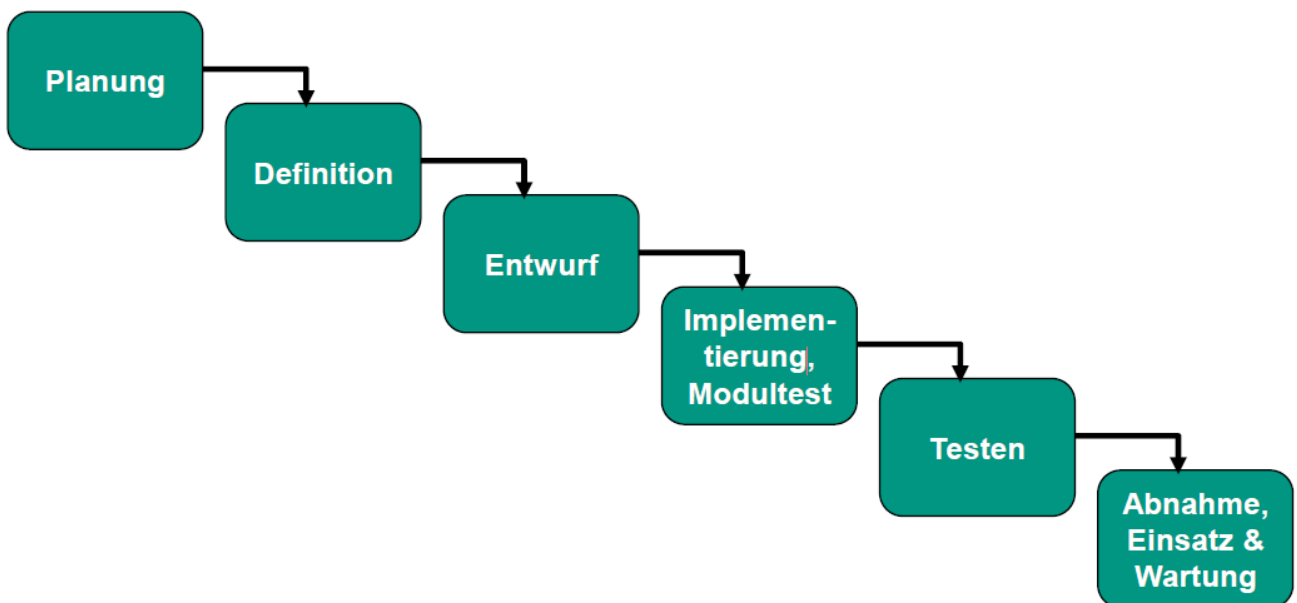


# Softwaretechnik I

Björn Holtvogt

Das Wasserfallmodell als einfaches Grundmodell einer softwaretechnischen Planung:



# Inhaltsverzeichnis

<b>1</b>	<b>Planungsphase</b>	<b>1</b>
1.1	Lastenheft . . . . .	1
1.2	Durchführbarkeitsuntersuchung . . . . .	2
<b>2</b>	<b>Definitionsphase</b>	<b>3</b>
2.1	Pflichtenheft . . . . .	3
2.2	Modellarten . . . . .	3
2.3	Gliederung . . . . .	4
2.4	Liskov'sches Substitutionsprinzip . . . . .	4
2.5	Folgerungen aus dem Substitutionsprinzip . . . . .	5
2.5.1	Varianzen gemäß dem Substitutionsprinzip und in Java . . . . .	6
2.6	Kapselungsprinzip . . . . .	6
2.7	Geheimnisprinzip . . . . .	6
2.8	Beispiele für Verbergung . . . . .	7
<b>3</b>	<b>Entwurfsphase</b>	<b>8</b>
3.1	Softwarearchitektur . . . . .	8
3.2	Entwurfsmethoden . . . . .	8
3.3	Modularer Entwurf . . . . .	8
3.4	Anforderungen an ein Modul . . . . .	9
3.5	Modul . . . . .	9
3.6	Objekt-orientierter Entwurf . . . . .	10
3.7	Ergebnis eines objekt-orientierten Entwurfs . . . . .	10
3.8	Architekturstile . . . . .	11
3.8.1	Schichtenarchitektur . . . . .	11
3.8.2	Klient/Dienstgeber . . . . .	13
3.8.3	Partnernetze . . . . .	13
3.8.4	Datenablage . . . . .	14
3.8.5	Modell-Präsentation-Steuerung . . . . .	14
3.8.6	Fließband . . . . .	15
3.8.7	Rahmenarchitektur . . . . .	15
3.8.8	Dienstorientierte Architekturen . . . . .	17
3.9	Entwurfsmuster . . . . .	19
3.9.1	Entkopplungsmuster . . . . .	21
3.9.2	Variantenmuster . . . . .	24
3.9.3	Zustandshandhabungsmuster . . . . .	27
3.9.4	Steuerungsmuster . . . . .	30
3.9.5	Bequemlichkeitsmuster . . . . .	31

<b>4</b>	<b>Implementierungsphase</b>	<b>33</b>
4.1	Grundlagen der Parallelität . . . . .	33
4.2	Parallelität in Java . . . . .	33
4.2.1	Erzeugen von Kontrollfäden . . . . .	33
4.2.2	Konstrukte zum Schützen kritischer Abschnitte . . . . .	34
4.2.3	Bewertung von parallelen Algorithmen . . . . .	37
<b>5</b>	<b>Testphase</b>	<b>38</b>
5.1	Fehlerarten . . . . .	38
5.2	Fehlerklassen . . . . .	38
5.3	Arten von Testhelfern . . . . .	38
5.4	Testphasen . . . . .	39
5.5	Klassifikation testender Verfahren . . . . .	39
5.5.1	Kontrollflussorientierte Testverfahren . . . . .	40
5.5.2	Funktionale Tests . . . . .	41
5.5.3	Leistungstests . . . . .	41
5.5.4	Manuelle Prüfung . . . . .	41
5.5.5	Prüfprogramme . . . . .	42
5.5.6	Integrationstests . . . . .	42
5.5.7	Systemtests . . . . .	42
5.5.8	Abnahmetests . . . . .	42
5.6	Inspektion . . . . .	43
<b>6</b>	<b>Abnahme- Einführungs-, Wartungs- und Pflegephase</b>	<b>45</b>
6.1	Abnahmephase . . . . .	45
6.2	Einführungsphase . . . . .	45
6.3	Wartungs- und Pflegephase . . . . .	45
<b>7</b>	<b>Aufwandsschätzung</b>	<b>46</b>
7.1	Schätzmethoden . . . . .	46
<b>8</b>	<b>Prozessmodelle</b>	<b>47</b>
8.1	Agile Prozessmodelle . . . . .	49
8.1.1	Extreme Programming . . . . .	50
8.1.2	Scrum . . . . .	51
<b>9</b>	<b>Werkzeugkette und Versionskontrolle</b>	<b>53</b>
9.1	Befehlskette . . . . .	53
9.2	Interaktion innerhalb Local- und Remote Repository . . . . .	53
9.3	Unterschied: GIT und SVN . . . . .	54
9.4	Versionskontrolle . . . . .	54
9.4.1	Vorwärtsdelta . . . . .	54
9.4.2	Rückwärtsdelta . . . . .	54
9.4.3	Ein- und Ausbuchen . . . . .	54

<b>10 Quellenverzeichnis</b>	<b>55</b>
10.1 Literatur . . . . .	55
10.2 Grafiken . . . . .	55

# 1 Planungsphase

Ziel:

- Beschreibung des Systems in Worten als **Lastenheft**
- **Durchführbarkeitsuntersuchung**

## 1.1 Lastenheft

- Zielbestimmung
- Produkteinsatz
- Funktionale Anforderungen
  - Beschreibt Funktionen, die das System unterstützen muss, unabhängig von der Implementierung
  - Als Aktionen formuliert: "Ersterfassung, Änderung und Kunden"
- Produktdaten
- Nichtfunktionale Anforderungen
  - Beschreiben Eigenschaften des Systems: "Reagiert innerhalb von zehn Sekunden"
  - Als Einschränkung (constraints) oder Zusicherung (assertions) formuliert
- Systemmodelle
  - Szenarien
  - Anwendungsfälle
- Glossar
  - Begriffslexikon zur einheitlichen Kommunikation mit dem Kunden

## 1.2 Durchführbarkeitsuntersuchung

- Fachliche Durchführbarkeit
  - Fachkräfte genügend qualifiziert?
- Alternative Lösungsvorschläge
  - Open-Source als Teilersatz?
- Personelle Durchführbarkeit
  - Genügend qualifizierte Fachkräfte?
- Risiken
- Ökonomische Durchführbarkeit
  - Projekt wirtschaftlich? (Aufwands- und Termschätzung, Wirtschaftlichkeitsrechnung)
- Rechtliche Gesichtspunkte
  - Datenschutz
  - Zertifizierung

## 2 Definitionsphase

Ziel:

- Erstellung eines **Pflichtenhefts**, mit Hilfe von **Objekt-** und **dynamischen Modellen**

### 2.1 Pflichtenheft

- **Definiert** das zu erstellende System **vollständig und exakt**
  - Ohne Nachfragen implementierbar!
  - Nicht wie, sondern nur was zu implementieren ist
- Verfeinerung des Lastenhefts

### 2.2 Modellarten

- Funktionales Modell (aus dem Lastenheft)
  - Szenarien
  - Anwendungsfalldiagramme
- Objektmodell
  - Klassendiagramm
  - Objektdiagramm
- Dynamisches Modell
  - Sequenzdiagramm
  - Zustandsdiagramm
  - Aktivitätsdiagramm

## 2.3 Gliederung

- Zielbestimmung
- Produkteinsatz
- *Produktumgebung*
- Funktionale Anforderungen
- Produktdaten
- Nichtfunktionale Anforderungen
- *Globale Testfälle*
- Systemmodelle
  - Szenarien
  - Anwendungsfälle
  - *Objektmodelle*
  - *Dynamische Modelle*
  - *Benutzerschnittstelle - Bildschirmskizze, Navigationspfade*
- Glossar

## 2.4 Liskov'sches Substitutionsprinzip

- In einem Programm, in dem U eine Unterklasse von K ist, kann **jedes Exemplar der Klasse K durch ein Exemplar von U ersetzt werden**, wobei das Programm weiterhin korrekt funktioniert



## 2.5 Folgerungen aus dem Substitutionsprinzip

- Signaturvererbung
  - Eine in der Oberklasse definierte und evtl. implementierte Methode überträgt **nur ihre Signatur** auf die Unterklasse
- Implementierungsvererbung
  - Eine in der Oberklasse definierte und implementierte Methode überträgt ihre **Signatur und ihre Implementierung** auf die Unterklasse
  - ⇒ Implementierungsvererbung setzt Signaturvererbung voraus!
- Anpassung geerbter Eigenschaften
  - Überladen
    - \* Eine geerbte Methode mit gleichem Namen, aber anderer Signatur wird definiert
  - Überschreiben
    - \* Eine geerbte, **dynamische** Methode mit gleichem Namen und gleicher Signatur wird **neu implementiert**
  - Verdecken
    - \* Eine geerbte, **statische** Methode mit gleichem Namen und gleicher Signatur wird **neu implementiert**

- Varianz
  - Definition
    - \* Parametermodifikation einer überschriebenen Methode
  - Invarianz
    - \* Der Parametertyp wird nicht modifiziert
  - Kovarianz
    - \* Der Parametertyp wird spezialisiert
  - Kontravarianz
    - \* Der Parametertyp wird allgemeiner

### 2.5.1 Varianzen gemäß dem Substitutionsprinzip und in Java

	Eingabeparameter			Ausgabeparameter		
	Invarianz	Kovarianz	Kontravarianz	Invarianz	Kovarianz	Kontravarianz
<b>Substitutionsprinzip</b>	✓		✓	✓	✓	
<b>Java</b>	✓			✓	✓	

## 2.6 Kapselungsprinzip

- Der **Zustand** ist zwar nach außen sichtbar, er wird aber **im Inneren des Objektes verwaltet** (und also nur kontrolliert geändert)

## 2.7 Geheimnisprinzip

- Jedes Modul verbirgt eine **wichtige Entwurfsentscheidung** hinter einer **wohl-definierten Schnittstelle** die sich bei einer Änderung der Entscheidung nicht mit ändert
  - Verborgenes und Unbenutztes kann ohne Risiko geändert werden

## 2.8 Beispiele für Verbergung

- **Datenstrukturen** (Wahl, Größe und Implementierung und Operationen an diesen)
- **Maschinennahe Details** (Gerätetreiber, Ein- und Ausgabe)
- **Betriebssystemnahe Details** (Ein- und Ausgabeschnittstellen, Dateiformate, Netzwerkprotokolle)
- **Grundsoftware** (Datenbanken, Oberflächenbibliotheken)
- **Benutzungsschnittstellen** (Kommandoschnittstelle, graphische Oberfläche, Gesten-gesteuerte Oberfläche, Web, Sprachsteuerung, Kombinationen davon,...)
- **Sprache** (Text von Dialogen, Beschriftungen)
- **Reihenfolge der Verarbeitung**

## 3 Entwurfsphase

Ziel:

- Aus gegebenen Anforderungen an einem Softwareprodukt wird eine softwaretechnische Lösung, die **Softwarearchitektur**, entwickelt

### 3.1 Softwarearchitektur

- Gliederung eines Softwaresystems in **Komponenten** (Module, Klassen) und **Subsysteme** (Pakete, Bibliotheken)
- **Spezifikationen** der Komponenten und Subsystemen
  - Aufstellung der **Benutzrelation**

### 3.2 Entwurfsmethoden

- **Modularer Entwurf**
- **Objekt-orientierter-Entwurf**
  - Erweiterung um Vererbung, Polymorphie und Datenmodellierung

### 3.3 Modularer Entwurf

- **Externer Entwurf**
  - **Modulführer (Grobentwurf)**
    - \* Gliederung in Komponenten und Subsysteme
    - \* Beschreibung der Modulfunktionen
    - \* Benutzt Entwurfsmuster

- **Modulschnittstellen**
  - \* Genaue Beschreibung der vom Modul zur Verfügung gestellten Elemente (Typen, Variablen)
  - \* Module mit Ein- und Ausgabe: Genaue Beschreibung der Formate (XML,..)
- Interner Entwurf
  - **Benutzrelation**
    - \* Gliederung in Komponenten und Subsysteme
    - \* Beschreibung der Benutzung von Modulen und Subsystemen untereinander
      - Azyklisch gerichteter Graph: Das ermöglicht einen inkrementellen/es Aufbau und Testen!
  - **Feinentwurf**
    - \* Beschreibung modulinterner Datenstrukturen und Algorithmen
    - \* Pseudocode (für Assembler)

### 3.4 Anforderungen an ein Modul

- Module sollen unabhängig voneinander bearbeitet und benutzt werden können
  - Ohne Kenntnis der späteren Nutzung entworfen, implementiert, getestet und überarbeitet

### 3.5 Modul

- Ist eine **Menge von Programmelementen** (Typen, Klassen, Konstanten, Variablen, Datenstrukturen, Prozeduren, Funktionen,..), die nach dem **Geheimnisprinzip** gemeinsam entworfen und geändert werden

## 3.6 Objekt-orientierter Entwurf

- Externer Entwurf
  - **Paket- und Klassenführer**
    - \* UML-Klassen und Paketdiagramme zur Dokumentation der Entwurfsentscheidung
  - **Schnittstellen (der Klassen)**
    - \* Klassen, abstrakte Klassen und reine Schnittstellen
- Interner Entwurf
  - **Benutztrelation** auf Ebene von Paketen und Klassen dokumentiert
  - **Feinentwurf** liefert Beschreibung der modulinternen Datenstrukturen und Algorithmen, sowie Pseudocode (wenn nötig)

## 3.7 Ergebnis eines objekt-orientierten Entwurfs

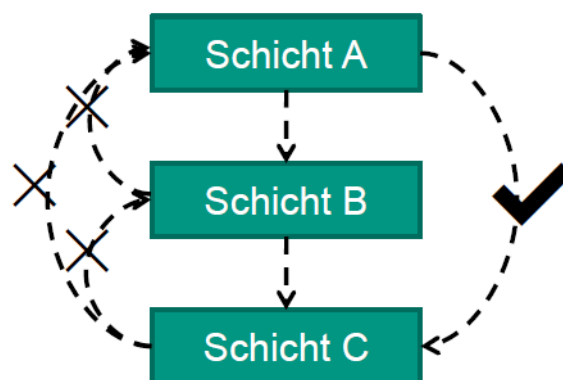
- **Mehrfach-Instanziierung** von Klassen
- **Vererbung** und **Polymorphie**
- **Variantenbildung** (Mehrfachimplementierung einer Schnittstelle)

### 3.8 Architekturstile

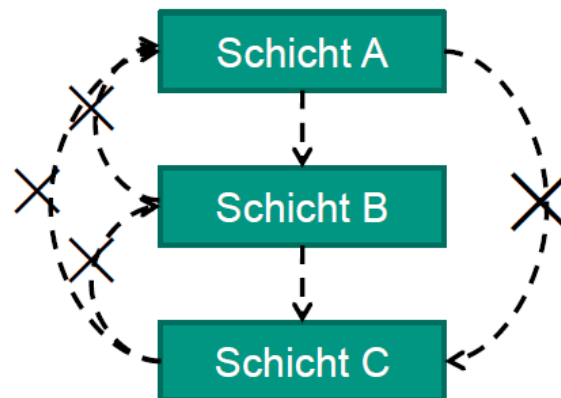
- Schichtenarchitektur
- Klient/Dienstgeber (Client/Server)
- Partnernetze (Peer-To-Peer)
- Datenablage (Repository)
- Modell-Präsentation-Steuerung (Model-View-Controller)
- Fließband (Pipeline)
- Rahmenarchitektur (Framework)
- Dienstorientierte Architektur (Service oriented architecture)

### 3.8.1 Schichtenarchitektur

- Gliederung einer Softwarearchitektur mit **hierarchischen Schichten**
  - Eine Schicht nutzt die darunter liegenden Schichten und diese stellen ihre Dienste den darüber liegenden Schichten zur Verfügung
- Transparent:



- Intransparent:



- 3-stufige Architektur

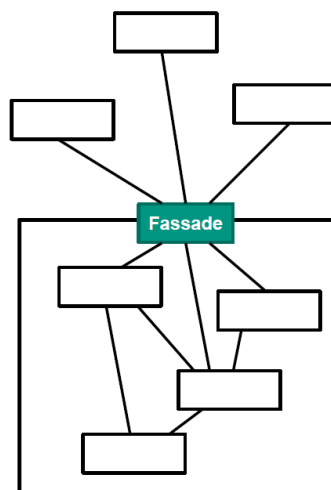
- 3-Schichten Architektur mit Schichten auf unterschiedlichen Rechnern

- 3-Schichten Architektur

- Benutzerschnittstelle, Anwendungskern und Datenbanksystem

Oft wird die Schichtenarchitektur mit dem Entwurfsmuster **Fassade** verwendet.

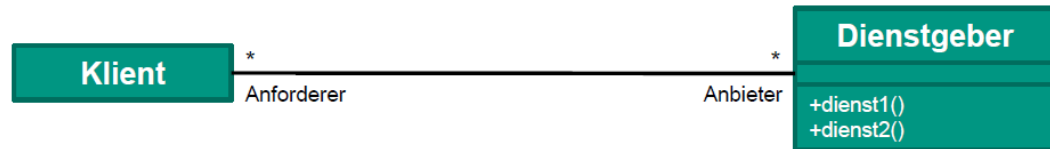
- Leitet an die eigentlichen Elemente in der Schicht weiter





### 3.8.2 Klient/Dienstgeber

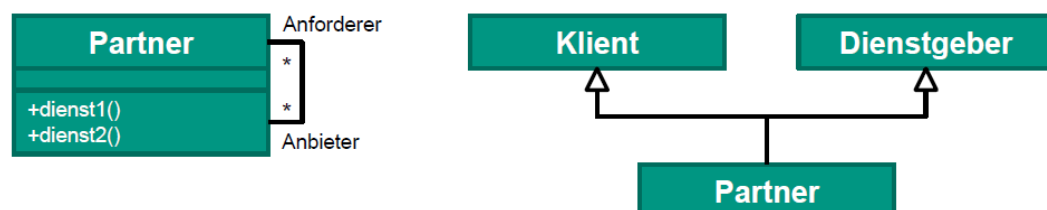
- Ein- oder mehrere **Dienstgeber** bieten Dienste für andere Subsysteme (Klienten) an



- Oft bei **Datenbankservern** verwendet
  - Front-End: Benutzeroberfläche für den Benutzer (Klient)
  - Back-End: Datenbankzugriff und Manipulation (Dienstgeber)
- **Klientenfunktionen**
  - Eingaben des Benutzer entgegennehmen und vor verarbeiten
- **Dienstgeberfunktionen**
  - Datenverwaltung, -integrität und -konsistenz
  - Sicherheit
- Beispiel: TCP/IP, DNS (Netzwerkebene)

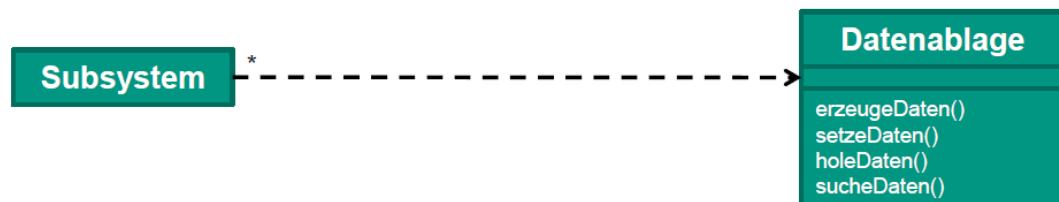
### 3.8.3 Partnernetze

- **Verallgemeinerung** von Klient/Dienstgeber
- Alle Subsysteme sind **gleichberechtigt**



### 3.8.4 Datenablage

- Subsysteme **verändern Daten** von einer zentralen Datenstruktur (Datenablage)
  - Sind **lose gekoppelt** und interagieren nur über die Datenablage
- Realisierung: **Lokaler- oder Fernzugriff**

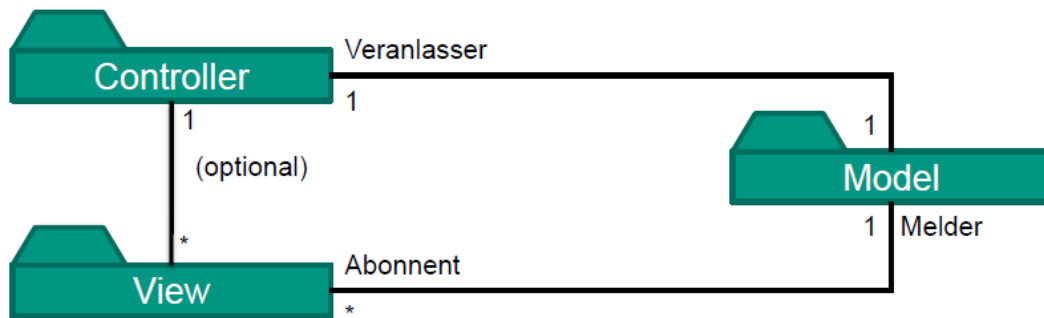


- Beispiele: Subversion, GIT

### 3.8.5 Modell-Präsentation-Steuerung

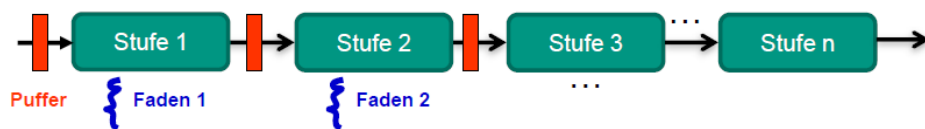
Problem	Lösung
System mit <u>hoher Kopplung</u>	MVC mit <b>Trennung</b> von Daten und Darstellung

- **Modell:**
    - Verantwortlich für anwendungsspezifische Daten
  - **Präsentation:**
    - Verantwortlich für die Darstellung der Objekte der Anwendung
  - **Steuerung:**
    - Verantwortlich für Benutzerinteraktion
    - Aktualisiert Modell
    - Weiterleitung der Änderung von Modelldaten an Präsentation
- ⇒ Entwurfsmuster: **Beobachter!**



### 3.8.6 Fließband

- Jede/r Stufe/Filter ist ein **eigenständiger- und ablaufender Prozess/Faden**
- Jede Stufe **verarbeitet vorherige Daten** und sendet sie an die nächste Stufe



Bei Parallelrechnern echt parallel ausführbar!

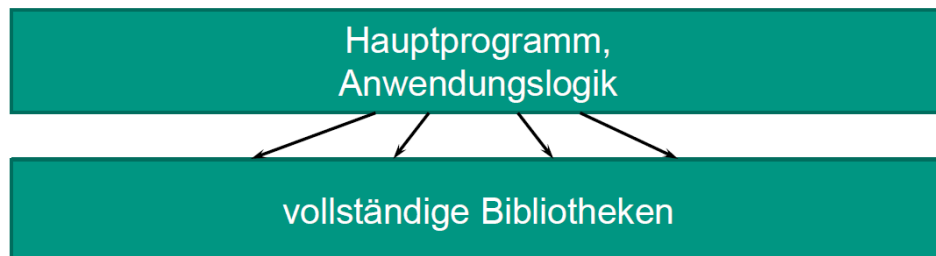
- Beispiel: Unix-Shell
  - **Anwendung:**
    - Datenströme (Videobearbeitung, Übersetzer, Stapelverarbeitung)
- ⇒ Für **gute Leistung**: Einzelne Stufen etwa gleich schnell ausführbar auf Parallelrechnern

### 3.8.7 Rahmenarchitektur

- Bietet **fast vollständiges Programm**, dass durch **Lücken/Erweiterungen** erweitert werden kann
- **Klassenabstrahierung** und **Methodenüberschreibung** vorgesehen
  - Rahmenprogramm führt Erweiterungen (**Plug-Ins**) richtig aus

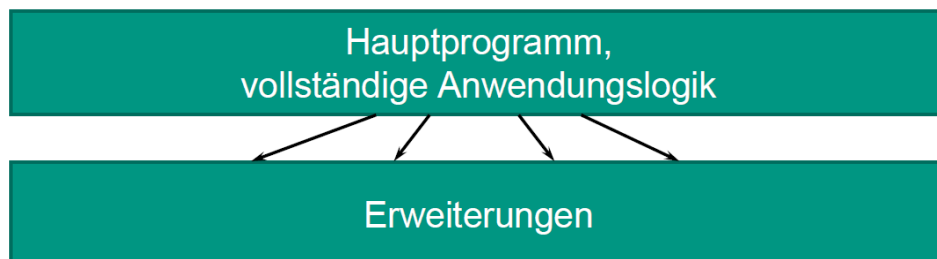
### Herkömmlich:

- Hersteller liefert Bibliotheken
- Benutzer schreibt Hauptprogrammlogik

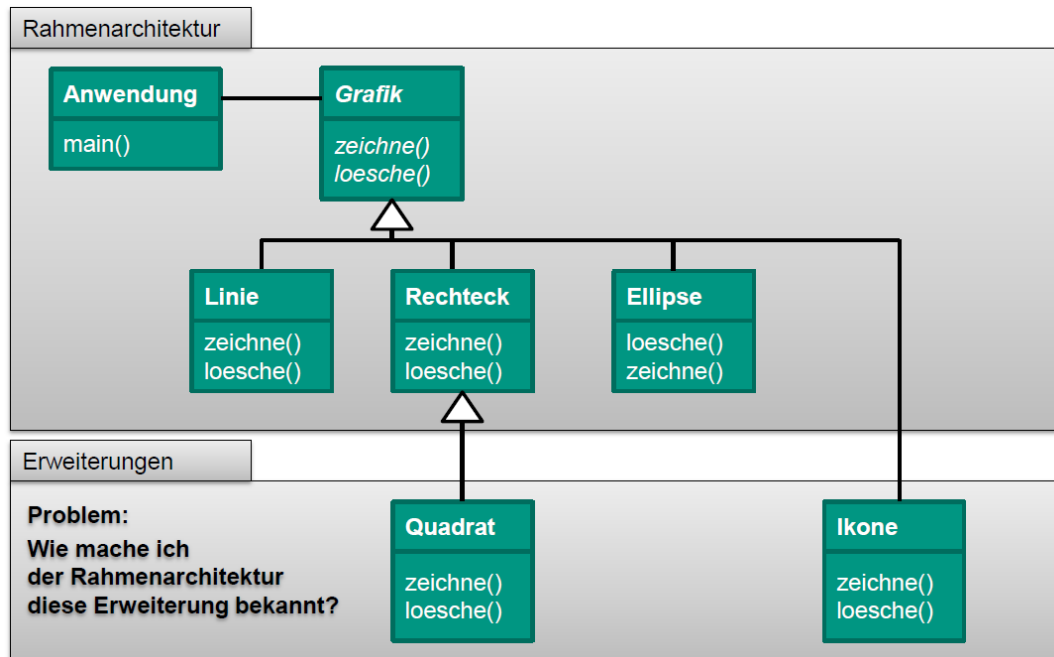


### Mit Rahmenarchitektur:

- **Hollywood-Prinzip:** Don't call us, we call you!
- Hauptprogramm vorhanden, Erweiterungen vom Benutzer werden aufgerufen



Beispiel:



- **Anwendung:**

- Grundversion der Anwendung schon funktionsfähig
- Erweiterung konsistent
- **Entwurfsmuster:** Strategie, Fabrikmethode, Abstrakte Fabrik und Schablonenmethode

### 3.8.8 Dienstorientierte Architekturen

- Anwendungen bestehen aus **unabhängigen Diensten**
  - Abstraktes Konzept
- Dienste als **zentrale Elemente** eines Unternehmens
  - Bereitstellen gekapselter Funktionalität an andere Dienste/Anwendungen
    - \* **Gemeinsame Schnittstelle** für standardisierte(r) Austausch/Kommunikation

- Merkmale/Ziele:

- **Lose Kopplung**

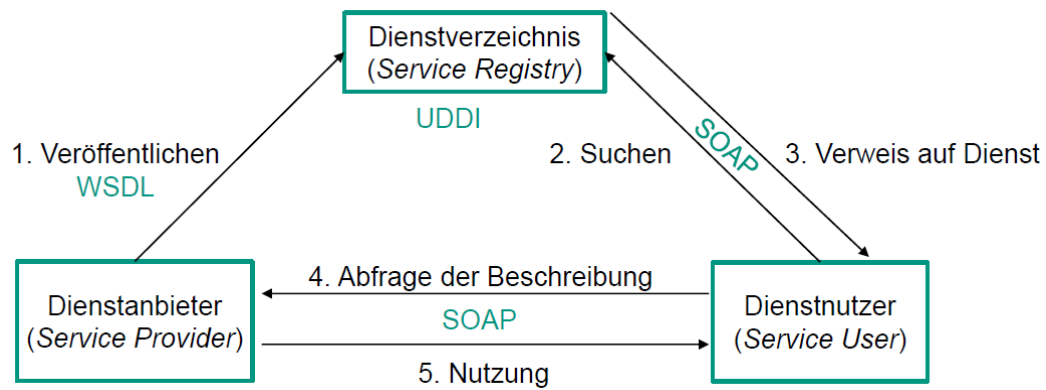
- \* Einfaches Ersetzen eines Dienstes zur Laufzeit
    - \* Dynamisches Binden durch Dienstverzeichnis

- **Unterstützung von Geschäftsprozessen**

- \* Dienste kapseln geschäftsrelevante Funktionalität

- **Verwendung von offenen Standards!**

- \* Programmiersprachen- und plattformunabhängige Bereitstellung von Diensten



⇒ **Dienstmodell** als Kern der Dienstorientierten Architektur

### 3.9 Entwurfsmuster



- **Entkopplungsmuster:**

- Teilt System in **unabhängige Einzelsysteme**

- \* Vorteil: Durch lokale Änderungen **verbesser-, anpass- und erweiterbar ohne** ganzes System zu modifizieren

- **Variantenmuster:**

- **Herausziehen von Gemeinsamkeiten** und platzieren an einer Stelle

- \* Vorteil: **Keine Codewiederholung**

- **Zustandshandhabungsmuster:**

- **Bearbeitung des Zustands** von Objekten unabhängig vom Zweck

- **Steuerungsmuster:**

- **Kontrollflusssteuerung** (Aufruf der richtigen Methode zur richtigen Zeit)

- **Bequemlichkeitsmuster:**

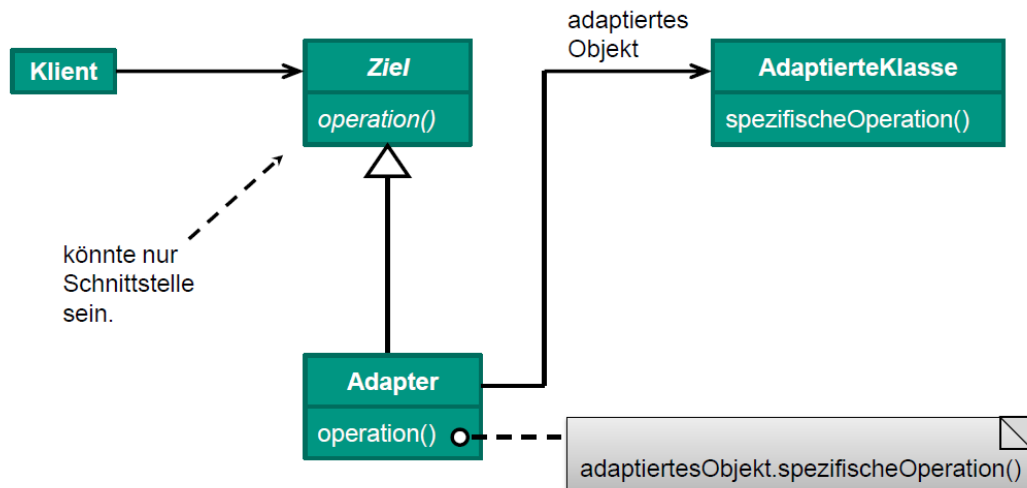
- Sparen von **Schreib- und Denkarbeit**



### 3.9.1 Entkopplungsmuster

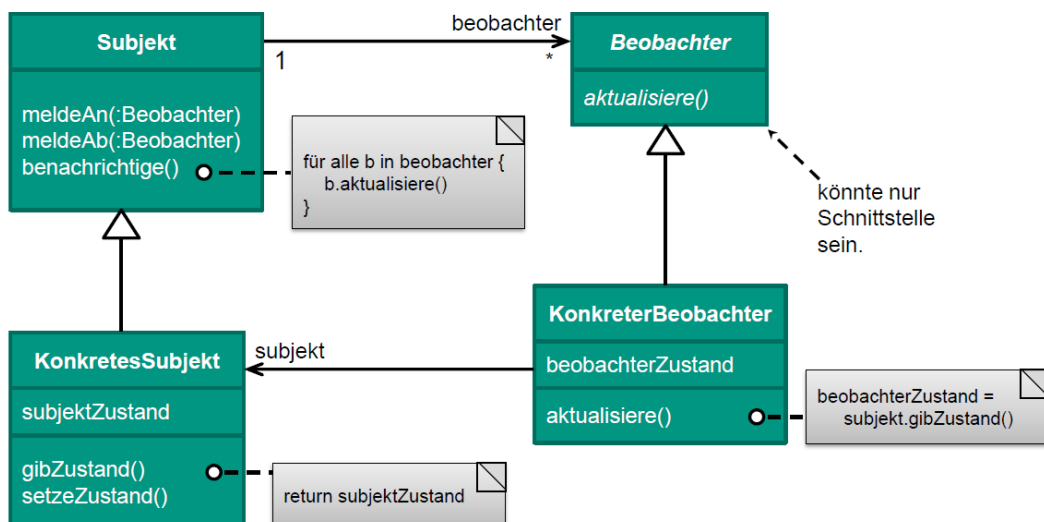
- **Adapter** (Wrapper)

- Passt die Schnittstelle einer Klasse an eine andere, vom Klienten erwartete Schnittstelle an



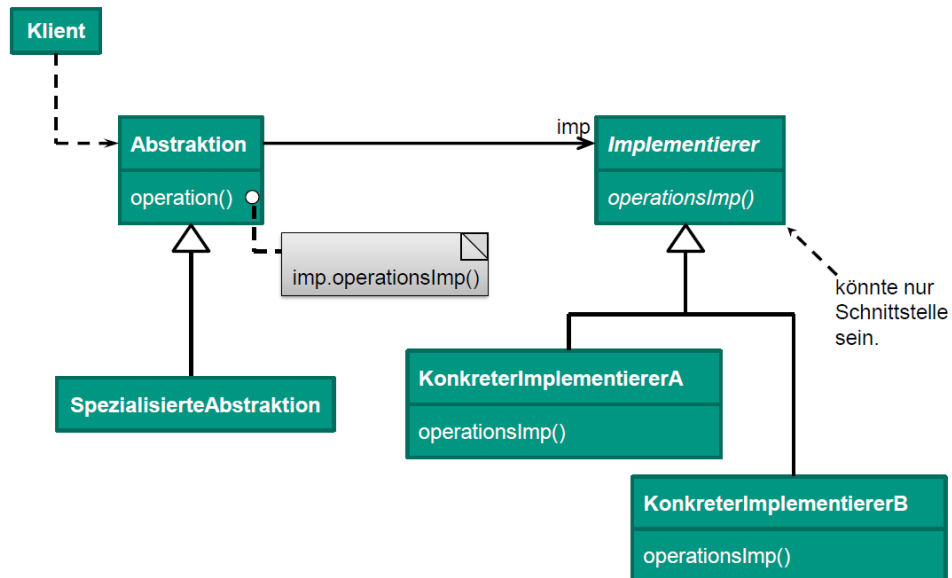
- **Beobachter** (Observer)

- 1:n **Abhängigkeit** zwischen Objekten, so dass die **Änderung eines Zustandes** eines Objektes dazu führt, dass alle abhängigen Objekte **benachrichtigt und aktualisiert** werden



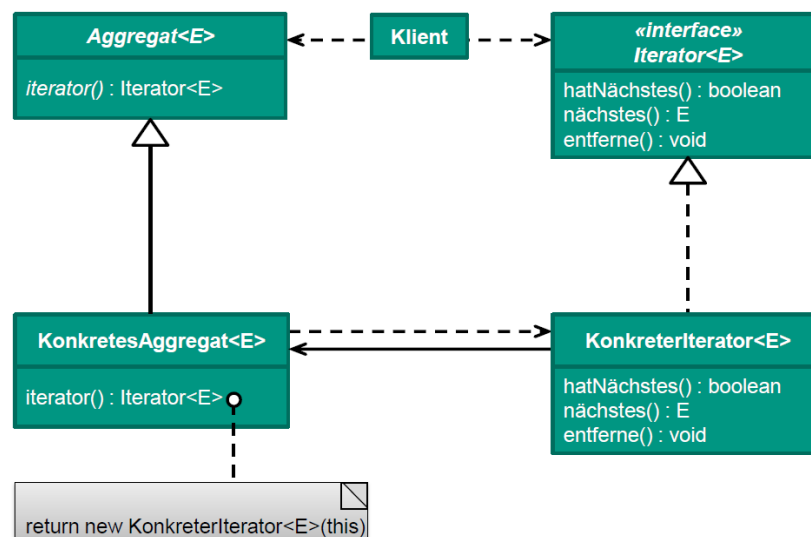
- **Brücke (Bridge)**

- **Entkoppelt eine Abstraktion von ihrer Implementierung**, so dass beide **unabhängig voneinander variiert** werden können



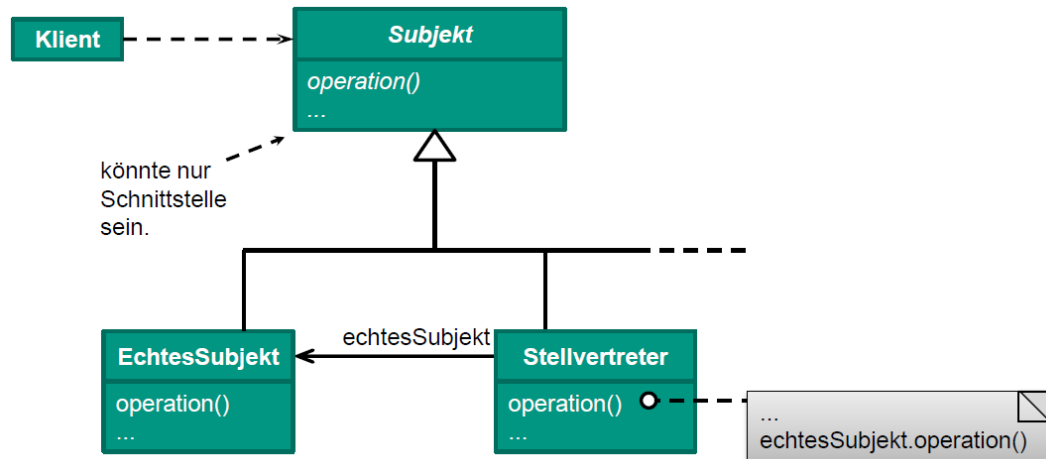
- **Iterator**

- Ermöglicht **sequentiellen Zugriff** auf die Elemente eines zusammengesetzten Objektes, **ohne seine zugrundeliegende Repräsentation offenzulegen**



- **Stellvertreter (Proxy)**

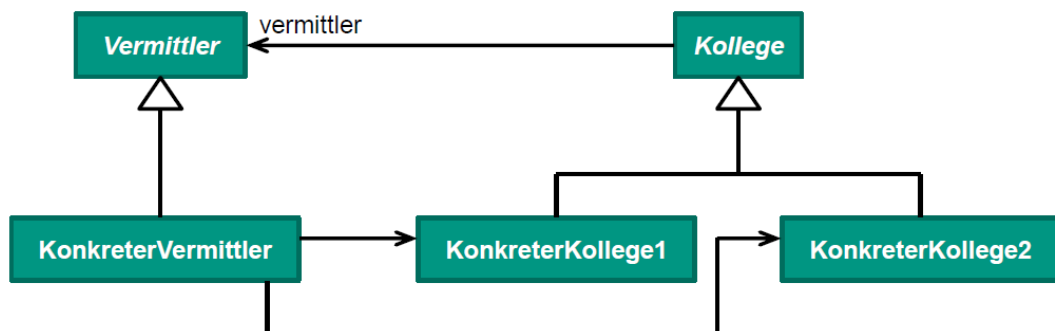
- Kontrolliert den Zugriff auf ein Objekt, mit Hilfe eines vorgelagerten Stellvertreterobjekts



- **Vermittler (Mediator)**

- Definiert ein Objekt, dass das Zusammenspiel einer Menge von Objekten in sich kapselt

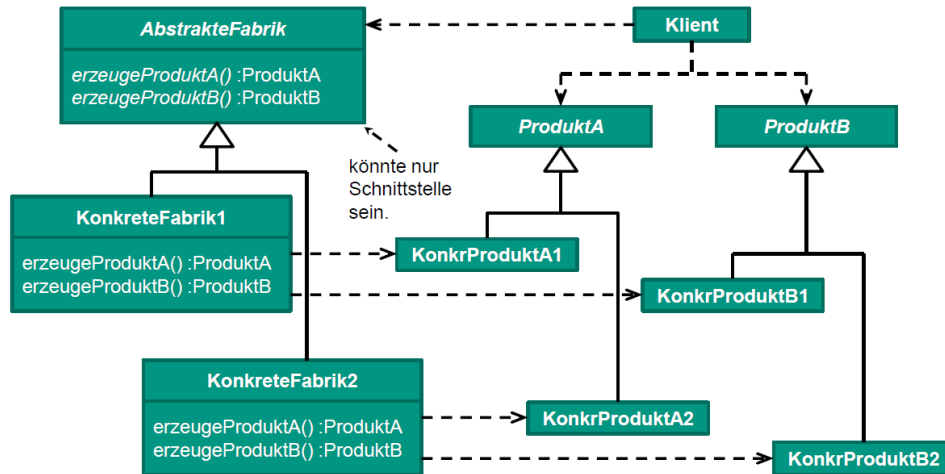
⇒ Zentralisieren!



### 3.9.2 Variantenmuster

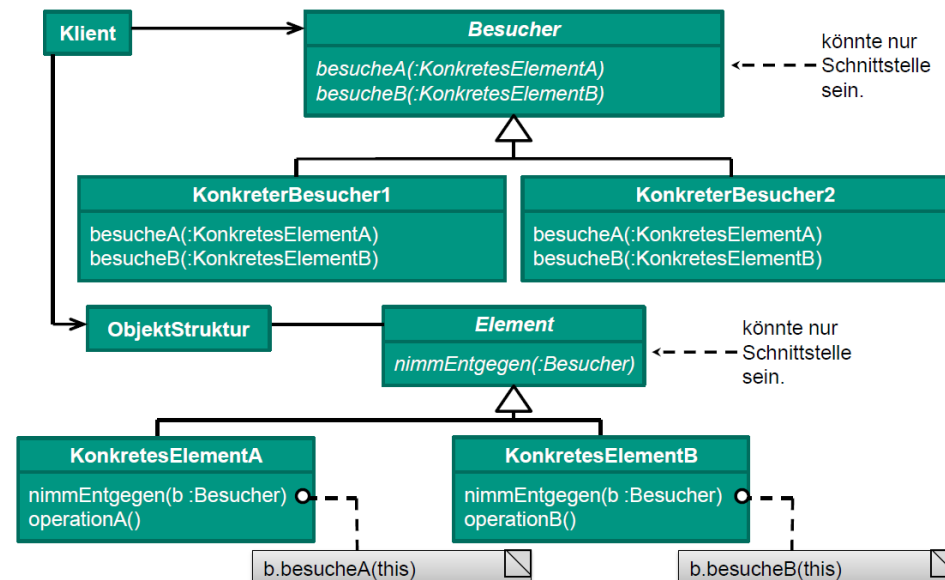
- **Abstrakte Fabrik**

- Bietet eine Schnittstelle zum **Erzeugen von Familien** verwandter oder voneinander abhängigen Objekte, ohne ihre konkreten Klassen zu benennen



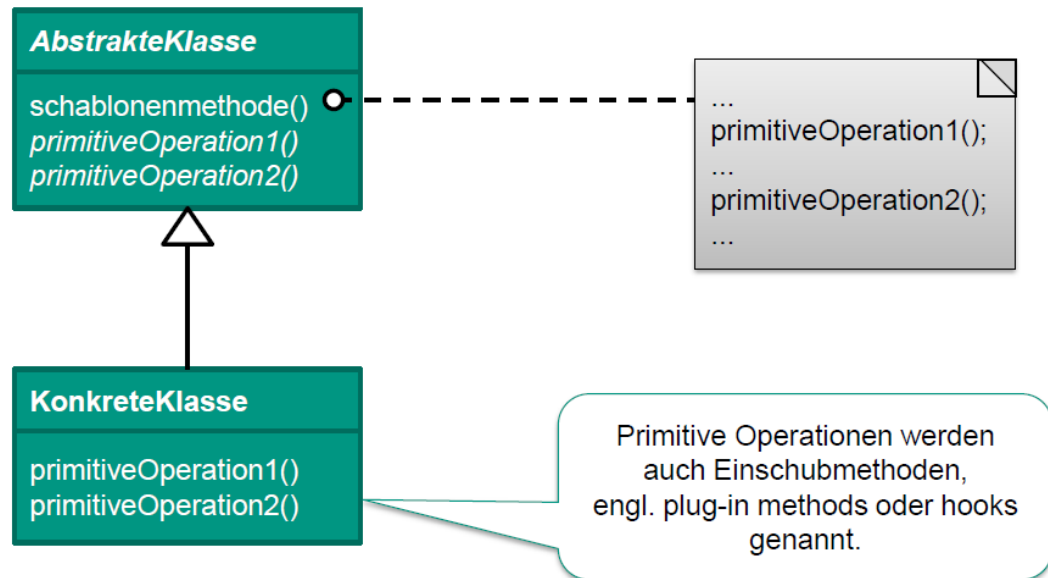
- **Besucher (Visitor)**

- **Kapselt** eine auf den Elementen einer Objektstruktur **auszuführenden Operation** als ein Objekt



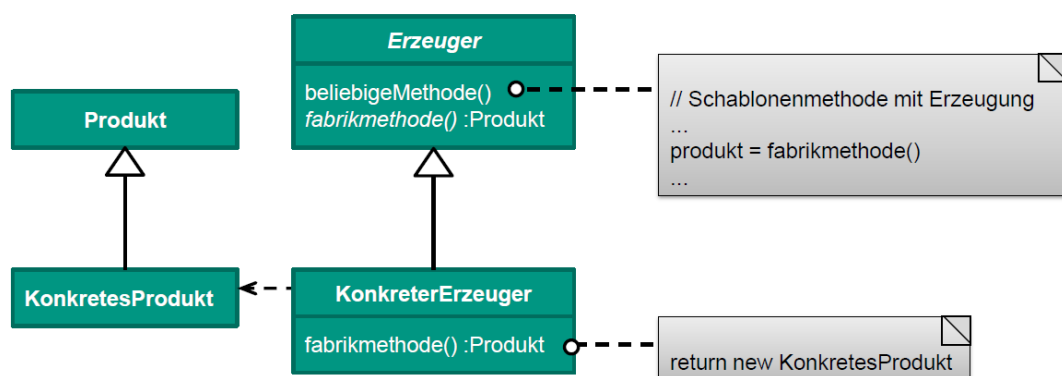
- **Schablonenmethode** (Template Method)

- Definiert **in einer Methode das Skelett eines Algorithmus** und überlässt einzelne Schritte den Unterklassen



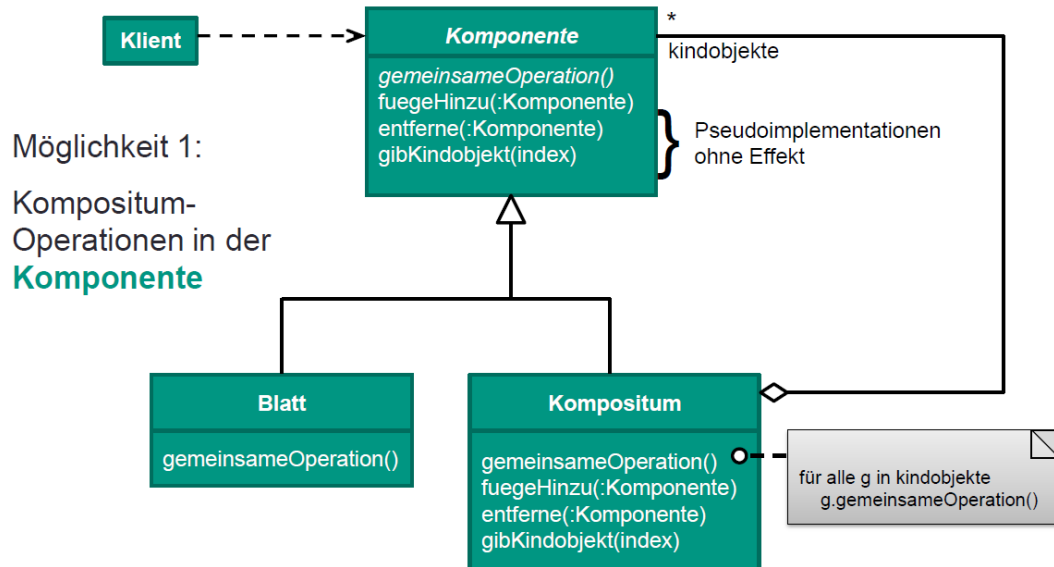
- **Fabrikmethode** (Factory Method)

- Definiert eine **Klassenschnittstelle mit Operationen zum Erzeugen** eines Objekts, aber **lässt Unterklassen entscheiden, von welcher Klasse** das zu erzeugende Objekt ist



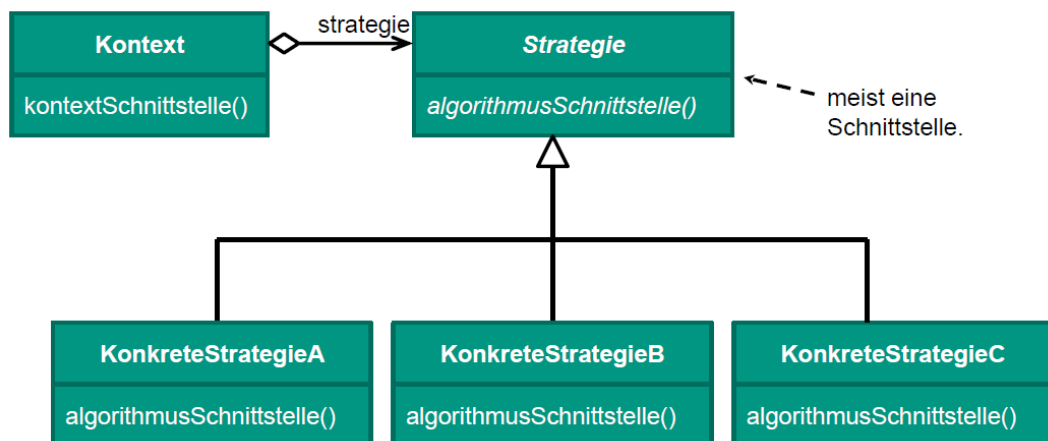
- **Kompositum**

- Fügt Objekte zu **Baumstrukturen** zusammen, um **Bestandshierarchien** zu repräsentieren. Ermöglicht, das **Klienten**, **einzelne Objekte** und **Aggregate einheitlich** behandelt werden



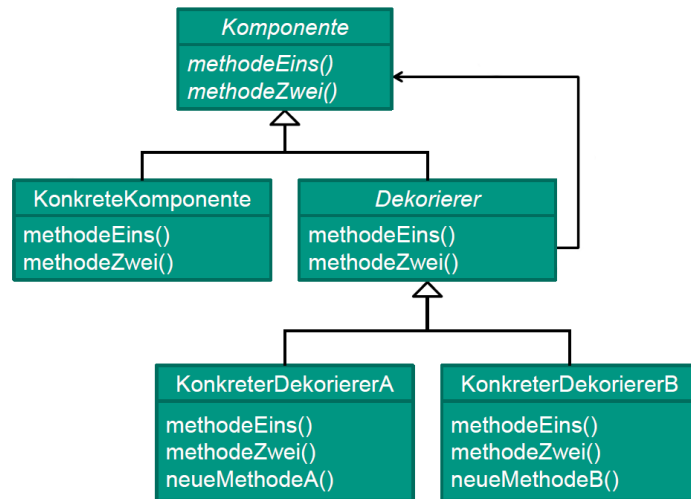
- **Strategie** (Stichwort: Switch-less programming)

- Definiert eine **Familie von Algorithmen**, kapselt sie und macht sie **austauschbar**



- Dekorierer

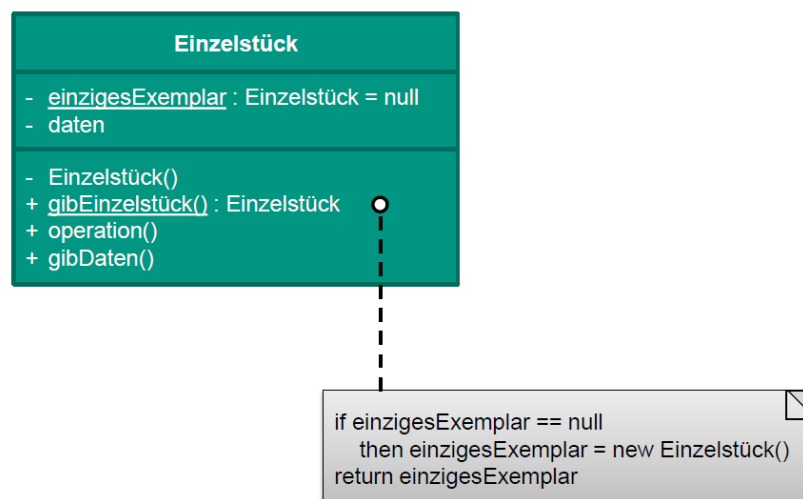
- Fügt **dynamisch zur Laufzeit neue Funktionalitäten** zu einem Objekt hinzu



### 3.9.3 Zustandshandhabungsmuster

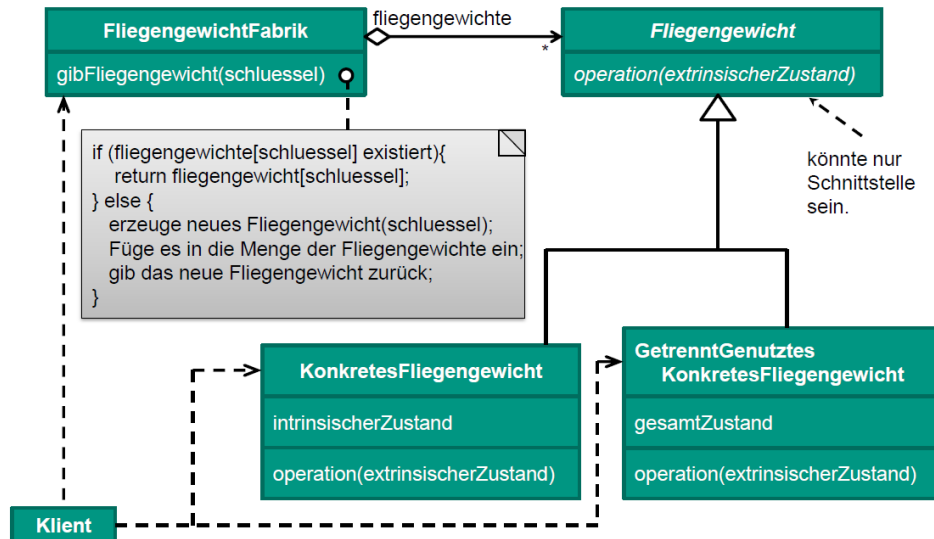
- Einzelstück (Singleton)

- Zusicherung, dass eine Klasse **genau ein Exemplar** besitzt mit **globalen Zugriffspunkt**



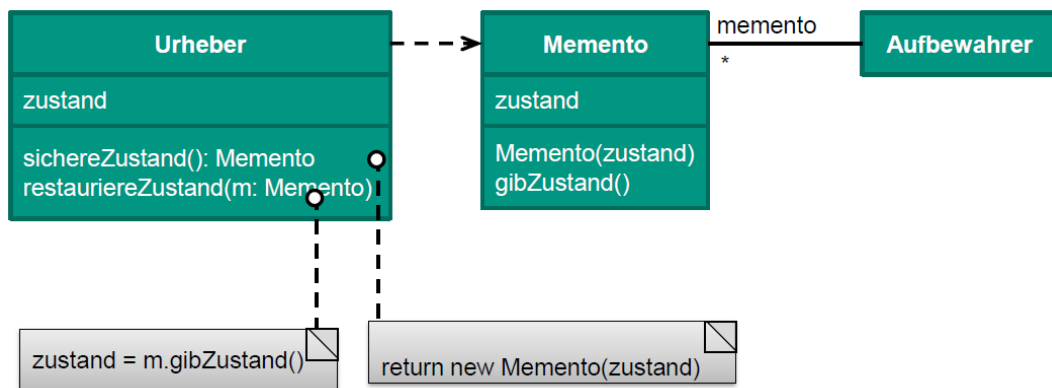
## • Fliegengewicht

- Nutzt Objekte **kleinster Granularität gemeinsam**, um große Mengen von ihnen **effizient zu speichern**



## • Memento

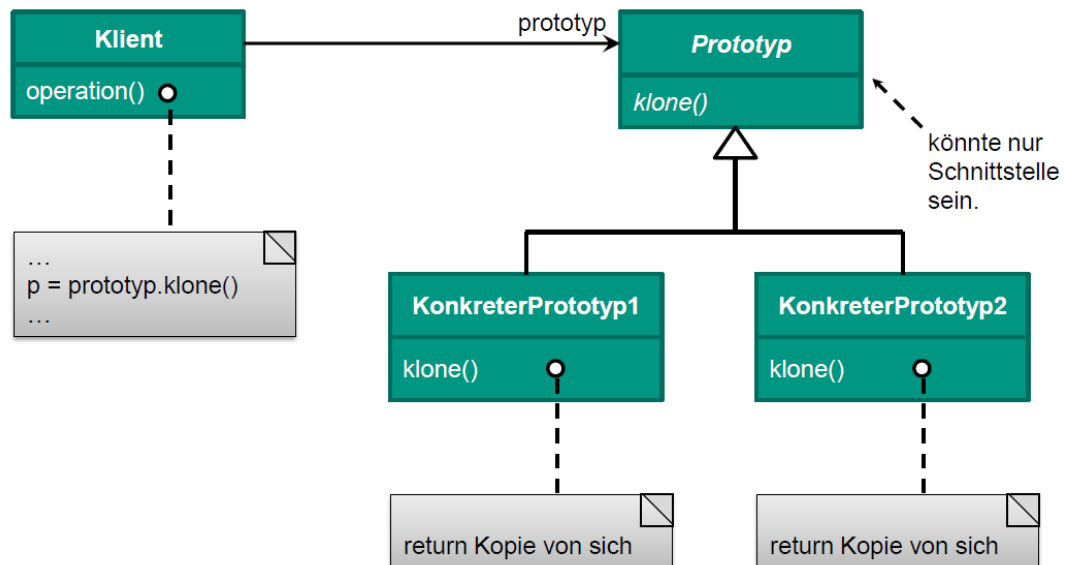
- Erfasst und **externalisiert den internen Zustand eines Objekts**, ohne seine Kapselung zu verletzen, so dass das Objekt **später in diesen Zustand zurückversetzt** werden kann





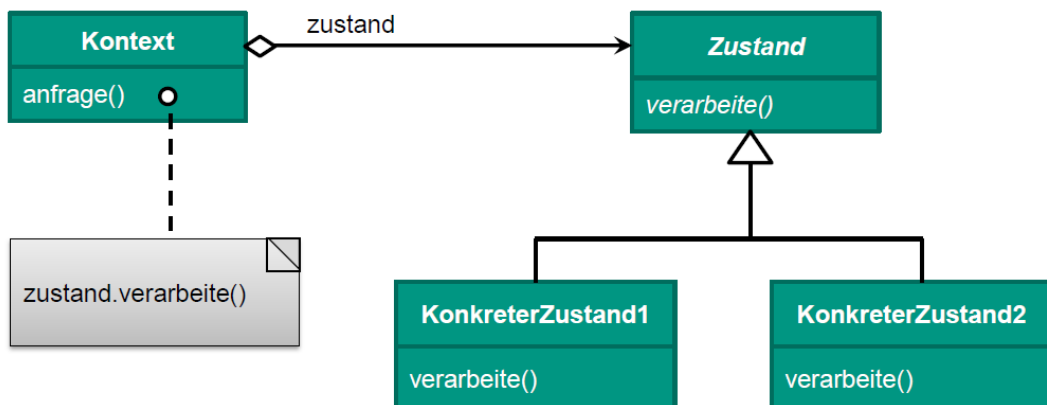
- **Prototyp**

- Bestimmt die **Arten zu erzeugender Objekte** durch die Verwendung eines typischen Exemplars und erzeuge neue Objekte durch **Kopieren dieses Prototyps**



- **Zustand**

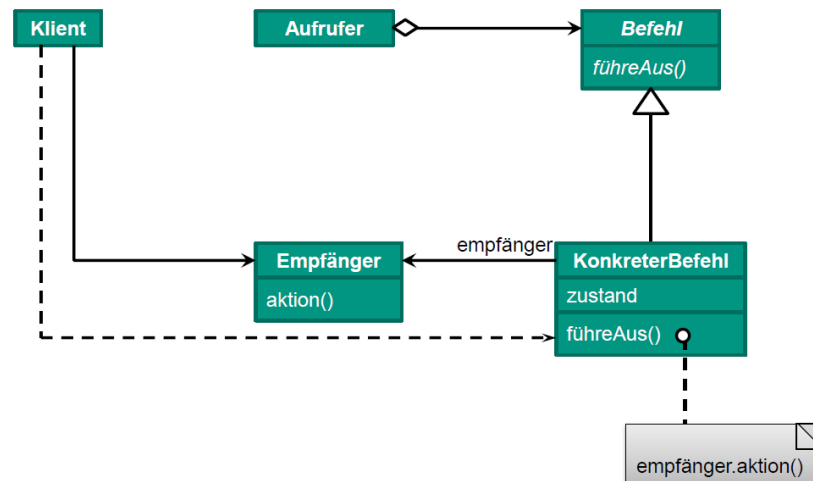
- Ändere das **Verhalten** des Objekts, wenn sich dessen **interner Zustand** ändert



### 3.9.4 Steuerungsmuster

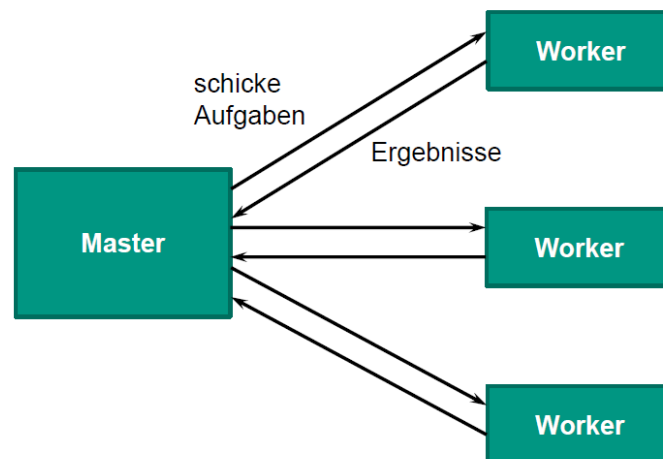
- Befehl

- Kapselt einen **Befehl als Objekt** und ermöglicht es, Klienten mit verschiedenen **Anfragen zu parametrisieren**, Operationen in eine **Warteschlange** zu stellen, ein Logbuch zu führen und **Operationen rückgängig** zu machen



- Master/Worker

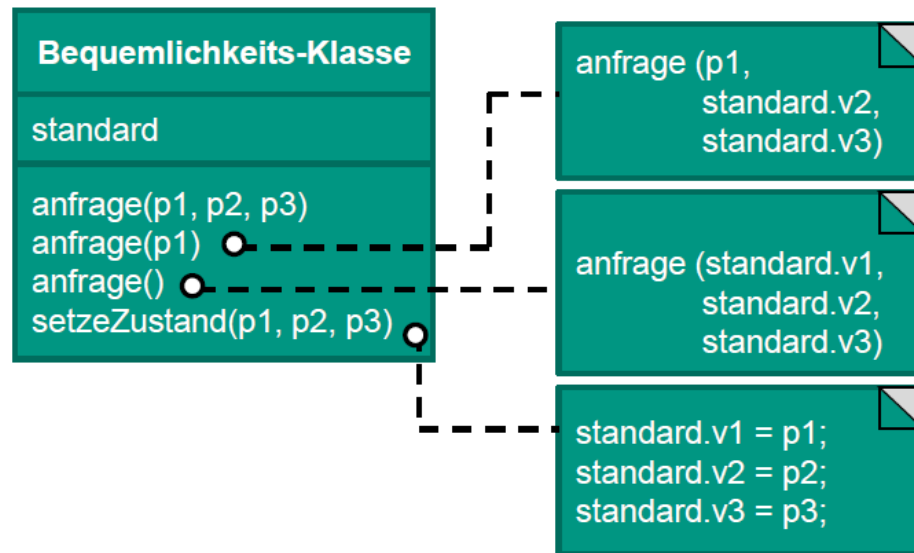
- Bietet **fehlertolerante- und parallele Berechnung**. Ein **Master verteilt die Arbeit an identische Worker** und **berechnet das Endergebnis** aus den Teilergebnissen, die die Worker zurückliefern



### 3.9.5 Bequemlichkeitsmuster

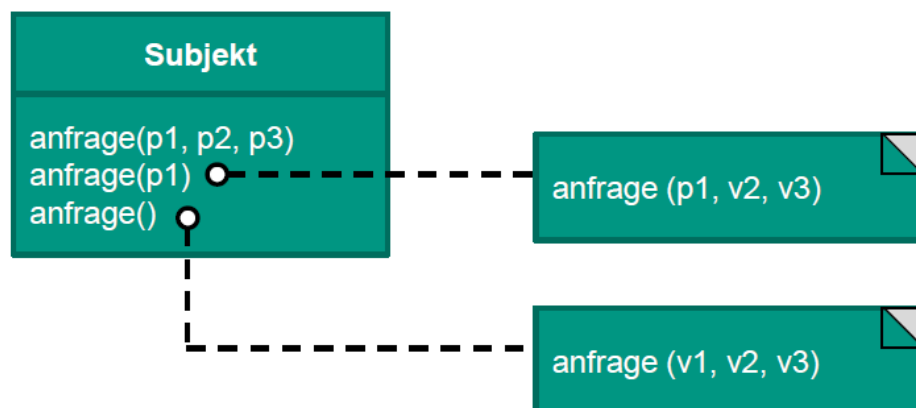
- Bequemlichkeitsklasse

- Vereinfachung von Methodenaufrufen durch Bereithaltung der Parameter in spezieller Klasse



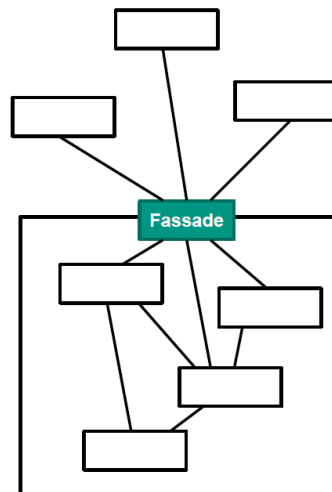
- Bequemlichkeitsmethode

- Vereinfachung von Methodenaufrufen durch Bereithaltung häufig genutzter Parameterkombinationen in zusätzlichen Methoden



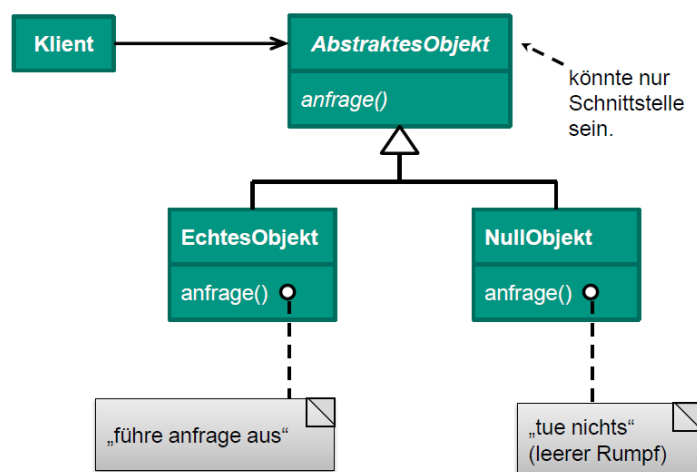
- **Fassade**

- Bietet **einheitliche Schnittstelle** zu einer **Menge von Schnittstellen** eines Subsystems
- \* Fassadenklasse bietet **abstrakte Schnittstelle**, die die Benutzung des Systems vereinfacht



- **Null-Objekt**

- Stellt **Stellvertreter** zur Verfügung, der die gleiche Schnittstelle bietet, aber **nichts tut**



## 4 Implementierungsphase

Ziel:

- Leistungssteigerung durch **Parallelität**

### 4.1 Grundlagen der Parallelität

- Architekturstile:
  - **Gemeinsamer Speicher** (Shared memory)
    - \* CPUs haben einen Speicher
  - Verteilter Speicher
    - \* CPU hat eigenen Speicher

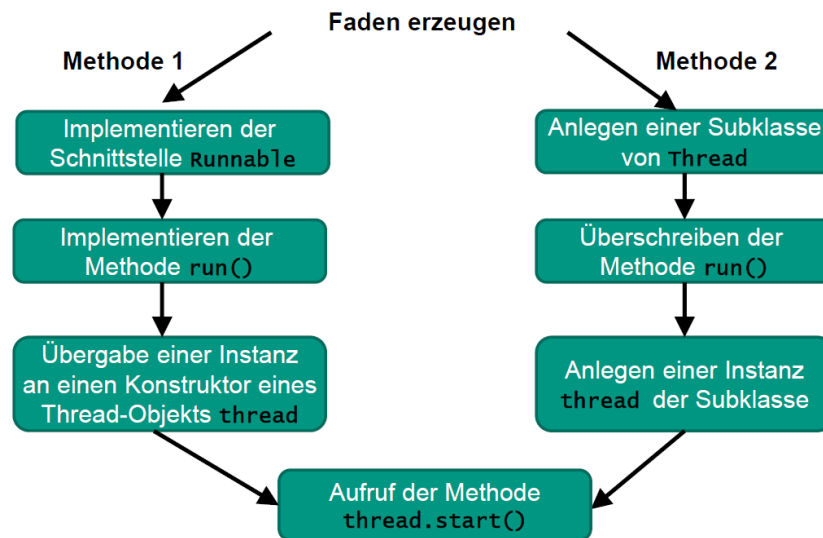
Prozess	Kontrollfaden (Thread)
- Aufgabe eines Programms ⇒ <b>Durch Betriebssystem erzeugt</b>	- Leichtgewichtige Aufgabe eines Prozesses - Greift auf <b>Daten des Prozess</b> zu

### 4.2 Parallelität in Java

#### 4.2.1 Erzeugen von Kontrollfäden

- Mit Hilfe von:
  - Interface **Runnable**
  - Klasse **Thread**

- Skizzierter Ablauf:



- Runnable vs. Thread:
  - **Bessere Modularisierung** mit Runnable
    - \* **Weniger Overhead** durch Kapselung
    - \* Aufgabe kann **über Netzwerk versendet** werden (**serialisierbar**)

#### 4.2.2 Konstrukte zum Schützen kritischer Abschnitte

- Koordination von:
  - **Wechselseitigem Ausschluss**
    - \* Eine Aktivität gleichzeitig!
  - **Warten auf Ereignisse/Benachrichtigungen**
  - **Unterbrechungen**
    - \* Aktivität wartet auf ein (nicht) eintretendes Ereignis

- Wechselseitiger Ausschluss

- Bei **gleichzeitigem Datenzugriff** kann es zu **Wettlaufsituationen** (race conditions) kommen:

- \* **Monitor** besetzt Aktivität bis zum Ende mit:

- **enter()**

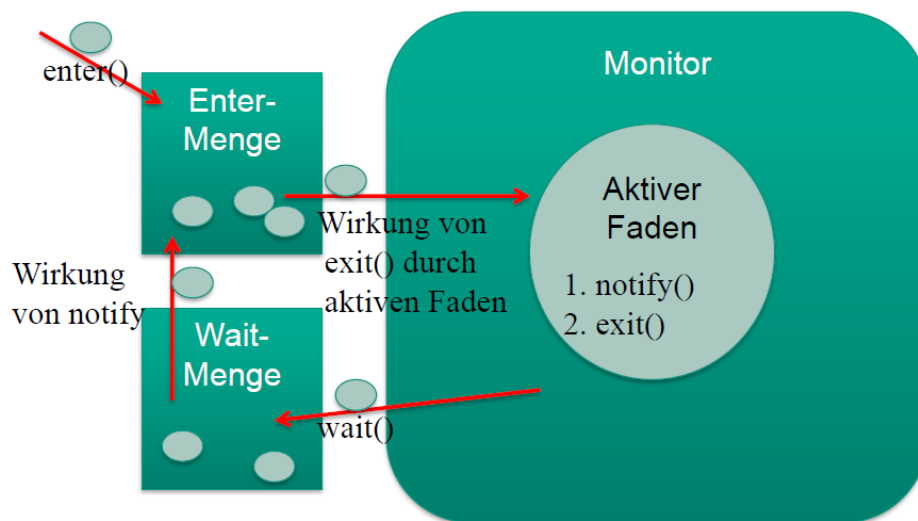
- **exit()**

- \* Synchronisierung nach gleichem Prinzip mit:

- **synchronized (obj)**

- **synchronized void foo()**

- Monitor:



- Warten auf Ereignisse/Benachrichtigungen

- Überprüfung, dass nur eine Aktivität gleichzeitig läuft, reicht nicht!

⇒ **Warteschlange** spielt eine Rolle!

- Methoden:
  - \* `wait()`, `notify()`, `notifyAll()` nur im **synchronized-Block**!
    - ⇒ Monitor ist "this" und kann weggelassen werden
    - ⇒ Falls nicht im synchronized-Block: **IllegalMonitorStateException**
- `wait()`:
  - \* Setzt Faden in **Wartezustand bis Signal** eintritt
    - ⇒ IMMER in einer Schleife!
    - ⇒ Bedingung VOR und NACH dem Warten prüfen!
- `notify()`, `notifyAll()`:
  - \* Schicken **Signale an wartende Aktivitäten**
    - ⇒ Sicher ist nur: `notifyAll()`
- Unterbrechnungen
  - Wie beendet man Aktivitäten, die auf nicht mehr eintreffende Signale warten?
    - ⇒ Durch **`interrupt()`**
  - Die Methode **`wait()`** kann auch eine **`InterruptedException`** werfen!
- Verklemmungen (Deadlocks)
  - **Semaphore**, die mit Anzahl an Genehmigungen initialisiert werden
    - \* **`acquire()`**: Blockiert bis Genehmigung verfügbar und dann Genehmigungen–
    - \* **`release()`**: Genehmigungen++
  - **CyclicBarrier**, die Gruppen von Fäden synchronisiert
    - \* Fäden rufen **`await()`** auf, die so lange blockiert, bis alle Fäden warten



### 4.2.3 Bewertung von parallelen Algorithmen

- **Beschleunigung:**

- $S(p) = \frac{T(1)}{T(p)}$

- $S(p)$  = Angabe, wie viel schneller mit p Prozessoren

- **Effizienz:**

- $E(p) = \frac{T(1)}{p \cdot T(p)} = \frac{S(p)}{p}$

- $E(p)$  = Anteil an Ausführungszeit der nützlich verrichteten Arbeit

- \* Ideal:  $S(p) = p$  oder  $E(p) = 1$

- **Gesamtlaufzeit:**

- $T(p) = \sigma + \frac{\pi}{p}$

- $\sigma$  = Zeit für Ausführung des sequentiellen Teils

- $\pi$  = Zeit für die sequentielle Ausführung des parallelen Teils

- $p$  = Anzahl an CPUs

- **Amdahl'sches Gesetz:**

- $S(p) \leq \frac{1}{f}$  mit  $f = \frac{\sigma}{\sigma + \pi}$

## 5 Testphase

Ziel:

- Softwarefehler möglichst **früh finden**
  - **Zeit ist Geld!**

### 5.1 Fehlerarten

- **Irrtum/Herstellungsfehler:** Menschliche Aktion, die zum Defekt führt
- **Defekt:** Mangel an Softwareprodukt
- **Versagen/Ausfall:** Abweichung des Softwareverhaltens

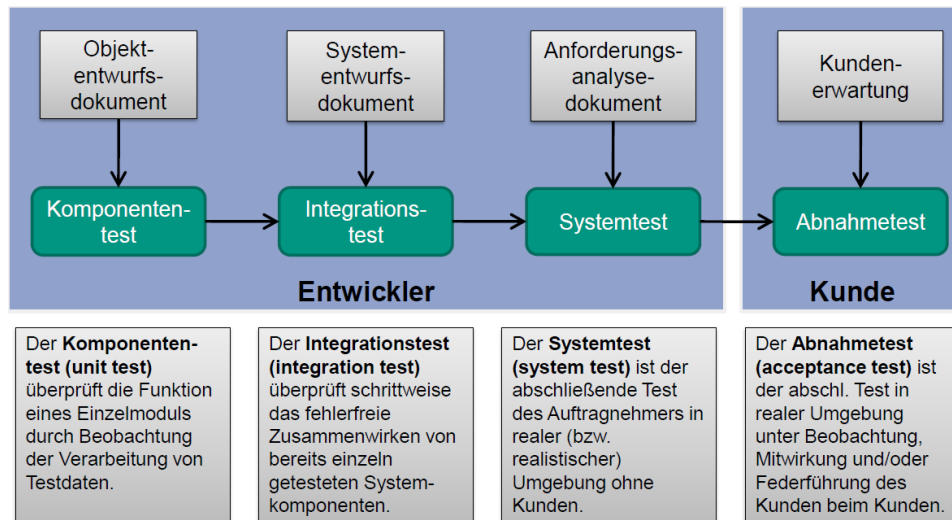
### 5.2 Fehlerklassen

- **Anforderungsfehler** (Defekt im Pflichtenheft)
  - Inkorrekte Benutzerwünsche,..
- **Entwurfsfehler** (Defekt in der Spezifikation)
  - Unvollständige/fehlerhafte Umsetzung der Anforderung,..
- **Implementierungsfehler** (Defekt im Programm)
  - Fehlerhafte Umsetzung der Spezifikation im Programm,..

### 5.3 Arten von Testhelfern

- **Stummel:** Rudimentär implementierter Softwareteil
- **Attrappe:** Simuliert die Implementierung zu Testzwecken
- **Nachahmung:** Attrappe mit zusätzlicher Funktion

## 5.4 Testphasen



## 5.5 Klassifikation testender Verfahren

- **Dynamische Verfahren**
  - Strukturtests
    - \* Kontroll- und datenflussorientierte Tests
  - Funktionale Tests
  - Leistungstests
- **Statische Verfahren**
  - Manuelle Prüfmethoden
  - Prüfprogramme

### 5.5.1 Kontrollflussorientierte Testverfahren

- Anweisungsüberdeckung

- Ausführung aller **Grundblöcke**

- Zweigüberdeckung

- **Traversierung** aller Zweige

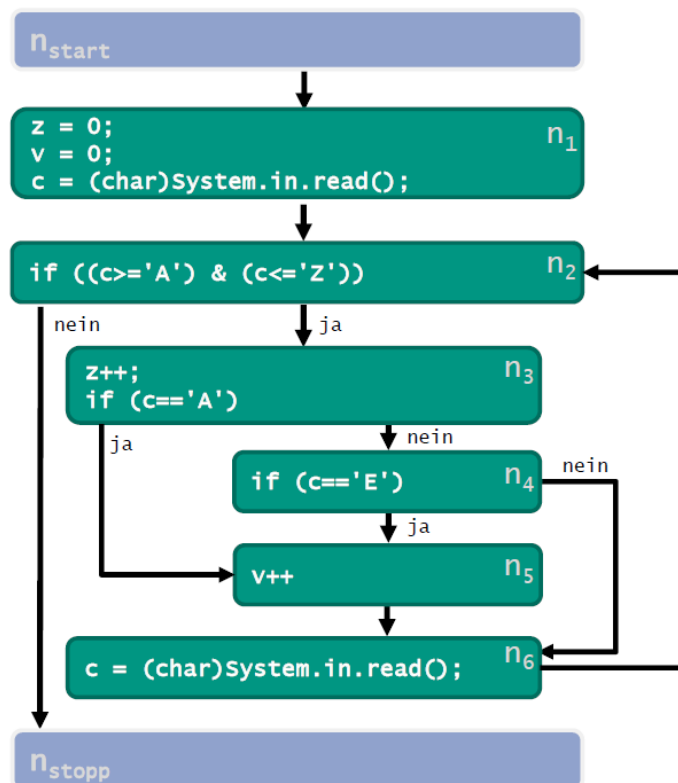
- Pfadüberdeckung

- Ausführung aller **unterschiedlichen, vollständigen Pfade** im Programm

\* Pfadanzahl **wächst bei Schleifen** enorm!

⇒ **Nicht praktikabel!**

- Beispiel:



### 5.5.2 Funktionale Tests

- **Funktionale Äquivalenzklassenbildung**
  - **Zerlege Wertebereich** der Eingabeparameter und **Definitionsbereich** der Ausgabeparameter in Äquivalenzklassen
- **Grenzwertanalyse**
  - Erweiterung von Äquivalenzklassenbildung mit **Grenzwerten**
- **Zufallstest**
  - **Zufällige Testfälle** (mit Testhelfern)
- **Test von Zustandsautomaten**
  - Testfälle aus **Zustandsübergängen**

### 5.5.3 Leistungstests

- **Lasttests**
  - Zuverlässigkeit und Einhalten der Spezifikation **im erlaubten Grenzbereich**
- **Stresstests**
  - Verhalten des System **beim Überschreiten** der definierten Grenzen

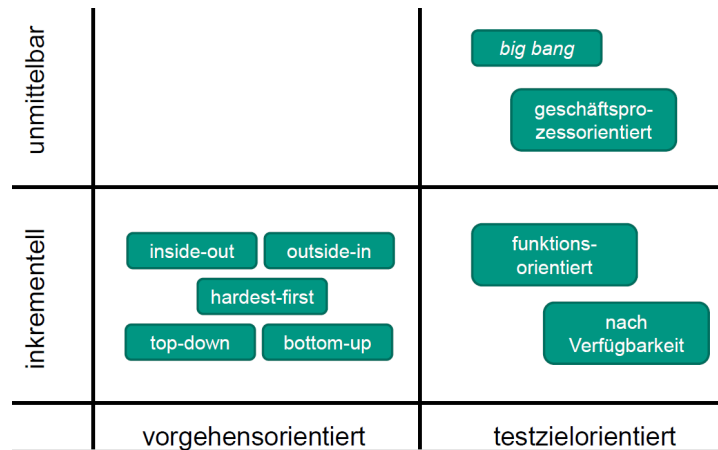
### 5.5.4 Manuelle Prüfung

- **Semantik** wird geprüft
- **Aufwendig** (20% der Erstellungskosten)

### 5.5.5 Prüfprogramme

- Warnungen, Fehler, Programmierstil, etc.

### 5.5.6 Integrationstests



### 5.5.7 Systemtests

- **Funktionaler Systemtest**

- Überprüfung funktionaler Qualitätsmerkmale, Korrektheit und Vollständigkeit

- **Nichtfunktionaler Systemtest**

- Überprüfung nichtfunktionaler Qualitätsmerkmale wie: Sicherheit, Benutzbarkeit,...

### 5.5.8 Abnahmetests

- Spezieller Systemtest: **Kunde beobachtet** oder **wirkt mit!**
- Formale Abnahme ist **bindende Erklärung der Annahme** durch den Auftraggeber

## 5.6 Inspektion

- **Phasen:**

- **Vorbereitung**

- \* **Teilnehmer/Rollen** festlegen
    - \* **Dokumente/Formulare** vorbereiten
    - \* **Zeitlichen Ablauf** planen

- **Individuelle Fehlersuche**

- \* **Inspektoren prüfen** Dokumente für sich
    - \* **Notieren der Problempunkte** und genaue Stelle im Dokument
    - \* Problempunkte: Mögliche Defekte, Verbesserungsvorschläge, Fragen

- **Gruppensitzung**

- \* **Problempunkte sammeln** und besprechen
    - \* **Verbesserungsvorschläge sammeln**

- **Nachbereitung**

- \* Liste der Problempunkte an **Editor**
    - \* Editor identifiziert **tatsächliche Defekte** und **klassifiziert** sie
    - \* **Alle Problempunkte** werden bearbeitet

- **Prozessverarbeitung**

- \* **Standards für Dokumente** erarbeiten
    - \* Defektklassifikationsschema, Planung und Durchführung verbessern

• **Rollen:**

- **Inspektionsleiter** (leitet alle Phasen)
- **Moderator** (leitet Gruppenphase)
- **Inspektor** (prüft Dokument)
- **Schriftführer** (protokolliert Defekte in Gruppensitzung)
- **Editor** (klassifiziert/behebt Defekte)
- **Autor** (verfasst Dokument)

+	-
- Anwendbar auf alle Softwaredokumente - Effektiv in industrieller Praxis	- Aufwendig - Teuer, da Zeitaufwand hoch



## 6 Abnahme- Einführungs-, Wartungs- und Pflegephase

### 6.1 Abnahmephase

- **Übergabe des Gesamtprodukts** inklusive vollständiger Dokumentation
  - Verbunden: **Abnahmeprotokoll und -test**

### 6.2 Einführungsphase

- **Installation** des Produkts
- **Schulung** der Benutzer und des Benutzerpersonals
- **Inbetriebnahme** des Produkts:
  - Direkte Umstellung
  - Parallellauf
  - Versuchslauf

### 6.3 Wartungs- und Pflegephase

- Kategorien:
  - Korrektive Tätigkeiten (**Wartung**)
    - \* Stabilisierung/Korrektur
    - \* Optimierung/Leistungsverbesserung
  - Progressive Tätigkeiten (**Pflege**)
    - \* Anpassung/Änderung
    - \* Erweiterung

## 7 Aufwandsschätzung

### 7.1 Schätzmethoden

- Analogiemethode
  - Vergleiche die zu schätzenden Entwicklungen mit **bereits abgeschlossenen Produktentwicklungen** anhand von **Ähnlichkeitskriterien**
- Basismethoden
  - Relationsmethode
    - \* Methode, die anhand von **Faktoren** (Programmiersprache/-erfahrung, Dateiorganisation) vergleicht, wie diese den **Aufwand beeinflussen: Auf- und Abschläge** mit etwa gleich großem, existierenden Produkt
  - Multiplikatormethode
    - \* **Zerlegung in Teilprodukte** mit Zuteilung feststehender Aufwände  
 $\Rightarrow \text{Anzahl Teilprodukte} \cdot \text{Aufwand Kategorie}$
  - Phasenaufteilung
    - \* Ermittlung aus abgeschlossenen Entwicklungen werden **auf einzelne Entwicklungsphasen verteilt** (Kuchendiagramm)
- COCOMO II
  - $PM = A \cdot (Size)^{1,01+0,01 \cdot \sum_{j=1}^5 SF_j} \cdot \prod_{i=1}^{17} EM_i$ 
    - \*  $PM$  = Personenmonate
    - \*  $A$  = Konstante für Kalibrierung des Modells (z.B. LOC)
    - \*  $Size$  = Geschätzter Umfang der Software in KLOC
    - \*  $SF_j$  = Skalierungsfaktoren
    - \*  $EM_i$  = Multiplikative Kostenfaktoren

## 8 Prozessmodelle

- **Programmieren durch Probieren** (Trial & Error)

- **Programm erzeugen und danach alles weitere planen, testen, warten,...**

+		-	
- Schnell		- Schlecht strukturiert	
- Ohne nutzlosen Zusatzaufwand		- Keine Dokumentation	

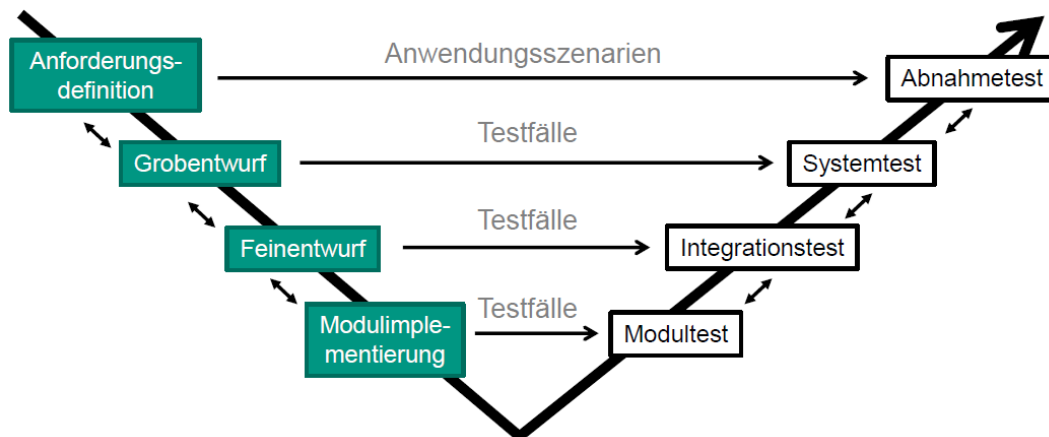
- **Wasserfallmodell**

- **Dokumentgetriebenes Modell**
- **Jede Aktivität in fester Reihenfolge** und anschließendem Dokument

+		-	
- Einfach		- Entwurf, Implementierung und Testen von nutzlosem Code	
- Verständlich		- Keine Rückkopplung	

- **V-Modell 97** (Vorgehensmodell)

- Jede Aktivität hat **eigenen Prüfungsschritt**



- **V-Modell XT**

- Entwicklungsstandard für IT-Systeme der **öffentlichen Hand**
- **Aktivitäten, Produkte und Verantwortlichkeit** werden festgelegt, jedoch keine Reihenfolge
- Aufteilung in **vier Submodelle** mit zusätzlichen **Vorgehensbausteinen**:
  - \* **Projektmanagement**
  - \* **Qualitätssicherung**
  - \* **Konfigurationsmanagement**
  - \* **Systemerstellung**

- **Prototypmodell**

- Geeignet für Systeme, für die keine vollständige Spezifikation **ohne explorative Entwicklung/Experimentation** erstellt werden kann
- Prototyp wird **WEGGEWORFEN!**

- **Iteratives Modell**

- **Teile der Funktionalität** lassen sich klar definieren/realisieren
  - \* Funktionalität wird **Schritt für Schritt** hinzugefügt

- **Synchronisiere und Stabilisiere**

- Idee:

Programmierer in **kleinen Teams**  
↓  
Regelmäßig **synchronisieren** (nächtlich)  
↓  
Regelmäßig **stabilisieren** (3-Monate)

– **Phasen:**

\* **Planungsphase** (3-12 Monate)

- Wunschbild, Spezifikation, Zeitplan und Teamstruktur

\* **Entwicklungsphase** (6-12 Monate)

- Manager koordinieren, Entwickler entwerfen und Tester testen parallel

\* **Stabilisierungsphase** (3-8 Monate)

- Manager koordinieren Beta-Tester und sammeln Rückmeldungen
- Entwickler stabilisieren Code
- Tester isolieren Fehler

+	-
<ul style="list-style-type: none"><li>- Effektiv durch kurze Produktzyklen</li><li>- Fortschritt ohne vollständige Spezifikation</li></ul>	<ul style="list-style-type: none"><li>- Ungeeignet für manche Art von Softwareproblemen</li><li>- Mangelnde Fehlertoleranz oder Echtzeitfähigkeit</li></ul>

## 8.1 Agile Prozessmodelle

• Idee:

- **Minimum** an Vorausplanung
- Planung erfolgt **inkrementell**
- **Schnelle Reaktion** auf Änderung

- Vertreter:
  - **Extreme Programming** (XP)
  - **Scrum!**
  - Crystal
  - Adaptive Software Development
  - Feature-Driven Development
  - Software Expedition

### 8.1.1 Extreme Programming

- **Paarprogrammierung**
  - Zwei Entwickler an einer Maus/Tastatur
  - Geeignet für:
    - \* **Vage und schnell ändernde** Anforderungen
    - \* **Kleines** Entwicklerteam
    - \* **Wenig** Verwaltungsaufwand

+	-
- Bessere Qualität des Quellcodes - Gut für unerfahrene Entwickler	- Doppelte Kosten - Vorteil gegenüber einzelner Programmierung mit Inspektionen nicht nachweisbar

### 8.1.2 Scrum

- Vorgehensmodell: **Agiles Projektmanagement**
- **Artefakte:**
  - **Anforderungsliste** (Product backlog)
    - \* Produktanforderungen und Liste aller Projektarbeiten
    - \* Anforderungen vor jedem Sprint priorisiert
  - **Aufgabenliste** (Sprint backlog)
    - \* Alle Aufgaben mit Beschreibungen für aktuellen Sprint
  - **Hindernisliste** (Impediment backlog)
    - \* Alle Hindernisse des Projekts, die Scrum-Master mit Team bespricht
- **Rollen:**
  - **Auftraggeber** (Product owner)
    - \* Legt Anforderungen/Auslieferungstermin fest
    - \* Stellt Budget
    - \* Priorisiert Anforderungen für Sprint
  - **Scrum-Master**
    - \* Sicherstellung der Scrumwerte und -techniken
    - \* Sorgt sich um vollständiges, funktionsfähiges und produktives Team
    - \* Beseitigt Hindernisse und kommuniziert zwischen den Rollen
  - **Entwicklungsteam**
    - \* Personen unterschiedlicher Fachrichtungen

- **Treffen:**

- **Sprintplanung**

- \* Entwicklungsteam wählt machbare Anforderungen aus, die im Sprint geschafft werden kann und erstellt mit Scrum-Master eine Aufgabenliste

- **Tägliches Scrumtreffen** (Daily Scrum)

- \* "Was hast du gestern getan?"
    - \* "Was wirst du heute tun?"
    - \* "Welche Hindernisse gibt es?"

- **Reviewtreffen** (Sprintende)

- \* Vorstellen der Sprintergebnisse
    - \* Keine Folien!

- **Retrospektive**

- \* Analyse vom letzten Sprint mit dem Entwicklungsteam, Scrum-Master, Auftraggeber und eventuell dem Endkunden



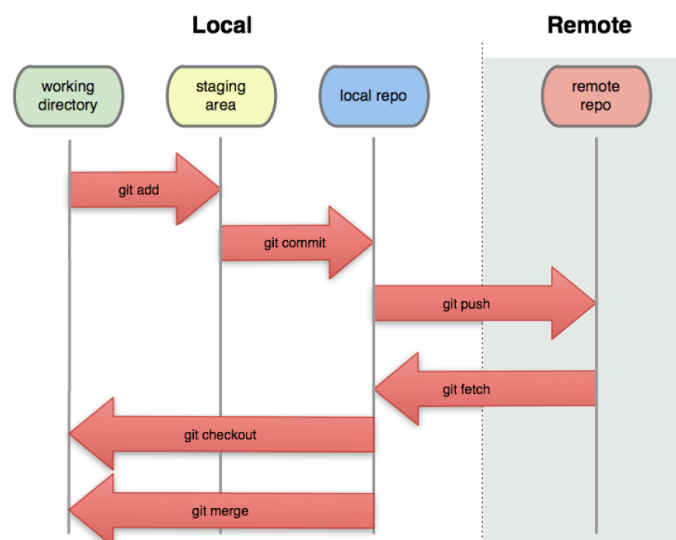
## 9 Werkzeugkette und Versionskontrolle

### 9.1 Befehlskette

git <Kommando>:

- **help:** Liste der Kommandos
- **init:** Depot anlegen
- **clone:** Depot eines Projekts laden
- **add:** Neue Datei hinzufügen und in Staging Area übernehmen
- **commit:** Änderungen in das eigene Depot übernehmen
- **push:** Commits in ein anderes Depot übertragen
- **fetch:** Änderungen von einem anderen Depot holen
- **merge:** Änderungen von einem Branch in einen anderen übertragen
- **pull:** Ausführen von fetch und merge

### 9.2 Interaktion innerhalb Local- und Remote Repository



## 9.3 Unterschied: GIT und SVN

GIT	SVN
<ul style="list-style-type: none"><li>- Depot ist für jeden User <b>lokal</b></li><li>- Operationen <b>offline ausführbar</b></li><li>- <b>Kryptografische Sicherung</b> der Historie</li><li>- Speichert <b>Schnappschüsse</b></li></ul>	<ul style="list-style-type: none"><li>- Depot wird <b>auf einem Server</b> abgelegt</li><li>- <b>Optimistisches Ausbuchen</b></li><li>- Versioniert gesamtes Depot</li><li>- Speichert <b>Deltas</b></li></ul>

## 9.4 Versionskontrolle

### 9.4.1 Vorwärtsdelta

**Anfangsversion als Ausgangspunkt** und alle Deltas danach werden gespeichert.

### 9.4.2 Rückwärtsdelta

**Neueste Version als Ausgangspunkt** und alle Deltas davor werden gespeichert.

	+	-
Vorwärtsdelta	Schneller Zugriff auf alte Version	Langsamer Zugriff auf alte Version
Rückwärtsdelta	Schneller Zugriff auf alte Version	Langsamer Zugriff auf neue Version

### 9.4.3 Ein- und Ausbuchen

- Optimistisch: **Mehrfaches Ein- und Ausbuchen ohne Änderungsreservierung**
- Strikt: Ein- und Ausbuchen **mit Änderungsreservierung**

## 10 Quellenverzeichnis

### 10.1 Literatur

- Alle Informationen wurden den Vorlesungsfolien aus dem Kurs Softwaretechnik 1 vom Sommersemester 2018 des Karlsruher Instituts für Technologie entnommen.

### 10.2 Grafiken

- Alle Grafiken wurden den Vorlesungsfolien aus dem Kurs Softwaretechnik 1 vom Sommersemester 2018 des Karlsruher Instituts für Technologie entnommen.