

# Software Engineering

## 1 Überblick

---

- Warum brauchen wir SE
- 

### 1.1 Was ist Software Engineering

**IEEE:**

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches.

**Balzert:**

Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen.

⇒ Es gibt also 2 Hauptaspekte:

- Ingenieurdisziplin
- Softwaresysteme

**Manifest der Softwaretechnik:**

Software Engineering zielt auf die ingenieurmäßige Entwicklung, Anpassung und Weiterentwicklung großer Softwaresysteme unter Verwendung bewährter systematischer Vorgehensweisen, Prinzipien, Methoden und Werkzeuge.

### 1.2 Warum ist Software Engineering wichtig?

**Immer mehr Systeme werden durch Software gesteuert:**

- Die Wirtschaft aller industrialisierten Länder hängt von Software ab.
- Die Aufwendungen für Software repräsentieren einen enormen Faktor im Bruttonzialprodukt aller Länder

**Managen von Softwarekosten:**

- Softwarekosten dominieren oft Systemkosten (Bsp.: Kosten der Software eines PCs oft höher als die Hardwarekosten)
- Wartung von Software ist teurer als ihre Entwicklung

**Verschärfung der Probleme:**

- Die Komplexität von Programmen nimmt ständig zu
- Umfang und Lebensdauer nehmen zu
- Neue Anwendungen werden für den Rechnereinsatz erschlossen
- Die Softwareentwicklung ist ein integraler Bestandteil der Systementwicklung
- Immer mehr Entwickler sind mit der Pflege von Altsystemen beschäftigt
- Softwarefehler sind allgegenwärtig

### 1.3 Soll-Eigenschaften guter Software

- Zuverlässig:
  - Software muss im Fall des Versagens physische oder ökonomische Schäden vermeiden
- Wartbar:
  - Software muss leicht erweiterbar und anpassbar an neue Anforderungen sein
  - Software sollte möglichst Plattform-unabhängig sein
  - Software-Code muss gut dokumentiert und lesbar sein
- Benutzerfreundlich:
  - Software muss sich nach Bedürfnissen von Nutzern richten
  - Benutzerschnittstelle (z.B. GUI) muss ergonomisch und intuitiv sein
  - Softwareprodukt muss gut dokumentiert sein
- Performant:
  - Software muss geforderte Funktionalität schnellstmöglich erbringen
- Effizient:
  - Software muss ressourcenschonend gegenüber Host-System sein

### 1.4 Ziele des Software Engineering

- Qualität:
  - Definition: Software-Produkt-Qualität ist die Gesamtheit von Eigenschaften eines Software Produkts, die sich auf die Eignung zur Erfüllung gegebener Erfordernisse beziehen.
  - Erfordernisse des Anwenders sind entscheidend
- Kosten:
  - Entwicklungskosten
    - Personal
    - Entwicklungsplatz, Rechner, Stromkosten, ...
  - Fehlerbeseitigung
  - Anpassung, Erweiterung
  - Organisatorische Implementierung
    - Customizing (Beispiel: SAP)
    - Schulung
  - Vertriebskosten
    - Marketing, Werbung, Sales, Promotion
    - Personal Selling
  - **Wartungskosten** ⇒ machen 2/3 der Kosten aus
- Ziele bezgl. Verfügbarkeit und Einsatzdauer:
  - Kurze Entwicklungsdauer ⇒ schnelle Marktpräsenz (time-to-market)
  - Konflikt mit Qualität ⇒ sog. "Bananenprodukte" (d.h., sie reifen beim Kunden)
  - Marktpräsenz/Einsatzdauer maximieren:
    - Lizenzennahmen (sog. "Milchkühe", "Cash-Cows")
    - Amortisation (nach welcher Zeit hat sich Investition rentiert?)
  - **Softwareentwicklung muss planbar sein**
- Umsatz und Kosteneinsparung:
  - Softwarelieferant (-produzent)
    - Lizenzennahmen ( $\text{Verkaufszahl} \cdot \text{Preis}$ )
    - Lizenzgebühr für Nutzungsrecht
    - Wartungsgebühr
    - Beratung, Schulung

- Kunde/Anwender
  - Kosteneinsparungen durch Rationalisierung
  - Stichwort: Amortisation der Anschaffungskosten
  - Schwer zu messende/ zu quantifizierende Komponenten:
    - Schnellere Reaktionszeit (Anfrage ... Lieferung)
    - Bessere Information
    - Zufriedenere Endkunden

## 1.5 Zusammenfassung

**Software nimmt immer mehr Bedeutung in allen Bereichen des Alltags ein:**

- Software als Wirtschaftsgut
- Anteil Softwarekosten gegenüber Hardwarekosten steigt stark
- Anteil der Software an der Wertschöpfung steigt erheblich (z.B. Automotive, Medizin) Bruttosozialprodukt aller Länder

**Software Engineering vs Programmieren:** (s. letztes Semester)

- Projekt-Management (großer und komplexer Projekte)
- Schätzung von Terminen und Kosten
- Erfassen von Kunden-, Markt- und Gesetzesanforderungen
- Änderungsmanagement
- Qualitätsmanagement (hohes Qualitätsniveau sicherstellen)
- Wartung und Weiterentwicklung von Produktivsystemen
- Pflege eines guten Programmierstils
- Entwicklungswerzeuge
- Basis-Prinzipien (Abstraktion, Strukturierung, Hierarchiebildung, Modularisierung)
- ...

⇒ **Softwareentwicklung ist mehr als nur Programmieren!**

**Aktivitäten und Ziele des Software Engineering:**

- Aktivitäten
  - Planung
  - Entwicklung (inkl. Anforderungsanalyse)
  - Betrieb
  - Pflege, Wartung
- Entwicklung qualitativ hochwertiger Software durch
  - Prinzipien
  - Methoden
  - Vorgehensweisen
  - Werkzeuge
  - Quantifizierbare Ziele
- Software Engineering: Lehre von der ingenieurmäßigen Entwicklung von Software und Softwarebasierten Systemen
- Vielfalt der Problembereiche, Projektgröße erfordert Portfolio an Techniken
- Kritischer Faktor ist Softwarequalität
- Software Engineering ist praxisnah:
  - ⇒ **Methoden und Techniken müssen geübt und angewandt werden**

## 2 Ethik

Ausgeschlossen

### 3 Projektphasen

- Überblick über SE und die Phasen

#### 3.1 Wie kommt man überhaupt zu einem Projekt

##### 1. Der Projektantrag

- Ziel: Darlegung der Projektidee in entsprechenden Gremien
- Inhalt: Darlegung häufig in werbenden Präsentationen
  - Ausgangslage
  - Ziele
  - Kosten/Nutzen
  - Organisation

##### 2. Der Projektauftrag

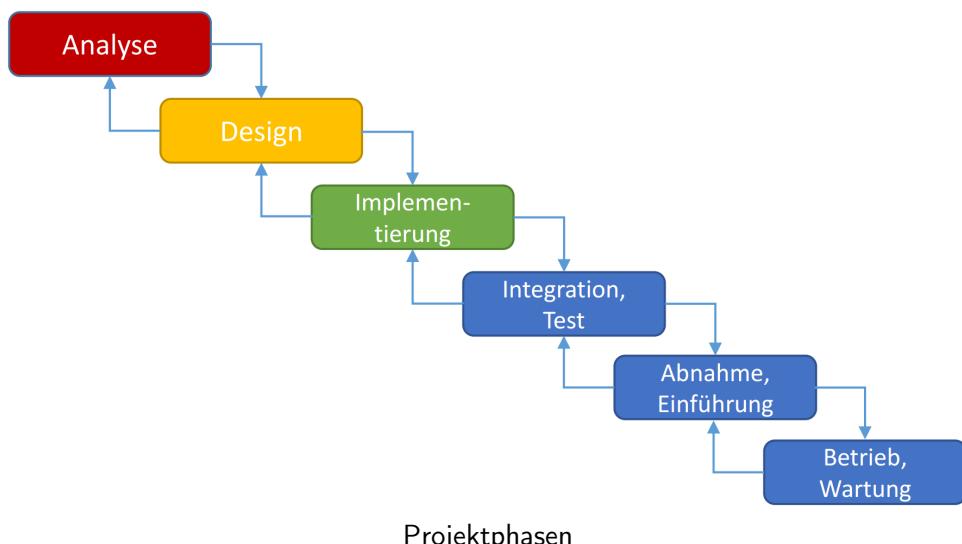
- Inhalt:
  - Projektbezeichnung
  - Auftraggeber
  - Projektbeginn und -ende
  - Kurzbeschreibung, Unternehmensbedarf und Ziele
  - Projektergebnisse
  - Projektbudget
  - Projektleiter und Projektteam
  - Annahmen und Beschränkungen
  - Terminvorgaben

⇒ Es gilt: **KEIN PROJEKT OHNE PROJEKTAUFRAG!**

##### 3. Der Projektantrag und -auftrag

- Sehr unterschiedliche Handhabung im Umgang mit Projektantrag und Projektauftrag
- Er hängt von Organisationsstruktur und Kultur des Unternehmens ab
- Bei mittelgroßen/großen Unternehmen gibt es normalerweise Vorlagen für den Projektauftrag

#### 3.2 Phasenübersicht eines Softwareprojekts



### **3.2.1 Analyse**

- Aufstellen von Leistungen, Einschränkungen und Zielen des Systems
- in Zusammenarbeit mit den Systembenutzern/dem Kunden
- anschließend detailliertere Beschreibung und Definition
- Ergebnis dient als Systemspezifikation

### **3.2.2 Design (Entwurf)**

Anforderungen werden in Hard- oder Softwaresysteme aufgeteilt in:

- dazu wird übergeordnete Systemarchitektur festgelegt
- beim Software Design geht es um Erkennen und Beschreiben
  - der grundlegenden (abstrakten) Softwaresysteme und
  - deren Beziehung zueinander

### **3.2.3 Implementierung (und Modultests)**

- Umsetzung des Softwareentwurfs durch Programme oder Programmeinheiten
- Testen der Module auf Einhaltung Ihrer Spezifikation

### **3.2.4 Test und Integration**

- Zusammenführung/Integration einzelner Programme oder Programmeinheiten
- Test des "Ganzen" auf Einhaltung der Anforderungen
- Auslieferung an Kunden

### **3.2.5 Abnahme und Einführung**

- Abnahmetests in mehreren Stufen
  - Technische Tests
  - Last-/Performancetests
  - Fachliche Tests
  - User-Acceptance Tests
  - ...
- Auslieferung an den Kunden
- ggfs Anwenderschulungen, Informationsmaßnahmen bis hin zu Marketing für die neue Software

### **3.2.6 Betrieb und Wartung**

- längste Phase im Software-Lifecycle
- System wird installiert und zur Nutzung freigegeben
- Elemente der Wartung
  - Beheben von Fehlern
  - Verbesserung der Implementierung von Systemeinheiten
  - Verbesserung des Systems, falls neue Anforderungen entstehen
- Teil der Wartung häufig noch in Projektkosten enthalten
- später Wartungsverträge

### 3.2.7 Allgemeine Hinweise

- Phasen nicht notwendigerweise sequentiell (s. Kapitel Vorgehensmodelle)
- Tätigkeiten innerhalb der Phasen erfordern verschiedene Qualifikationen
- ABER: nicht immer ist (vollständige) Durchführung aller Phase sinnvoll
- Tätigkeiten müssen koordiniert werden → Projektmanagement
- Softwareprojekte laufen immer in Teams ab (sowohl innerhalb des Unternehmens als auch mit Kooperationspartnern/Kunden)

⇒ Kommunikation innerhalb des Teams und zwischen Team und Kunden ist extrem wichtig

## 4 Objekt Orientierte Analyse

---

- Anforderungen
  - Eigenschaften die Anforderungen
- Ziel
- Aktivitäten von Phasen
- Doku
  - Pflichtenheft vs Lastenheft
- Qualifikation von Anforderungen
  - Kano Modellieren
  - Risiken
  - best practices
- Makro Prozess
  - Dynamisches/statisches Modell
- Modelle
  - Use Case Diagramm
  - Aktivitätsdiagramm (mit token)
  - Zustandsdiagramm
  - Klassendiagramm (mit Objektdiagramm)
  - Komponentendiagramm
  - Wozu sind die da und in welcher Phase werden die gebraucht?
  - Welche Fragen klären die Diagramme, sind die wichtig?
- Architekturdiagramm vs Verhaltensdiagramme
- Analysemuster
  - Erklären können + Beispiele

## 4.1 Requirements Engineering - Anforderungsanalyse

Motivation: Was soll realisiert werden?

### 4.1.1 Beispiel 2: wie es nicht geht

Ein realistischeres weiteres kleines Beispiel:

#### 1. Ist-Zustand

- Zur Stundenerfassung und Abrechnung werden von den Projektmitarbeitern spezielle Excel-Tabellen jeden Freitag ausgefüllt und am Montag vom Projektleiter bei der Verwaltung abgegeben.
- Der zuständige Sachbearbeiter überträgt dann die für den Projektüberblick relevanten Daten manuell in ein SAP-System. Dieses System generiert automatisch eine Übersicht, aus der die Geschäftsführung ablesen kann, ob die Projekte wie gewünscht laufen.
- Dieser Bericht liegt meist am Freitag der Folgewoche vor. Die Bearbeitungszeit ist der Geschäftsführung zu lang, deshalb soll der Arbeitsschritt automatisiert werden.

#### 2. Projektplanung

- Es wird ein Projekt "Projektberichtsautomatisierung" (ProAuto) beschlossen.
- Der Leiter der hausinternen IT-Abteilung wird über die anstehende Aufgabe informiert. Er erhält eine Beschreibung der Excel-Daten und der gewünschten SAP-Daten.
- Der Leiter stellt fest, dass seine Abteilung das Know-how und die Kapazität hat, das Projekt durchzuführen und legt der Geschäftsführung einen Projektplan mit einer Aufwandsschätzung vor.
- Die Geschäftsführung beschließt, das Projekt intern durchzuführen zu lassen und kein externes Angebot einzuholen.

#### 3. Die Schritte zum Projektmiss Erfolg

- Die IT-Abteilung analysiert die Excel-Daten und wie die Daten in das SAP-System eingefügt werden können.
- Kurz nach dem geschätzten Projektende liegt eine technisch saubere Lösung vor. Excel wurde um einen Knopf erweitert, so dass die Projektleiter per Knopfdruck die Daten nach SAP überspielen können.
- Vier Wochen nach Einführung des Systems wird der Leiter der IT-Abteilung entlassen, da die Daten zwar jeden Montag vorliegen, sich aber herausgestellt hat, dass sie nicht nutzbar sind und die erzürnte Geschäftsleitung falsche Entscheidungen getroffen hat. Das Projekt wird an eine Beratungsfirma neu vergeben.

Wie es besser geht: Geschäftsprozessanalyse:

- Ist-Zustand modellieren
- Ist-Zustand analysieren und optimieren
- Soll-Zustand modellieren
- Prozess implementieren

### 4.1.2 Analysephase (RE + OOA)

Ziel:

- Klärung was realisiert werden soll

Aktionen:

- Klärung der Visionen und Ziele
- Definition des Systemkontextes (Einsatzfeld, Benutzer, Systemumgebung)
- Analyse des Ist-Zustandes
- Analyse der Machbarkeit (Risiko, Kosten)
- Bestimmung der Anforderungen (funktional vs. nicht-funktional)

Ergebnis:

- Pflichtenheft
- Konzept der Benutzeroberfläche (frühes Feedback)
- Analysemodell (statisch und dynamisch)

### Requirements Engineering als erster Schritt der Analysephase :

RE frei nach Sommerville:

*Requirements Engineering ist der Prozess des Herausfindens, Analysierens, Dokumentierens und Überprüfens der Dienste und Beschränkungen eines zu erstellenden Systems.*

Requirements Engineering ist kooperativer, iterativer, inkrementeller Prozess mit Ziel:

- Alle relevanten Anforderungen sind bekannt und in erforderlichem Detaillierungsgrad verstanden
- Involvierte Stakeholder stimmen über Anforderungen überein
- Anforderungen sind konform zu Dokumentationsvorschriften dokumentiert bzw. zu Spezifikationsvorschriften spezifiziert.

Eigenschaften von Anforderungen: sie müssen ...

- vollständig
- notwendig ("WAS statt WIE")
- eindeutig
- richtig (abgestimmt als Teil einer Zielhierarchie") sein.

#### 4.1.3 Bedeutung des Requirement Engineerngs

Hauptgründe für Projektabbruch:

1. Änderung der Anforderungen und des Umfangs
2. Mangelnde Einbindung des höheren Managements
3. Engpass im Budget
4. Fehlende Projektmanagement-Fähigkeiten

Anforderungen müssen:

- ermittelt:
  - Systemkontext festlegen, Anforderungen ermitteln
- dokumentiert:
  - adäquat beschreiben
  - geprüft und abgestimmt
  - prüfen sowie abstimmen und dadurch Qualitätssichern
  - verwaltet werden
  - Änderungen sind nachzuverfolgen

⇒ Aktivitäten des RE

## **Ergebnis:**

- Anforderungsspezifikation (Lasten-, Pflichtenheft):
  - enthält alle ermittelten, spezifizierten, analysierten
  - und validierten Anforderungen aus Kundensicht.
- Fachliche Lösung bzw. Produktmodell:
  - Beschreit die analysierte und verifizierte fachliche
  - Lösung aus Auftragsnehmersicht

⇒ Artefakte des RE

## **Aufbau eines Requirement Dokuments nach IEEE 830-1998**

1. Einleitung
  - Zielsetzung
  - Produktziele
  - Definitionen, Akronyme, Abkürzungen
  - Referenzen
  - Überblick
2. Übersichtsbeschreibung
  - Produkt-Umgebung
  - Produkt-Funktionen
  - Benutzer-Eigenschaften
  - Restriktionen
  - Annahmen und Abhängigkeiten
3. Spezifische Anforderungen
  - Externe Schnittstellenanforderungen
  - Funktionale Anforderungen
  - Leistungsanforderungen
  - Entwurfsrestriktionen
  - Eigenschaften des Softwaresystems
  - Andere Anforderungen

### **4.1.4 Visionen, Ziele und Rahmenbedingungen**

#### **Vision:**

- beschreibt, was erreicht werden soll, aber nicht wie
- hat geringe Detailtiefe; dient als Leitgedanke
- wird im Rahmen eines Projektauftrags formuliert, der genehmigt werden muss

Dadurch können Anforderungen später immer mit den Visionen und Zielen abgeglichen werden. **Denkfalle des Auftraggebers:**

Visionen und Ziele werden sich im Laufe der Anforderungserstellung schon ergeben.

**Ziel:** ausgehend von einer Vision dienen Ziele dazu, die Vision zu verfeinern und zu operationalisieren  
Beispiel: Ziel für Fensterheber Die erwarteten Stückzahlen betragen 20.000 Einheiten p.a.

#### **Regeln zur Zielformulierung:**

- kurz und prägnant formulieren
- Aktivformulierung
- überprüfbare und realistische Ziele formulieren
- nicht überprüfbare Ziele nicht verfeinern
- Mehrwert eines Ziel hervorheben
- Begründung für das Ziel liefern
- keine Lösungsansätze formulieren

#### 4.1.5 Rahmenbedingungen

##### Rahmenbedingung:

- legt organisatorische und technische Restriktionen für das Softwaresystem bzw. den Entwicklungsprozess fest
- organisatorische Rahmenbedingungen:
- Anwendungsbereiche (z.B. Textverarbeitung im Büro)
- Zielgruppen (z.B. Sekretärinnen, Schreibkräfte)
- Betriebsbedingungen (z.B. Büroumgebung, mobiler Einsatz)
- technische Rahmenbedingungen:
- technische Produktumgebung
- Anforderungen an die Entwicklungsumgebung

##### Rahmenbedingungen können:

- keine der Anforderungen einschränken
- die mögliche Realisierung von Anforderungen einschränken
- zu Änderung von Anforderungen führen
- zu nicht realisierbaren Anforderungen führen

##### Denkfalle des Auftraggebers:

- Rahmenbedingungen legt der Auftragnehmer fest.

#### 4.1.6 Anforderungen

Balzert:

*Anforderungen legen fest, was man von einem Softwaresystem als Eigenschaften erwartet“*

##### Häufig verwendete Unterkategorisierung:

- Funktionale Anforderungen:
  - beschreiben, was ein System tun soll
  - Dienste, die das System zur Verfügung stellt
  - Reaktion auf bestimmte Eingaben
  - Verhalten in bestimmten Situationen
- Nicht-funktionale Anforderungen:
  - Beschränkungen
  - Beziehen sich eher auf das Gesamtsystem als auf einzelne Dienste

**Merke:** nicht die Unterscheidung in die beiden Arten ist essentiell sondern die Berücksichtigung beider bei der Softwareentwicklung

##### Weitere Kategorisierung von Anforderungen

- Funktionale Anforderungen
  - Funktionen, Daten, Stimuli, Reaktionen, Verhalten, ...
  - Bsp.: Das System muss Kunden-, Firmen-, Seminar-, Veranstaltungs- und Dozentendaten permanent speichern
- Leistungsanforderungen
  - Zeit, Geschwindigkeit, Umfang, Durchsatz, ...
  - Bsp.: Alle Reaktionszeiten auf Benutzeraktionen müssen unter 5 Sek. Liegen
- Qualitätsanforderungen
  - Zuverlässigkeit, Benutzbarkeit, Sicherheit, Portabilität, Wartbarkeit, ...
  - Randbedingungen/ Einschränkungen
  - physikalisch, rechtlich, kulturell, Umgebung, Schnittstellen, ...

##### Beispiele: nicht-funktionale Anforderungen

- Technische Anforderungen:
  - Das System muss mit Java entwickelt werden und in der Version "Oracle Java 1.8" laufen
- Ergonomische Anforderungen:
  - Die Benutzerführung erfolgt in deutsch
- Anforderungen an die Dienstqualität:
  - Das System muss jede Anfrage des Benutzers innerhalb von 2 Sekunden beantworten
  - Der Speicherbedarf darf 2 GB nicht übersteigen
  - Das System muss 8 Anfragen pro Sekunde beantworten können
- Anforderungen an die Zuverlässigkeit:
  - Der Dienst muss eine Verfügbarkeit von 999/1000 haben
- Anforderungen an den Entwicklungsprozess:
  - Entwickler muss mit Kunden monatliche Reviews der zu erstellenden Dokumente machen
- Rechtlich-vertragliche Anforderungen:
  - Der Kunde leistet für jeden Meilenstein ein Viertel der vertraglich für die Systementwicklung vereinbarte Summe
  - Die deutschen Datenschutzrichtlinien müssen erfüllt sein

#### 4.1.7 Aktivitäten des Requirement Engineering

#### 4.1.8 Systemkontext festlegen

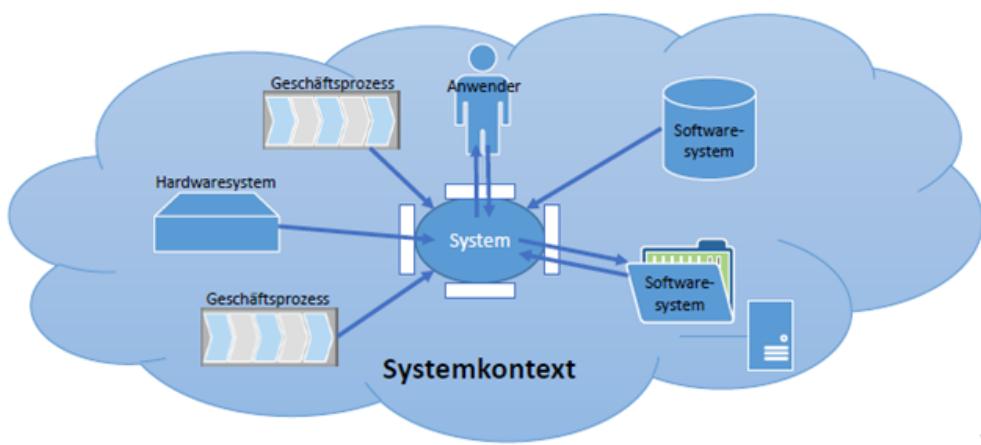
Früh im Projekt wird der Systemkontext erfasst und dokumentiert  
 → Textuell und in Form eines Kontextdiagramms (keine feste Form)

##### Ziel:

- Abgrenzen des System von der Umgebung
- Identifikation von für Anforderungen relevanter Umgebungsteile

##### Typen und Aspekte im Systemkontext:

- Personen (Stakeholder oder Stakeholdergruppen)
- System im Betrieb (andere technische Umsysteme)
- Prozesse (technisch oder physikalisch)
- Dokumente (z.B. Gesetze, Standards, Systemdokumentation)



#### 4.1.9 Stakeholder

*Systembetroffener*

##### Stakeholder:

- sind Quellen für Anforderungen
- Übersehen eines Stakeholders führt zu lückenhaften Anforderungen

#### 4.1.10 Regeln für die Definition von Zielen

##### Eigenschaften von Zielen:

- vollständig
- korrekt
- konsistent gegenüber anderen Zielen und in sich konsistent
- testbar
- verständlich für alle Stakeholder
- umsetzbar/ realisierbar
- notwendig
- eindeutig und positiv formuliert
- Lösungsneutralität
- einschränkende Rahmenbedingungen

⇒ Ziele sind abstrakte Top-Level-Anforderungen

#### 4.1.11 Anforderungen

##### Anforderungen erfassen Quellen für Anforderungen:

- Stakeholder
  - involvierte Benutzergruppen
- Dokumentation
  - Gesetze, Normen
  - Branchen-/Organisations-spezifische Dokumentation
- Systeme im Betrieb
  - Legacy- bzw. Vorgängersystem

##### Ermittlungstechniken:

- Befragungstechniken:
  - Stakeholder wird direkt zu seinen Anforderungen befragt:
  - Setzt Willen zur Mitarbeit und
  - die Möglichkeit des "sich Ausdrückens" voraus
  - Interview zwischen Requirements-Engineer und Stakeholder
  - Fragebogen
  - Osborn-Checkliste
- Dokumentengetrieben:
  - Systemarchäologie (Analyse der Dokumentation bestehender Systeme)
  - Perspektivenbasiertes Lesen
  - Wiederverwendung bereits erstellter Anforderung
- Beobachtungstechniken:
  - Feldbeobachtung: Beobachtung des Stakeholders in dessen gewohnter Umgebung, Dokumentation der Abläufe, Prozesse, Handgriffe, usw.
  - Apprenticing (=in die Lehre gehen): Requirements-Engineer lernt die Tätigkeit des Stakeholders. Unklare Schritte werden hinterfragt. Gut geeignet für die Ermittlung "selbstverständlicher Abläufe"
- Kreativitätstechniken:
  - Brainstorming
  - Perspektivenwechsel (Sechs-Hüte-Denken)

##### Anforderungen dokumentieren :

*Anforderungsspezifikation ist ein Dokument, das spezifizierte Anforderungen enthält, d.h. Anforderungen, die definierten Spezifikationskriterien genügen.*

##### Zentrale Bedeutung von Anforderungen und deren Dokumentation:

- Ausgangspunkt für nachfolgende Phasen wie Systementwicklung, Test und Abnahme
- Anforderungen sind rechtlich relevant
- Anforderungen sind komplex (z.B. Umfang und Vernetzung)
- Anforderungen sollen allen Beteiligten zur Verfügung stehen

#### **Benutzeranforderungen (Was? Und Wofür?):**

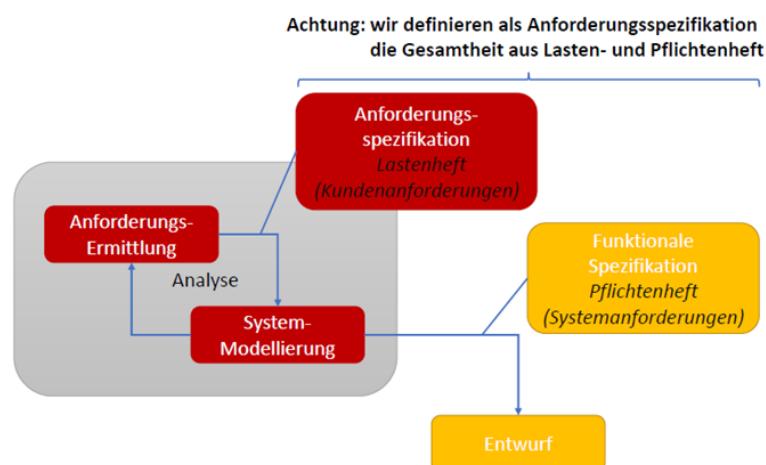
- Aussagen in natürlicher Sprache
- Diagramme
- beschreiben Dienste, die System leisten soll
- beschreiben Randbedingungen unter denen System betrieben wird
- Systembeschreibung aus Kundensicht (→ Lastenheft)

#### **Systemanforderungen (Wie? Und Womit?):**

- Detaillierte Festlegung von Funktionen, Diensten und Beschränkungen
- Beschreibung, was implementiert werden soll
- Systembeschreibung aus technischer Sicht (→ Pflichtenheft)

#### **4.1.12 Lastenheft vs. Pflichtenheft**

- Lastenheft:
  - wird von Auftraggeber gestellt
  - Beschreibung des "was" und "Wofür"
  - "grobes" Pflichtenheft
  - Details werden bewusst offen gelassen
- Pflichtenheft:
  - wird vom Auftragnehmer erstellt
  - Beschreibung des "Wie" und "Womit"
  - zu lieferndes System wird detailliert
  - Systembeschreibung aus technischer Sicht



- Template: Lastenheft
  1. Vision und Ziele
  2. Rahmenbedingungen
  3. Kontext und Überblick
  4. Funktionale Anforderungen
  5. Qualitätsanforderungen
- Template: Pflichtenheft
  1. Vision und Ziele

2. Rahmenbedingungen
3. Kontext und Überblick
4. Funktionale Anforderungen
5. Qualitätsanforderungen
6. Abnahmekriterien
7. Subsystemstruktur (optional)
8. Glossar

⇒ Das Pflichtenheft ist eine Verfeinerung des Lastenheft

#### **4.1.13 Klassifikation von Anforderungen**

Beispiel für Klassifikation von Anforderungen

##### **Das Kano-Modell (5 Merkmale)**

###### **1. Basis-Merkmale:**

- so grundlegend und selbstverständlich, dass sie Kunden erst bei Nichterfüllung bewusst werden (implizite Erwartung)
- werden die Grundforderungen nicht erfüllt, entsteht Unzufriedenheit
- werden sie erfüllt, entsteht aber keine Zufriedenheit
- Nutzensteigerung im Vergleich zur Differenzierung am Wettbewerber ist sehr gering
- Am Beispiel Auto: Sicherheit, Rostschutz

###### **2. Leistungs-Merkmale:**

- sind dem Kunden bewusst
- sie beseitigen Unzufriedenheit oder schaffen Kundenzufriedenheit abhängig vom Ausmaß der Erfüllung
- Am Beispiel Auto: Fahreigenschaften, Beschleunigung, Lebensdauer, Verbrauch

###### **3. Begeisterungs-Merkmale:**

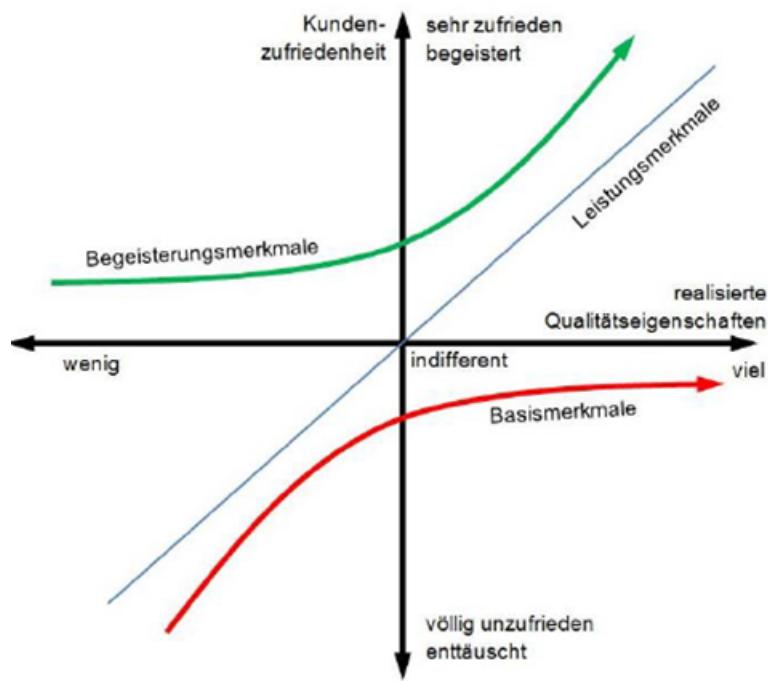
- sind Nutzen stiftende Merkmale, mit denen Kunde nicht unbedingt rechnet
- sie zeichnen das Produkt gegenüber der Konkurrenz aus und rufen Begeisterung hervor
- Kleine Leistungssteigerung kann zu einem überproportionalen Nutzen führen
- Differenzierungen gegenüber der Konkurrenz können gering sein, der Nutzen enorm
- Am Beispiel Auto: Sonderausstattung, besonderes Design, Hybridtechnologie

###### **4. Unerhebliche Merkmale:**

- sind sowohl bei Vorhandensein wie auch bei Fehlen ohne Belang für Kunden
- sie können daher keine Zufriedenheit stiften, führen aber auch nicht zu Unzufriedenheit
- Am Beispiel Auto: je nach Kundengruppe z.B. Automatikgetriebe, Schiebedach

###### **5. Rückweisungs-Merkmale:**

- Führen bei Vorhandensein zu Unzufriedenheit
- bei Fehlen jedoch nicht zu Zufriedenheit
- Am Beispiel Auto: Rostflecken, abgelaufener TÜV



#### 4.1.14 Erfassung von Anforderungen mithilfe von UseCases

ToDo

#### 4.1.15 Zusammenfassung Requirements Engineering



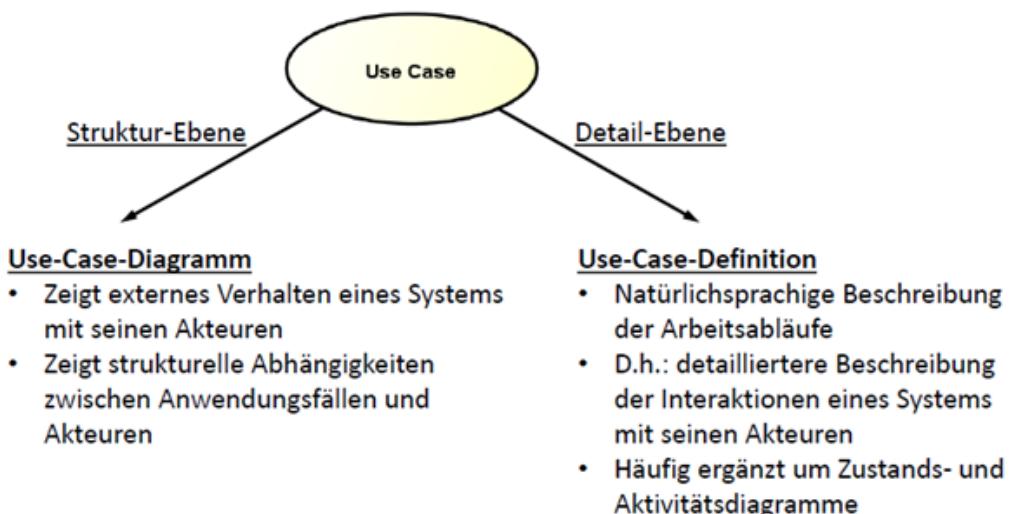
## 4.2 Use Cases

Unterschiedliche Notationen für Use Cases (Anwendungsfälle):

- Use-Case-Definition
- tabellarische Use-Case-Definition
- Use-Case-Diagramm

### Definition „Use Case“:

- spezifiziert Sequenz von Aktionen einschließlich möglicher Varianten, die das System in Interaktionen mit Akteuren auslöst
- Abstrahiert von konkreter technischer Lösung
- wird durch bestimmtes Ereignis ausgelöst
- wird ausgeführt, um ein konkretes Ziel zu erreichen/ gewünschtes Ergebnis zu erstellen
- ist als Black-Box zu verstehen
- beschreibt also extern wahrnehmbares Verhalten
- geht nicht auf interne Struktur oder Realisierungsdetails ein



### Zwei Beschreibungsvarianten

#### 4.2.1 Beschreibungsvariante: Use-Case-Definition

Usecase wird in einer Tabelle mit natürlicher Sprache festehalten

#### Regeln zum Aufbau einer Use-Case-Definition:

Use-Case zeigt die Absicht der Akteure an (nicht das Verhalten des Systems)

- Falsch:

1. Das System fragt nach dem Namen
2. Der Nutzer gibt den Namen ein
3. Das System fragt nach der Adresse
4. Der Nutzer gibt die Adresse ein
5. Der Nutzer klickt auf OK
6. Das System zeigt das Profil des Nutzers

- Richtig:

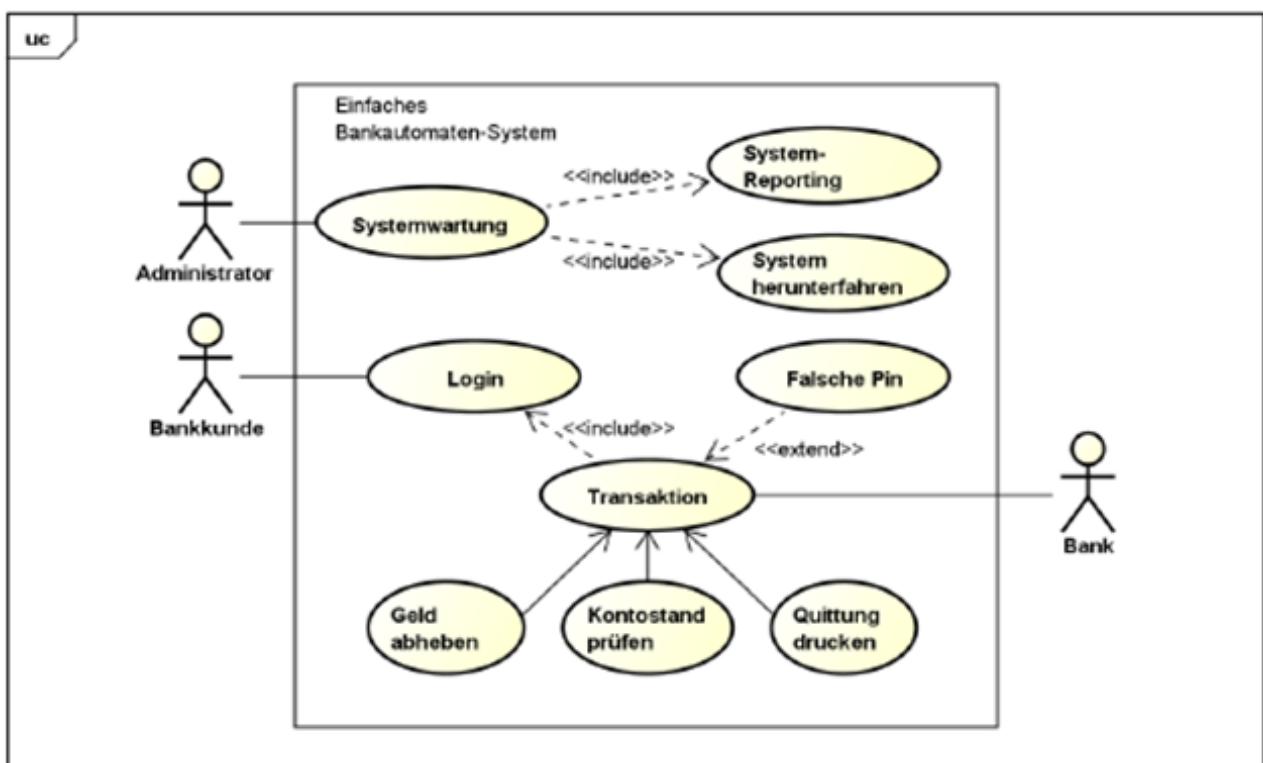
1. Der Nutzer gibt Namen und Adresse ein
2. Das System zeigt das Profil des Nutzers an

## Formulierungsregeln

- einfache Grammatik benutzen
  - Beispiel: Namen und Adresse eingeben (oder: Der Nutzer gibt Namen und Adresse ein)
  - Beispiel: Kontostand aktualisieren (oder: Das System aktualisiert den Kontozustand)
  - Ziel: Klarheit
- Lesbarkeit beachten
  - zu viele Schritte in einem Use Case vermeiden (Empfohlen: max. 9 Schritte)
- Präzise formulieren:
  - Falsch: Das System prüft, ob das Passwort richtig ist.
  - Richtig: Das System verifiziert, dass das Passwort richtig ist
- Falls Wiederholungen notwendig sind: „Mache Schritte i-j bis bedingung k erfüllt ist“
  - Beispiel:
    1. Der Verbraucher wählt ein Produkt aus
    2. Das System fügt das Produkt in den Einkaufswagen
    3. Der Verbraucher wiederholt die Schritte 1-2, bis alle gewünschten Produkte gewählt
    4. Der Verbraucher zahlt die Produkte im Einkaufswagen

### 4.2.2 Use-Case Diagramm

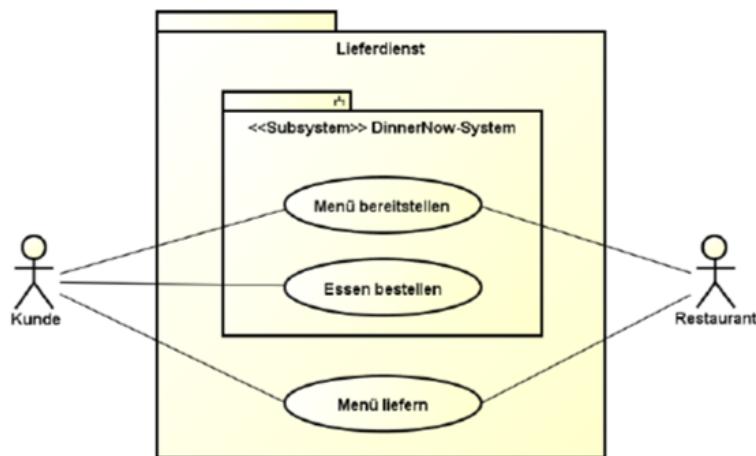
beschreibt strukturelle Abhängigkeit zwischen Anwendungsfällen und Akteuren:



Notation	Name	Bedeutung
	Use-Case	<ul style="list-style-type: none"> <li>- Definiert fachlichen Anwendungsfall</li> <li>- Verhalten wird beschrieben (keine internen Realisierungsdetails)</li> <li>- Detailliertere Darstellung der Aktionen durch andere UML-Diagramme oder im Klartext</li> <li>- Wird von einem Akteur gestartet</li> <li>- Führt zu einem fachlichen Ergebnis</li> </ul>
	Subsystem	<ul style="list-style-type: none"> <li>- Teil des Systems</li> <li>- Beinhaltet Use Cases, die vom Subsystem unterstützt werden</li> </ul>
	Akteur	<ul style="list-style-type: none"> <li>- Externes Objekt oder Person, die mit System interagiert</li> <li>- Use Case wird stets von einem Akteur gestartet</li> </ul>
	Akteur	<ul style="list-style-type: none"> <li>- Links: Fremdsystem als Akteur</li> <li>- Rechts: Zeitgesteuertes Ereignis als Akteur</li> </ul>
	Abhängigkeit	<ul style="list-style-type: none"> <li>- Aufbau des Quell-Use-Cases hängt vom Aufbau des Ziel-Use-Cases ab</li> </ul>

Notation	Name	Bedeutung
	Assoziation (ungerichtet)	<ul style="list-style-type: none"> <li>- Akteur ist an Use Case beteiligt bzw.</li> <li>- Akteur kommuniziert mit dem System</li> <li>- Evtl. mit Multiplizität: wie oft kann Element an Aktion gleichzeitig teilnehmen</li> </ul>
	Assoziation (gerichtet)	<ul style="list-style-type: none"> <li>- Beschreibt welcher Teil der Aktive ist</li> <li>- (beschreibt nicht Richtung des Informationsflusses)</li> </ul>
	Generalisierung/Vererbung	<ul style="list-style-type: none"> <li>- Ziele, Use Cases, etc. der Generalisierung werden von der Spezialisierung geerbt</li> </ul>
	<<include>>-Beziehung	<ul style="list-style-type: none"> <li>- Use Case1 importiert Verhalten eines anderen Use Cases</li> <li>- Beziehung ist nicht optional, Verhalten wird also immer importiert</li> </ul>
	<<extend>>-Beziehung	<ul style="list-style-type: none"> <li>- Verhalten von UseCase2 kann durch UseCase1 erweitert werden</li> <li>- Zeitpunkt, an dem Verhalten erweitert werden kann: Extension Point (UseCase darf mehrere EP haben)</li> <li>- Optionale &lt;Bedingung&gt;: wenn true, EP ausführen</li> </ul>

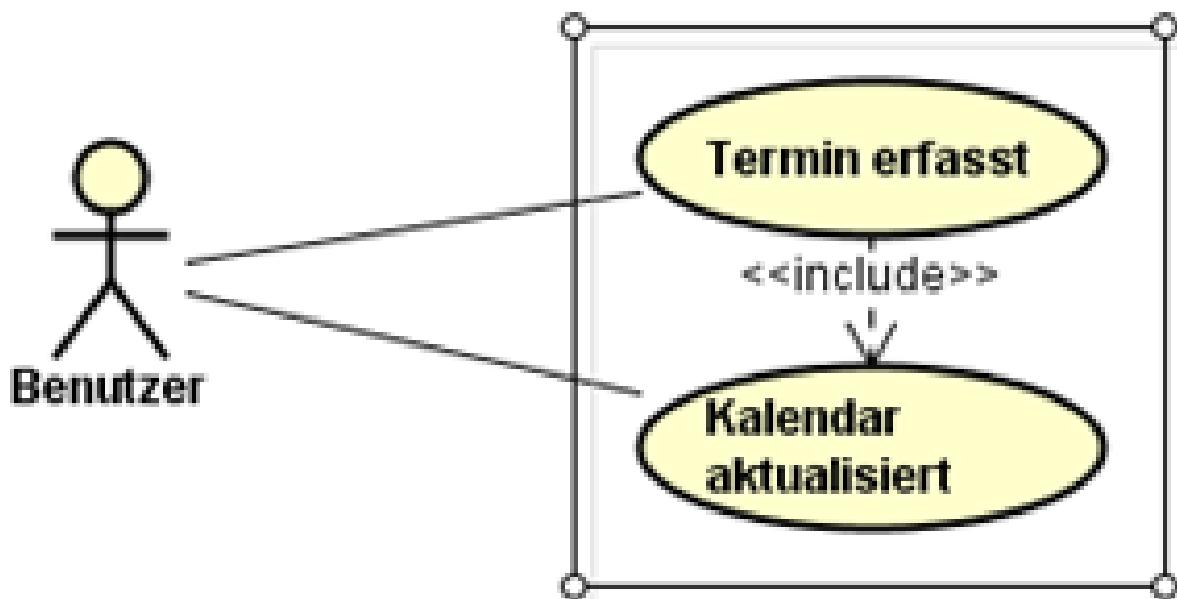
## Notation



Bsp: Bank

#### Use-Case-Diagramm: «include»

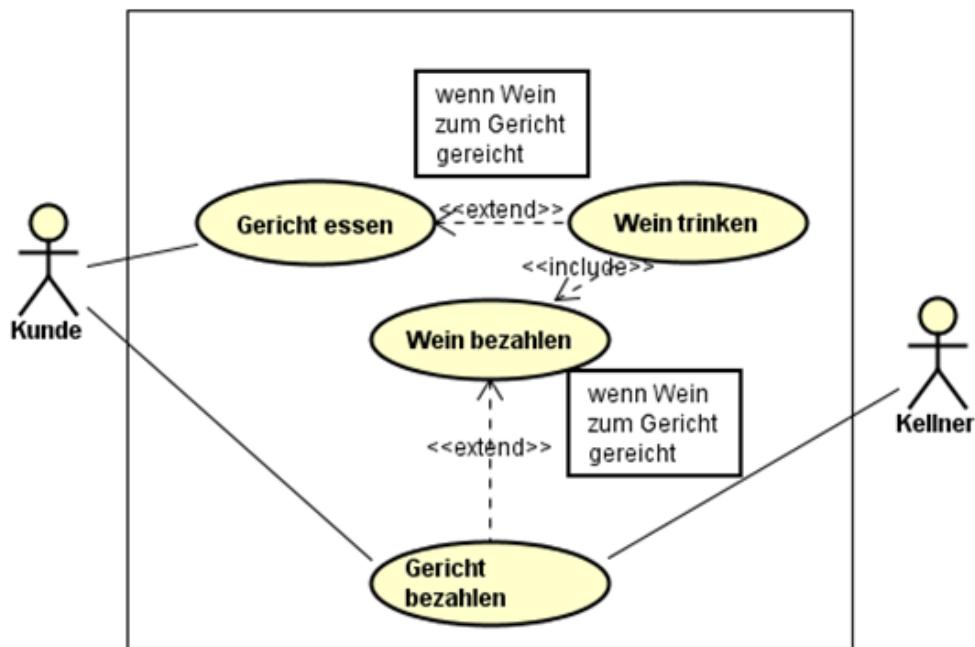
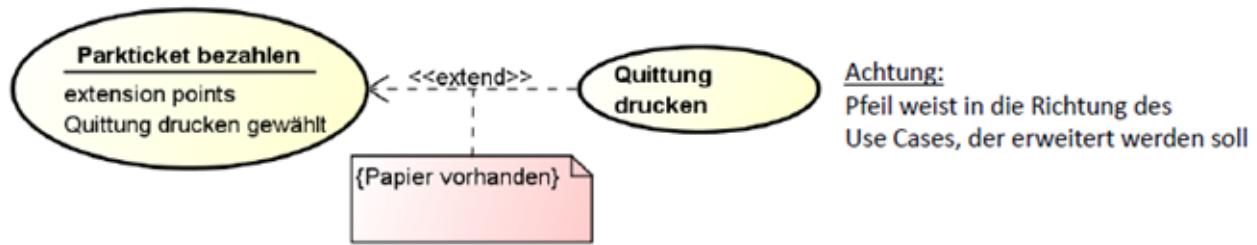
- visualisiert, dass Use Case A das Verhalten von Use Case B importiert
- das Verhalten wird immer importiert (Verhalten ist nicht optional)
- Erst die Beschreibung der Aktionen definiert, an welcher Stelle B inkludiert wird



Kann auch in der Use-Case Definition unter 'Essentielle Schritte' eingetragen werden.

#### Use-Case-Diagramm: «extend»

- Verhalten von Use-Case 2 kann durch Use-Case 1 erweitert werden (...muss aber nicht)
- Zeitpunkt, an dem ein Verhalten eines Use-Case erweitert werden kann, bezeichnet man als Erweiterungspunkt (extension point)
- ein Use Case darf mehrere Erweiterungspunkte besitzen
- um optionale Bedingung ergänzbar (nur wenn Bedingung wahr ist, wird Erweiterung ausgeführt)



Bsp: Restaurant

#### 4.2.3 Anwendungsbereiche

- erlauben Black-Box-Sicht auf zu entwickelndes System ohne Realisierungsdetails
- erlauben Abgrenzung des Systems von der Umwelt
- Einsatzgebiete:
- funktionale Dienstleistungen des Systems auf einen Blick zeigen
- System aus Nutzersicht in handhabbare, logische Teile zerlegen
- Außenschnittstellen und Kommunikationspartner des Systems modellieren
- komplexe Systeme auf hohem Abstraktionsniveau darstellen
- planbare Einheiten (inkrementale) für Entwicklung bestimmen

#### 4.2.4 Use-Case-Erstellung: Checkliste (Auszug aus Heide Balzert)

1. Akteure ermitteln
2. Use Cases für Standardverarbeitung ermitteln:
  - auf Basis von: Akteuren, Ereignissen (externe Systeme als Akteure), Aufgabenbeschreibungen (Gesamtziele, Top-10-Aufgaben, etc.)
3. Use Cases für Sonderfälle ermitteln:
  - «extends»-Beziehung verwenden, um Standardfälle zu erweitern
4. Aufteilen komplexer Anwendungsfälle:

- komplexe Schritte als eigene Use Cases (mit «include» einbinden)
- Use Cases mit vielen Sonderfällen aufspalten (für Gemeinsamkeiten «include» nutzen)
- umfangreiche Erweiterungen mit «extend» spezifizieren

## 5. Gemeinsamkeiten mit «include» wiederverwenden

### 4.2.5 Use Case vs. Szenario

#### Use Case (Anwendungsfall):

- beschreibt abstrakte Interaktion zwischen Akteuren und Systemen

#### Szenario:

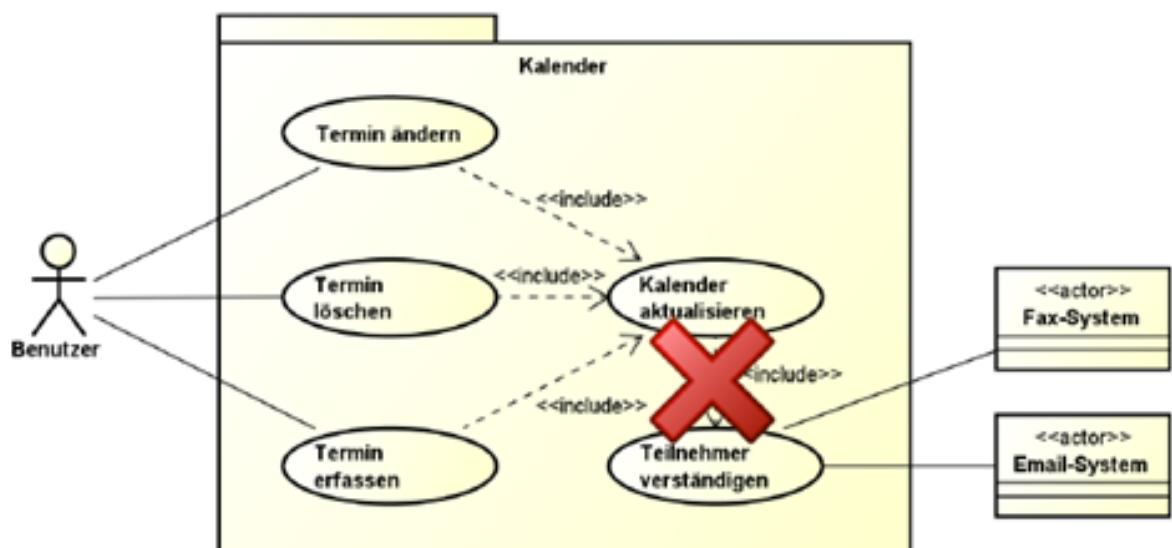
- ist eine konkrete Ausprägung eines Use Cases (also ein möglicher Ablauf mit konkreten Werten)
- ist eine Instanz eines Use Cases

#### Zusammengefasst:

- Szenarios werden durch Sequenzdiagramme oder textuell beschrieben
- Zu einem Use Case existieren i.d.R. mehrere Szenarios

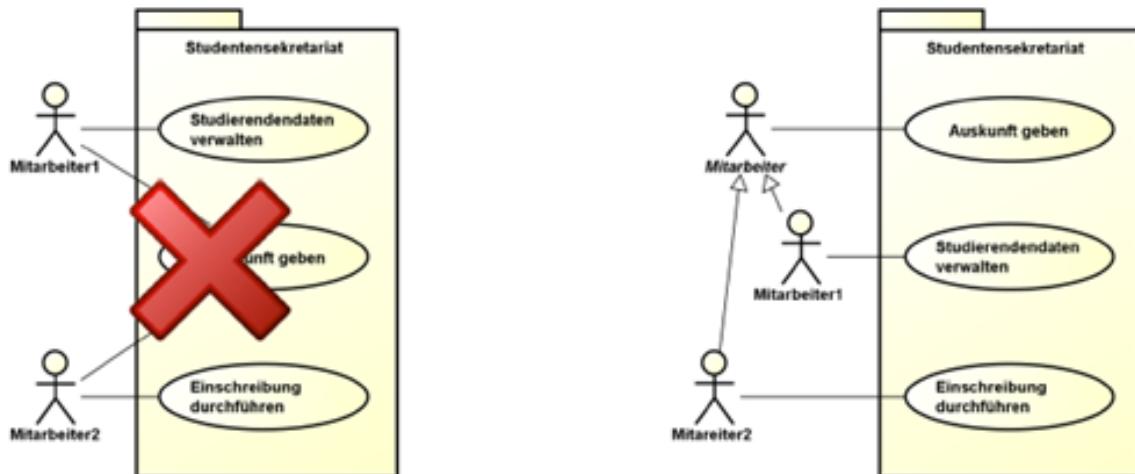
### 4.2.6 Typische Modellierungsfehler

Use-Case-Diagramme modellieren keine Abläufe

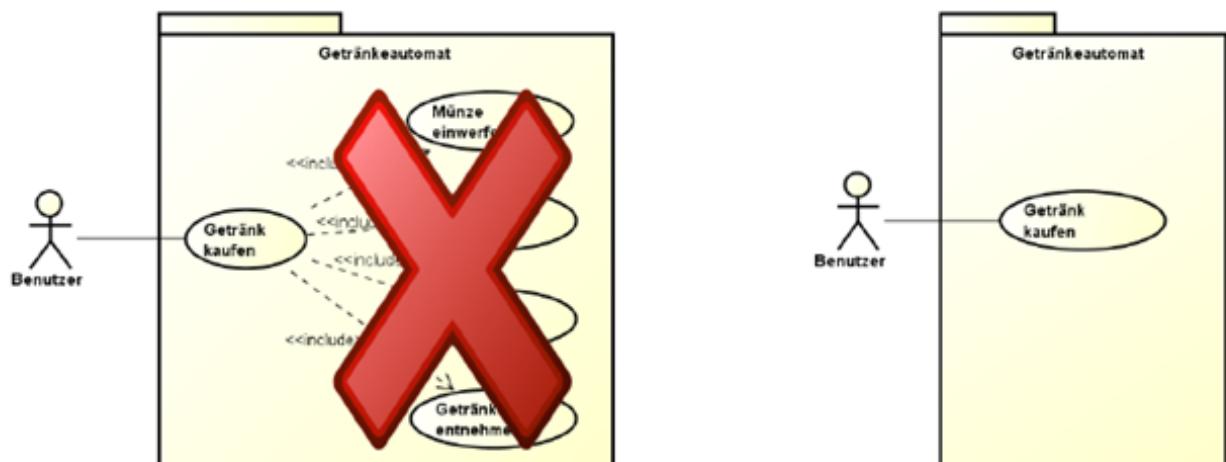


- “Teilnehmer verständigen” ist nicht Teil von “Kalender aktualisieren”

- Wir wollen ausdrücken, dass entweder Mitarbeiter1 oder Mitarbeiter2 den Anwendungsfall "Auskunft geben" ausführen. Was ist das Problem?



- links würden beide Mitarbeiter den Use Case "Auskunft geben" gemeinsam ausführen

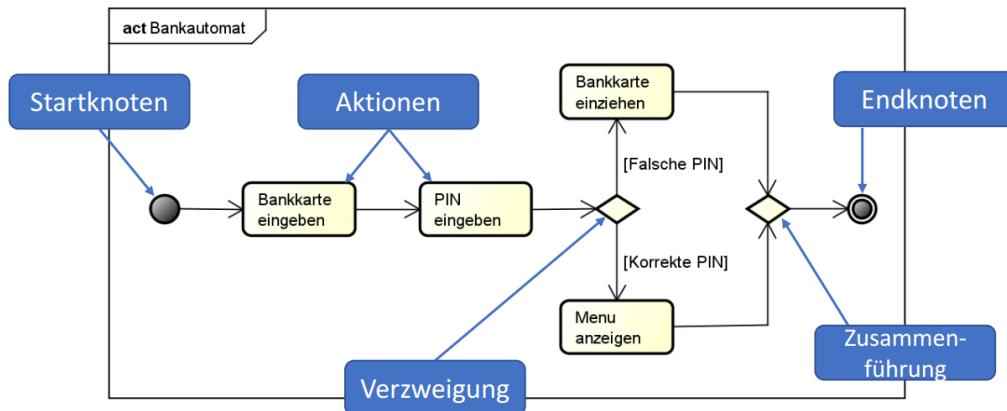


- Nicht alle Einzelschritte als Use Cases modellieren
- Wichtig ist: Anwendungsfall soll für den Nutzer messbaren Nutzen erzeugen

#### 4.3 Aktivitätsdiagramme

Wie realisiert mein System ein bestimmtes Verhalten?

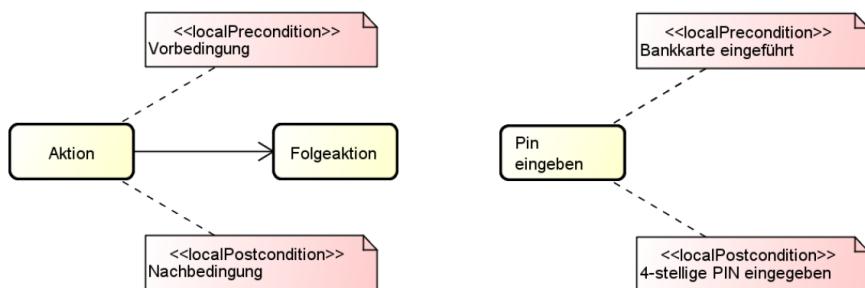
Beispiel:



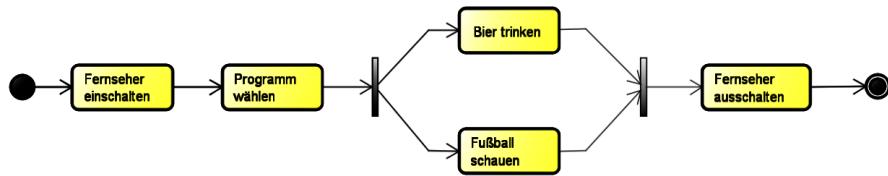
Notation:

Notation	Bedeutung
	- Token gelangt zu „Aktion“ führt diese aus - Sobald Aktion beendet, wandert es entlang der ausgehenden Kante weiter
	- Jede aus Startknoten führende Kante erhält ein Token (Geburt)
	- Token das in Endknoten läuft wird konsumiert (Tod) - Erreicht ein Token einen Endknoten wird die Instanz des Diagramms beendet
	- Token das in Ablaufendknoten läuft wird konsumiert (Tod) - Die Instanz des Diagramms läuft weiter
	- Weiche für Tokenfluss (→ Schleifen realisierbar) - Bedingungen müssen sich gegenseitig ausschließen ([else] vordefiniert) - Das Gegenstück der Weiche ist die Zusammenführung (auch Rautensymbol)
	- Modellierung der Aufspaltung von Abläufen - Eingehendes Token teilt sich, so dass alle Ausgänge ein Token erhalten
	- Führt nebenläufige Token zusammen - Vereinigung der Kontrolltoken, sobald an allen eingehenden Kanten ein Token - Nur ein Token wird weitergeleitet (Datentoken werden alle weitergeleitet)

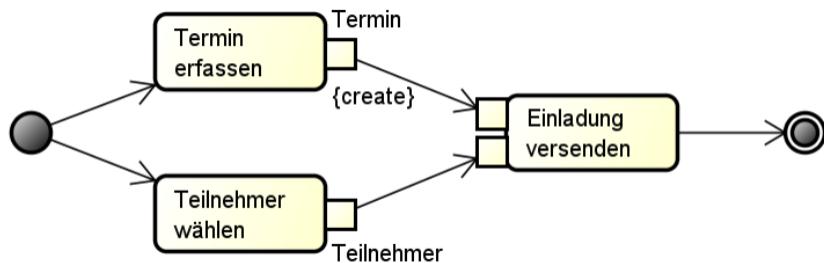
Optionale Vor- und Nachbedingung:



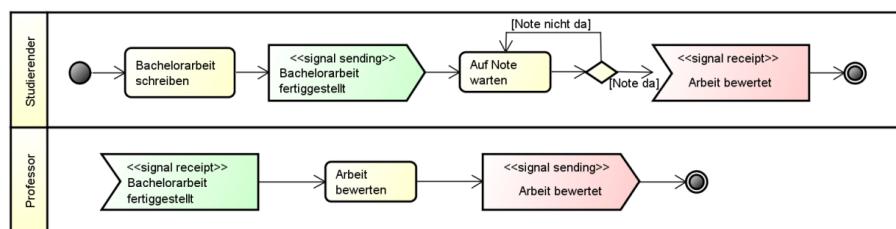
Fork/Join:



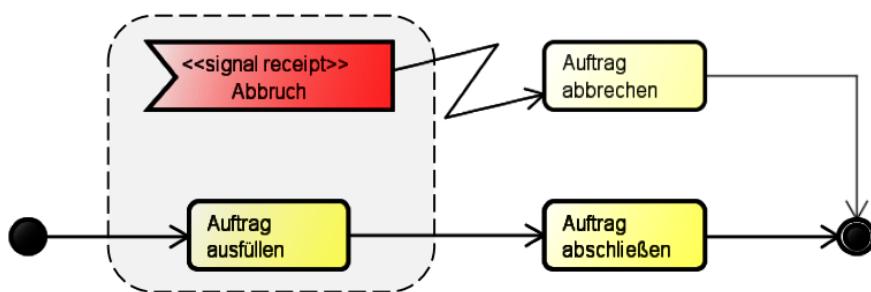
## Objekte:



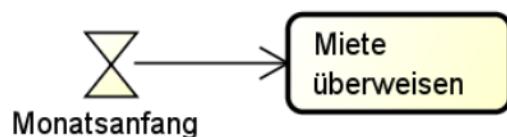
## Swimlanes:



## **Unterbrechung:**



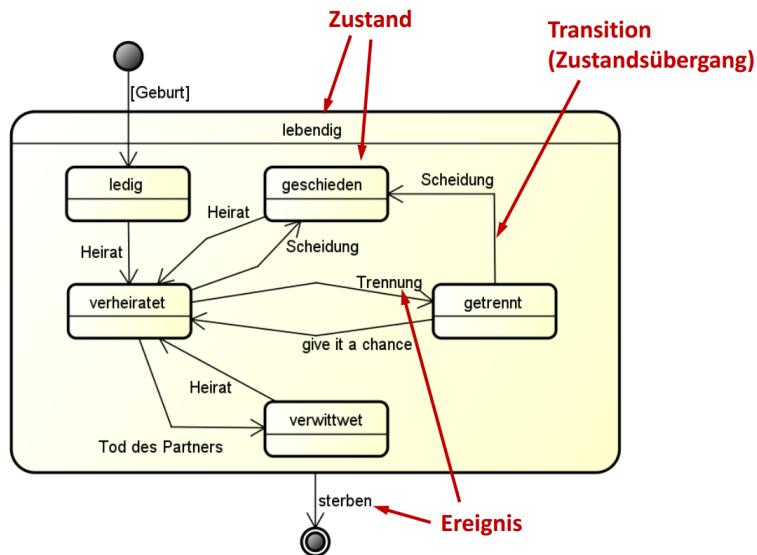
## Zeitgesteuertes Event:



## 4.4 Zustandsdiagramme

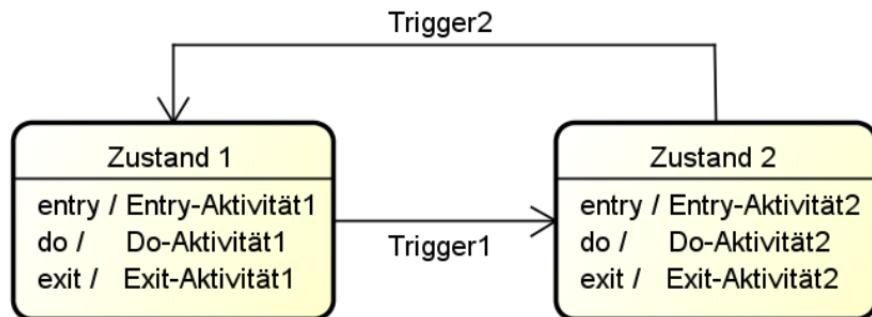
Wie verhält sich System in bestimmtem Zustand bei gewissen Ereignissen?

## Zustand:



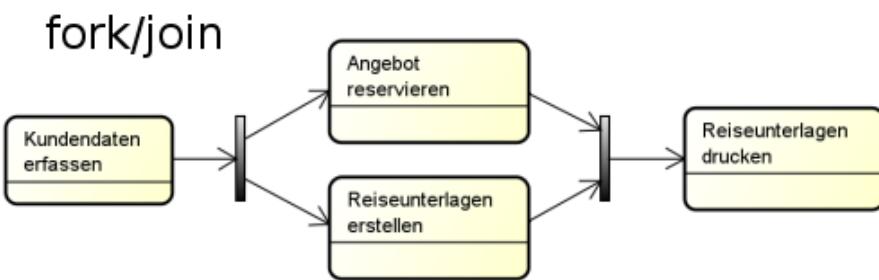
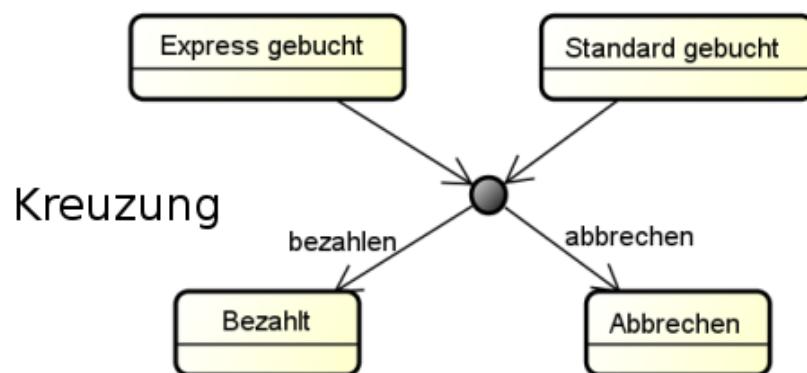
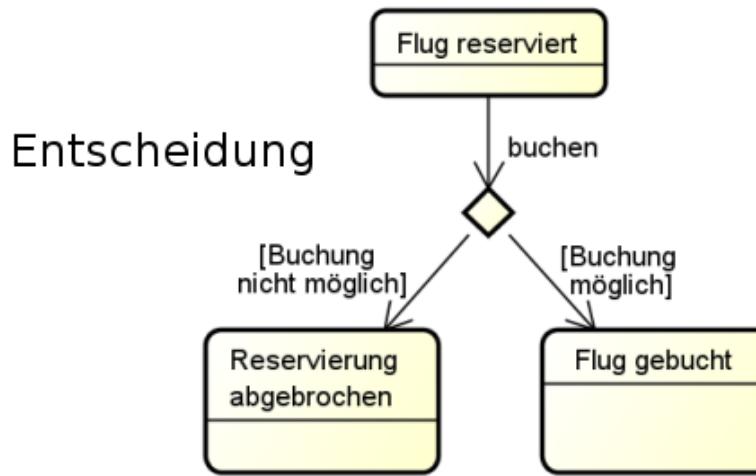
- Zustand = Objekt in bestimmter Situation
- kann auf äußere Ereignisse reagieren
- Notation: Zustandsname + Aktivität:
  - entry/: Aktivitäten bei Eintritt in Zustand
  - do/: Aktivitäten während Aufenthalt im Zustand
  - exit/: Aktivitäten bei Verlassen des Zustandes

#### Transition:



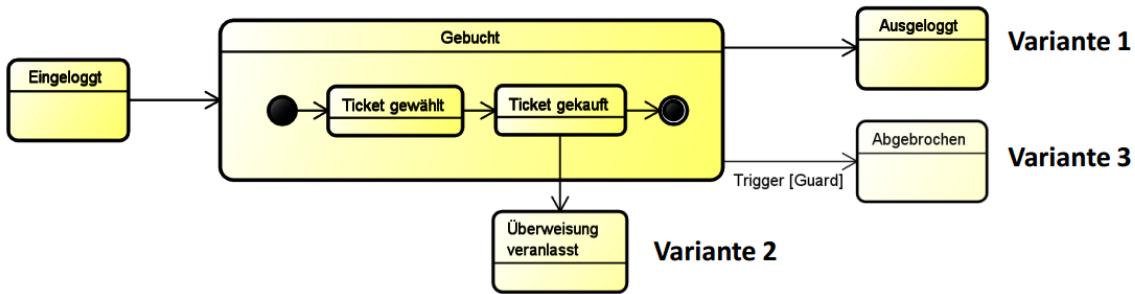
- modelliert Übergang von einem Quell- in einen Zielzustand
- Quellzustand kann gleich Zielzustand sein (Selbsttransition)
- Transition schaltet (wird durchlaufen), wenn zugehöriges Triggerereignis eintritt
- Konvention: Durchlaufen der Transition nimmt keine Zeit in Anspruch
- Notation:
  - Trigger: externes Ereignis, das Transition auslöst  
→ Bsp.: Wenn der Nutzer den Transfer-Button klickt
  - Guard: Durchlaufbedingung für Transition (nur bei true kann Transition schalten)  
→ Bsp.: Wenn die gemessene Temperatur 32 Grad Celsius übersteigt
  - Aktion: beim Durchlauf durchzuführende Aktion  
→ Bsp.: Ventil zwischen Behälter1 und 2 öffnen

#### Pseudozustände:



- Entscheidung:
  - beschreiben Entscheidungen, die vom Wert des Transitionsübergangs abhängen
  - Guards der ausgehenden Transitionen werden ausgewertet, nachdem eingehende Transition durchlaufen
- Kreuzung/Verbindung (Junction-Point):
  - Genutzt um viele Transitionen zu wenigen zusammenzuführen (Übersichtlichkeit)
- Teilung und Synchronisation (Fork/Join):
  - Fork: Aufteilung einer eingehenden Transition auf mehrere ausgehende Zustände (keine Guards auf ausgehenden Tr.)
  - Join: Zusammenfassung paralleler Zweige (keine Guards auf eingehenden Transitionen)

Hierarchische Zustände:



Eintritt in hierarchischen Zustand:

- Eingehende Transition endet am Rand des zusammengesetzten Zustands
- Verarbeitung von „Gebucht“ beginnt am Startzustand des hierarchischen Zustands

Verlassen des hierarchischen Zustands:

- Variante 1: hierarchischer Zustand wird explizit über inneren Endzustand verlassen
- Variante 2: hierarchischer Zustand wird explizit über ausgehende Transition verlassen
- Variante 3: hierarchischer Zustand wird verlassen, sobald in einem der Unterzustände das Ereignis "Trigger" eintritt und die Bedingung "Guard" erfüllt ist

#### 4.4.1 Zustandsdiagramm erstellen

1. Prüfen, ob ein nicht-trivialer Lebenszyklus für eine Klasse existiert:
  - Das gleiche Ereignis kann (in Abhängigkeit des aktuellen Zustandes) unterschiedliches Verhalten bewirken
  - Operationen sind nur in bestimmten Situationen (Zuständen) auf ein Objekt anwendbar und werden sonst ignoriert
2. Zustände des Automaten bestimmen:
  - Ausgangsbasis ist der Startzustand
  - Ereignisse bestimmen, die zu Zustandswechseln führen
  - Folgezustände ermitteln
3. Endzustände bestimmen (falls vorhanden):
  - das Objekt hört auf zu existieren oder
  - das Objekt existiert weiterhin, aber sein dynamisches Verhalten ist nicht länger von Interesse
4. Aufzurufende Operationen bestimmen:
  - zustandsabhängige Operationen aus dem Klassendiagramm eintragen
  - evtl. weitere Operationen definieren, falls notwendig
5. Zu berücksichtigende Ereignisse ermitteln:
  - externe Ereignisse: vom Benutzer oder anderen Objekten
  - zeitliche Ereignisse: Zeitdauer, Zeitpunkt
  - intern generierte Ereignisse des betrachteten Objekts

## 4.5 Paketdiagramme

Motivation

- Problem: In großen Softwaresystemen ist es schwer den Überblick zu behalten
  - es existieren meist hunderte von Use Cases, Aktivitätsdiagrammen, Klassen etc.
- Ziel: auf hoher Ebene Struktur in die Systembeschreibung bringen
- Lösung: Einteilung des Systems in Pakete (Abstraktion)

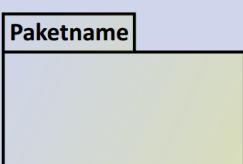
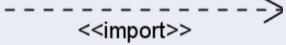
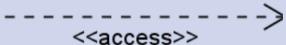
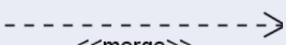
Was sind Pakete / wie bündelt man Pakete?

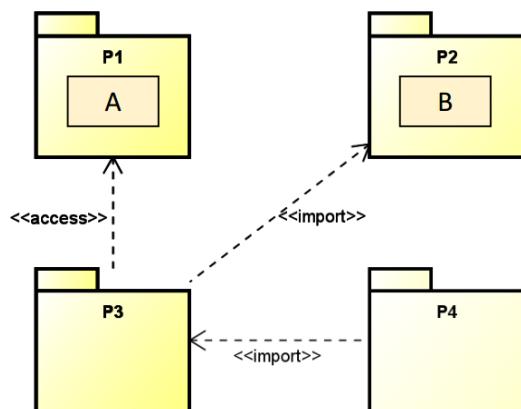
- Paket bündelt Elemente der UML sinnvoll, d.h. nach einem Zusammenhang

- Paket entspricht einem in sich geschlossenen Teilsystem
- Pakete können hierarchisch geschachtelt sein (ein Paket enthält Pakete)

Verwendung von Paketsichten:

- Paketsicht dient der Strukturierung des Systems
- Strukturierung kann nach unterschiedlichen Kriterien erfolgen:
  - Funktionale Gliederung
  - Definition von Schichten
  - ...

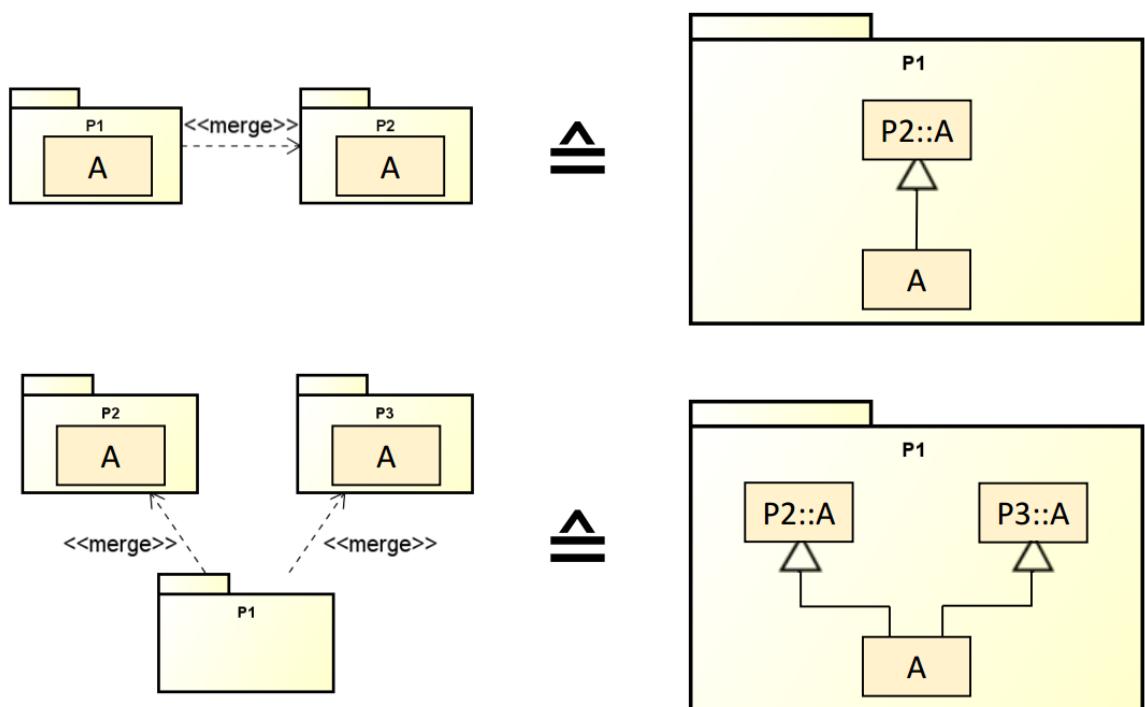
Notation	Bedeutung, Verwendung
	<ul style="list-style-type: none"> <li>- Paket fasst paketierbare Elemente zusammen (oft: Klassen, andere Pakete)</li> <li>- Paket definiert einen Namensraum</li> <li>- Paketname in der Tasche oben links (Name kann auch eine URI enthalten)</li> <li>- Innerhalb des unteren Rechtecks können zugehörige Elemente eingetragen werden</li> <li>- Sichtbarkeit der Elemente wie üblich mit "+" (public), "-" (private)</li> </ul>
	<ul style="list-style-type: none"> <li>- Namen aller <u>öffentlichen Elemente</u> eines Pakets werden dem importierenden System <u>als öffentlich hinzugefügt</u> (public import)</li> <li>- Abhängigkeit möglich zwischen: 2 Paketen, 2 Klassen, Operation und Klasse, Klasse und Schnittstelle (auch für &lt;&lt;access&gt;&gt; und &lt;&lt;merge&gt;&gt;)</li> </ul>
	<ul style="list-style-type: none"> <li>- Namen aller <u>öffentlichen Elemente</u> eines Pakets werden dem importierenden System <u>als privat hinzugefügt</u> (private import)</li> </ul>
	<ul style="list-style-type: none"> <li>- Nicht private Inhalte des Zielpakets werden in die Inhalte des Quellpakets verschmolzen</li> </ul>



- P3 importiert A (P1) als **private**
- P3 importiert B (P2) als **public**
- P4 importiert P3

Ergebnis:

- P4 darf auf B zugreifen
- P4 darf auf A nicht zugreifen



Vorgehen bei der Erstellung eines Paketdiagramms:

- Konstruktive Schritte zum Identifizieren von Paketen:
  - Top-down: Bei großen Projekten noch vor der Erstellung der Use Cases
  - Bottom-up: Bei kleinen oder mittelgroßen Projekten nach der Erstellung der Geschäftsprozesse oder spätestens nach der statischen Modellierung
- Analytische Schritte:
  - Bildet das Paket eine abgeschlossene Einheit? (Häufig bilden Klassen mit vielen Beziehungen untereinander ein Paket)
  - Ist der Paketname geeignet?

Fehlerquellen:

- zu kleine Pakete
- schlechte oder falsche Bezeichner
  - Pakete müssen eindeutige Namen haben
- Bezeichnung der Beziehung fehlt (import (Standardfall), access, merge)
- Zirkuläre «merge»-Beziehung
  - Pakete dürfen sich nicht gegenseitig mergen
- Falsche Hierarchie:
  - Klassen dürfen Subklassen, aber keine Pakete beinhalten

## 4.6 Klassendiagramme

### 4.6.1 Schema zur Definition von Klassenattributen

Sichtbarkeit [/] Name [:Typ] [Multiplizität] [=Vorgabewert] (Eigenschaft)\*

- Sichtbarkeit (des Attributs):
  - “+“ public: öffentlich; für alle Elemente sichtbar
  - “~“ package: sichtbar für alle Elemente im gleichen Paket
  - “#“ protected: sichtbar für Instanzen und Instanzen abgeleiteter Klassen
  - “-“ private: sichtbar nur für Instanzen dieser Klasse
- Typ (des Attributs):
  - Elementare Datentypen (z.B. int, String, double, etc.)
  - Datentypen, die durch Klassen definiert sind
- Multiplizität: (Anzahl der Instanzen, die unter diesem Attribut ablegbar sind)
  - Nutzung der Schreibweise: “[min,max]“
  - “[0..1]“: Optionalität
  - “[1..1]“= “[1]“: Standard (zwingendes Attribut)
  - “[0..\*]“= “[\*]“: 0 oder beliebig viele
  - “[1..N]“: mindestens eins, höchstens N
- Vorgabewert (Angabe eines Standardwertes):
  - Konstante
  - Berechenbarer Ausdruck
- /:
  - Attribut ist abgeleitet (kann zur Laufzeit aus anderen Attributen berechnet werden)
- Eigenschaft:
  - beschreibt eine Eigenschaft des Attributs (z.B.: unique, ordered, readonly, id)

#### 4.6.2 Schema zur Definition von Klassenoperationen

Sichtbarkeit Name (Parameterliste) : Rückgabetyp Eigenschaften

- Sichtbarkeit (des Attributs):

“+“ public: öffentlich; für alle Elemente sichtbar  
“~“ package: sichtbar für alle Elemente im gleichen Paket  
“#“ protected: sichtbar für Instanzen und Instanzen abgeleiteter Klassen  
“-“ private: sichtbar nur für Instanzen dieser Klasse

- Rückgabetyp (der Operation):

- Datentyp, Klasse, void

- Eigenschaften (Einschränkungen):

- Preconditions (zu erfüllende Bedingungen zur Aktivierung der Operation)  
- Postconditions (Zusicherung über Zustand des Systems nach der Ausführung)  
- Body conditions (Einschränkungen des Rückgabewerts)

- Richtung (Übergaberichtung):

- “in“: Wert des Parameters wird von aufrufender Methode übergeben  
- “out“: Wert des Parameters wird an die aufrufende Methode zurückgegeben  
- “InOut“: erst “in“ dann “out“

- Name/Typ/Multiplizität/Eigenschaften

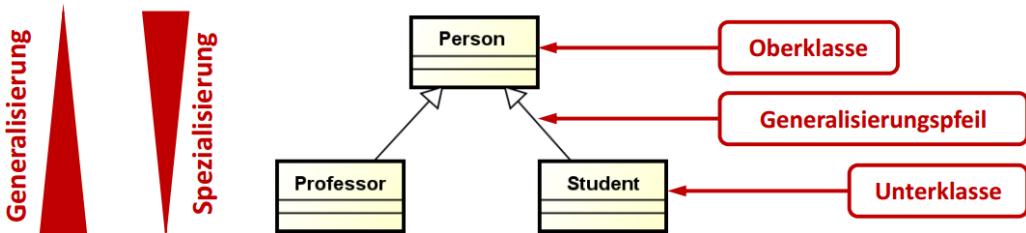
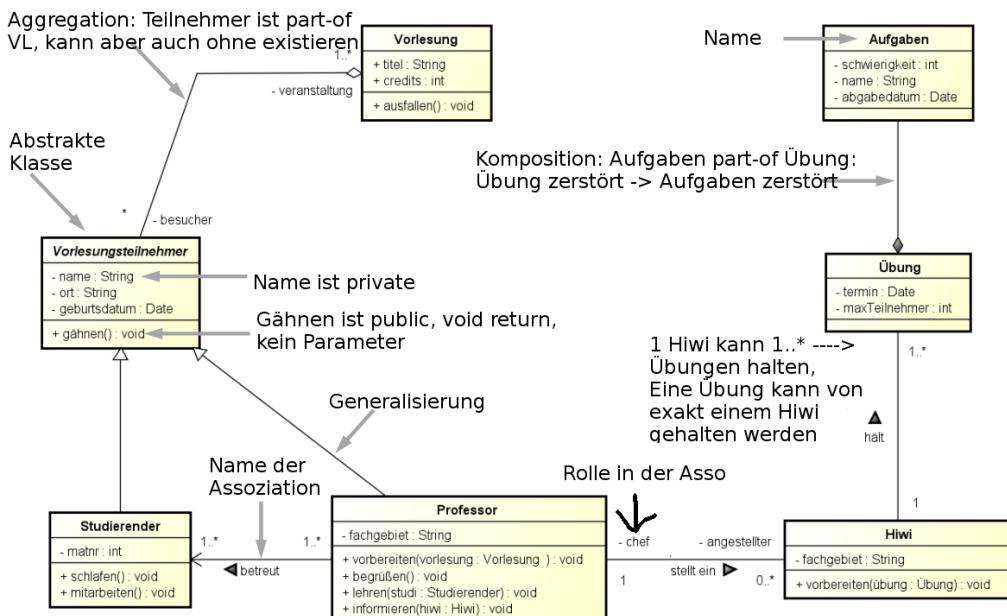
- analog zu Attributen

- Defaultwert:

- Wert des Parameters, falls kein Wert explizit übergeben wurde

#### 4.6.3 Beispiele

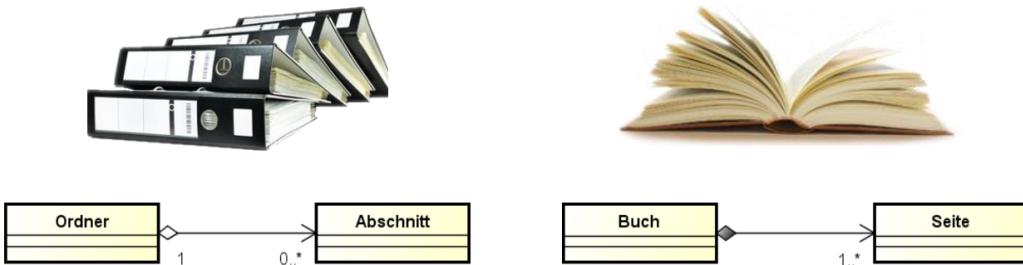
Automobil
+ farbe : Color[1] = rot {readonly}
- leistung : int = 100
+ starten() : void
+ tanken(volumen : double) : double
- verbrauchMessen() : double



## Vergleich: Aggregation

vs.

## Komposition



## 4.7 Analysemuster

Motivation für Analysemuster:

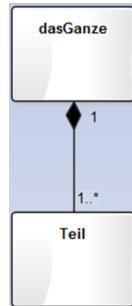
- Nutzung erprobter Vorgehensweisen bei Modellierung mit Klassendiagrammen
- unterstützen Wiederverwendung
- vereinfachen Design
- definieren gemeinsame Sprache
- erleichtern Dokumentation und Wissenstransfer
- erleichtern ingenieurmäßiges Vorgehen bei der Softwareentwicklung

### 4.7.1 Liste

Modellierung:

- Komposition (Teil kann ohne Ganzes nicht existieren)
- Es gibt genau eine "Teil"-Klasse

- Dem "Ganzen" ist meist mindestens ein "Teil" zugeordnet
  - Multiplizität also: 1..\*
- Objekte vom Typ "Teil" sind einem "Ganzen" fest zugeordnet
  - sie können entfernt werden, bevor das "Ganze" gelöscht wird
- manche Attributwerte des Aggregat-Objekts gelten auch für die Teilobjekte



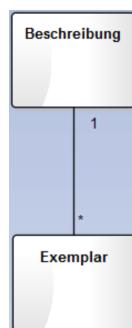
#### 4.7.2 Exemplartyp

Idee:

- Ziel: Redundanzen vermeiden
- es gibt Eigenschaften, die für alle Exemplare gleich sind
- zusätzlich hat jedes Exemplar eigene Eigenschaften

Modellierung:

- Einfache Assoziation (keine Teile-Ganzes-Beziehung)
- Objektbeziehungen werden nicht geändert
- Name der Klasse, die den Exemplartyp beschreibt enthält Begriffe wie z.B.: -Typ, -Gruppe, -Beschreibung, -Spezifikation
- Beschreibung kann unabhängig von konkreten Exemplaren existieren (daher die Kardinalität "\*")



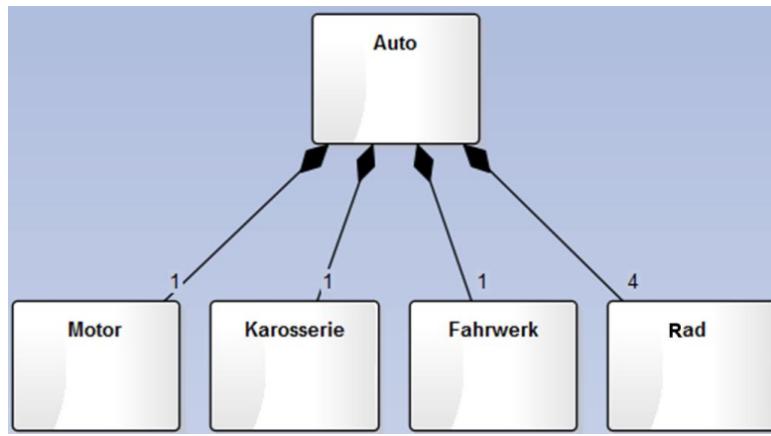
#### 4.7.3 Baugruppe

Motivation:

- Ganzes existiert nur gemeinsam mit seinen Einzelteilen
- Ganzes kann ohne seine Einzelteile seine Aufgabe nicht ausführen
- getroffene Zuordnung besteht über längeren Zeitraum
- Ganzes und Einzelteile sind physische Objekte

Modellierung:

- Komposition
- "Ganzes" kann ohne "Teile" nicht funktionieren
- "Ganzes" kann aus unterschiedlichen "Teilen" bestehen
- Ein "Teil"-Objekt kann von seiner Baugruppe getrennt werden (Ganzes ist dann u.U. nicht mehr funktionsfähig)



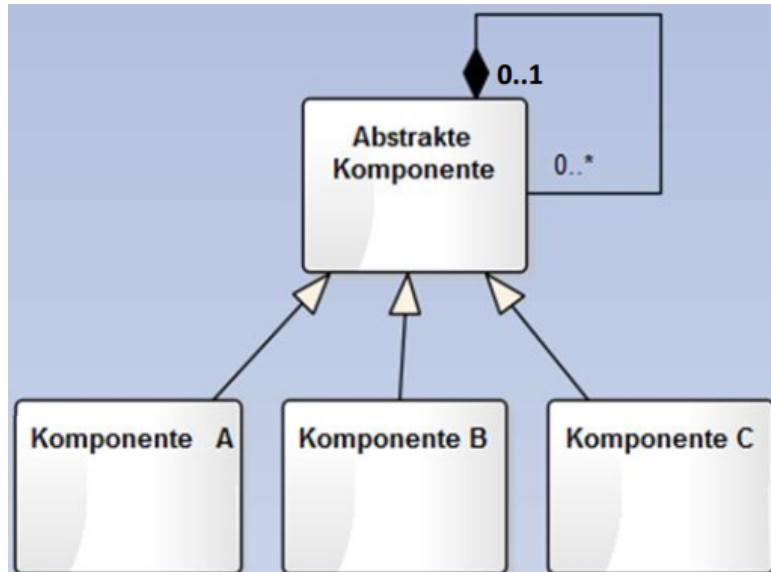
#### 4.7.4 Stückliste

Motivation:

- rekursive Beziehung zwischen Komponenten; die entstehende Struktur soll als Einheit behandelbar sein
- Komponente kann Komponenten enthalten (rekursive Struktur)
- Enthaltene Teile sind einzeln behandelbar

Modellierung:

- Komposition
- "Teil"-Objekte können "Aggregat"-Objekten zugeordnet werden
- Kardinalität an Aggregat: 0..1



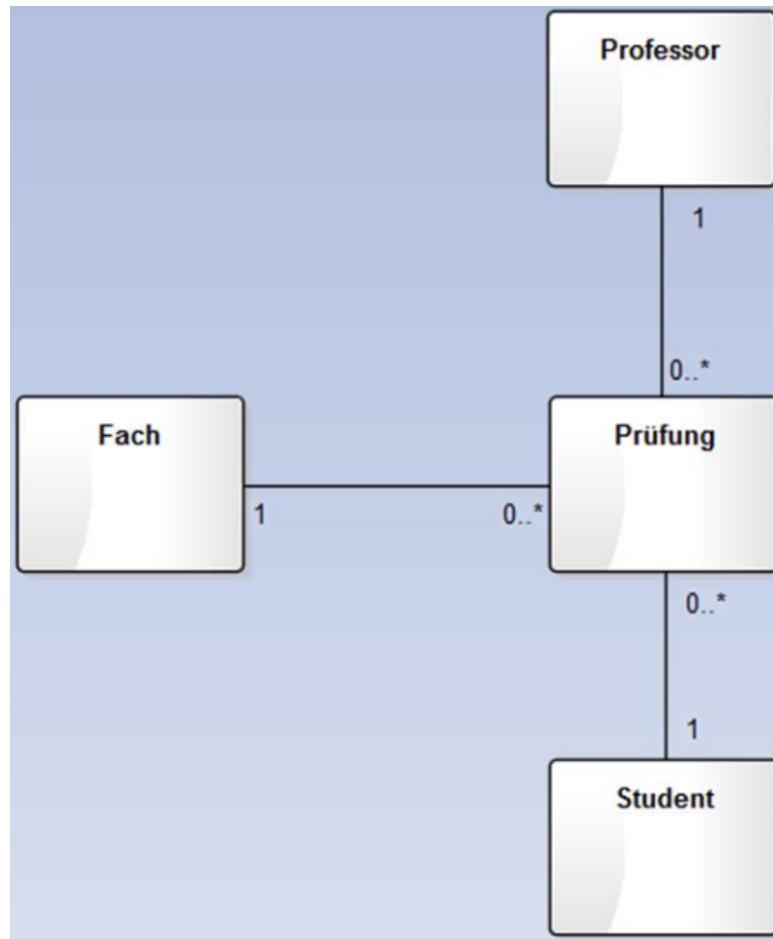
#### 4.7.5 Koordinator

Motivation:

- Mehrere Dinge stehen in Beziehung zueinander
- über die Beziehung werden Daten vorgehalten
- Oft stehen Objekte über ein drittes Objekt in Beziehung

Modellierung:

- Einfache Assoziationen
- Koordinator ersetzt n-äre Assoziation mit assoziativer Klasse
- Koordinator besitzt wenige Attribute bzw. Operationen, dafür aber mehrere Assoziationen zu anderen Klassen



#### 4.7.6 Rollen

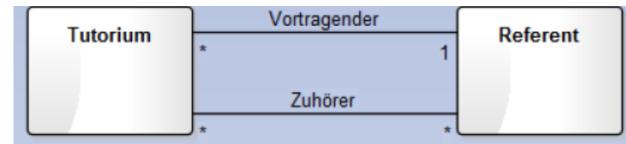
Motivation:

- Ein Objekt kann – zu einem Zeitpunkt – in Bezug auf die Objekte der anderen Klasse verschiedene Rollen spielen
- Unabhängig von der Beziehung werden die gleichen Attribute beziehungsweise Operationen innerhalb der beteiligten Objekte benötigt

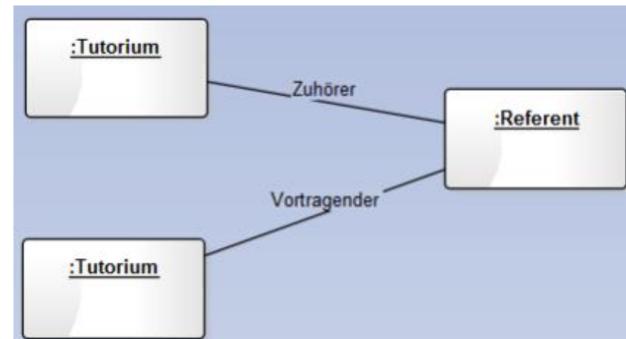
Modellierung:

- es existieren mehrere einfache Assoziationen zwischen zwei Klassen
- Ein Objekt kann in Bezug auf die Objekte der anderen Klasse verschiedene Rollen spielen
- Objekt in verschiedenen Rollen benötigen die gleichen Operationen und Attribute

Klassendiagramm:



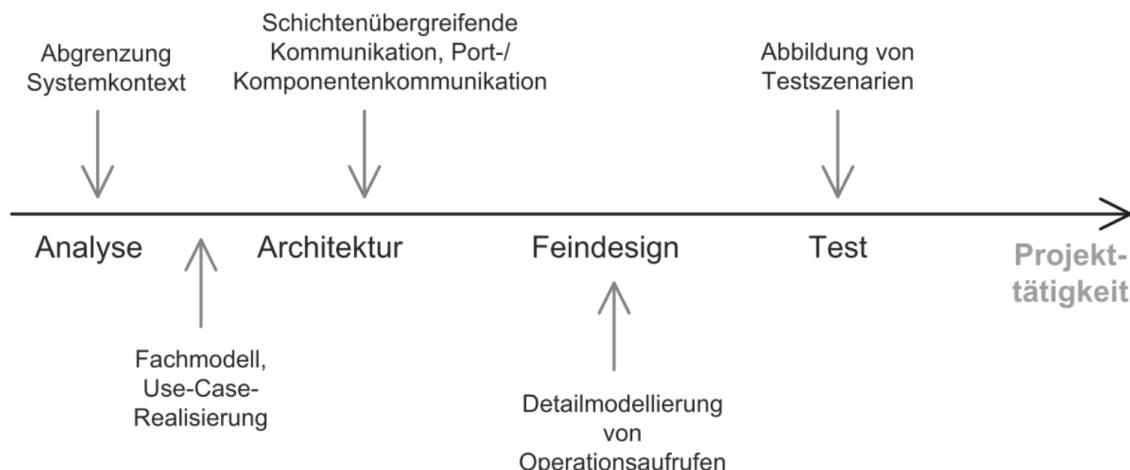
Objektdiagramm:



## 4.8 Sequenzdiagramme

*“Wie läuft die Kommunikation in meinem System ab?”*

- Das Sequenzdiagramm ist das meist verwendete Interaktionsdiagramm
- Es ist außerdem das mächtigste Interaktionsdiagramm



Fazit:

- Es lassen sich mit Sequenzdiagrammen Interaktionen auf verschiedenen Abstraktionsniveaus modellieren
- Sequenzdiagramme sind zu jedem Zeitpunkt im Projekt sinnvoll einsetzbar

Sequenzdiagramme zeigen den Informationsaustausch zwischen beliebigen Kommunikationspartnern innerhalb oder zwischen Systemen:

- häufig sind Kommunikationspartner Objekte von Klassen
- erlauben Modellierung von festen Reihenfolgen, zeitlichen und logischen Ablaufbedingungen, Schleifen und Nebenläufigkeit
- Kommunikationspartner werden durch Rechtecke mit Lebenslinien (gestrichelte Linie) gekennzeichnet
- Die Zeit schreitet von oben nach unten fort

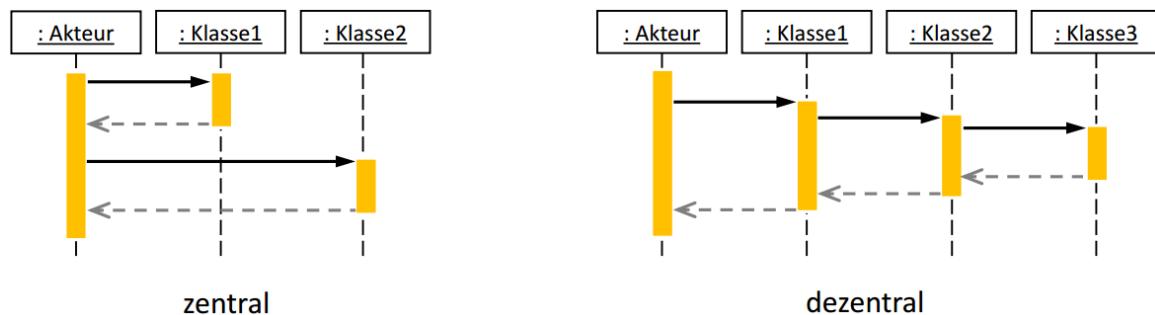
#### 4.8.1 Konstruktion von Sequenzdiagrammen

Grundidee:

- Identifikation aller relevanten Szenarien
- Erstellen eines Sequenzdiagramms für jedes Szenario

Vorgehen:

1. Pro Use Case mehrere Szenarios entwickeln
  - Standardausführung und Alternativen berücksichtigen
  - positive und negative Fälle unterscheiden (Verhalten das passieren soll oder nicht passieren darf)
2. Entwicklung eines Sequenzdiagramms
  - Beteiligte Klassen finden
  - Aufgaben in Operationen zerlegen
  - Reihenfolge der Operationen prüfen
3. Identifizierte Operationen den einzelnen Klassen zuordnen
4. Wie ist das Szenario zu strukturieren?
  - zentrale Struktur (fork diagram)
  - dezentrale Struktur (stair diagram)



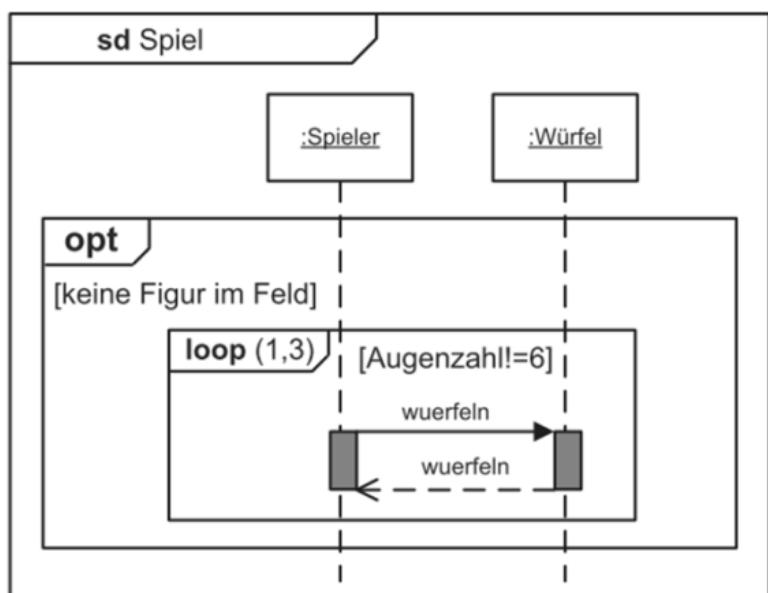
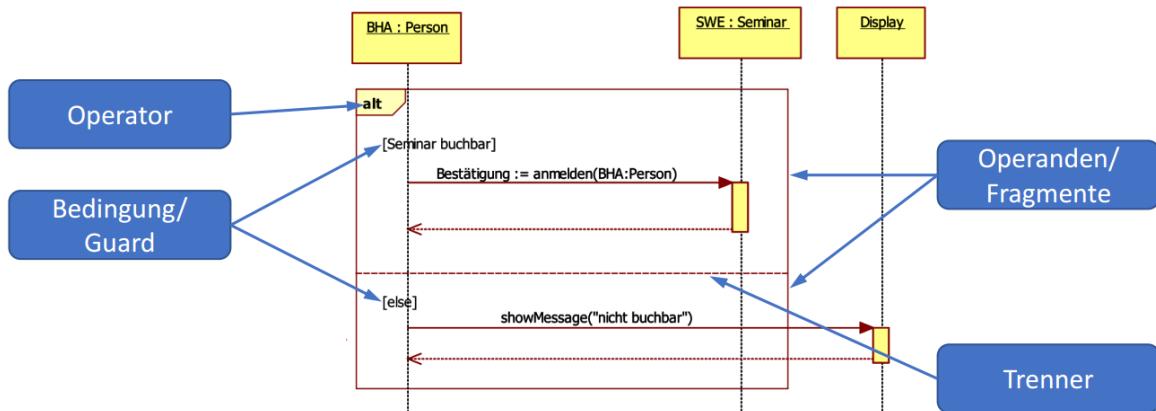
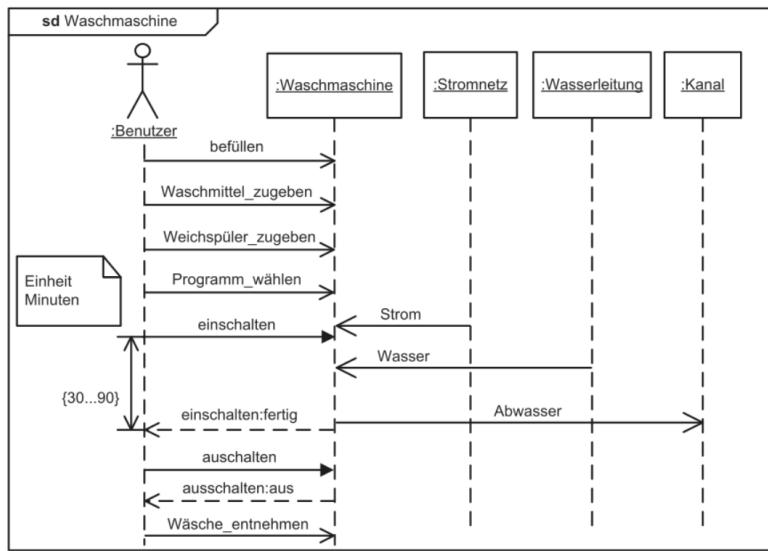
**Fehlerquelle:**

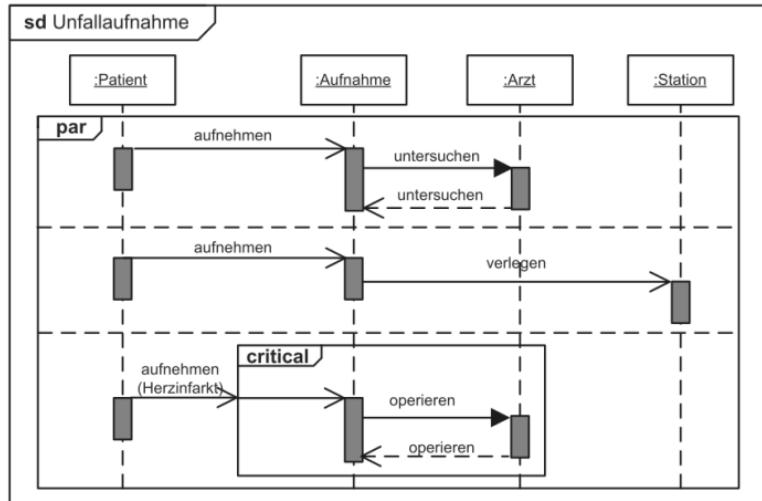
⇒ zu viele Details beschreiben

Stattdessen:

- auf die wichtigsten Szenarien beschränken
- nicht jeden Sonderfall beschreiben
- auf unnötige Details verzichten

#### 4.8.2 Notation&Beispiele





## 4.9 Kommunikationsdiagramme

*“Welche Teile einer komplexen Struktur arbeiten wie zusammen, um eine bestimmte Funktion zu erfüllen?”*

Das Kommunikationsdiagramm:

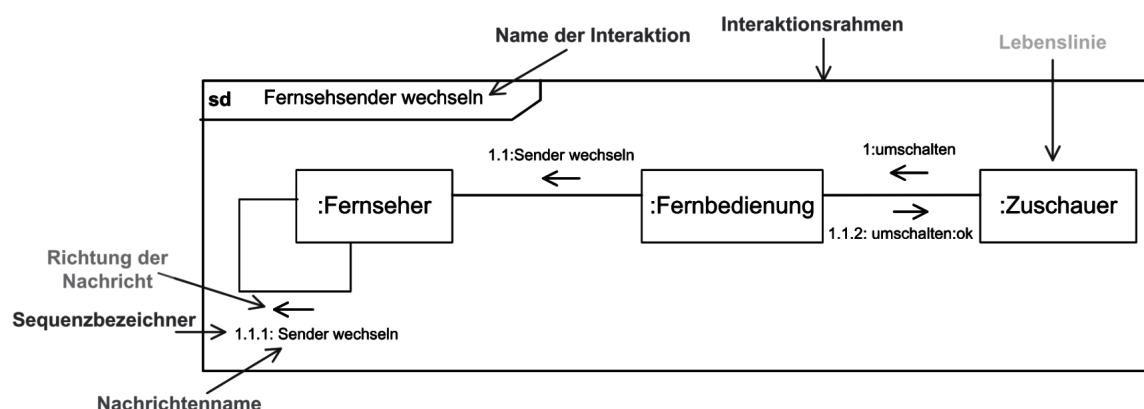
- zeigt Wechselspiel und Nachrichtenaustausch von Teilen einer komplexen Struktur

Anwendung im Projekt: (Geschäftsprozess-) Analyse

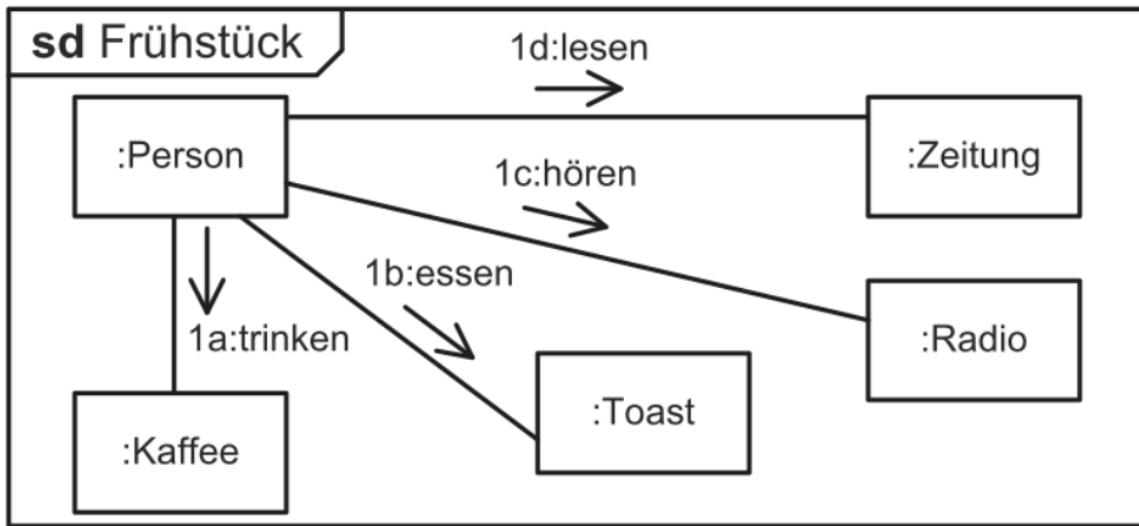
- Kommunikationsdiagramme integrieren daten- oder entitätsorientierte Modelle und Interaktionen auf eine sehr einfache Weise
- damit für alle Stakeholder verständlich
- Tipp: nach erstem Wurf des statischen Analysemodells (oder Geschäftsprozessmodells):
  - dynamische Zusammenhänge mittels Kommunikationsdiagramm visualisieren
  - Kommunikationspartner (nicht die Abläufe) stehen im Vordergrund des Diagramms

Beispiele:

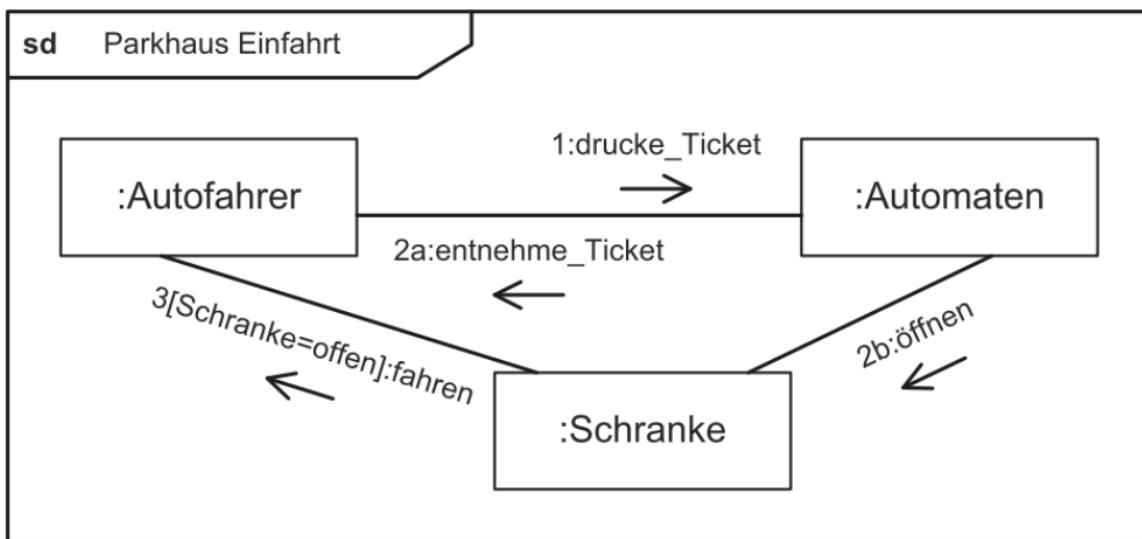
Sequenz



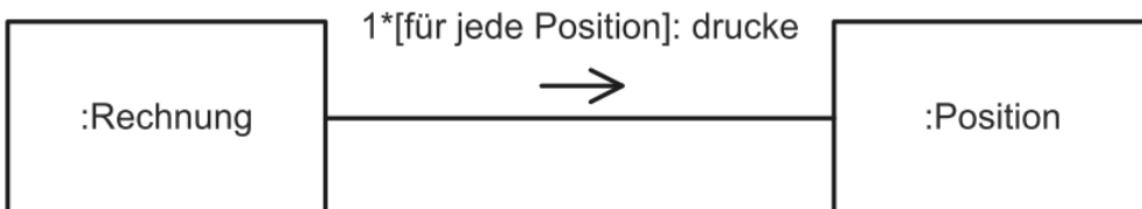
Nebenläufigkeit



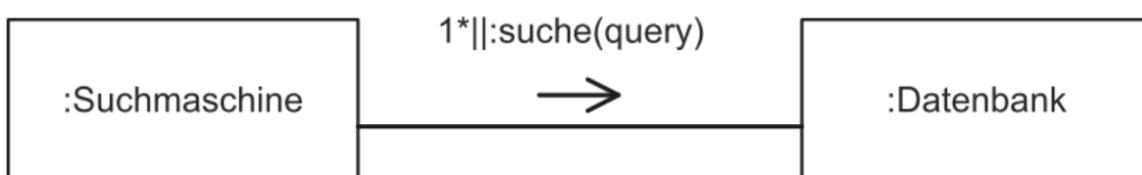
#### Bedingte Nachrichten



#### Iterative Nachrichten



#### Broadcast-Nachrichten



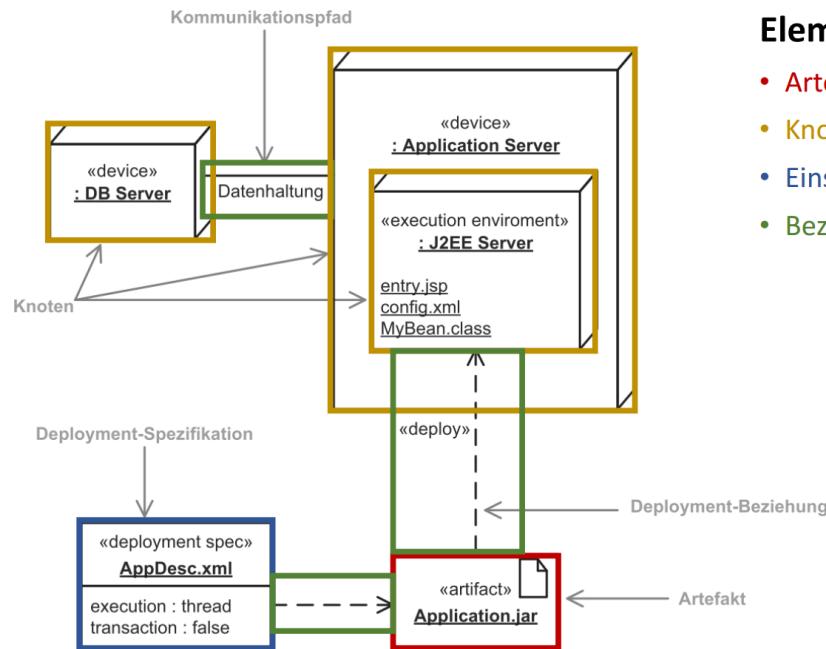
## 4.10 Verteilungsdiagramm

“Wie werden die Artefakte des Systems zur Laufzeit verteilt?”

(Verteilung: Installation, Konfiguration, Bereitstellung oder Ausführung)

Das Verteilungsdiagramm:

- zeigt die Zuordnung von Artefakten (also u.a. Softwarekomponenten) auf Hardware-Einheiten, die als Knoten bezeichnet werden
- stellt die Kommunikationsverbindungen und Abhängigkeiten zwischen Knoten dar



### Elemente:

- Artefakte
- Knoten
- Einsatzspezifikationen
- Beziehungen zwischen Knoten

Grundsätzlich:

- nur sinnvoll, wenn es auch wirklich etwas zu verteilen gibt

Dann ist es an zwei Stellen sinnvoll:

- Design-Phase:  
→ “Wie sieht eine Verteilung auf Umgebungen im Prinzip aus?”
- Übergabe in den Betrieb:  
→ “Wie sieht die konkrete Verteilung aus (Rechnernamen, IP-Adressen, Portnummern, etc.)?”

## 4.11 Rueckblick

Rueckblick UML

## 5 Objekt Orientiertes Design

- Ziele Aktion der Designphase
- Sichtenmodel
- Architekturprinzipien/Muster

- Pro/Contras
  - Anwenden können
- 

## 5.1 Grundlagen

### 5.1.1 Ziele/Aktion der Designphase

Ziel:	Wie und womit erfolgt die Realisierung?
Aktionen:	<ul style="list-style-type: none"> <li>• Ermitteln und/oder Festlegen von Umgebungs- und Randbedingungen</li> <li>• Grundsatzentscheidungen treffen</li> <li>• Spezifikation der Systemkomponenten</li> <li>• Programmierung im Großen</li> </ul>
Ergebnis:	<ul style="list-style-type: none"> <li>• Softwarearchitekturmodell</li> <li>• Systemkomponenten und Beziehung untereinander</li> <li>• Schnittstellen zwischen Komponenten</li> <li>• Schnittstellen zur Umgebung des Produkts</li> </ul>

### 5.1.2 Sichtenmodell

*“Eine Sicht ist eine Repräsentation eines Gesamtsystems aus einer festgelegten Perspektive“*

#### Problem:

- Softwarearchitekturen i.d.R. komplex
- Einzelne Darstellung kann Vielschichtigkeit und Komplexität nicht ausdrücken
- Architekturbeschreibung für verschiedene Stakeholder mit unterschiedlichsten Informationsbedürfnissen wichtig

#### Beispiel:

Auftraggeber oder Projektmanager benötigen andere Informationen als Entwickler, Qualitätsmanagement oder Betreiber der Software

## Lösung: Sichten

- unterschiedliche Sichten ermöglichen Konzentration auf das jeweils Wesentliche
- reduzieren Darstellungskomplexität (s. Beispiel Hausbau)

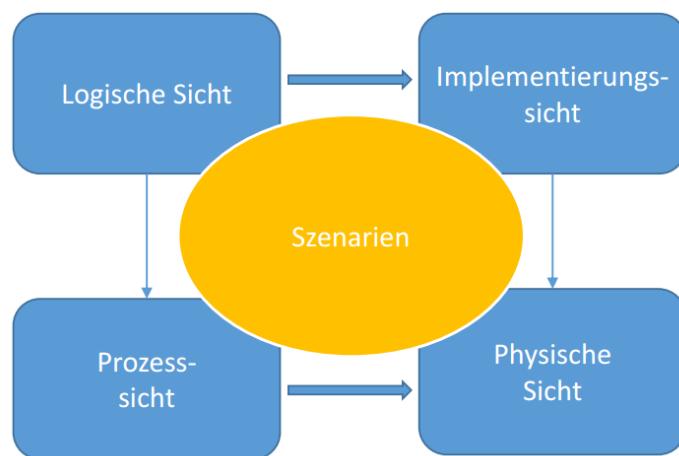
## Eigenschaften einer Sicht:

- beschreibt nur gewisse Eigenschaften eines Gesamtsystems (Selektivität)
- braucht eine Beschreibungstechnik (Plan oder Modell)
- ist meist nicht vollst. unabhängig von anderen Sichten (Idealfall: Orthogonalität)
- Sichten sollten parallel bearbeitet werden können (Schnittstellenproblematik)
- Zwei Sichten eines Systems sollten sich nicht widersprechen (Konsistenz)
- Summe aller Sichten sollte in eine Gesamtsicht münden

## Analogie zum Hausbau:

Plan/Sicht	Bedeutung	Format	Nutzer
Grund- und Aufriss	Lage und Beschaffenheit von Mauern, Maueröffnungen, Böden, Decken	Normiert nach DIN	Maurer, Käufer
Elektroplan	Lage von spannungsführenden Leitungen, Schaltern, Steckdosen, ...	Normiert nach DIN	Architekt, Käufer, Elektriker, Küchenbauer, ...
Heizungs-, Wasser- und Sanitärplan	Lage von Wasser- und Abwasserleitungen, Heizungsrohren sowie Gasleitungen	Normiert nach DIN	Architekt, Heizungsbauer, Käufer, Küchenbauer, ...
3D-Modell	Dreidimensionale Darstellung des Gebäudes im Ganzen oder Teilen	Beliebig, Bilder oder Filme	Käufer, Verkäufer
Raumplan	Zweidimensionale Darstellung von Zimmern und Einrichtung	Beliebig, angelehnt an DIN	Käufer, Innenarchitekt, Küchenbauer

## 5.1.3 4+1 Sichten Modell nach Kruchten



### Logische Sicht:

Adressat: Endanwender

Fokus:

- Funktionalität für Endanwender
- funktionale Anforderungen
- welche Dienste werden dem Nutzer vom System bereitgestellt

Hilfsmittel u.a.:

- Klassendiagramme
- Sequenzdiagramme
- Kommunikationsdiagramme

**Implementierungssicht:**

Adressat: Entwickler, SW-Manager

Fokus:

- Systembeschreibung aus Entwicklersicht
- statische Organisation der SW
- Software-Management

Hilfsmittel:

- Komponentendiagramm
- Paketdiagramm

**Prozesssicht:**

Adressat: Integrierer

Fokus:

- dynamische Aspekte des Systems
- nicht-funktionale Anforderungen (Skalierbarkeit, Parallelität, Verteilung, Performanz)

Hilfsmittel:

- Aktivitätsdiagramm

**Physische Sicht:**

Adressat: System-Engineers

Fokus:

- nicht-funktionale Anforderungen mit Hinblick auf Hardware (Zuverlässigkeit, Erreichbarkeit, Performanz)
- Verteilung und Kommunikation der HW-komponenten (physische Ebene)

Hilfsmittel:

- Verteilungsdiagramm

**Szenarien:** (wichtige Anwendungsfälle)

Adressat: alle Stakeholder

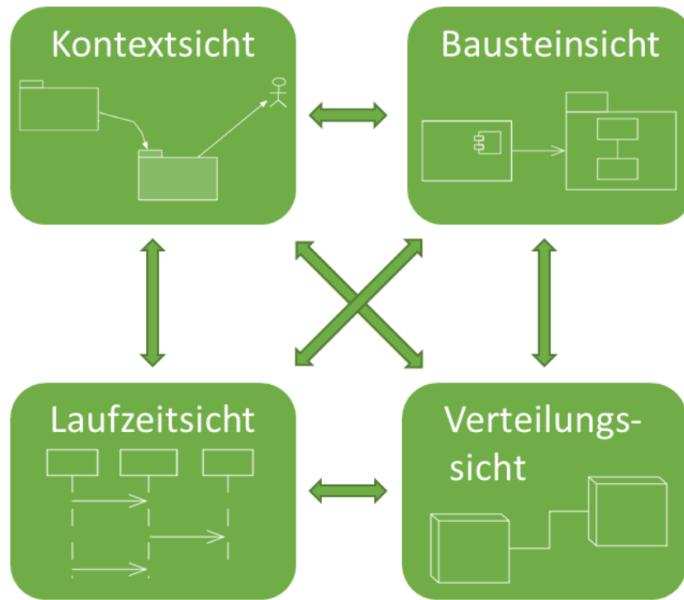
Fokus:

- Systemkonsistenz
- Ablaufbeschreibung zwischen Komponenten
- Architektur überprüfen

Hilfsmittel:

- Use-Case-Diagramm
- Sequenzdiagramm

#### 5.1.4 Sichten nach Starke



**Kontextsicht:** Wie ist System in Umgebung eingebettet?

Beschreibt:

- System als Black-Box (Außenansicht)
- aus Vogelperspektive
- Schnittstellen zu Nachbarsystemen
- Interaktion mit wichtigen Stakeholdern
- wesentliche Teile der Infrastruktur

Zweck:

- abstrakte Beschreibung (Vision) des Systems

⇒ **Kontextsicht eine Abstraktion der übrigen Sichten mit Fokus auf das Umfeld des Systems**

Darstellung der Systemstruktur:

Klassendiagramme angereichert um Pakete und Komponenten

Darstellung von Schnittstellen zur Umwelt:

in Klassendiagrammen über Assoziationen zu Fremdsystemen oder Akteuren

Anwendungsfälle oder Abläufe:

dynamische UML-Diagramme (Sequenz-, Kommunikations-, Aktivitätsdiagramme)

Technische Systemumgebung:

Verteilungsdiagramme

**Bausteinsicht:** Wie ist das System intern aufgebaut?

Beschreibt:

- statische Strukturen des Systems
- Subsysteme, Module, Pakete, Komponenten, Klassen etc.
- Abhängigkeiten zwischen den Bausteinen

Zweck:

- Unterstützen Projektleiter und Auftraggeber bei Projektüberwachung
- dienen Zuteilung von Arbeitspaketen
- Referenz für Softwareentwickler

Notation:

- Klassendiagramme (Kardinalitäten, Aggregation, Komposition, Generalisierung)
- Komponentendiagramme
- Paketdiagramme

**Laufzeitsicht:** Wie läuft das System ab?

Beschreibt:

- welche Bausteine existieren zur Laufzeit
- wie interagieren Bausteine miteinander
- dynamische Strukturen (im Gegensatz zur statischen Bausteinsicht)

Beispiele:

- Welche Instanzen von Architekturbausteinen gibt es zur Laufzeit und wie werden diese gestartet, überwacht und beendet?
- Wie startet das System (z.B.: notwendige Startskripte, Abhängigkeiten von externen Subsystemen, Datenbanken, Kommunikationssystemen etc.)?

Notation:

- Sequenzdiagramme
- Kommunikationsdiagramme
- Aktivitätsdiagramme

**Verteilungssicht: (Infrastruktursicht)** In welcher Umgebung läuft das System ab?

Beschreibt:

- Hardwarekomponenten
- Rechner, Prozessoren, Speicher, Netztopologie
- sonstige Bestandteile der physischen Systemumgebung

Beispiele:

- welche Architekturbausteine laufen auf welchen Hardwarekomponenten
- Z.B. Computer, Netzwerktopologie, Netzwerkprotokolle und sonstige Bestandteile der physischen Systemumgebung (z.B. Speicher, Router, Firewalls)

**Entwurf von Sichten:** Entwurfsprozess von Sichten ist von deren starken Wechselwirkungen und Abhängigkeiten geprägt

⇒ **Iteratives Vorgehen**

Zunächst:

- Kontextsicht

Anschließend:

- Bausteinsicht:  
wenn Sie bereits ähnliche Systeme entwickelt und genaue Vorstellung von benötigten Implementierungskomponenten haben
- Laufzeitsicht:  
wenn Sie bereits erste Vorstellungen wesentlicher Architekturbausteine haben und deren Verantwortlichkeiten und Zusammenspiel klären wollen
- Verteilungssicht:  
wenn Sie viele Randbedingungen und Vorgaben durch die technische Infrastruktur, Rechenzentrum oder Administratoren des Systems bekommen

## 5.2 Architektur

### 5.2.1 Merkmale schlechter Software

- steif (rigid)
  - selbst kleine Änderungen sind schwierig/aufwendig/riskant
  - State-of-the-art-Funktionen bleiben unimplementiert
  - Symptome: nicht-kritische Fehler bleiben unbeseitigt
- zerbrechlich (fragile)
  - kleine Fehler lösen Fehler in entfernten Softwarebereichen aus
  - Symptome: Großteil der Entwicklungsarbeit fließt in Bugfixing
- unbeweglich (immobile)
  - viele Querverbindungen/Abhängigkeiten
  - Software nicht einmal in Teilen wiederverwendbar
  - Symptome: parallele Produktlinien mit ähnlicher Funktionalität und
- zäh (viscous)
  - Änderungen gegen den Entwurfsgedanken sind leichter umsetzbar als Änderungen im Sinne des Entwurfsgedankens
  - Symptome: nicht genutzter und redundanter Code, irrelevante Dokumentation

### 5.2.2 Eigenschaften eines guten OO-Entwurfs

- Einhaltung von Prinzipien:
  - SOLID: Single Responsibility, Open/Closed Principle, LSP, Interface Segregation, Dependency Inversion
  - GRASP: General Responsibility Assignment Patterns (Menge von Entwurfsmustern)
  - ...
- Die Prinzipien führen auf eine Struktur der Lösung
  - Softwarearchitektur
  - Softwaredesign
- Korrektheit
  - Erfüllung der Anforderungen
  - Wiedergabe aller Funktionen des Systemmodells
  - Sicherstellung der nichtfunktionalen Anforderungen
- Verständlichkeit & Präzision
  - Gute Dokumentation
- Anpassbarkeit
- Hohe Kohäsion innerhalb der Komponenten
- Schwache Kopplung zwischen den Komponenten
- Wiederverwendung

⇒ Diese Kriterien gelten auf allen Ebenen des Entwurfs (Architektur, Subsysteme, Komponenten).

### 5.2.3 Kohäsion

*Kohäsion ist ein Maß für die Zusammengehörigkeit der Bestandteile einer Komponente*

- Hohe Kohäsion einer Komponente erleichtert:

- Verständnis, Wartung und Anpassung
- Hohe Kohäsion wird erreicht durch:
  - Prinzipien der Objektorientierung (Datenkapselung)
  - Einhaltung von Regeln zur Paketbildung
  - Verwendung geeigneter Muster zu Kopplung und Entkopplung

#### 5.2.4 Kopplung

*Kopplung ist ein Maß für die Abhängigkeiten zwischen Komponenten*

- Geringe Kopplung erleichtert die Wartbarkeit und macht Systeme stabiler
- Arten der Kopplung:
  - Datenkopplung (gemeinsame Daten)
  - Schnittstellenkopplung (gegenseitiger Aufruf)
  - Strukturkopplung (gemeinsame Strukturelemente)
- Reduktion der Kopplung:
  - Kopplung kann nie auf Null reduziert werden!
  - Datenkopplung vermeiden!  
aber durch Objektorientierung meist gegeben
  - Strukturkopplung vermeiden!  
z.B. keine Vererbung über Paketgrenzen hinweg
- Entkopplungsbeispiel: getter/setter-Methoden statt direkter Attributzugriff

#### 5.2.5 Interne Wiederverwendung

*Interne Wiederverwendung (reuse) ist ein Maß für die Ausnutzung von Gemeinsamkeiten zwischen Komponenten*

- Reduktion der Redundanz
- Erhöhung der Stabilität und Ergonomie
- Hilfsmittel für Wiederverwendung:
  - im objektorientierten Entwurf: Vererbung, Parametrisierung
  - im modularen und objektorientierten Entwurf: Module/Objekte mit allgemeinen Schnittstellen (Interfaces)
  - Strukturkopplung (gemeinsame Strukturelemente)
- Aber: Wiederverwendung kann die Kopplung erhöhen:
  - Schnittstellenkopplung und Strukturkopplung

#### 5.2.6 Probleme zerlegen

- In Scheiben schneiden:
  - Zerlegen Sie ein System in (horizontale) Schichten
  - Jede Schicht stellt einige klar definierte Schnittstellen zur Verfügung
  - nutzt Dienste von darunter liegenden Schichten
- In Stücke schneiden:
  - Zerlegen Sie ein System in (vertikale) Teile
  - Jeder Teil übernimmt eine bestimmte fachliche oder technische Funktion

- Kapselung:
  - Teile (Komponenten) als Black-Box betrachten
  - mit der Umgebung über klar definierte Schnittstellen kommunizieren
- Verwenden Sie etablierte und erprobte Strukturen wieder. Erfinden Sie möglichst wenig neu, sondern halten Sie sich an Bewährtes.
- Entwerfen Sie in Iterationen. Überprüfen Sie die Stärken und Schwächen eines Entwurfes anhand von Prototypen oder technischen Durchsichten.
- Bewerten Sie diese Versuche explizit und überarbeiten daraufhin Ihre Strukturen – wenden Sie es beim Entwurf von Strukturen intensiv und dauernd an.
- Dokumentieren Sie die Entscheidungen, die zu einer bestimmten Struktur führen. Andere Projektbeteiligte werden sie künftig verstehen müssen.

### 5.2.7 Fachdomäne nach Evans

#### Isolierung der Fachdomäne

User Interface (Presentation Layer)	Darstellung und Informationsanzeige, nimmt Eingaben und Kommandos von Nutzern entgegen
Application Layer	beschreibt oder koordiniert Geschäftsprozesse, delegiert an den Domain Layer oder den Infrastructure Layer
Domain Layer (Fachmodellschicht)	Kern von DDD (Domain Driven Design)! Schicht repräsentiert Fachdomäne. Hier „lebt“ das Modell mit seinem aktuellen Zustand. Persistenz seiner Entitäten delegiert diese Schicht an Infrastructure Layer
Infrastructure Layer	Allgemeine technische Services (z.B. Persistenz, Kommunikation mit anderen Systemen)

#### Verwaltung der Fachdomäne

- **Aggregate:**

Sie kapseln vernetzte (d.h. miteinander assoziierte) Domänenobjekte. Ein Aggregat hat grundsätzlich eine einzige Entität als Wurzelobjekt. Diese Wurzel ist der einzige Einstiegspunkt in das Aggregat, sämtliche mit der Wurzel verbundene Domänenobjekte sind lokal. Objekte von außen dürfen nur Referenzen auf die Wurzelentität halten.

- **Factories:**

Entities und insbesondere Aggregate können komplexe Strukturen vernetzter Objekte bilden, die Sie nicht über triviale Konstruktoraufrufe erzeugen können oder wollen. Verwenden Sie Factories, um die Erzeugung von Aggregaten und Entitäten zu kapseln. Factory-Objekte arbeiten ausschließlich innerhalb der Domäne und haben keinen Zugriff auf den Infrastruktur-Layer.

- **Repositories:**

Alle Arten von Objekten (sowohl aus dem Domain Layer wie auch dem Application Layer) benötigen eine Möglichkeit, die Objektreferenzen anderer Objekte zu erhalten. Repositories kapseln die technischen Details der Infrastrukturschicht gegenüber den Domänenobjekten. Dadurch bleibt das Domänenmodell auch in dieser Hinsicht „technologiefrei“. Repositories beschaffen beispielsweise Objektreferenzen von Entitäten, die aus Datenbanken gelesen werden

### 5.2.8 Architekturprinzipien

#### Principles vs Patterns (Gebot vs Rezept)

- Principle (Prinzip): "Was Du nicht willst, das man Dir tut, . . ."
  - abstrakte Handlungsanweisung (z.B.: DRY)
  - auf einen „höheren“ Zweck gerichtet (Gebot)
  - kann auch ein (nicht-konstruktives) Bewertungskriterium sein
- Pattern (Muster): "Wenn Dir einer Eine Ohrfeige gibt, dann halte auch. . ."
  - Lösungsschablone (erprobte Lösung für ein verbreitetes Problem)
  - auf einen konkreten Zweck gerichtet (Rezept: „Wenn DIES, tue DAS“)
  - ist konstruktiv

## Beispiele

- Lose Kopplung
- Hohe Kohäsion
- Open Closed Principle
- Information Hiding
- Separation of Concerns
- Abhängigkeit nur von Abstraktionen
- Dependency Injection
- Liskovsches Substitutionsprinzip
- Abtrennung von Schnittstellen
- Keine zyklischen Abhängigkeiten
- . . .

### 5.2.9 Hauptprinzipien des Architekturentwurfs

Eine gute Software-Architektur verfügt über:

1. Geringe Kopplung (leichter Austausch von Komponenten)
  - Datenkopplung (gemeinsame Daten)
  - Schnittstellenkopplung (gegenseitiger Aufruf)
  - Strukturkopplung (gemeinsame Strukturelemente)
2. Hohe Kohäsion ("Unteilbarkeit" der Komponenten)

#### Ziel: Reduktion von Kopplung

- Kopplung kann nie auf Null reduziert werden
- Schnittstellenkopplung ist akzeptabel
- Datenkopplung vermeiden
- Strukturkopplung vermeiden (z.B. keine Vererbung über Paketgrenzen)

Generelles Ziel:

→ Reduktion der Kopplung zwischen verschiedenen Modulen

#### Ziel: Hohe Kohäsion

- Kohäsion = Zusammenhalt
- Dinge in Struktureinheiten zusammenfassen, die inhaltlich zusammengehören
- Kohäsion ist ein Maß für die Zusammengehörigkeit der Bestandteile einer Komponente
- Hohe Kohäsion einer Komponente erleichtert Verständnis, Wartung und Anpassung

Erreichung hoher Kohäsion durch:

- Prinzipien der Objektorientierung (Datenkapselung)
- Einhaltung von Regeln zur Paketbildung
- Verwendung geeigneter Design-Patterns zu Kopplung und Entkopplung

Generelles Ziel

→ hohe Kohäsion innerhalb eines Moduls oder Subsystems

### 5.2.10 Architekturprinzipien

#### Don't Repeat Yourself DRY Principle

- Wiederholungen:
  - machen Arbeit
  - werden irgendwann inkonsistent
  - erschweren das Verständnis
  - verschlechtern Performance beim Kompilieren und zur Laufzeit
- Vermeide Wiederholungen auf allen Ebenen der Softwareerstellung:
  - Code
  - Dokumentation
  - Tests
  - Datenbankschemata
  - ...

#### Open Closed Principle OCP

Module (= Klassen, Funktionen, Packages, ... )

- sollen für Erweiterungen offen und
- für Veränderungen/Modifikationen geschlossen sein

→ Eine **Erweiterung** in diesem Sinne:

- verändert das vorhandene Verhalten der Einheit nicht,
- vielmehr erweitert es die Einheit um zusätzliche Funktionen oder Daten

→ Eine **Modifikation** hingegen würde das bisherige Verhalten der Einheit ändern

Ziele:

- neue Eigenschaften hinzufügen, ohne bestehenden Code modifizieren zu müssen
- sicherstellen, dass künftige Erweiterungen ohne Nebenwirkungen funktionieren

Generelle Anmerkungen zu OCP:

- Schlüssel zu OCP ist Polymorphie (und dynamische Bindung)
- Klassen können durch Ableitung zusätzliches Verhalten bereitstellen, ohne die Basisklasse zu verändern
- OO-Sprachen machen Umsetzung des OCP daher leicht
- Was tut man bei nicht-OO-Sprachen?
  - ⇒ OCP im Konzept verwenden

## **Interne Wiederverwendung reuse**

ist ein Maß für die Ausnutzung von Gemeinsamkeiten zwischen Komponenten

- Reduktion der Redundanz
- Erhöhung der Stabilität und Ergonomie
- Hilfsmittel für Wiederverwendung:
  - im objektorientierten Entwurf: Vererbung, Parametrisierung
  - im modularen und objektorientierten Entwurf: Module/Objekte mit allgemeinen Schnittstellen (Interfaces)
- Aber: Wiederverwendung kann die Kopplung erhöhen:
  - Schnittstellenkopplung und Strukturkopplung

Generelles Ziel:

→ hohe Wiederverwendbarkeit (aber nicht um jeden Preis! )

## **Information Hiding Geheimhaltungsprinzip**

- Verbergen der Funktionsweise eines Systemteils im Inneren des Moduls
- nach außen nur Informationen bekannt, die über Schnittstelle explizit zur Verfügung gestellt werden
- Schnittstelle gibt so wenig wie möglich nach außen bekannt
- Umsetzung mithilfe der Objektorientierung:
  - Kapselung → Information Hiding
    - Kapselung: Attribute und Methoden sinnvoll zusammengefügt
    - Information Hiding: innere Struktur geht Clients nichts an und Änderungen daran betreffen sie nicht
  - Zugriff auf Objektzustand nur über wohldefinierte Methoden
  - Objekt selbst ist verantwortlich und kann über die Zulässigkeit entscheiden

## **Separation of Concerns SoC**

- Concern:
  - Menge von Informationen/Wissensbereiche, die den Programmcode beeinflussen
  - Concerns sollten nach Möglichkeit orthogonal zueinander sein
- Separation of Concerns:
  - Concerns in spezialisierte Funktionseinheiten (Systemteile) ausgliedern
  - jede Komponente einer Architektur nur für eine einzige Aufgabe zuständig
- Beispiel:
  - Logging, Transaktionalität, Caching
- Warum?
  - Wenn Codeeinheit keine klare Aufgabe hat, ist es schwer sie zu verstehen, zu korrigieren oder zu erweitern
- Ziel:
  - Entwicklung und Wartung von Programmen vereinfachen

## Dependency Inversion Principle DIP

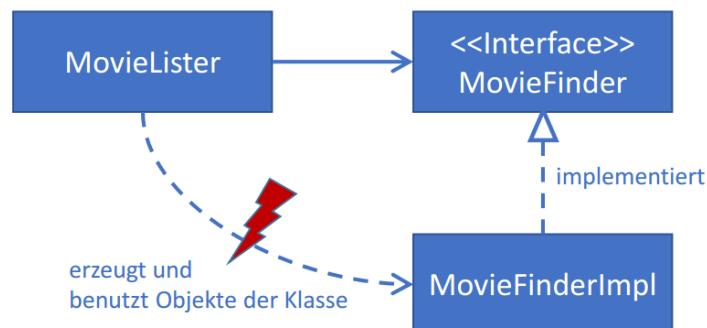
- Abhängigkeiten nur von Abstraktionen erlauben, nicht von konkreten Implementierungen
- Warum?
  - Hängen Module voneinander ab oder sogar Abstraktionen von Implementierungen, so besteht die Gefahr von
    - erhöhter Kopplung und
    - zyklischen Abhängigkeiten
- Ziel:
  - Invertierung der Abhängigkeiten
  - Module auf höheren Ebenen definieren Schnittstelle, niedrigere realisieren sie
  - Beide hängen von Abstraktionen ab

## Dependency Injection DI

Frage:

Wie bekommt man zur Laufzeit Objekte des benutzten Interfaces? (Informationsbereitstellung zur Laufzeit)

Wenn MovieLister das selbst regelt, erhält man wieder eine Kopplung der konkreten Klassen

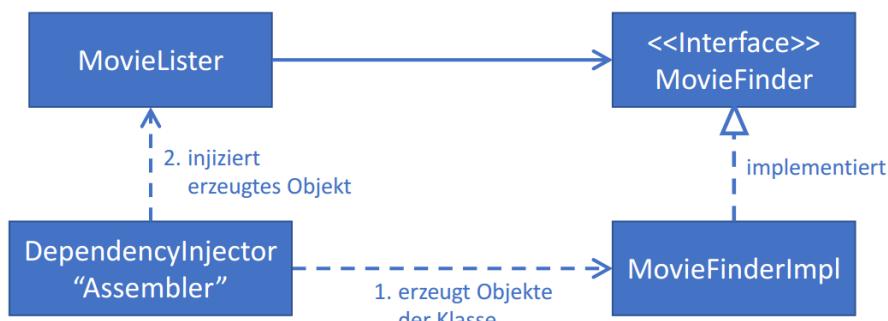


Idee:

Objekte zur Laufzeit erzeugen und dem MovieLister übergeben (inversion of control)

Lösung:

DependencyInjector (Assembler) als eigener Baustein



Formen von Dependency Injection:

- Constructor Injection
- Interface Injection
- Setter Injection

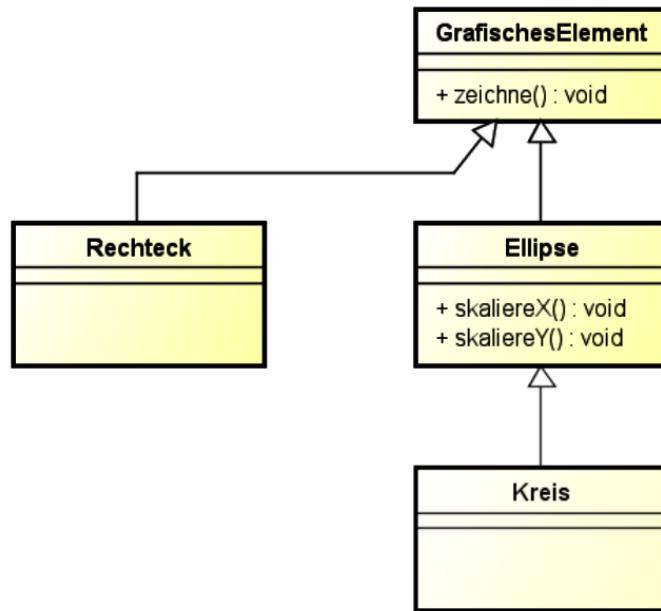
## Liskovsches Substitutionsprinzip LSP

Forderung:

Programm das Objekte einer Basisklasse T verwendet muss auch mit Objekten einer abgeleiteten Klasse S korrekt funktionieren, ohne dabei das Programm zu verändern

Beispiel:

Kreis-Ellipse-Problem (Verletzung des LSP)



Verletzung des LSP, da Kreis die Methoden `skaliereX(...)` und `skaliereY(...)` nicht haben darf (er besitzt nur einen Radius) damit darf hier Kreis nach dem LSP keine Unterklasse von Ellipse sein

Beachte:

die Entscheidung ist jeweils abhängig vom konkreten Fall:

würde Ellipse hier die beiden Memberfunktionen nicht aufweisen, dürfte Kreis Unterklasse von Ellipse sein

## 5.3 Architekturmuster

Idee:

- System in mehrere Schichten aufteilen
- Jede Schicht fasst logisch zusammengehörige Komponenten zusammen
- Jede Schicht stellt Dienstleistungen über Schnittstellen zur Verfügung
- Jede Schicht darf nur auf jeweilige Vorgängerschicht zugreifen (strikte Architektur)

Eigenschaften (Vorteile):

- Schichten sind nur gekoppelt, wenn sie benachbart sind
- Koppelung auf Schnittstellenebene (vertretbare Kopplung)
- Änderungen wirken sich meist nur lokal aus
- Eine Schicht kann aus mehreren entkoppelten Teilen mit intern hohem Zusammenhalt bestehen

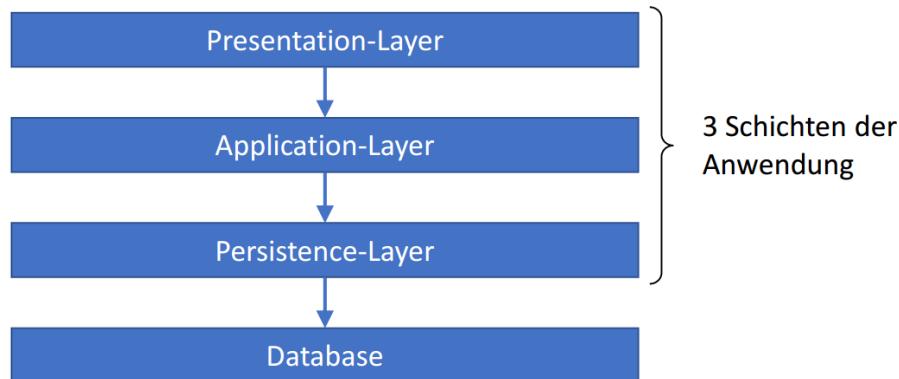
Unterscheidung von Schichtenarchitekturen:

- Strikte Schichtenarchitektur/Protokollbasierte Schichten:
  - Zugriff ausschließlich auf die nächst niedrigere Schicht

- Objektorientierte Schicht:
  - Zugriff auf alle tieferen Schichten

### 5.3.1 Die klassische 3-Schichten-Architektur

- Häufig eingesetzt für interaktive Systeme
- Protokollbasierte Schichtung



**Vorteile** von Schichtenarchitekturen:

- Schichten voneinander unabhängig (sowohl bei Erstellung als auch im Betrieb)
- Schichten über verschiedene Rechnerknoten verteilbar
- Implementierung austauschbar (sofern die gleichen Dienste angeboten werden)
- Schichtenbildung minimiert Abhängigkeiten zwischen Komponenten
- leicht verständliches Strukturkonzept

**Nachteile** von Schichtenarchitekturen:

- kann Performance beeinträchtigen  
(z.B. bei Anfragen durch mehrere Schichten)
- Schicht-übergreifende Änderungen werden schlecht unterstützt  
(z.B. neues Datenfeld, das sowohl gespeichert als auch in Nutzeroberfläche angezeigt werden soll zieht Änderungen in allen Schichten mit sich)

Hinweise:

- Vermeide Aspekte der Fachdomäne in die Präsentationsschicht zu verlagern:
  - Resultat: schwer wartbare Systeme
  - Reduziert Möglichkeit der Wiederverwendung innerhalb der Fachdomäne
- Wenn viele externe Ressourcen (Fremdsysteme) integriert werden:
  - Aufteilung der Infrastrukturschicht in Integrationsschicht und Ressourcenschicht

Beispiel:

- OSI-Schichtenmodell

### 5.3.2 Pipes und Filter

Idee:

- Struktur für Systeme, die Datenstrom verarbeiten
- Verarbeitungsschritte in Filter gekapselt (Vorteil: getrennt entwickelbar)
- Daten werden von einem Subsystem zum nächsten transportiert (pipe)
- Jedes Subsystem transformiert (filtert) die Daten

Herkunft:

- Pipes in der Unix-Shell (einfache Aufgaben zu komplexeren verknüpfen)

Beispiel:

- Compiler als Pipe:



Arten von Filtern:

- Aktiv: eigenständig laufende Prozesse oder Threads (von übergeordnetem Programm aufgerufen, ab dann laufen sie selbstständig)
- Passiv: durch Aufruf eines benachbarten Filterelementes aktiviert
  - Push-Mechanismus: Filter wird aktiv, wenn ihm Daten zugeschoben werden
  - Pull-Mechanismus: Filter wird aktiv, wenn nachfolgender Filter Daten anfordert

Zusammenarbeit zwischen Pipes und Filtern:

- Filter stellt Verarbeitung vollständig fertig, übergibt aktiv das Ergebnis an Pipe, die es zum nachfolgenden Filter transportiert
- Pipe erfragt das nächste Ergebnis bei Eingangsfilter und übergibt es an Ausgangsfilter
- Zentrale Steuerung koordiniert das Zusammenwirken der Filter
- Filter übergeben Ergebnisse "stückchenweise" über Pipes an nachfolgenden Filter; viele Pipes und Filter sind gleichzeitig aktiv (parallele Verarbeitung)

Pipes entkoppeln Filter:

- Pipes können direkt oder zeitversetzt Daten transportieren
- Pipes können entscheiden, an welche Instanz eines Filters sie die aktuellen Daten weiterreichen
- Pipes können kapseln, welche Filter als nächstes in Verarbeitungskette folgen

Fazit:

- Pipes dienen als flexible Puffer zwischen Filtern
- Pipes sollen Kopplung im System verringern

### Vorteile

- Leicht verständliches Muster → wir sind mit solchen Abfolgen von Arbeitsschritten vertraut
- existierender Filter leicht durch neue Komponente austauschbar
- einfache, definierte Schnittstellen zwischen Subsystemen
- Filter evtl. zu komplexeren Verarbeitungseinheiten kombinierbar

### Nachteile

- Fehlerbehandlung: Filter kennen sich nicht gegenseitig, Folgefehler schwer behebbar
- Konfiguration: über zentrale Steuerung oder Intelligenz in die Filter verlagern
- Zustand: Filter kennen keinen globalen Zustand; Verarbeitungsinformationen müssen also in Daten übertragen oder zentraler Steuerung mitgegeben werden

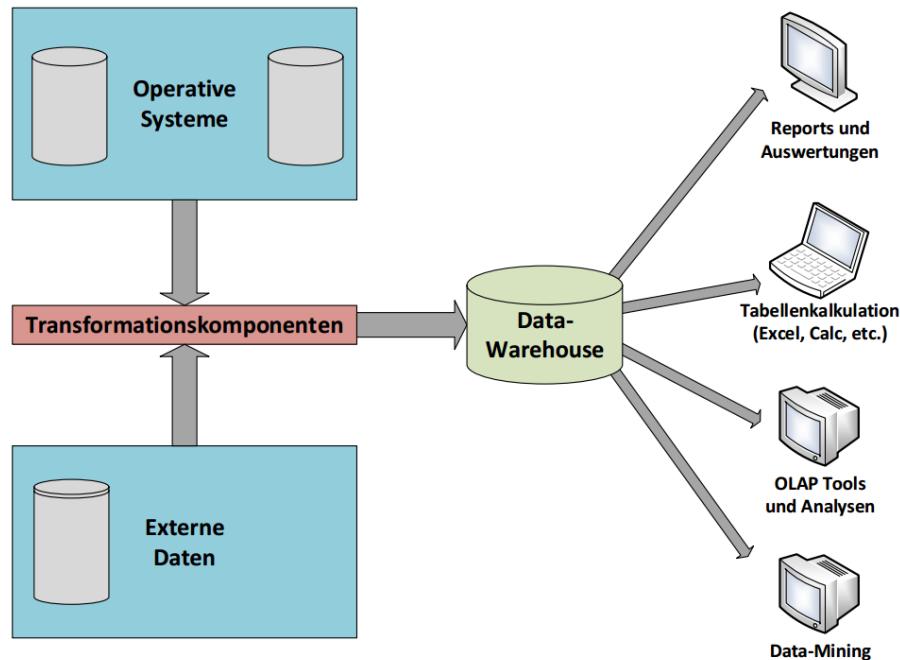
### 5.3.3 Blackboard

Idee:

- System ist Ansammlung unabhängiger Subsysteme
- Jedes Subsystem ist auf Teilaufgabe spezialisiert (Wissensquelle)
- Alle Subsysteme greifen auf ein gemeinsames Blackboard (Speicher/Datenbank/Repository) zu
- Datenaustausch untereinander über das Blackboard
- Zentrale Komponente bewertet Zustand und koordiniert Subsysteme

Beispiele:

- KI-Anwendungen (z.B. Bild- und Spracherkennung)
- Integrierte Entwicklungsumgebungen
- Management-Informationssysteme, Data Warehousing



### Vorteile

- effiziente Methode für gemeinsame Datennutzung (ohne direkte Kommunikation)
- Datengenerierende Subsysteme brauchen keinerlei Rücksicht darauf nehmen, wie Daten von anderen Subsystemen verarbeitet werden (→ schwache Kopplung)
- Zentralisierung von Aufgaben (Datensicherung, Schutz von Daten, Zugriffskontrolle) (→ hohe Kohäsion)

### Nachteile

- Subsysteme müssen dasselbe Datenformat nutzen
- Integrieren neuer Subsysteme kann so schwierig werden
- Blackboard-Komponente als Flaschenhals

### 5.3.4 Peer-to-Peer

Idee:

- gleichberechtigte Komponenten (Peers), die über Netzwerk verbunden sind
- Komponenten nehmen gleichzeitig Rolle von Server und Client wahr
- teilen Ressourcen (CPU, Speicher, Dateien, etc.)

Zweck: Ausfallsichere Datenverteilung:

- Filesharing
- Digitale Telefonie, Instant Messaging
- Verteiltes Rechnen: komplexe Rechenaufgaben (Spektralanalyse, Primfaktorzerlegung) auf Peers verteilt (Beispiel: Seti@Home)

Vorteile:

- Hohe Ausfallsicherheit und Parallelität
- Kein Flaschenhals

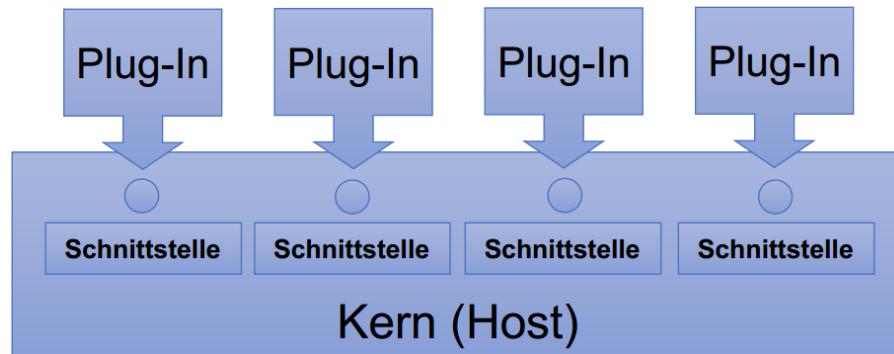
Nachteile:

- Auffinden von Peers in großen Netzen schwierig
- Fehlerbehandlung schwierig

### 5.3.5 Plug In

Idee:

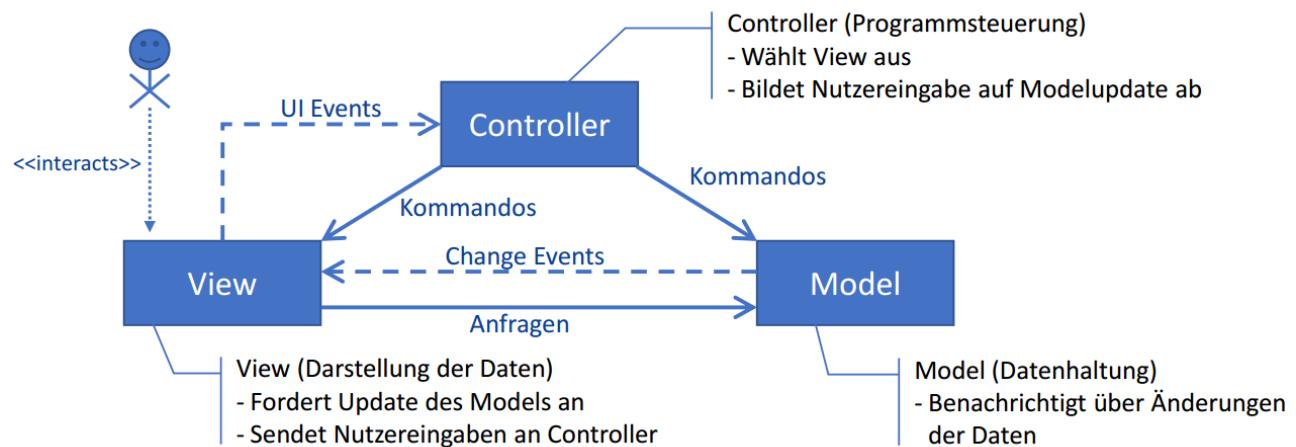
- System dynamisch um neue Funktionen erweitern, ohne Kernsystem zu modifizieren
- System bietet Plug-In-Mechanismus an, über den externe Module hinzugefügt werden können
- System bietet Schnittstellen, die durch die Plug-Ins implementiert werden



### 5.3.6 Model View Controller (MVC)

Idee:

- Aufteilung eines Programms in Komponenten:
  - Model (darzustellende Daten, evtl. auch Geschäftslogik)
  - View (Präsentation der Daten, Entgegennahme von Benutzerinteraktionen)
  - Controller (Steuerung)



### 5.3.7 Thin vs. Fat Client

- Thin Client:
  - Nur die Benutzerschnittstelle ist auf dem Client vorhanden
- Fat Client:
  - Teile oder die gesamte Fachlogik auf dem Client vorhanden
  - Hauptfunktion des Servers: Datenhaltung
  - Dadurch Entlastung des Servers
  - Aber zusätzliche Anforderungen an die Clients

# 6 Entwurf & Implementierung

---

- Detailaspekte der Implementierung
    - Kostensparen
    - Versionskontrolle
    - Build-Systeme
  - Design Pattern
    - Nutzen beschreiben
    - Pro/Contra
    - Kleines Beispiel
    - Als PseudoCode oder auch als Klassendiagramm
- 

## 6.1 Detailaspekte der Implementierungsphase

### 6.1.1 Wiederverwendung

Zwischen 1960 und 1990:

- viel Neuentwicklung, kaum Wiederverwendung
- höchstens in Programmiersprachenbibliotheken
- Dieser Ansatz wurde immer weniger tragfähig:
  - wegen zunehmenden Kosten, Termindruck

Aktuell:

- immer mehr Wiederverwendung
- auf ganz unterschiedlichen Ebenen

Abstraktionsebene:

- keine direkte Wiederverwendung von Software
- sondern Nutzung erfolgreicher Entwurfs- und Architekturmuster

Objektebene:

- Verwendung von Objekten aus Bibliotheken
- dazu: Auffinden von Bibliotheken, die gewünschte Funktionalität bieten

Komponentenebene:

- Verwendung von Frameworks

Systemebene:

- Wiederverwendung von gesamten Anwendungssystemen
- üblich: Neukonfiguration des Systems

Vorteile: Systeme können:

- schneller
- mit weniger Entwicklungsrisiko
- kostengünstiger
- zuverlässiger (da bereits in anderen Anwendungen getestet) entwickelt werden

Mögliche Kosten bei wiederverwendbaren Systemen:

- Kosten durch Suchen und Evaluierung wiederverwendbarer Software
- Kaufkosten wiederverwendbarer Software
- Kosten für Customizing und Konfiguration
- Kosten für Integration von Komponenten verschiedener Hersteller

### 6.1.2 Konfigurationsverwaltung/Konfigurationsmanagement

*Konfigurationsverwaltung ist der Prozess, ein sich veränderndes Softwaresystem zu verwalten*

Ziel:

- Unterstützung des Systemintegrationsprozesses, so dass:
- alle Entwickler greifen auf kontrollierte Art auf Code und Dokumentation zu
- alle können herausfinden, welche Änderungen wann gemacht wurden
- ...

#### Grundlegende Aktivitäten:

Versionsmanagement:

- Verwaltung und Kontrolle verschiedener Versionen der erstellten Artefakte
- Koordinierung der Entwicklung von mehreren Programmierern
- Verhindert Codeüberschreibungen von unterschiedlichen Programmierern

Systemintegration:

- Unterstützung von Entwicklern bei Festlegung, welche Softwareversion welche Komponenten verwendet
- soll automatisierte Erzeugung eines Systems ermöglichen (Buildsysteme)

Problemverfolgung:

- Tracking von Programmierfehlern und anderen Problemen
- Verfolgung, wer gerade welche Probleme löst, wann sie korrigiert sein werden, etc.

### 6.1.3 Host-Ziel-Entwicklung

Üblicherweise Entwicklung von Systemen auf einem Computer (Host) und Betrieb auf einem anderen Computer (Ziel/Target)

Allgemein:

- Development Platform
- Production/Execution Platform
- dabei ist Plattform mehr als nur Hardware (z.B. OS, DBMS, IDE, ...)

Werkzeuge von Entwicklungsplattformen:

- Syntaxorientierter Editor (Code-Erstellung, -Editierung)
- Integrierter Compiler (Code-Kompilierung)
- Debugger
- Graphische Bearbeitungstools (z.B. UML-Tools)
- Testwerkzeuge (z.B. CppUnit/JUnit zur Unterstützung automatisierter Entwicklertests)
- Weitere Werkzeuge, z.B.:
  - Analyseprogramme
  - Profiler,
  - Werkzeuge für Softwaremetriken und zur Projektunterstützung
  - Dokumentationswerkzeuge (Javadoc, Doxygen, ...)

## 6.2 Design Pattern

*Design Patterns sind wiederverwendbare Lösungen zu wiederkehrenden Design-Problemen*

Was sind Design Patterns nicht?

- sie sind kein konkretes Design
- sie sind keine konkrete Implementierung (wie etwa ein Algorithmus)

Design Patterns:

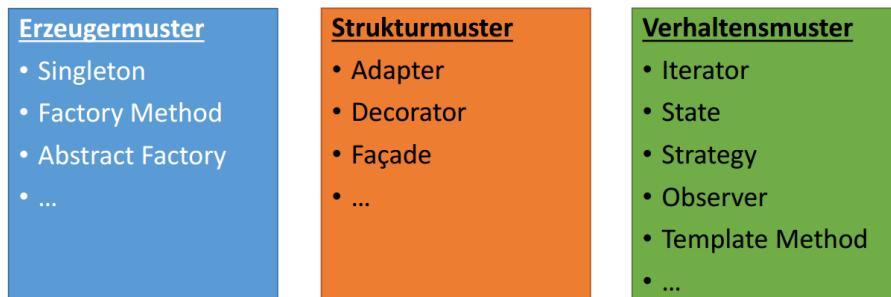
- unterstützen Wiederverwendung,
- vereinfachen das Design,
- erleichtern Dokumentation und Wissenstransfer,
- definieren eine gemeinsame Sprache,
- erleichtern ingenieurmäßiges Vorgehen bei der Softwareentwicklung,
- fördern Fähigkeit, selbstständig gutes Design zu machen,
- erleichtern den Zugang zu Klassenbibliotheken und Frameworks, erleichtern den Zugang zu objektorientierten Sprachen

Wichtige Prinzipien von Design Patterns:

- Programmiere gegen ein Interface, nicht gegen die Implementierung
- Ziehe Objekt-Komposition der Vererbung vor
- Koppele Objekte lose
- Kapsele variable Konzepte

⇒ Design Patterns liefern eine Transformation von einem spezifischen Design-Problem in eine generische Lösung

Wichtige Entwurfsmuster:



Helfen dabei:

- das Entwickler-Vokabular zu standardisieren
- den Code einfacher wartbar machen
- Kosten zu reduzieren
- Implementierungen austauschbar machen
- das Verhalten änderbar gestalten (späte/dynamische Konfigurierbarkeit)

### 6.2.1 Factory Pattern

Factory Method :

Intention:

- Objekte verschiedenen Typs auf Bestellung erzeugen

Umsetzung:

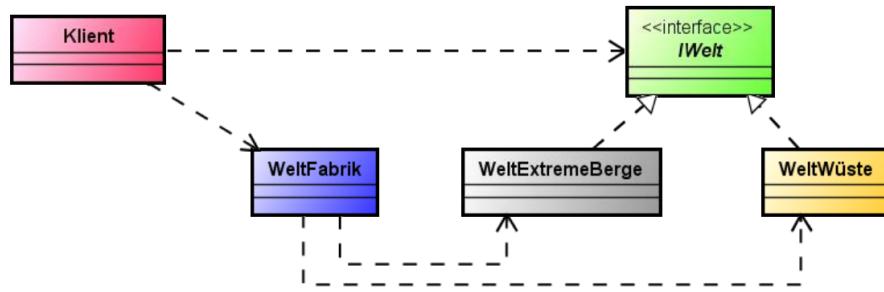
- eigene Erstellerklasse als Fabrik mit Fabrikmethode
- Erzeugung verschiedener Objekte über Fabrikmethode

Klient:

- Kein Aufruf von Konstruktor (new)
- sondern von Fabrikmethode (z.B. mit Welt-Typ)

Ziele:

- Entkopplung der Herstellung von der Verwendung eines Objekts
- Leichte Austauschbarkeit der Implementierung oder des Produktionsprozesses



### Vorteile

- Lose Kopplung zwischen Klient und Produkten
- Kapselung der Objekterstellung in Fabrik
- Trennung von Produkterstellung und Produktnutzung
- Reduzierte Komplexität aus Klientensicht
- Lesbarkeit und Wiederverwendbarkeit des Programmcodes

### Nachteile

- Mehr Komplexität im Design durch Subklassenbildung (eine konkrete Fabrik-Subklasse für jedes Produkt)
- Mehraufwand durch zusätzlichen Programmcode

### Abstract Factory :

Angenommen wir möchten eine WüstenWelt erstellen:

- Was passiert, wenn wir nun weitere Welten erstellen wollen?

Problem:

- Fehlende Abstraktion,
- Enge Kopplung von Klient und Welten
- Inkonsistenzen (Klient kann Elemente der Welten beliebig vermischen)

Ziel:

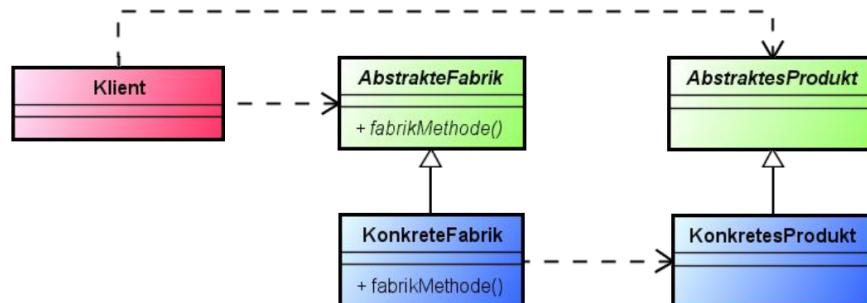
- Entkopplung von Klient und Welten

Klient entscheidet: *wann wird ein Objekt erzeugt*

→ Aufruf einer abstrakten Fabrikmethode

Subklasse der Fabrik entscheidet über: *wie wird Objekt erzeugt*

→ Implementierung der abstrakten Fabrikmethod



## Vorteile

- Kapselung der Objekterstellung in Fabrik (→ Entkopplung)
- Klient kennt konkr. Implementierungen nicht (→ Entkopplung)
- Wechsel zwischen Produktfamilien (→ z.B. Look and Feel)
- Leichte Erweiterung um neue Produktfamilien (→ z.B. Dschungel)
- Konsistente Benutzung von Produktfamilien (→ z.B. kein Kaktus im extremen Gebirge, keine Motif-Scrollbar im Windows 8 Fenster)

## Nachteile

- Hohe Komplexität (s. Klassendiagramm)
- Neue Produkte erfordern Änderungen in gesamter Fabrik-Hierarchie

## Lessons learned :

Auswirkungen auf die Software-Architektur in der Praxis

- Fabrikmethode:
  - ermöglicht lose Kopplung zwischen Nutzer und Produkten
  - Viele Anwendungsfälle, wird häufig genutzt (Komplexität aus Klient auslagern)
  - Erleichtert Testbarkeit
- Abstrakte Fabrik:
  - ermöglicht Realisierung von Produktfamilien und leichte Ersetzbarkeit
  - Anwendungsfall selten, daher wenig Nutzung
  - Nutzung z.B. in Klassenbibliotheken (JDK Look and Feels)
- Beide Entwurfsmuster:
  - erhöhen Komplexität der Klassenhierarchie (insb. bei abstrakter Fabrik)
  - Sie sind immer mit Bedacht einzusetzen (keine „Entwurfsmusteritis“)
  - Verständnis ist wichtiger als starrer Mustereinsatz

### 6.2.2 Singleton

*Sichere ab, dass eine Klasse genau ein Exemplar besitzt und stelle einen globalen Zugriff darauf bereit*

Ziel:

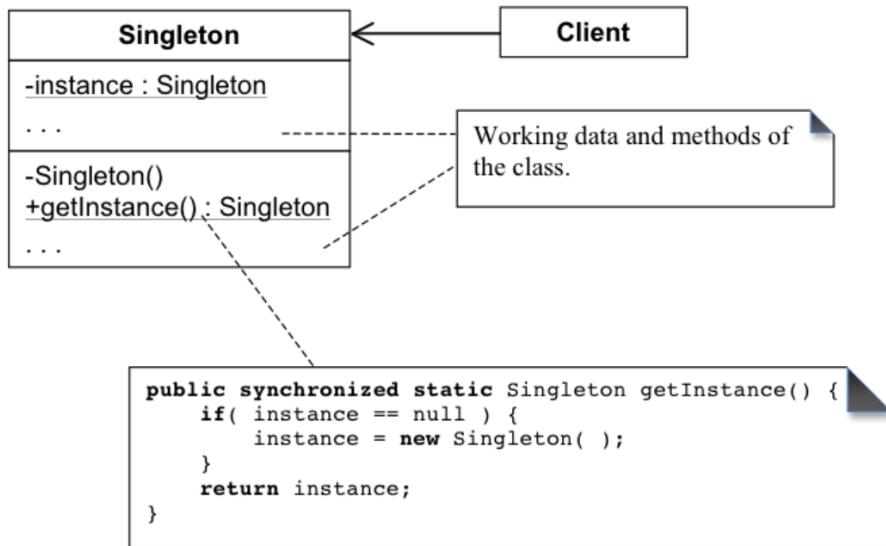
- stellt sicher, dass nicht mehr als eine Instanz einer Klasse erzeugt werden kann
- stellt einen globalen Zugriffspunkt zu dieser Instanz bereit

Lösungsidee:

- Konstruktor privat machen, um zu verhindern, dass Clients Instanzen erzeugen können
- öffentliche Methode getInstance() erstellen, die einzige Instanz der Klasse zurückgibt
- beim ersten Aufruf wird die Instanz erzeugt, gecached und zurückgegeben
- bei nachfolgenden Aufrufen wird nur die gecachte Instanz zurückgegeben

Performance:

- synchronized: macht getInstance() thread-safe
- es kann aber die Performance beeinträchtigen



falls Thread-safety kein Thema ist:

- auf Schlüsselwort synchronized verzichten

Problem:

- implizite Kopplung zwischen den Modulen durch geteilten Zugriff auf Singleton

### 6.2.3 Iterator

Bietet eine Möglichkeit, um auf die Elemente eines zusammengesetzten Objektes sequentiell zugreifen zu können

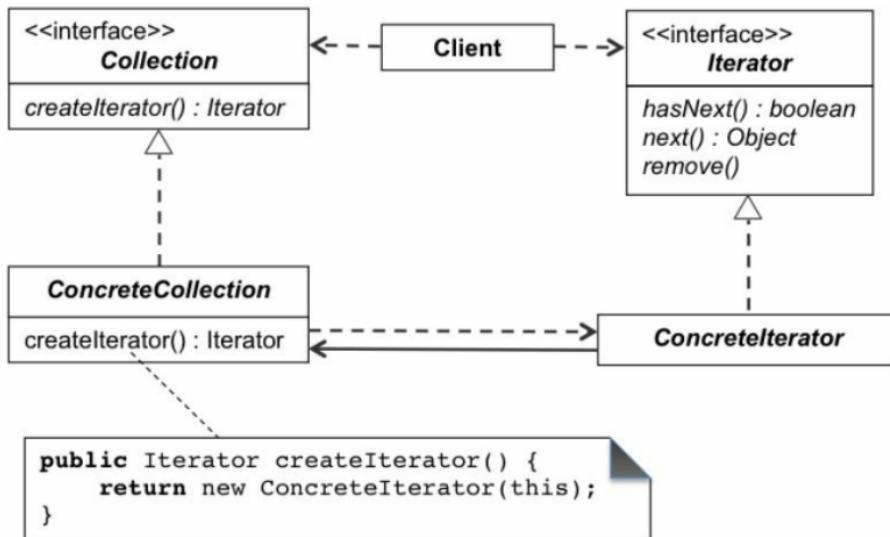
Ziel:

- bietet einen einheitlichen Weg zur Traversierung aller Elemente eines Collection Objekts

Beispiele für Collection Objekte:

- Listen, Sets, Dictionaries, ...
- konkret: Produktportfolio, das alle Produkte eines Händlers speichert:

Lösung:



#### 6.2.4 Decorator

Bietet die Möglichkeit, flexibel das Verhalten bestehender Klasse zu erweitern

Wie kann ich das Verhalten einer Klasse erweitern?

- Code der Klasse ändern
  - Open-Closed-Prinzip verletzt (Klassen sollen für Erweiterungen offen aber Modifikationen geschlossen sein)
- Erweiterung durch Vererbung
  - potentielle Explosion der Anzahl an Klassen bei Kombination von Features

Ein Online-Store möchte Ebook-Reader in drei Versionen anbieten:

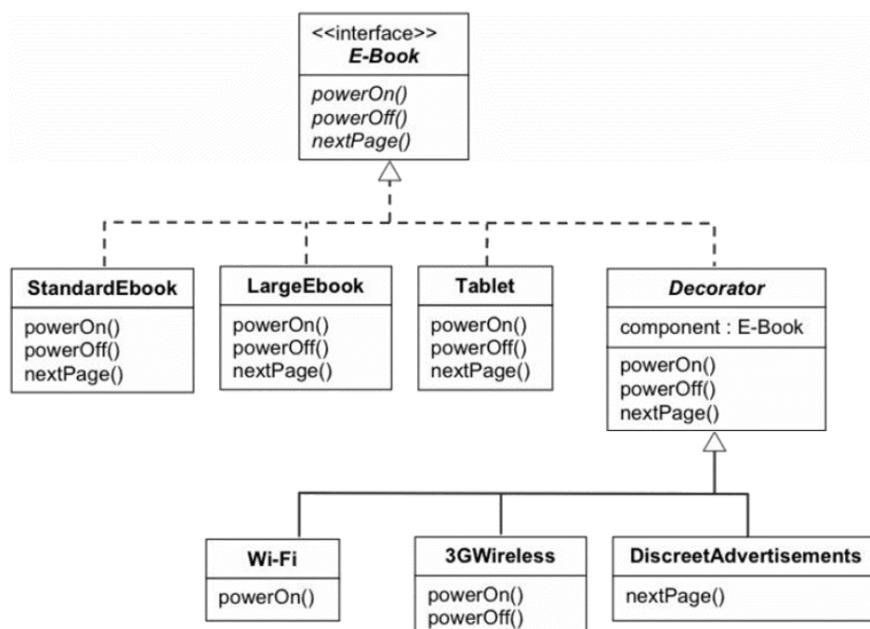
- Standard
- Groß
- Tablet

außerdem soll es zu jedem Modell Kombinationen der folgenden Features geben:

- mit WIFI
- mit 3G Wireless
- mit Werbeeinblendung (30 Euro Subventionierung)

**Lösung:**

- es gibt 3 Basismodelle des Ebook-Readers (standard, groß, Tablet) und
- verschiedene "Folien" für die einzelnen Features
- Um einer Komponente ein Feature hinzuzufügen:
  - erzeugt man Instanz der Komponenten und
  - übergibt diese Instanz dem Konstruktor des konkreten Decorators



### 6.2.5 Fassade

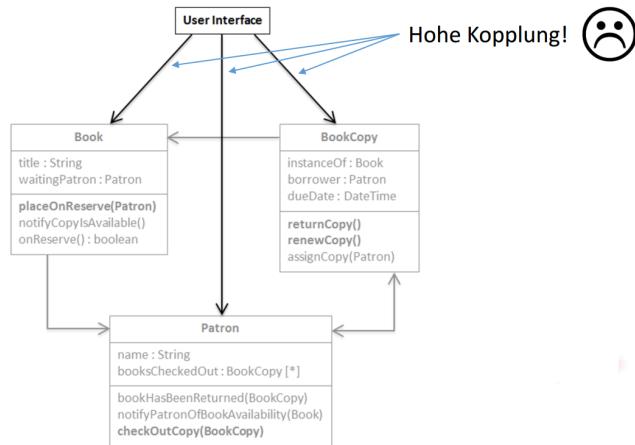
Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems

Vorteile:

- geringere Kopplung, da User-Interface-Komponente i.a. aus vielen Klassen besteht

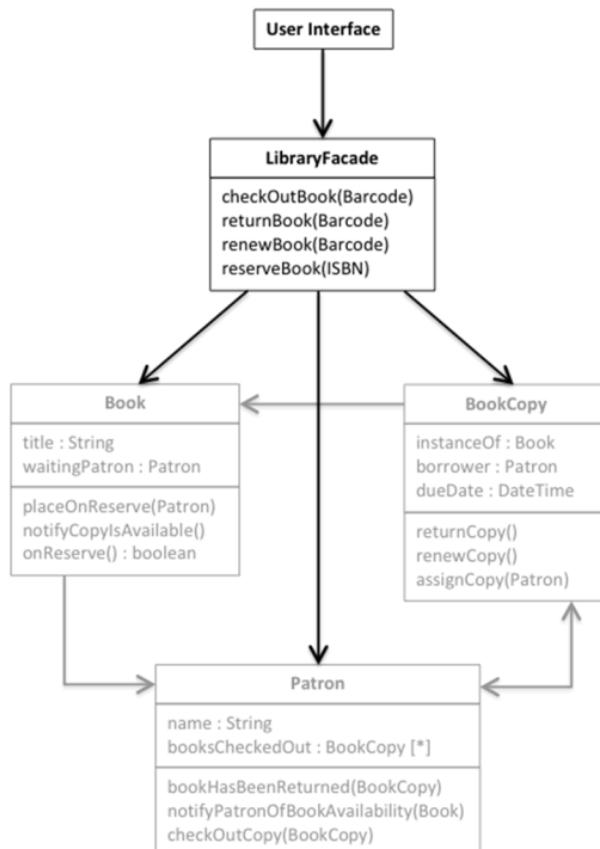
vorher:

- alle Klassen des UI hatten Referenzen zu den drei Klassen
- die Interfaces waren medium komplex



nachher:

- alle UI-Klassen haben nur noch eine Referenz
- sie greifen über ein einfaches Interface auf die drei Klassen zu



# 7 Testing

---

- Arten von Tests
  - Testphasen
  - Das "Tool"
    - Und wie toll es ist
- 

## 7.1 Allgemein

### 7.1.1 Ziel der Testphase

- Ausführen des Programms mit künstlichen Daten
- Validierungstests:
  - "nachweisen, dass Programm das tut, was es soll"
  - Entwicklern und Kunden zeigen, dass die Anforderungen erfüllt sind
- Fehlertests:
  - "Fehler finden, bevor das System produktiv benutzt wird"
  - Situationen aufspüren, in denen Software falsch/unerwünscht/nicht spezifikationsgemäß verhält

Aber:

- Abwesenheit von Fehlern kann durch Testen nicht nachgewiesen werden
- Testen ist Teil eines umfassenderen Verifikations- und Validierungsprozesses

### 7.1.2 Verifikation vs Validierung

Verifikation:

- Klärt die Frage: „Bauen wir das Produkt richtig?“
- Formales Prüfen der Implementierung gegen die Spezifikation

Validierung:

- Klärt die Frage: „Bauen wir das richtige Produkt?“
- Ausführen des Codes und prüfen gegen die wirklichen Bedürfnisse des Nutzers

**Verifikation:** Korrektheit formal beweisen

- Sehr aufwendig, daher sehr teuer
- In kritischen Systemen (Luftfahrt, AKWs, ...) vorgeschrieben
- Werkzeugunterstützung durch so genannte „Model Checker“, „Hoare Kalkül“

**Testen:** stichprobenartig Experimente durchführen

- Modultest (Unit Test): „Korrektheit“ der Implementierung eines Bausteins gegenüber Spezifikation (Schnittstelle)
- Integrationstest: „Korrektheit“ der Implementierung eines Pakets/des Gesamtsystems gegenüber Schnittstelle/Anforderungsdefinition
- Installations-/Abnahmetest: „Korrektheit“ des Gesamtsystems auf Zielmaschine und hinsichtlich der Vorstellungen des Auftraggebers

**Prüfen:** menschliche (nicht automatisierte) Begutachtung

- Inspektion (meistens für ausführbare Dokumente/Quelltext):
  - Ein Test-Team sucht Defekte im Dokument mittels Checklisten, bewertet sie;
  - Autor darf Fragen klären; fester Ablauf Suche, Treffen, Korrektur, Prüfung
- Walkthrough (nur für ausführbare Dokumente/Quelltext):
  - manueller Programmdurchlauf, Test-Team spielt Computer
- Review (für alle Dokumente):
  - Beurteilung eines Dokuments durch eine Person,
  - auch als Teil von Inspektion oder Walkthrough

#### Vorteile

- Beim Testen können bestehende Fehler andere Fehler überdecken
- Unvollständige Software-Versionen können inspiziert aber evtl. nicht getestet werden
- Über die Fehlersuche hinaus können weitere Qualitätsmerkmale geprüft werden

#### Nachteile:

- häufig höhere Kosten
- Inspektionen können prüfen, ob Spezifikationen eingehalten werden, aber nicht, ob die wirklichen Benutzeranforderungen erfüllt sind
- Inspektionen können nicht/nur schwer nicht-funktionale Charakteristika überprüfen (Performance, Usability)

⇒ **Inspektionen und Tests ergänzen sich**

## 7.2 Testphasen

### 7.2.1 Entwicklertests

Testen des Systems während der Entwicklung durch das Entwicklerteam

**Modultests:** Testen von einzelnen Einheiten (z.B. Funktionalität von Objekten und Methoden)

dabei sollten alle Eigenschaften des Objekts abgedeckt werden:

- alle zum Objekt definierten Operationen
- alle Attributwerte des Objekts
- alle möglichen Zustände des Objekts (Ereignisse simulieren, die zu Zustandswechseln führen)

Vererbung erschwert das Testen, da z.B. zu testende Funktionalität auch in erbenden Klassen getestet werden muss

Zwei Arten von Test Cases:

- Normale Programmausführung (die Komponente tut, was sie soll)
- Ungewöhnliche/falsche Eingaben (die Komponente kann damit umgehen, ohne abzustürzen)

Ziel:

- wenn möglich, Modultests automatisieren

Durchführung:

- Verwendung von Testframeworks (z.B. Junit, CppUnit), um Tests zu schreiben, durchzuführen und zu protokollieren

Vorteil:

- Ermöglicht, bei jeder Änderung alle Tests laufen zu lassen und das Ergebnis graphisch darzustellen

Auswahl der Testfälle: (2 Strategien)

- Klassenbasierte Tests:
  - Identifikation von Gruppen von Eingabedaten mit ähnlichem Verhalten
  - mindestens ein Test pro Gruppe
- Richtlinienbasierte Tests:
  - Verwendung von Richtlinien zur Testfalldefinition
  - Richtlinien spiegeln vorausgegangene Erfahrung mit Art von Fehlern wider, die Programmierern häufig unterlaufen

**Komponententests:** Testen von zusammengesetzten, interagierenden Objekten

Zugriff über definierte Schnittstellen

⇒ **Hauptfokus: Schnittstellen**

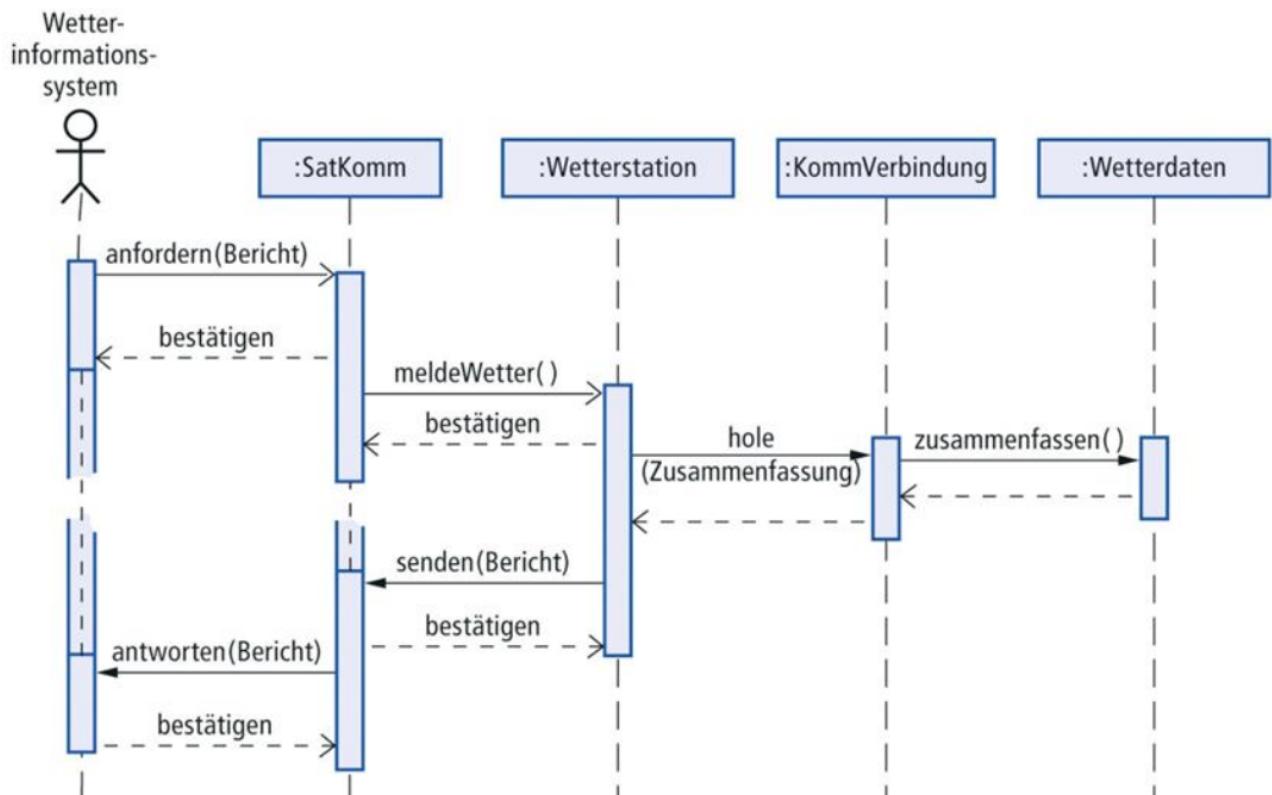
**Systemtests:** Testen des Gesamtsystems

- Getestete Komponenten (s. vorherige Detailstufe) werden integriert
- Evtl. zugekaufte Komponenten werden integriert
- Es wird geprüft, ob Zusammenspiel der Komponenten korrekt ist

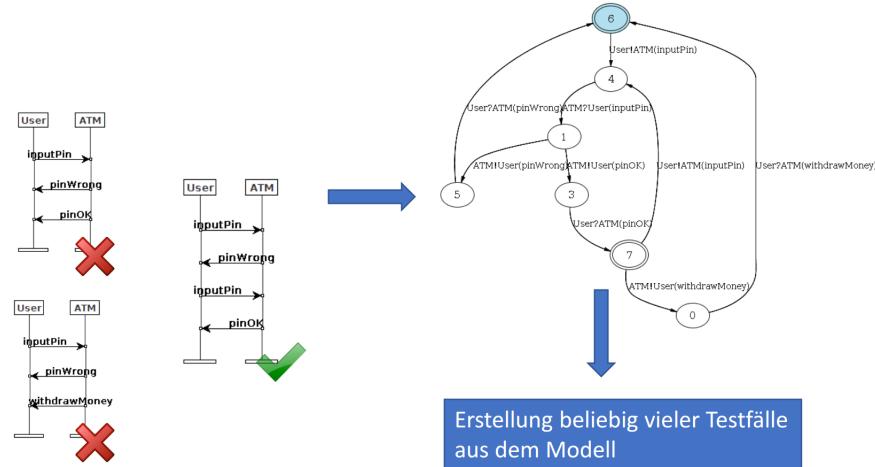
⇒ **Hauptfokus: Interaktion der Komponenten**

Sequenzdiagramme

- helfen, Systemtests zu definieren
- stellen nämlich Interaktion zwischen Komponenten dar



## “Smyle“



## Testgetriebene Entwicklung

Ansatz zur Programmentwicklung, bei dem Codeerstellung und Testen ineinander greifen

Vorgehen:

- Identifizierung geforderter Funktionalität (kleinschrittig)
- Automatisierten Test für Funktionalität schreiben
- Implementieren der Funktionalität
- Testen der Funktionalität

TDD ist oft Teil von agilen Methoden (z.B. Extreme Programming, s. nächstes Kapitel)

### 7.2.2 Freigabetests

Separates Testteam testet vollständige Version des Systems, bevor dieses für Benutzer freigegeben wird

Ziel:

- Nachweis, dass System gebrauchsfähig/ gut genug für externen Gebrauch ist
- Nachweis der spezifizierten Funktionalität, Performance, Zuverlässigkeit während des normalen Gebrauchs

Übliches Mittel:

- Blackbox Testing

Unterscheidung von Freigabetests in:

- Anforderungsbasiertes Testen
- Szenariobasiertes Testen
- Leistungstests

### 7.2.3 Benutzertests

Benutzer eines Systems testen das System in ihrer eigenen Umgebung

Beispiele:

- Marketinggruppe, die entscheidet, ob Produkt verkauft werden kann
- Alphatests: Benutzer arbeiten mit Entwicklern zusammen
- Betatests: Benutzer erhalten Release der Software und experimentieren damit; Probleme werden mit Entwicklern besprochen
- Abnahmetests mit Kunden; Kunden testen und entscheiden, ob System abgenommen und installiert wird

## 7.3 Grundregeln für Testen (Quiz)

- Tests können Anwesenheit aber **nicht die Abwesenheit** von Fehlern zeigen!
- Komponente soll:
  - nicht nur das tun, was sie soll, sondern
  - **auch mit Fehleingaben umgehen können**
- Immer mit **richtigen und falschen** Eingaben testen
- Testdaten XXX festlegen oder Tests XXX implementieren
- Test darf **nicht** vom Entwickler durchgeführt werden // (gemeint ist der finale Test)
- Auf durch Korrekturen XXX prüfen
- Auf **sinngemäß: wichtigste** Komponenten konzentrieren
- Tests XXX und Ergebnisse XXX und XXX

## 8 Vorgehensmodelle

---

- Welche gibt es
  - Vor/Nachteile der Modelle
  - Beispiele
  - Kategorien
- 

### 8.1 Generelle Prinzipien von Vorgehensmodellen

- Planung und Koordination
  - Risikominimierung: alle Beteiligten können im Voraus erkennen, was wann getan werden muss
- Korrektur und kontinuierliche Verbesserung
  - Prozess so gestalten, dass unvermeidlich auftretende Fehler gut ausgeglichen werden können
- Iteration
  - Risikominimierung: Projekt bringt in definierten Abständen einsetzbare Versionen des Softwareproduktes hervor

⇒ Softwareentwicklungsprozess wird transparenter und somit:

- planbar
- nachvollziehbar
- kontrollierbar
- lehrbar

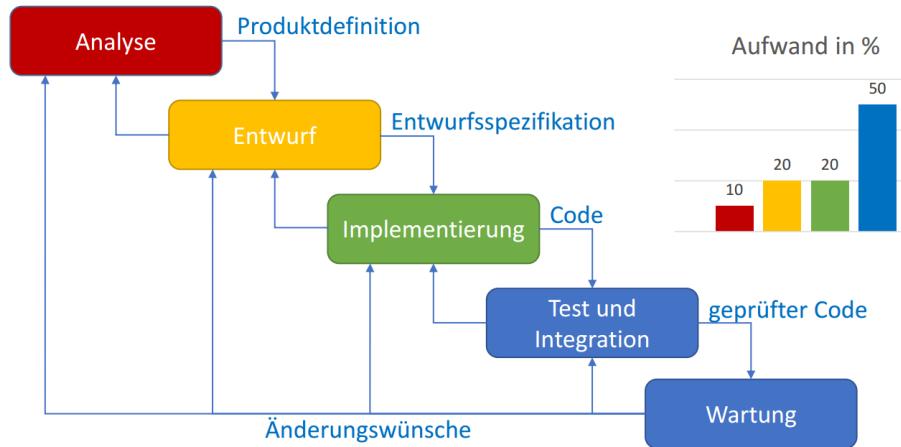
Auswirkung auf das Software-Produkt:;

- höhere Qualität
- effizientere Produktion
- bessere Wartbarkeit
- und dadurch:
  - schnellere Fehlerbehebung
  - erhöhte Änderungsfreundlichkeit

### 8.2 Wasserfall-Modell

Eigenschaften:

- Alle Phasen werden sequentiell durchgeführt
- Nächste Phase beginnt erst, wenn vorhergehende abgeschlossen ist (d.h.: Ergebnis der vorhergehenden Phase liegt vor)
- Rücksprung in vorherige Phase ist möglich (Erweiterung nach Boehm)



### Vorteile

- einfach zu verstehen
- einfach zu managen
- einfach zu überwachen (definierte Phasenübergänge)

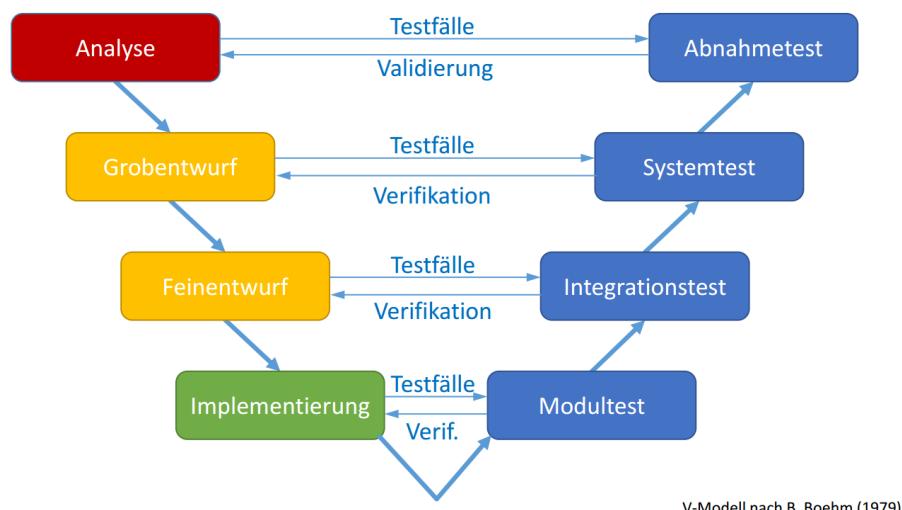
### Nachteile

- Anforderungen müssen zu 100% feststehen
- Auftraggeber nur in der ersten Phase eingebunden
- Entwicklungsrisiken werden spät erkannt
- Änderungen aufwendiger
- Verzögerungen durch sequentielles Arbeiten
- Nichteinhalten der Projektplanung führt zu Vernachlässigen später Phasen (Testen)
- Testen nur am Ende des Entwicklungszylkus

## 8.3 V-Modell

Eigenschaften:

- Erweiterung des Wasserfall-Modells
- Sehr umfangreiches Modell
- Rücksprung in vorherige Phase ist möglich (Erweiterung nach Boehm)



V-Modell nach B. Boehm (1979)

### Vorteile

- Zu jeder Entwicklungsphase (links) existiert eine Qualitätssicherungsphase (rechts)
- Qualitätssicherung also in Prozess integriert (Validierung und Verifikation)
- Kann für konkrete Projekte angepasst werden (tailoring)
- Organisationsneutral: setzt keine speziellen Strukturen beim Anwender voraus

### Nachteile

- Muss für konkrete Projekte angepasst werden (tailoring) → Mitarbeiterschulung
- viel Bürokratie (Dokumentation)

Verbindliches Vorgehensmodell für Bundeswehr und Behörden  
(V-Modell XT = Extreme Tailoring)

## 8.4 Nachteile von Wasserfall- und V-Modell

- Alle Requirements müssen von Beginn an feststehen
- Alle Testaktivitäten geschehen am Ende des Entwicklungsprozesses
- Aus der Praxis:
  - genannte Modelle legen viel Wert auf Dokumentation und Modelle
  - üblicherweise werden diese aber in der Praxis während der Wartungsphase aus Zeitmangel nur unzureichend angepasst
  - Probleme entstehen, wenn auf dieser Basis dann neue Software-Version erstellt werden soll
- Fehlende Rückkopplung zwischen Auftraggeber, Entwickler und Anwender

## 8.5 Prototyp

*Vorabversion (von Teilen) des zu entwickelnden Systems*

Ziel:

- Über Anforderungen klar werden
- schnell zu ersten Ergebnissen kommen
- frühes Feedback
- grundsätzliche Machbarkeit relevanter Teile prüfen
- Probleme und Änderungswünsche früh erkennen
- weniger Aufwand bei Behebung (der erkannten Probleme) als nach Fertigstellung
- Experimentieren und über das Projekt und die Requirements lernen

⇒ Risikominimierung!!

Arten von Prototyping (Auszug):

- Evolutionäres Prototyping
- Wegwerf-Prototyping

Vorteile von Prototyping:

- Integration in andere Vorgehensmodelle (z.B. Erweiterung Wasserfall-Modell) mögl.
- Reduzierung des Entwicklungsrisikos

Nachteile von Prototyping:

- evtl. höherer Entwicklungsaufwand
- Beschränkungen und Grenzen des Prototypen oft nicht bekannt
- Anforderungen müssen nahezu 100%-ig sein
- Prototyp (illegal) in die Entwicklung übernehmen

### 8.5.1 Evolutionäres Prototyping

Zeichnet sich aus durch:

- Prototyp wird in Iterationen immer weiter verfeinert
- In jeder Iteration: gelerntes Wissen aus den vorhergehenden Iterationen nutzen, um aktuelle Prototypversion zu verbessern
- Prototyp wird dabei stets lauffähig gehalten
- Prototyp entwickelt sich schließlich zum finalen Produkt

#### Vorteile

- Prototyp als zentrales Kommunikationsobjekt zwischen Kunde und Entwicklung
- Erhöhte Akzeptanz beim Kunden, da er in Entwicklungsprozess integriert ist
- Risikominimierung einer Fehlentwicklung
- Beschleunigte Einführung

#### Nachteile

- Schwierig Datum für Produktfertigstellung vorherzusagen
- Anwendung in großen Projekten gefährlich, da sich Schnittstellen zu anderen Teilprojekten ändern können

### 8.5.2 Wegwerf-Prototyping

*throw-away prototyping*

Zeichnet sich aus durch:

- Prototyp wird erzeugt
- Prototyp wird mit Kunden diskutiert
- nach Übereinkunft über seine Korrektheit weggeworfen
- Wissen aus Erstellung und Diskussion über Prototyp fließt nun in Erstellung des finalen Systems ein

#### Vorteile

- Lernen aus Prototyp

#### Nachteile

- Eingang des Wegwerfprototypen in Produkt (aus Zeitmangel)
- Zusätzlicher Entwicklungsaufwand (Kosten!)

Entwicklung eines Prototyps nützlich, wenn:

- Anforderungen noch nicht hinreichend klar sind
- Anforderungen anders nicht geklärt werden können
- Benutzerschnittstellen

Erfolgsfaktoren:

- Direkter Kontakt Anwender ↔ Entwickler
- Dokumentation / Auswertung

## 8.6 Iterative Entwicklung

Iterative Modelle

- Der Entwicklungsprozess besteht aus Folge von Zyklen (Iterationen)
- Am Ende jeder Iteration: neue (ausführbare) Version des Produktes
- Neue Version verbessert bzw. erweitert Funktionsumfang der letzten

Merkmale:

- Erweiterung der Prototypidee;
- Software wird in Iterationen entwickelt
- In jeder Iteration wird das System weiter verfeinert
- In ersten Iterationen Schwerpunkt auf Analyse und Machbarkeit; später auf Realisierung

#### Vorteile

- dynamische Reaktion auf Risiken
- Teilergebnisse mit Kunden diskutierbar

### Nachteile

- schwierige Projektplanung
- schwierige Vertragssituation
- Kunde erwartet zu schnell Endergebnis
- Kunde sieht Anforderungen als beliebig änderbar

## 8.7 Inkrementelle Entwicklung

- Anforderungen vollständig erfassen und modellieren
- Auf Kernanforderungen des Auftraggebers konzentrieren
  - Zunächst nur Kernsystem entwickeln (Null-Version)
  - Kernsystem ausliefern
  - Anwender sammeln Erfahrungen und äußern Wünsche

### 8.7.1 Inkrementell vs. Evolutionär

Inkrementell:

- Anforderungsanalyse und Konzeption nur zu Beginn der Entwicklung
- Jede Iteration erzeugt weiteres Stück der Lösung
- Schneller zu ersten Ergebnissen
- ABER: anfällig bei Anforderungsänderungen

Evolutionär:

- Anforderungsanalyse und Konzeption in jeder Iteration

Beide Verfahren:

- Wichtig ist enge Zusammenarbeit von Auftraggeber und -nehmer

## 8.8 Iterativ-Inkrementelle Entwicklung (State of the Art)

Merkmale:

- Projekt in kleine Teilschritte zerlegt
- pro Schritt neue Funktionalität (Inkrement) + Überarbeitung existierender Ergebnisse (Iteration)
- n+1-ter Schritt kann Probleme des n-ten Schritts lösen

### Vorteile

- siehe „iterativ“
- flexible Reaktion auf neue funktionale Anforderungen

### Nachteile

- siehe „iterativ“ (etwas verstärkt)

Optimierung/Anpassung:

- Anforderungsanalyse am Anfang intensiver durchführen

## 8.9 Spiralmodell (B. Boehm, 1988)

- Eines der ersten Entwicklungsmodelle, die die Probleme von Wasserfall- und V-Modell überwinden
- Iteratives (inkrementelles) Entwicklungsmodell
- Phasen werden in einer Spirale durchlaufen
- Unterstützt iterative Entwicklung von Anforderungen und Prototypen
- Am Ende jeder Iteration: Planung der folgenden Iteration

## 8.10 Kritik an klassischen Vorgehensmodellen

Nachteile klassischer Modelle:

- Es müssen viele Dokumente erzeugt und gepflegt werden
- Eigene Wissenschaft, Modelle wie V-Modelle zu verstehen und zurecht zu schneidern
- Prozessbeschreibungen hemmen Kreativität
- Anpassung an neue Randbedingungen, z. B. Technologien (Web-Services) in Prozessen und Werkzeugen ist extrem aufwendig

Alternativer Ansatz:

- "Menschen machen Projekte erfolgreich, trau den Menschen"  
→ Agile Prozesse (urspr. Name: leichtgewichtige Prozesse)

Agile Manifesto (K. Beck et al., 2001)

- Beinhaltet 12 Grundprinzipien agiler Softwareentwicklung, u.a.:
  - Individuen und Interaktion: wichtiger als Prozesse und Werkzeuge
  - Funktionierende Software: wichtiger als ausführliche Dokumentation
  - Zusammenarbeit mit Kunden: wichtiger als Vertragsverhandlungen
  - Reagieren auf Veränderung: wichtiger als Befolgen eines Plans
- in >55 Sprachen übersetzt

Agile Methoden:

- Extreme Programming (XP)
- SCRUM

Generelles Credo:

- Methoden lernen von einander
- Es gibt nicht die eine agile Methode

Variante 1: Konkrete Prozessbeschreibung

- Vorschlagen konkreter Verfahren für verschiedene Software-Entwicklungsphasen
- Dokumentieren der Abhängigkeiten der Verfahren (Wer "A" macht, muss auch "B" machen)  
→ Möglichkeit zur individuellen Optimierung
- Beispiele:
  - eXtreme Programming (XP) (u.a. Kent Beck, Ward Cunningham)
  - Dynamic Systems Development Method (Konsortium)

Variante 2: Beschreibung auf Metaprozessebene

- Grundregeln zur Projektorganisation
- Wenige Hinweise zur konkreten Umsetzung
- Vorgehensweisen in Projekten werden vom Team festgelegt
- Beispiele:
  - Scrum (u.a. Ken Schwaber, Jeff Sutherland)
  - Crystal Methodenfamilie (Alistair Cockburn)

### 8.10.1 Scrum als populäres Beispiel

A scrum (*short for scrummage*) is a method of restarting play in rugby that involves players packing closely together with their heads down and attempting to gain possession of the ball

Motivation

- 60% aller IT-Projekte sind nicht im Plan
- Falsche Annahmen:
  - Anforderungen sind klar
  - Anforderungen sind stabil
  - Entwicklungsprozess ist vorhersehbar
- Ansatz von Scrum: empirisch, inkrementell, iterativ
- Prinzipien von Scrum:
  - Zerlegung
  - Transparenz
  - Überprüfung
  - Anpassung

Idee:

- in kurzen Zyklen releasefähige Software ausliefern
- Produkt schreitet in Abschnitten von monatlichen Sprints fort
- Anforderungen sind als Listeneinträge im Product-Backlog festgehalten

**Product Owner** ist verantwortlich für:

- die Wertmaximierung des Produkts, sowie
- die Arbeit des Entwicklungsteams

ist die einzige Person, die für das Management des Product-Backlogs verantwortlich ist

**Scrum Master** sorgt für:

- Verständnis und Durchführung von Scrum verantwortlich
- dass Team die Theorie, Praktiken und Regeln von Scrum einhält

**Entwicklungsteam** ist

- bestehend aus Profis, die am Ende eines jeden Sprints ein fertiges Inkrement übergeben, welches potentiell auslieferbar ist
- Interdisziplinär
- Selbstorganisierend
- Selbstbestimmend: Niemand (auch nicht Scrum Master) schreibt Team vor, wie es aus Product-Backlog auslieferbare Funktionalität machen soll

Nur Mitglieder erstellen das Produkt-Inkrement

**Meetings** :

Sprint Planning:

- Was ist in Produkt-Inkrement enthalten (kommender Sprint)?
- Wie wird für Lieferung des Produkt-Inkrementen erforderliche Arbeit erreicht?

Daily Scrum: jedes Teammitglied beantwortet drei Fragen:

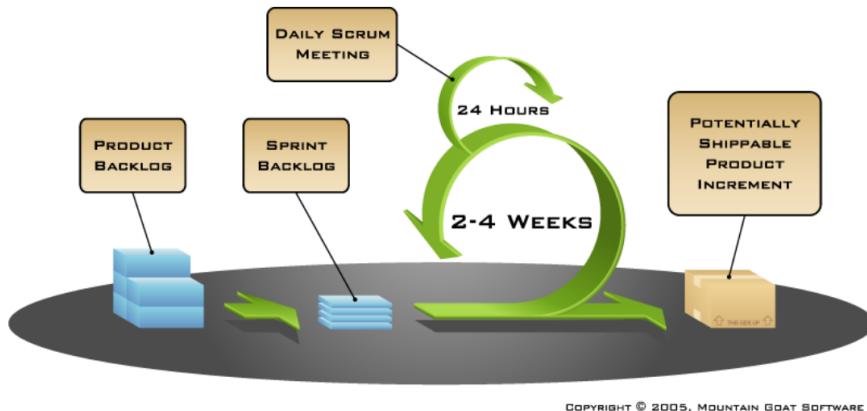
- Was habe ich gestern gemacht?
- Was werde ich heute tun?
- Was hindert mich bei meiner Arbeit?

Sprint Review: (Scrum Team und Stakeholder)

- Inkrement überprüfen und ggf. Product-Backlog anpassen

### Sprint Retrospektive:

- Team überprüft sich selbst
- erstellt Verbesserungsplan für kommenden Sprint



COPYRIGHT © 2005, MOUNTAIN GOAT SOFTWARE

Scrum-Projekte schreiten in Serien von Sprints voran

- typische Sprintdauer: 2 – 4 Wochen (bzw. nicht länger als ein Kalendermonat)
- konstante Dauer führt zu einem besseren Rhythmus
- Das Produkt wird während des Sprints
  - entworfen
  - kodiert und
  - getestet
- Keine Änderungen während des Sprints (die Sprint-Ziel gefährden könnten)
  - Sprintdauer ist vorab festgelegt
  - Sprintdauer hängt davon ab, wie lange Veränderungen vom Sprint ferngehalten werden

## 8.11 Agile Methoden

- haben viele Innovationen in verstaubte Entwicklungsprozesse gebracht
- Einsetzbarkeit stark von technischen und menschlichen Fähigkeiten abhängig
- große Erfolge möglich, wenn "Dream-Team" gefunden wurde
- Agiles Manifest interessant als Ergänzung für alle SW-Entwicklungsprozesse

**Generell: das gewählte Vorgehensmodell muss zum Projekt und den Menschen im Projekt passen**

## 8.12 Wann sind welche Modelle angemessen?

Strikte, stark planende Modelle

- Wenn wohldefiniertes Resultat in definierter Zeit erreicht werden muss
  - Wenn sehr große (insbesondere verteilte) Projektgruppen koordiniert werden müssen
- ⇒ **sind die Pläne und Dokumente zur Koordination unverzichtbar**

Agile Modelle

- Wenn Projekte komplex sind
- hohe Unsicherheit über die Anforderungen besteht
- hohe Unsicherheit über technische Realisierung

## 8.13 Hauptunterschied zwischen Vorgehensmodellen

Wie viel Planung?

Wichtigste Dimension zur Unterscheidung: wie präzise/strikt/weit voraus wird geplant?

- Viel: Wasserfallmodell
  - möglichst präzise und strikt, für das gesamte Projekt im Voraus
- Mittel: Iterative Modelle
  - präzise, wo möglich
  - nicht strikt (nötige Veränderungen werden akzeptiert)
  - nur für wenige Iterationen im Voraus
- Wenig: Agile Prozesse
  - nur so viel Planung wie unbedingt nötig
  - lieber Ziele als Pläne (→ um flexibel zu bleiben)

# Inhaltsverzeichnis

<b>1 Überblick</b>	<b>1</b>
1.1 Was ist Software Engineering . . . . .	1
1.2 Warum ist Software Engineering wichtig? . . . . .	1
1.3 Soll-Eigenschaften guter Software . . . . .	2
1.4 Ziele des Software Engineering . . . . .	2
1.5 Zusammenfassung . . . . .	3
<b>2 Ethik</b>	<b>3</b>
<b>3 Projektphasen</b>	<b>4</b>
3.1 Wie kommt man überhaupt zu einem Projekt . . . . .	4
3.2 Phasenübersicht eines Softwareprojekts . . . . .	4
3.2.1 Analyse . . . . .	5
3.2.2 Design (Entwurf) . . . . .	5
3.2.3 Implementierung (und Modultests) . . . . .	5
3.2.4 Test und Integration . . . . .	5
3.2.5 Abnahme und Einführung . . . . .	5
3.2.6 Betrieb und Wartung . . . . .	5
3.2.7 Allgemeine Hinweise . . . . .	6
<b>4 Objekt Orientierte Analyse</b>	<b>6</b>
4.1 Requirements Engeneering - Anforderungsanalyse . . . . .	7
4.1.1 Beispiel 2: wie es nicht geht . . . . .	7
4.1.2 Analysephase (RE + OOA) . . . . .	7
4.1.3 Bedeutung des Requirement Engineerings . . . . .	8
4.1.4 Visionen, Ziele und Rahmenbedingungen . . . . .	9
4.1.5 Rahmenbedingungen . . . . .	10
4.1.6 Anforderungen . . . . .	10
4.1.7 Aktivitäten des Requirement Engineerings . . . . .	11
4.1.8 Systemkontext festlegen . . . . .	11
4.1.9 Stakeholder . . . . .	11
4.1.10 Regeln für die Definition von Zielen . . . . .	12
4.1.11 Anforderungen . . . . .	12
4.1.12 Lastenheft vs. Pflichtenheft . . . . .	13
4.1.13 Klassifikation von Anforderungen . . . . .	14
4.1.14 Erfassung von Anforderungen mithilfe von UseCases . . . . .	15
4.1.15 Zusammenfassung Requirements Engineering . . . . .	15
4.2 Use Cases . . . . .	15
4.2.1 Beschreibungsvariante: Use-Case-Definition . . . . .	16
4.2.2 Use-Case Diagramm . . . . .	17
4.2.3 Anwendungsbereiche . . . . .	20
4.2.4 Use-Case-Erstellung: Checkliste (Auszug aus Heide Balzert) . . . . .	20
4.2.5 Use Case vs. Szenario . . . . .	21
4.2.6 Typische Modellierungsfehler . . . . .	21

4.3	Aktivitätsdiagramme . . . . .	22
4.4	Zustandsdiagramme . . . . .	24
4.4.1	Zustandsdiagramm erstellen . . . . .	27
4.5	Paketdiagramme . . . . .	27
4.6	Klassendiagramme . . . . .	30
4.6.1	Schema zur Definition von Klassenattributen . . . . .	30
4.6.2	Schema zur Definition von Klassenoperationen . . . . .	31
4.6.3	Beispiele . . . . .	31
4.7	Analysemuster . . . . .	32
4.7.1	Liste . . . . .	32
4.7.2	Exemplartyp . . . . .	33
4.7.3	Baugruppe . . . . .	33
4.7.4	Stückliste . . . . .	34
4.7.5	Koordinator . . . . .	34
4.7.6	Rollen . . . . .	35
4.8	Sequenzdiagramme . . . . .	36
4.8.1	Konstruktion von Sequenzdiagrammen . . . . .	37
4.8.2	Notation&Beispiele . . . . .	38
4.9	Kommunikationsdiagramme . . . . .	39
4.10	Verteilungsdiagramm . . . . .	41
4.11	Rueckblick . . . . .	41
<b>5</b>	<b>Objekt Orientiertes Design</b>	<b>41</b>
5.1	Grundlagen . . . . .	42
5.1.1	Ziele/Aktion der Designphase . . . . .	42
5.1.2	Sichtenmodell . . . . .	42
5.1.3	4+1 Sichten Modell nach Kruchten . . . . .	43
5.1.4	Sichten nach Starke . . . . .	45
5.2	Architektur . . . . .	47
5.2.1	Merkmale schlechter Software . . . . .	47
5.2.2	Eigenschaften eines guten OO-Entwurfs . . . . .	47
5.2.3	Kohäsion . . . . .	47
5.2.4	Kopplung . . . . .	48
5.2.5	Interne Wiederverwendung . . . . .	48
5.2.6	Probleme zerlegen . . . . .	48
5.2.7	Fachdomäne nach Evans . . . . .	49
5.2.8	Architekturprinzipien . . . . .	49
5.2.9	Hauptprinzipien des Architekturentwurfs . . . . .	50
5.2.10	Architekturprinzipien . . . . .	51
5.3	Architekturmuster . . . . .	54
5.3.1	Die klassische 3-Schichten-Architektur . . . . .	55
5.3.2	Pipes und Filter . . . . .	55
5.3.3	Blackboard . . . . .	56
5.3.4	Peer-to-Peer . . . . .	57
5.3.5	Plug In . . . . .	58

5.3.6 Model View Controller (MVC) . . . . .	58
5.3.7 Thin vs. Fat Client . . . . .	58
<b>6 Entwurf &amp; Implementierung</b>	<b>59</b>
6.1 Detailaspekte der Implementierungsphase . . . . .	59
6.1.1 Wiederverwendung . . . . .	59
6.1.2 Konfigurationsverwaltung/Konfigurationsmanagement . . . . .	60
6.1.3 Host-Ziel-Entwicklung . . . . .	60
6.2 Design Pattern . . . . .	60
6.2.1 Factory Pattern . . . . .	61
6.2.2 Singleton . . . . .	63
6.2.3 Iterator . . . . .	64
6.2.4 Decorator . . . . .	65
6.2.5 Fassade . . . . .	66
<b>7 Testing</b>	<b>67</b>
7.1 Allgemein . . . . .	67
7.1.1 Ziel der Testphase . . . . .	67
7.1.2 Verifikation vs Validierung . . . . .	67
7.2 Testphasen . . . . .	68
7.2.1 Entwicklertests . . . . .	68
7.2.2 Freigabetests . . . . .	70
7.2.3 Benutzertests . . . . .	71
7.3 Grundregeln für Testen (Quiz) . . . . .	71
<b>8 Vorgehensmodelle</b>	<b>72</b>
8.1 Generelle Prinzipien von Vorgehensmodellen . . . . .	72
8.2 Wasserfall-Modell . . . . .	72
8.3 V-Modell . . . . .	73
8.4 Nachteile von Wasserfall- und V-Modell . . . . .	74
8.5 Prototyp . . . . .	74
8.5.1 Evolutionäres Prototyping . . . . .	74
8.5.2 Wegwerf-Prototyping . . . . .	75
8.6 Iterative Entwicklung . . . . .	75
8.7 Inkrementelle Entwicklung . . . . .	76
8.7.1 Inkrementell vs. Evolutionär . . . . .	76
8.8 Iterativ-Inkrementelle Entwicklung (State of the Art) . . . . .	76
8.9 Spiralmodell (B. Boehm, 1988) . . . . .	77
8.10 Kritik an klassischen Vorgehensmodellen . . . . .	77
8.10.1 Scrum als populäres Beispiel . . . . .	78
8.11 Agile Methoden . . . . .	79
8.12 Wann sind welche Modelle angemessen? . . . . .	79
8.13 Hauptunterschied zwischen Vorgehensmodellen . . . . .	80