für Nicht-Programmierer unsichtbar. Es gilt das unsichtbare, also den Fortschritt, die Ideen und Konzepte (Architektur, Design, Schnittstellen) mit Diagrammen (Domainmodell, Activitydiagramme, Statediagramme ,Contextdiagramme, Sequnezdiagramme, Package und Deployment Diagramme) sichtbar zu machen. Es gibt noch weitere Variaten um den Projektverlauf zu visualisieren:

- Burn-down chart
- Bug count
- Time to fix bugs
- Build breaks
- Metriken (Klassenlänge, Verschachtlungen)

## 13.2. Scope Keeper

Der ProjektManager muss den Funktionsumfang des Projekts im Zaum halten. Dies ist der **wichtigste Punkt**! Wenn man dies nicht umsetzt, dann kann man noch so ein tolles Team haben, die sauberste Architektur entworfen haben, mit den besten Werkzeugen und Techniken arbeiten: Man kann das Projekt rauchen.

- 2% an unerwarteten Erweiterungen (Scope Creep) pro Monat sind normal. Folglich können 20 Monate lange Projekte um 40% umfangreicher werden.
- Plane kein Projekt über mehr als neun Monate. Ansonsten sollte es aufgeteilt und auch so verkauft werden.

## 13.3. Communication is Key

**Software ist Kommunikations-intensiv**

- Das Team sollte an einem Ort sein und die selbe Sprache sprechen
- Teamwand einrichten. (SCRUM Board)
- Viel und regelmässig austauschen (Daily Standup Meeting)

## 13.4. Kosten

**Entfernung ist teuer**
Im Idealfall arbeitet das ganze Team im selben Gebäude und Stockwerk. Ansonsten gibt es höhere Kosten

- +10%: Alle arbeiten im selben Gebäude, aber nicht mehr auf einem Stock
- +10%: Alle arbeiten in derselben Ortschaft, aber nicht mehr im gleichen Gebäude
- +10%: Alle arbeiten im gleichen Land, ...
- +10%: Alle arbeiten in derselben Zeitzone
- +10%: nicht mehr gleiche Zeitzone, Zeitverschiebung mehr als 4 Std.
- +20%: Nicht alle haben dieselbe Muttersprache (= Kommunikationssprache)

**Übergaben sind teuer**
Wenn man eine Arbeit jemand anders übergeben muss, dann kostet das einen erheblichen Aufwand und es ist mit Kommunikationsfehlern zu rechnen. Um diesem Problem entgegen zu wirken, kann man wichtigen Rollen auf zwei Personen aufteilen (Pilot/Copilot). Ebenfalls sollte der Software Architekt bereits früh im Projekt involviert sein. Auch hier begünstigt die örtliche Nähe den Informationsaustausch.

## 13.5. Praktische Tipps

**Nichts vergessen dank Listen**

- Tasks für das Entwicklungsteam in Ticketing Tool erfassen
- ToDo Liste führen über Dinge, die aus dem Weg geräumt werden müssen
- Maximal drei Prioritäten definieren (1. dringen, 2. wichtig, 3. machen wenn Zeit)

**Immer iterativ vorgehen**
Agile Methoden haben sich bewährt weil:

- Kunde sieht immer wieder Resultate und kann Feedback geben
- Das Team hat Zwischenresultate, die ein Gefühl von “wir haben was geschafft!“ vermitteln.
- Es ist einfacher, ein System schrittweise auf- und auszubauen (mit Refactoring) als eine “Big Bang” Integration gegen Schluss zu machen.

- Fehler und Fehleinschätzungen („das will der Kunde so“) werden früher entdeckt.
- Auch Requirements iterativ erfassen (mit OOA zwischendrin)
- Sprints: 2-3 Wochen Dauer, Kunden-Demos: alle 2-3 Monate

**Inspect - Adapt**
Ein Projekt sollte als eine Reise betrachtet werden. Unerwartete Erreignisse sollten erkannt und die Reise entsprechen angepasst werden. Das Gegenteil wäre ein abgeschossener Pfeil, dessen Richtung sich nicht mehr ändert.

**Projektkontrolle**

- Budgetkontrolle
- Fortschrittskontrolle. Aktiv nachfragen und Probleme lösen. Es darf nicht vorkommen, dass ein Programmierer über Wochen am gleichen Problem arbeitet.
- Management by walking around: Anwensenheit schafft Verbindlichkeit
- Hotspots identifizieren: Hotspots sind Dinge, die niemand übernehmen möchten. Dinge, die immer wieder geändert werden müssen.

**Gehen Sie auf die Baustelle**

- Ein Projektleiter muss selbständig den aktuelle Stand der Baustelle (Versionierungstool) kennen.
- Commits im Versionierungstool überprüfen. Warum gibt es keine Commits? (Code anschauen, Diffs anschauen, Tests überprüfen)
- Welche Codeteile haben sich oft geändert?
- Welche Codeteile werden oft gebraucht?

**Geschichtsbeschreibung**
Das erste Projekt wird wahrscheinlich schief laufen. Man sollte sich deshalb die wichtigsten Punkte (Lessons learned) notieren und eine Schlusspräsentation halten. (Export von Redmine, Screenshots von Milestones und Schwierigkeiten, Sitzungsprotokolle, Mitarbeiterzufriedenheit, Kundenzufriedenheit, Fehlerquellen)

# 14. Redmine

Redmine ist ein Open Source Projektmanagement Werkzeug wie JIRA. Es wird hauptsächlich für folgende Dinge verwendet:

- Tasks erfassen (Features)
- Bug, Issue Tracking
- Versionskontrolle Integration (Git, SVN)
- Wiki (für Protokolle, How-To's, Milestone Reviews), User Forum, Dokumente, Files
- Release Planung
- Rollenbasierte Zugriffskontrolle für Benutzer
- Stundenerfassung

# A. Listings

# B. Abbildungsverzeichnis

# C. Tabellenverzeichnis