

## Einführung und Überblick

### Software Engineering

- Disziplinen: Anforderungen, Architektur, Implementierung, Test und Wartung.
- Ziel: Strukturierte Prozesse für Qualität, Risiko- und Fehlerminimierung.
- «Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen.» (H. Balzert)
- Aufgrund des hohen Aufwandes zur Erstellung und Wartung komplexer Software erfolgt die Entwicklung durch Softwareentwickler anhand eines strukturierten (Projekt-)Planes.

### Modellierung in der Softwareentwicklung

- Modelle als Abstraktionen: Anforderungen, Architekturen, Testfälle.
- Einsatz von UML: Skizzen, detaillierte Blueprints, vollständige Spezifikationen.
- Zweck:
  - Verstehen eines Gebildes
  - Kommunizieren über ein Gebilde
  - Gedankliches Hilfsmittel zum Gestalten, Bewerten oder Kritisieren
  - Spezifikation von Anforderungen
  - Durchführung von Experimenten

**Modellierungsumfang bestimmen** Folgende Fragen zur Bestimmung des notwendigen Modellierungsumfangs:

- Wie komplex ist die Problemstellung?
- Wie viele Stakeholder sind involviert?
- Wie kritisch ist das System?
- Analogie: Planung einer Hundehütte vs. Haus vs. Wolkenkratzer

**Prüfungsfrage zur Modellierung** Erklären Sie anhand eines selbst gewählten Beispiels, warum der Modellierungsaufwand je nach Projekt stark variieren kann. Nennen Sie mindestens drei Faktoren, die den Modellierungsumfang beeinflussen.

Mögliche Antwort:

- Beispiel: Entwicklung einer Smartphone-App vs. Medizinisches Gerät
- Faktoren:
  - Komplexität der Domäne
  - Regulatorische Anforderungen
  - Anzahl beteiligter Stakeholder
  - Sicherheitsanforderungen

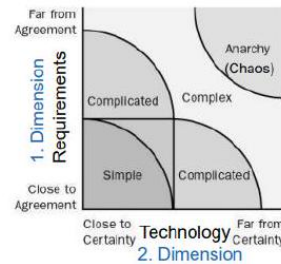
### Wrap-up

- Solide Analyse- und Entwurfskompetenzen sind essenziell.
- Iterativ-inkrementelle Modelle fördern agile Entwicklung.

## Softwareentwicklungsprozesse

### Klassifizierung Software-Entwicklungs-Probleme

Wir betrachten Wasserfall, iterativ-inkrementelle und agile Softwareentwicklungsprozesse.



Quelle: Agile Project Management with Scrum, Ken Schwaber, 2003



3. Dimension  
Skills, Intelligence Level, Experience  
Attitudes, Prejudices

### Prozesse im Softwareengineering Kernprozesse

- Anforderungserhebung
- Systemdesign/technische Konzeption
- Implementierung
- Softwaretest
- Softwareeinführung
- Wartung/Pflege

### Unterstützungsprozesse

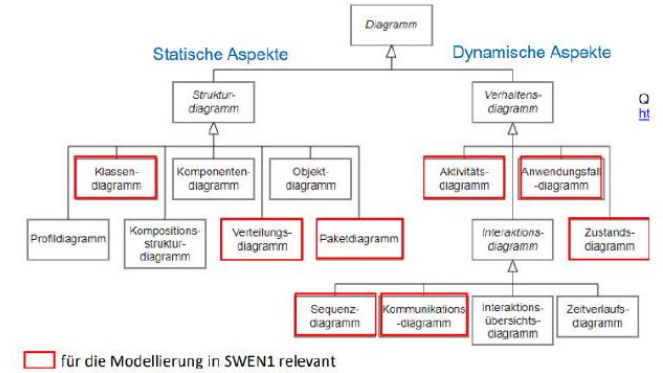
- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

### Begriffe

- Warum wird modelliert: Um Analyse- und Designentwürfe zu diskutieren, abstimmen und zu dokumentieren bzw. zu kommunizieren.
- Modell: Ein Modell ist ein konkretes oder gedankliches Abbild eines vorhandenen Gebildes oder Vorbild für ein zu schaffendes Gebilde (hier Softwareprodukt).
- Original: Das Original ist das abgebildete oder zu schaffende Gebilde.
- Modellierung: Modellierung gehört zum Fundament des Software Engineering

### Modelle in der Softwareentwicklung

- Software ist vielfach (immer?) selbst ein Modell
- Anforderungen sind Modelle der Problemstellung
- Architekturen und Entwürfe sind Modelle der Lösung
- Testfälle sind Modelle des korrekten Funktionierens des Codes usw.



### Code and Fix

Vorgehen, bei dem Codierung oder Korrektur im Wechsel mit Ad-hoc-Tests die einzigen bewussten ausgeführten Tätigkeiten der Software-Entwicklung sind: Schnell, Agil, Einfach am Anfang, Schlecht Planbar, Schlecht Wartbar, Änderungen s. Aufwändig

### Wasserfallmodell

Die Software-Entwicklung wird als Folge von Aktivitäten/Phasen betrachtet, die durch Teilergebnisse (Dokumente) gekoppelt sind. Die Reihenfolge der Aktivitäten ist fest definiert. : gut planbar, klare Aufteilung in Phasen, Schlechtes Risikomanagement, nie alle Anforderungen zu Anfang bekannt

### Iterativ-inkrementelle Modelle

Software wird in mehreren geplanten und kontrolliert durchgeführten Iterationen schrittweise (inkrementell) entwickelt: Flexibles Modell, Gutes Risikomanagement, Frühe Einsetzbarkeit, Planung upfront hat Grenzen, Kunde Involviert über ganze Entwicklung  
Agile Softwareentwicklung Basiert auf iterativ-inkrementellen Prozessmodell, Fokussiert auf gut dokumentierten und getesteten Code statt auf ausführlicher Dokumentation

### Charakteristiken iterativ-inkrementeller Prozesse

- Projekt-Abwicklung in Iterationen (Mini-Projekte)
- In jeder Iteration wird ein Stück der Software entwickelt (Inkrement)
- Ziele der Iterationen sind Risiko-getrieben
- Iterationen werden reviewed und die Learnings fließen in die nächsten Iterationen ein
- Demming-Cycle: Plan, Do, Check, Act

### Zweck und den Nutzen von Modellen in der Softwareentwicklung

Modell von Requirements (close to/ far from Agreement) & Technology (known / unknown)

Ein Modell ist ein konkretes oder gedankliches Abbild eines vorhandenen Gebildes oder Vorbild für ein zu schaffendes Gebilde (hier Softwareprodukt).

**Typische Prüfungsaufgabe: Prozessmodelle vergleichen** Vergleichen Sie das Wasserfallmodell mit einem iterativ-inkrementellen Ansatz anhand folgender Kriterien:

- Umgang mit sich ändernden Anforderungen
- Risikomanagement
- Planbarkeit
- Kundeneinbindung

Musterlösung:

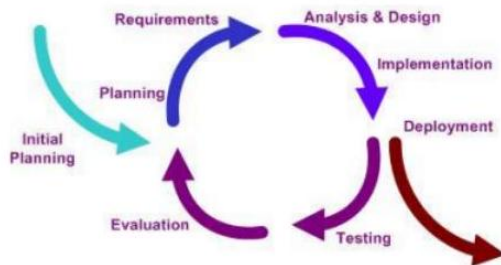
- Wasserfall:
  - Änderungen schwierig zu integrieren
  - Risiken erst spät erkennbar
  - Gut planbar durch feste Phasen
  - Kunde hauptsächlich am Anfang und Ende involviert
- Iterativ-inkrementell:
  - Flexibel bei Änderungen
  - Frühes Erkennen von Risiken
  - Planung pro Iteration
  - Kontinuierliches Kundenfeedback

### Unified Modelling Language (UML)

UML ist die Standardsprache für die graphische Modellierung von Anforderungen, Analyse und Entwürfen im Software Engineering (objektorientierte Modellierung). (As a sketch, blueprint, programming language)

### Incremental Model

Artefakte in einem iterativ-inkrementellen Prozess illustrieren und einordnen



## Die drei Säulen der Benutzererfahrung:

- 
- UX starts by being useful.
- Functionality, people must be able to use it.
- The way it looks and feels must be pleasing.
- This helps create an overall brand experience.
- Brand Experience**
- Desirability**
- Usability**
- Utility**  
It is useful to me.  
It meets my needs.
- I am able to use the product easily.
- I like the way the product looks and feels.
- My overall feeling about the brand/product (in the abstract) is good.

Source: User Experience 2008, nnGroup Conference Amsterdam

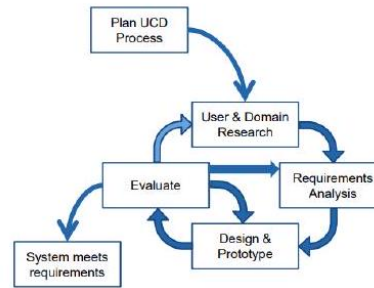
## Die drei Hauptdimensionen der Usability:

- ## ISO 9241-110: Usability-Anforderungen

Die sieben Grundprinzipien:

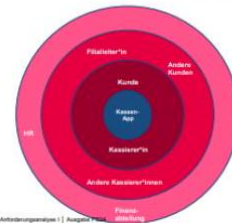
- Aufgabenangemessenheit
- Lernförderlichkeit
- Individualisierbarkeit
- Erwartungskonformität
- Selbstbeschreibungsfähigkeit
- Steuerbarkeit
- Fehlertoleranz

Ein iterativer Prozess zur nutzerzentrierten Entwicklung:



- Personas: Repräsentative Nutzerprofile
- Usage-Szenarien: Konkrete Anwendungsfälle
- Mentales Modell: Nutzerverständnis
- Domänenmodell: Fachliches Verständnis
- Service Blueprint: Geschäftsprozessmodell
- Stakeholder Map: Beteiligte und Betroffene
- UI-Artefakte: Skizzen, Wireframes, Designs

- Stakeholder Map
  - Zeigt die wichtigsten Stakeholders im Umfeld der Problemdomäne



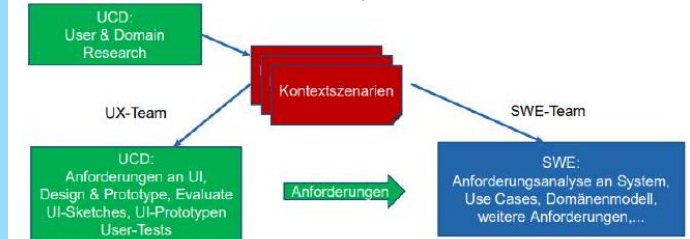
(E 02 - Arbeitsgruppenanalyse 1) | Ausgabe 1/2019

1. Zielgruppe identifizieren
  - Wer sind die Benutzer?
  - Was sind ihre Aufgaben/Ziele?
  - Wie sieht ihre Arbeitsumgebung aus?
2. Daten sammeln durch
  - Contextual Inquiry
  - Interviews
  - Beobachtung
  - Fokusgruppen
  - Nutzungsauswertung
3. Ergebnisse dokumentieren in
  - Personas
  - Usage-Szenarien
  - Mentales Modell

**Lösung: Sarah Schmidt, 34, Projektmanagerin**

- ## Requirements Engineering

- Leistungsfähigkeiten oder Eigenschaften
- Explizit oder implizit
- Müssen mit allen Stakeholdern erarbeitet werden
- Entwickeln sich während des Projekts



- Was soll das System tun?
- Dokumentiert in Use Cases
- Messbar und testbar

**Nicht-funktionale Anforderungen:**

- Wie soll das System sein?
- Performance, Sicherheit, Usability
- Nach ISO 25010 kategorisiert

- Technische Einschränkungen
- Gesetzliche Vorgaben
- Budget und Zeitrahmen

## Use Cases

Textuelle Beschreibung einer konkreten Interaktion zwischen Akteur und System:

- Aus Sicht des Akteurs
- Aktiv formuliert
- Konkreter Nutzen
- Essentieller Stil (Logik statt Implementierung)

## Akteure in Use Cases

- **Primärakteur:** Initiiert den Use Case, erhält Hauptnutzen
- **Unterstützender Akteur:** Hilft bei der Durchführung
- **Offstage-Akteur:** Indirekt beteiligter Stakeholder

## Use Case Erstellung

Schritte zur Erstellung eines vollständigen Use Cases:

1. **Identifikation:**
  - Systemgrenzen definieren
  - Primärakteure identifizieren
  - Ziele der Akteure ermitteln
2. **Dokumentation:**
  - Brief/Casual für erste Analyse
  - Fully-dressed für wichtige Use Cases
  - Standardablauf und Erweiterungen
3. **Review:**
  - Mit Stakeholdern abstimmen
  - Auf Vollständigkeit prüfen
  - Konsistenz sicherstellen

## Brief Use Case Verkauf abwickeln

Kunde kommt mit Waren zur Kasse. Kassier erfasst alle Produkte. System berechnet Gesamtbetrag. Kassier nimmt Zahlung entgegen und gibt ggf. Wechselgeld. System druckt Beleg.

## Fully-dressed Use Case UC: Verkauf abwickeln

- **Umfang:** Kassensystem
- **Primärakteur:** Kassier
- **Stakeholder:** Kunde (schnelle Abwicklung), Geschäft (korrekte Abrechnung)
- **Vorbedingung:** Kasse ist geöffnet
- **Standardablauf:**
  1. Kassier startet neuen Verkauf
  2. System initialisiert neue Transaktion
  3. Kassier erfasst Produkte
  4. System zeigt Zwischensumme
  5. Kassier schliesst Verkauf ab
  6. System zeigt Gesamtbetrag
  7. Kunde bezahlt
  8. System druckt Beleg

## Use Case Verfeinerung Schritte zur Verfeinerung eines Use Cases:

1. **Brief Use Case**
  - Kurze Zusammenfassung
  - Hauptablauf skizzieren
  - Keine Details zu Varianten
2. **Casual Use Case**
  - Mehrere Absätze
  - Hauptvarianten beschreiben
  - Informeller Stil
3. **Fully-dressed Use Case**
  - Vollständige Struktur
  - Alle Varianten
  - Vor- und Nachbedingungen
  - Garantien definieren

**Typische Prüfungsaufgabe: Use Case Analyse Aufgabe:** Analysieren Sie den folgenden Use Case und identifizieren Sie mögliche Probleme:

**Use Case:** "Der Benutzer loggt sich ein und das System zeigt die Startseite. Er klickt auf den Button und die Daten werden in der Datenbank gespeichert."

**Probleme:**

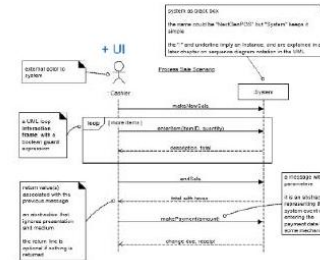
- Zu technisch/implementierungsnah
- Fehlende Akteurperspektive
- Unklarer Nutzen/Ziel
- Fehlende Alternativszenarien
- Keine Fehlerbehandlung

**Verbesserter Use Case:** "Der Kunde möchte seine Bestelldaten speichern. Er bestätigt die Eingaben und erhält eine Bestätigung über die erfolgreiche Speicherung."

## Systemsequenzdiagramm (SSD)

Formalisierte Darstellung der System-Interaktionen:

- Zeigt Input/Output-Events
- Identifiziert Systemoperationen
- Basis für API-Design
- Links ist Primärakteur aufgeführt
  - Hier Cashier
    - Inkl. seiner Benutzerschnittstelle
    - Initiiert die Systemoperationen (via UI)
      - UI findet zusammen mit Akteur heraus, was dieser tun möchte
      - UI ruft sodann entsprechende Systemoperation auf
- Mitte das System (:System)
  - Muss die Systemoperationen zur Verfügung stellen
- Rechts
  - Sekundärakteure, falls nötig



## Systemsequenzdiagramme

### SSD Erstellung

1. Use Case als Grundlage wählen
2. Akteur und System identifizieren
3. Methodenaufrufe definieren:
  - Namen aussagekräftig wählen
  - Parameter festlegen
  - Rückgabewerte bestimmen
4. Zeitliche Abfolge modellieren
5. Optional: Externe Systeme einbinden

**Äpgjy Aufgabe:** Erstellen Sie einen fully-dressed Use Case für ein Online-Bibliothekssystem. Fokus: "Buch ausleihen"

**Lösung:**

- **Umfang:** Online-Bibliothekssystem
- **Primärakteur:** Bibliotheksnutzer
- **Stakeholder:**
  - Bibliotheksnutzer: Möchte Buch einfach ausleihen
  - Bibliothek: Korrekte Erfassung der Ausleihe
- **Vorbedingung:** Nutzer ist eingeloggt
- **Standardablauf:**
  1. Nutzer sucht Buch
  2. System zeigt Verfügbarkeit
  3. Nutzer wählt Ausleihe
  4. System prüft Ausleihberechtigung
  5. System registriert Ausleihe
  6. System zeigt Bestätigung
- **Erweiterungen:**
  - 2a: Buch nicht verfügbar
  - 4a: Keine Ausleihberechtigung

**SSD Übungsaufgabe Aufgabe:** Erstellen Sie ein Systemsequenzdiagramm für den Use Case "Geld abheben an einem Bankautomaten."

**Wichtige Aspekte:**

- Kartenvvalidierung
- PIN-Eingabe
- Betragseingabe
- Kontostandsprüfung
- Geldausgabe
- Belegdruck

**Essentielle Systemoperationen:**

- validateCard(cardNumber)
- checkPIN(pin)
- withdrawMoney(amount)
- printReceipt()

## Domänenmodellierung

### Domänenmodell

Ein Domänenmodell ist ein vereinfachtes UML-Klassendiagramm zur Darstellung der Fachdomäne:

- Konzepte als Klassen
- Eigenschaften als Attribute (ohne Typangabe)
- Beziehungen als Assoziationen mit Multiplizitäten
- Optional: Aggregationen/Kompositionen

### Domänenmodell Erstellung

#### Schritt 1: Konzepte identifizieren

- Substantive aus Anforderungen extrahieren
- Kategorien prüfen:
  - Physische Objekte
  - Kataloge
  - Container
  - Externe Systeme
  - Rollen von Personen
  - Artefakte (Pläne, Dokumente)
  - Zahlungsinstrumente

- **Wichtig:** Keine Softwareklassen modellieren!

#### Schritt 2: Attribute definieren

- Nur wichtige/einfache Attribute
- Typische Kategorien:
  - Transaktionsdaten
  - Teil-Ganzes Beziehungen
  - Beschreibungen
  - Verwendungszwecke

- **Wichtig:** Beziehungen als Assoziationen, nicht als Attribute!

#### Schritt 3: Beziehungen modellieren

- Assoziationen zwischen Konzepten identifizieren
- Multiplizitäten festlegen
- Art der Beziehung bestimmen
- Richtung der Assoziation falls nötig

### Domänenmodell Zweck

- Visualisierung der Fachdomäne für alle Stakeholder
- Grundlage für das spätere Softwaredesign
- Gemeinsames Verständnis der Begriffe und Zusammenhänge
- Dokumentation der fachlichen Strukturen
- Basis für die Kommunikation zwischen Entwicklung und Fachbereich

**Prüfungsaufgabe: Konzeptidentifikation** **Aufgabentext:** Ein Bibliothekssystem verwaltet Bücher, die von Mitgliedern ausgeliehen werden können. Jedes Buch hat eine ISBN und mehrere Exemplare. Mitglieder können maximal 5 Bücher gleichzeitig für 4 Wochen ausleihen. Bei Überschreitung wird eine Mahngebühr fällig."

#### Identifizierte Konzepte:

- Buch (Beschreibungsklasse)
- Exemplar (Physisches Objekt)
- Mitglied (Rolle)
- Ausleihe (Transaktion)
- Mahnung (Artefakt)

#### Begründung:

- Buch/Exemplar: Trennung wegen mehrfacher Exemplare (Beschreibungsmuster)
- Ausleihe: Verbindet Exemplar und Mitglied, hat Zeitbezug
- Mahnung: Entsteht bei Fristüberschreitung

## Analysemuster im Domänenmodell

Bewährte Strukturen für wiederkehrende Modellierungssituationen:

### 1. Beschreibungsklassen

- Trennung von Instanz und Beschreibung
- Beispiel: Artikel vs. Artikelbeschreibung
- Vermeidet Redundanz bei gleichen Eigenschaften

### 2. Generalisierung/Spezialisierung

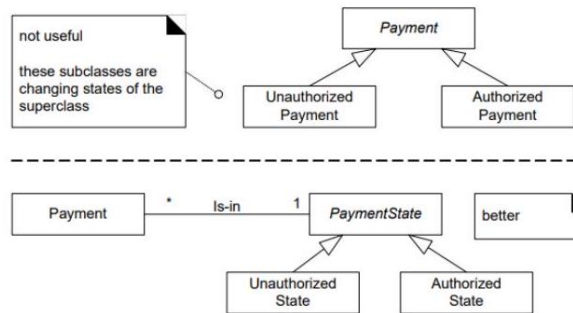
- 100% Regel: Alle Instanzen der Spezialisierung sind auch Instanzen der Generalisierung
- IS-A-Beziehung
- Gemeinsame Attribute/Assoziationen als Grund für Generalisierung

### 3. Komposition

- Starke Teil-Ganzes Beziehung
- Existenzabhängigkeit der Teile

### 4. Zustandsmodellierung

- Zustände als eigene Hierarchie
- Vermeidet problematische Vererbung

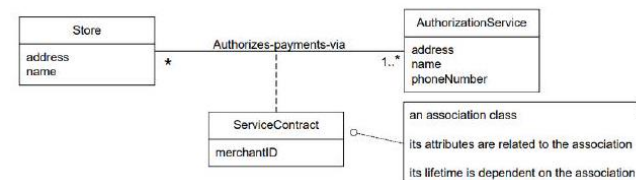


### 5. Rollen

- Unterschiedliche Rollen eines Konzepts
- Als eigene Konzepte oder Assoziationen

### 6. Assoziationsklassen

- Attribute einer Beziehung
- Eigene Klasse für die Assoziation



### 7. Wertobjekte

- Masseinheiten als eigene Konzepte
- Zeitintervalle als Konzepte
- Vermeidet primitive Obsession

**Analysemuster Anwendung** Systematisches Vorgehen bei der Anwendung von Analysemustern:

#### 1. Muster identifizieren

- Beschreibungsklassen bei gleichartigen Objekten
- Generalisierung bei ist-ein-Beziehungen
- Komposition bei existenzabhängigen Teilen
- Zustände bei Objektlebenszyklen
- Rollen bei verschiedenen Funktionen
- Assoziationsklassen bei Beziehungsattributen
- Wertobjekte bei komplexen Werten

#### 2. Muster anwenden

- Struktur des Musters übernehmen
- An Kontext anpassen
- Konsistenz prüfen

#### 3. Muster kombinieren

- Überschneidungen identifizieren
- Konflikte auflösen
- Gesamtmodell harmonisieren

**Domänenmodell Online-Shop** **Aufgabe:** Erstellen Sie ein Domänenmodell für einen Online-Shop mit Warenkorb-Funktion.

#### Lösung:

##### • Konzepte identifizieren:

- Artikel (physisches Objekt)
- Artikelbeschreibung (Beschreibungsklasse)
- Warenkorb (Container)
- Bestellung (Transaktion)
- Kunde (Rolle)

##### • Attribute:

- Artikelbeschreibung: name, preis, beschreibung
- Bestellung: datum, status
- Kunde: name, adresse

##### • Beziehungen:

- Warenkorb gehört zu genau einem Kunde (Komposition)
- Warenkorb enthält beliebig viele Artikel
- Bestellung wird aus Warenkorb erstellt

**Komplexes Domänenmodell: Reisebuchungssystem** **Anforderung:** Modellieren Sie ein System für Pauschalreisen mit Flügen, Hotels und Aktivitäten.

#### Verwendete Analysemuster:

##### • Beschreibungsklassen:

- Flugverbindung vs. konkreter Flug
- Hotelkategorie vs. konkretes Zimmer
- Aktivitätstyp vs. konkrete Durchführung

##### • Zustände:

- Buchungszustände: angefragt, bestätigt, storniert
- Zahlungszustände: offen, teilbezahlt, vollständig

##### • Rollen:

- Person als: Kunde, Reiseleiter, Kontaktperson

##### • Wertobjekte:

- Geldbetrag mit Währung
- Zeitraum für Reisedauer

[Hier könnte ein Beispiel-Diagramm folgen]



## Review eines Domänenmodells

Checkliste für die Überprüfung:

- **Fachliche Korrektheit**
  - Alle relevanten Konzepte vorhanden?
  - Begriffe aus der Fachdomäne verwendet?
  - Beziehungen fachlich sinnvoll?
- **Technische Korrektheit**
  - UML-Notation korrekt?
  - Multiplizitäten angegeben?
  - Assoziationsnamen vorhanden?
- **Modellqualität**
  - Angemessener Detaillierungsgrad?
  - Analysemuster sinnvoll eingesetzt?
  - Keine Implementation vorweggenommen?

## Typische Modellierungsfehler vermeiden

- **Keine Softwareklassen modellieren**
  - Manager-Klassen vermeiden
  - Keine technischen Helper-Klassen
- **Keine Operationen modellieren**
  - Fokus auf Struktur, nicht Verhalten
  - Keine CRUD-Operationen
- **Richtige Abstraktionsebene wählen**
  - Nicht zu detailliert
  - Nicht zu abstrakt
  - Fachliche Begriffe verwenden
- **Assoziationen statt Attribute**
  - Beziehungen als Assoziationen modellieren
  - Keine Objekt-IDs als Attribute

## Typische Prüfungsaufgabe: Modell verbessern

Fehlerhaftes Modell:

- Klasse "userManager" mit CRUD-Operationen
- Attribute "customerID" und "orderID" statt Assoziationen
- Operation "calculateTotal()" in Bestellung
- Technische Klasse "DatabaseConnection"

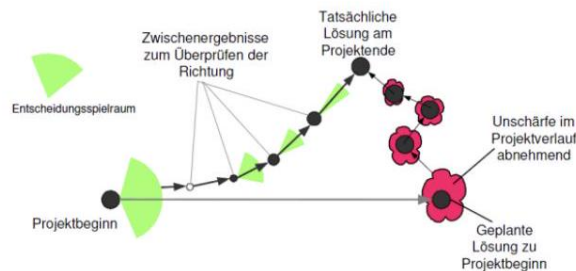
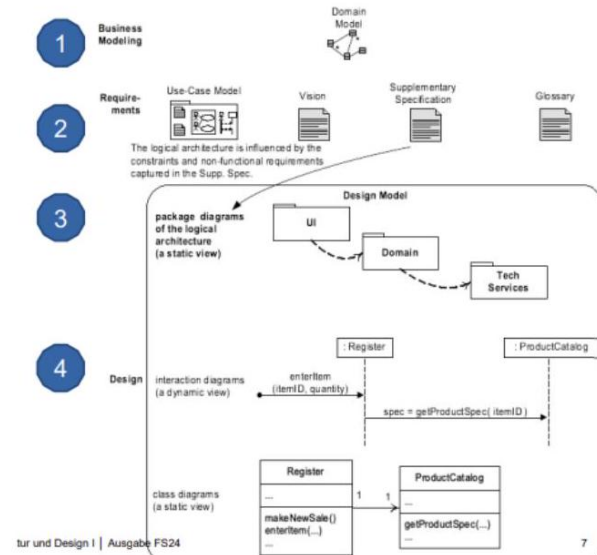
## Verbesserungen:

- userManager entfernen, stattdessen Beziehungen modellieren
- IDs durch direkte Assoziationen ersetzen
- Operationen entfernen (gehören ins Design)
- Technische Klassen entfernen

## Überblick Softwareentwicklung

Die Entwicklung von Software erfolgt in verschiedenen Ebenen:

- Business Analyse (Domänenmodell, Requirements)
- Architektur (Logische Struktur)
- Entwicklung (Konkrete Umsetzung)



## Softwarearchitektur

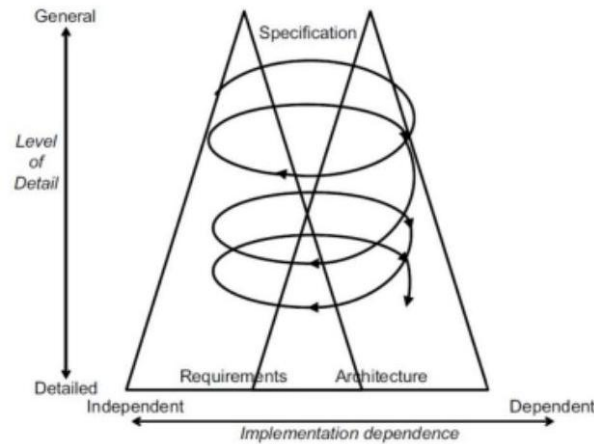
Die Architektur definiert:

- Grundlegende Strukturen und Komponenten
- Heutige und zukünftige Anforderungen
- Weiterentwicklungsmöglichkeiten
- Beziehungen zur Umgebung

## Architekturanalyse

Die Analyse erfolgt iterativ mit den Anforderungen:

- Analyse funktionaler und nicht-funktionaler Anforderungen
- Abstimmung mit Stakeholdern
- Kontinuierliche Weiterentwicklung



## ISO 25010 vs FURPS+

**ISO 25010:**

- Hierarchische Struktur für nicht-funktionale Anforderungen
- Definierte Hauptcharakteristiken und Subcharakteristiken
- Messbare Metriken für jede Anforderung
- Präzise Formulierung und Verifikation

**FURPS+:**

- Functionality (Funktionalität)
- Usability (Benutzbarkeit)
- Reliability (Zuverlässigkeit)
- Performance (Leistung)
- Supportability (Wartbarkeit)
- + (Implementation, Interface, Operations, Packaging, Legal)

## Modulkonzept

Ein Modul (Baustein, Komponente) wird bewertet nach:

- **Kohäsion:** Innerer Zusammenhang
- **Kopplung:** Externe Abhängigkeiten

**Eigenschaften:**

- Autarkes Teilsystem
- Minimale externe Schnittstellen
- Enthält alle benötigten Funktionen/Daten
- Verschiedene Formen: Paket, Library, Service

**Architekturentscheidungen treffen** Systematischer Ansatz für Architekturentscheidungen:

- 1. Anforderungen analysieren**
  - Funktionale Anforderungen gruppieren
  - Nicht-funktionale Anforderungen priorisieren
  - Randbedingungen identifizieren
- 2. Einflussfaktoren bewerten**
  - Technische Faktoren
  - Organisatorische Faktoren
  - Wirtschaftliche Faktoren
- 3. Alternativen evaluieren**
  - Vor- und Nachteile abwägen
  - Proof of Concepts durchführen
  - Risiken analysieren
- 4. Entscheidung dokumentieren**
  - Begründung festhalten
  - Verworfen Alternativen dokumentieren
  - Annahmen dokumentieren

**Typische Prüfungsaufgabe: Architekturanalyse Aufgabenstellung:** Analysieren Sie folgende Anforderungen und leiten Sie architektonische Konsequenzen ab:

- System muss 24/7 verfügbar sein
- 10.000 gleichzeitige Benutzer
- Reaktionszeit unter 1 Sekunde
- Jährliche Wartungsfenster maximal 4 Stunden

**Lösung:**

• **Architekturentscheidungen:**

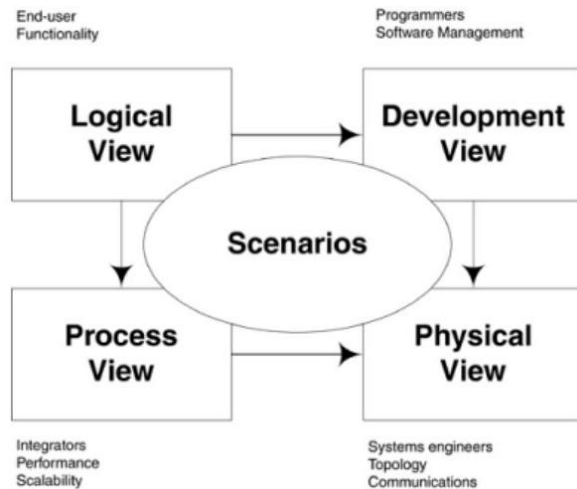
- Verteilte Architektur für Hochverfügbarkeit
- Load Balancing für gleichzeitige Benutzer
- Caching-Strategien für Performanz
- Blue-Green Deployment für Wartung

• **Begründungen:**

- Verteilung minimiert Single Points of Failure
- Load Balancer verteilt Last gleichmäßig
- Caching reduziert Datenbankzugriffe
- Blue-Green erlaubt Updates ohne Downtime

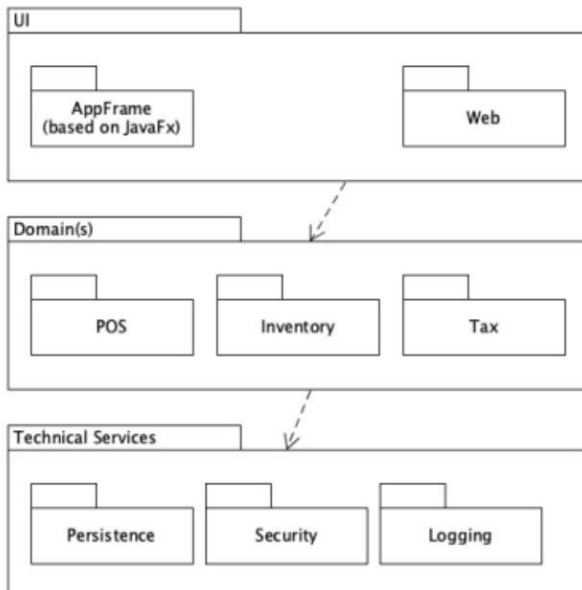
## Architektursichten

Das N+1 View Model beschreibt verschiedene Perspektiven:



### UML-Paketdiagramm:

- Definition von Teilsystemen
- Gruppierung von Elementen
- Abhängigkeiten zwischen Paketen



## UML Diagrammauswahl Entscheidungshilfe für die Wahl des UML-Diagrammtyps:

### 1. Strukturbeschreibung benötigt:

- Klassendiagramm für Typen und Beziehungen
- Paketdiagramm für Modularisierung
- Komponentendiagramm für Bausteinsicht
- Verteilungsdiagramm für Deployment

### 2. Verhaltensbeschreibung benötigt:

- Sequenzdiagramm für Interaktionsabläufe
- Aktivitätsdiagramm für Workflows
- Zustandsdiagramm für Objektlebenszyklen
- Kommunikationsdiagramm für Objektkollaborationen

### 3. Abstraktionsebene wählen:

- Analyse: Konzeptuelle Diagramme
- Design: Detaillierte Spezifikation
- Implementation: Codenahes Design

**GRASP Anwendung Szenario:** Online-Shop Warenkorb-Funktionalität

### GRASP-Prinzipien angewandt:

- **Information Expert:**
  - Warenkorb kennt seine Positionen
  - Berechnet selbst Gesamtsumme
- **Creator:**
  - Warenkorb erstellt Warenkorbpositionen
  - Bestellung erstellt aus Warenkorb
- **Controller:**
  - ShoppingController koordiniert UI und Domain
  - Keine Geschäftslogik im Controller
- **Low Coupling:**
  - UI kennt nur Controller
  - Domain unabhängig von UI

## Architektur-Review durchführen Vorgehen:

1. **Vorbereitung**
  - Architektur-Dokumentation zusammenstellen
  - Review-Team zusammenstellen
  - Checklisten vorbereiten
2. **Durchführung**
  - Architektur vorstellen
  - Anforderungen prüfen
  - Entscheidungen hinterfragen
  - Risiken identifizieren
3. **Nachbereitung**
  - Findings dokumentieren
  - Maßnahmen definieren
  - Follow-up planen

### Prüfkriterien:

- Anforderungserfüllung
- Technische Machbarkeit
- Zukunftssicherheit
- Best Practices

## UML-Modellierung

### Statische vs. Dynamische Modelle

#### Statische Modelle (Struktur):

- UML-Klassendiagramm
- Fokus auf Pakete, Klassen, Attribute
- Keine Methodenimplementierung

#### Dynamische Modelle (Verhalten):

- UML-Interaktionsdiagramme
- Fokus auf Logik und Verhalten
- Implementierung der Methoden

### UML-Diagrammtypen

#### 1. Klassendiagramm:

- Klassen und aktive Klassen
- Attribute und Operationen
- Sichtbarkeiten und Beziehungen
- Interfaces und Realisierungen

#### 2. Sequenzdiagramm:

- Lebenslinien und Nachrichten
- Synchrone/Asynchrone Kommunikation
- Aktivierung und Deaktivierung
- Alternative Abläufe

#### 3. Zustandsdiagramm:

- Zustände und Übergänge
- Start- und Endzustände
- Composite States
- Historie und Parallelität

#### 4. Aktivitätsdiagramm:

- Aktionen und Aktivitäten
- Kontroll- und Datenflüsse
- Verzweigungen und Zusammenführungen
- Partitionen (Swimlanes)

### Responsibility Driven Design (RDD)

Design basierend auf Verantwortlichkeiten:

- Klassenentwurf nach Rollen
- Kollaborationsbeziehungen
- Implementierung durch Attribute/Methoden
- Anwendbar auf allen Ebenen

### GRASP Prinzipien

General Responsibility Assignment Software Patterns:

- **Information Expert:** Verantwortung basierend auf Information
- **Creator:** Objekterstellung bei starker Beziehung
- **Controller:** Zentrale Steuerungslogik
- **Low Coupling:** Minimale Abhängigkeiten
- **High Cohesion:** Starker innerer Zusammenhang
- **Polymorphism:** Flexibilität durch Schnittstellen
- **Pure Fabrication:** Künstliche Klassen für besseres Design
- **Indirection:** Vermittler für Flexibilität
- **Protected Variations:** Kapselung von Änderungen



**Architekturentwurf Aufgabe:** Entwerfen Sie die grundlegende Architektur für ein Online-Banking-System.

**Lösung:**

- **Anforderungsanalyse:**
  - Sicherheit (ISO 25010)
  - Performance (FURPS+)
  - Skalierbarkeit
- **Architekturentscheidungen:**
  - Mehrschichtige Architektur
  - Microservices für Skalierbarkeit
  - Sicherheitsschicht
- **Module:**
  - Authentifizierung
  - Transaktionen
  - Kontoführung

**Architekturentwurf Schritte:**

1. Anforderungen analysieren
2. Architekturstil wählen
3. Module identifizieren
4. Schnittstellen definieren
5. Mit Stakeholdern abstimmen

**Qualitätskriterien:**

- Änderbarkeit
- Wartbarkeit
- Erweiterbarkeit
- Testbarkeit

**Prüfungsaufgabe: UML-Modellierung Aufgabe:** Modellieren Sie für ein Bibliothekssystem die Ausleihe eines Buches mit:

- Klassendiagramm der beteiligten Klassen
- Sequenzdiagramm des Ausleihvorgangs
- Zustandsdiagramm für ein Buchexemplar

**Bewertungskriterien:**

- Korrekte UML-Notation
- Vollständigkeit der Modellierung
- Konsistenz zwischen Diagrammen
- Angemessener Detaillierungsgrad

## Use Case Realisation

### Use Case Realization

Die Umsetzung von Use Cases erfolgt durch:

- Detaillierte Szenarien aus den Use Cases
- Systemantworten müssen realisiert werden
- UI statt System im SSD
- Systemoperationen sind die zu implementierenden Elemente

### Use Case Realization Ziele

- Umsetzung der fachlichen Anforderungen in Code
- Einhaltung der Architekturvorgaben
- Implementierung der GRASP-Prinzipien
- Erstellung wartbaren und testbaren Codes
- Dokumentation der Design-Entscheidungen

### UML im Implementierungsprozess

UML dient als:

- Zwischenschritt bei wenig Erfahrung
- Kompakter Ersatz für Programmiercode
- Kommunikationsmittel (auch für Nicht-Techniker)

Analyse eines System Sequence Diagrams    Use Case: Geld abheben am Bankomat

#### Systemoperationen identifizieren:

- validateCard(cardNumber)
- verifyPIN(pin)
- selectAmount(amount)
- withdrawMoney()
- printReceipt()

#### Operation Contract für withdrawMoney():

- **Vorbedingungen:**
  - Karte validiert
  - PIN korrekt
  - Betrag ausgewählt
- **Nachbedingungen:**
  - Kontosaldo aktualisiert
  - Transaktion protokolliert
  - Geld ausgegeben

## Vorgehen bei der Use Case Realization

### 1. Vorbereitung:

- Use Case auswählen und SSD ableiten
- Systemoperation identifizieren
- Operation Contract erstellen/prüfen

### 2. Analyse:

- Aktuellen Code/Dokumentation analysieren
- DCD überprüfen/aktualisieren
- Vergleich mit Domänenmodell
- Neue Klassen gemäß Domänenmodell erstellen

### 3. Realisierung:

- Controller Klasse bestimmen
- Zu verändernde Klassen festlegen
- Weg zu diesen Klassen festlegen:
  - Parameter für Wege definieren
  - Klassen bei Bedarf erstellen
  - Verantwortlichkeiten zuweisen
  - Verschiedene Varianten evaluieren
- Veränderungen implementieren
- Review durchführen

Use Case Realization: Verkauf abwickeln    1. Vorbereitung:

- **Use Case:** Verkauf abwickeln
- **Systemoperation:** makeNewSale()
- **Contract:** Neue Sale-Instanz wird erstellt

### 2. Analyse:

- **Klassen:** Register, Sale
- **DCD:** Beziehung Register-Sale prüfen
- **Neue Klassen:** Payment, SaleLineItem

### 3. Implementierung:

- Register als Controller
- Sale-Klasse erweitern
- Beziehungen implementieren

Design Class Diagram (DCD) erstellen    1. Klassen identifizieren

- Aus Domänenmodell übernehmen
- Technische Klassen ergänzen
- Controller bestimmen

### 2. Attribute definieren

- Datentypen festlegen
- Sichtbarkeiten bestimmen
- Validierungen vorsehen

### 3. Methoden hinzufügen

- Systemoperationen verteilen
- GRASP-Prinzipien anwenden
- Signaturen definieren

### 4. Beziehungen modellieren

- Assoziationen aus Domänenmodell
- Navigierbarkeit festlegen
- Abhängigkeiten minimieren

Vollständige Use Case Realization    Use Case: Bestellung aufgeben

### 1. Systemoperationen:

- createOrder()
- addItem(productId, quantity)
- removeItem(itemId)
- submitOrder()

### 2. Design-Entscheidungen:

- OrderController als Fassade
- Order aggregiert OrderItems
- OrderService für Geschäftslogik
- Repository für Persistenz

### 3. GRASP-Anwendung:

- Information Expert:
  - Order berechnet Gesamtsumme
  - OrderItem verwaltet Produktdaten
- Creator:
  - Order erstellt OrderItems
  - OrderService erstellt Orders
- Low Coupling:
  - Repository-Interface für Persistenz
  - Service-Interface für Geschäftslogik

### 4. Implementierung:

```
1 public class OrderController {
2     private OrderService orderService;
3     private Order currentOrder;
4
5     public void createOrder() {
6         currentOrder = orderService.createOrder();
7     }
8
9     public void addItem(String productId, int
10        quantity) {
11         currentOrder.addItem(productId, quantity);
12     }
13
14     public void submitOrder() {
15         orderService.submitOrder(currentOrder);
16     }
17 }
```

Implementierung prüfen    1. Funktionale Prüfung

- Use Case Szenarien durchspielen
- Randfälle testen
- Fehlersituationen prüfen

### 2. Strukturelle Prüfung

- Architekturkonformität
- GRASP-Prinzipien
- Clean Code Regeln

### 3. Qualitätsprüfung

- Testabdeckung
- Wartbarkeit
- Performance

**Typische Prüfungsaufgabe**   **Aufgabe:** Gegeben ist folgender Use Case:

"Kunde meldet sich an". Erstellen Sie:

- System Sequence Diagram
- Operation Contracts
- Design Class Diagram
- Implementierung der wichtigsten Methoden

**Bewertungskriterien:**

- Vollständigkeit der Modellierung
- Korrekte Anwendung der GRASP-Prinzipien
- Sinnvolle Verteilung der Verantwortlichkeiten
- Testbare Implementierung

#### Typische Implementierungsfehler vermeiden

- **Architekturverletzungen:**
  - Schichtentrennung beachten
  - Abhängigkeiten richtig setzen
- **GRASP-Verletzungen:**
  - Information Expert beachten
  - Creator Pattern richtig anwenden
  - High Cohesion erhalten
- **Testbarkeit:**
  - Klassen isoliert testbar halten
  - Abhängigkeiten mockbar gestalten

## Design Patterns

### Grundlagen Design Patterns

Bewährte Lösungsmuster für wiederkehrende Probleme:

- Beschleunigen Entwicklung durch vorgefertigte Lösungen
- Verbessern Kommunikation im Team
- Bieten Balance zwischen Flexibilität und Komplexität
- **Wichtig:** Design Patterns sind kein Selbstzweck

## Grundlegende Design Patterns

### Adapter Pattern

**Problem:** Inkompatible Schnittstellen

- Objekte mit unterschiedlichen Interfaces sollen zusammenarbeiten
- Externe Dienste sollen austauschbar sein

**Lösung:** Adapter-Klasse als Vermittler

### Simple Factory Pattern

**Problem:** Komplexe Objekterzeugung

- Objekterzeugung erfordert viele Schritte
- Konfiguration bei Erzeugung notwendig

**Lösung:** Eigene Klasse für Objekterzeugung

### Singleton Pattern

**Problem:** Genau eine Instanz benötigt

- Globaler Zugriffspunkt notwendig
- Mehrfachinstanzierung verhindern

**Lösung:** Statische Instanz mit privater Erzeugung

### Dependency Injection Pattern

**Problem:** Abhängigkeiten zu anderen Objekten

- Lose Kopplung erwünscht
- Flexibilität bei Abhängigkeiten

**Lösung:** Abhängigkeiten werden von außen injiziert

### Proxy Pattern

**Problem:** Zugriffskontrolle auf Objekte

- Verzögertes Laden
- Zugriffsbeschränkungen
- Netzwerkkommunikation

**Lösung:** Stellvertreterobjekt mit gleichem Interface

- **Remote Proxy:** Für entfernte Objekte
- **Virtual Proxy:** Für spätes Laden
- **Protection Proxy:** Für Zugriffsschutz

### Chain of Responsibility Pattern

**Problem:** Unklare Zuständigkeit für Anfragen

- Mehrere mögliche Handler
- Zuständigkeit erst zur Laufzeit klar

**Lösung:** Verkettete Handler-Objekte

## Erweiterte Design Patterns

### Decorator Pattern

**Problem:** Dynamische Erweiterung von Objekten

- Zusätzliche Verantwortlichkeiten
- Nur für einzelne Objekte

**Lösung:** Wrapper-Objekt mit gleichem Interface

### Observer Pattern

**Problem:** Abhängige Objekte aktualisieren

- Lose Kopplung erwünscht
- Typ des Empfängers unbekannt

**Lösung:** Observer-Interface für Benachrichtigungen

### Strategy Pattern

**Problem:** Austauschbare Algorithmen

- Verschiedene Implementierungen
- Zur Laufzeit wechselbar

**Lösung:** Interface für Algorithmus-Klassen

### Composite Pattern

**Problem:** Baumstrukturen verwalten

- Einheitliche Behandlung
- Teil-Ganzes Hierarchie

**Lösung:** Gemeinsames Interface für Container und Inhalt

### Design Pattern Auswahl

**Schritt 1: Problem analysieren**

- Art des Problems identifizieren
- Anforderungen klar definieren
- Kontext verstehen

**Schritt 2: Pattern evaluieren**

- Passende Patterns suchen
- Vor- und Nachteile abwägen
- Komplexität bewerten

**Schritt 3: Implementation planen**

- Klassenstruktur entwerfen
- Schnittstellen definieren
- Anpassungen vornehmen

### Pattern-Analyse für Prüfung

**Systematisches Vorgehen:**

1. **Problem identifizieren**
  - Was ist das Kernproblem?
  - Welche Flexibilität wird benötigt?
  - Welche Einschränkungen gibt es?
2. **Pattern auswählen**
  - Welche Patterns lösen ähnliche Probleme?
  - Wie unterscheiden sich die Patterns?
  - Welche Trade-offs gibt es?
3. **Lösung skizzieren**
  - Klassenstruktur beschreiben
  - Beziehungen definieren
  - Vor- und Nachteile nennen

**Prüfungsaufgabe: Pattern-Identifikation** **Szenario:** Ein Dokumentensystem soll verschiedene Dateitypen (.pdf, .doc, .txt) einheitlich behandeln. Jeder Dateityp benötigt eine spezielle Verarbeitung für Öffnen, Speichern und Drucken.

**Aufgabe:**

1. Identifizieren Sie geeignete Design Patterns
2. Begründen Sie Ihre Auswahl
3. Skizzieren Sie die Struktur der Lösung

**Musterlösung:**

- **Mögliche Patterns:**
  - Strategy (für Verarbeitungslogik)
  - Factory (für Dokumenterstellung)
  - Adapter (für einheitliche Schnittstelle)
- **Begründung Strategy:**
  - Unterschiedliche Algorithmen pro Dateityp
  - Austauschbarkeit der Verarbeitung
  - Erweiterbar für neue Dateitypen
- **Struktur:**
  - Interface DocumentProcessor
  - Konkrete Prozessoren pro Dateityp
  - Context-Klasse Document

**Pattern-Vergleich: Adapter vs. Facade** **Gegeben sind zwei Patterns. Vergleichen Sie diese:**

**Adapter:**

- **Zweck:** Inkompatible Schnittstellen vereinen
- **Struktur:** Wrapper um einzelne Klasse
- **Anwendung:** Bei existierenden, inkompatiblen Klassen

**Facade:**

- **Zweck:** Komplexes Subsystem vereinfachen
- **Struktur:** Neue Schnittstelle für mehrere Klassen
- **Anwendung:** Bei komplexen Subsystemen

**Kernunterschiede:**

- Adapter ändert Interface, Facade vereinfacht
- Adapter für einzelne Klasse, Facade für Subsystem
- Adapter für Kompatibilität, Facade für Vereinfachung

**Pattern-Kombination** **Schritte zur Kombination mehrerer Patterns:**

1. **Abhängigkeiten analysieren**
  - Welche Patterns ergänzen sich?
  - Wo gibt es Überschneidungen?
  - Welche Reihenfolge ist sinnvoll?
2. **Struktur entwerfen**
  - Gemeinsame Schnittstellen identifizieren
  - Verantwortlichkeiten zuordnen
  - Komplexität im Auge behalten
3. **Integration planen**
  - Übergänge zwischen Patterns definieren
  - Konsistenz sicherstellen
  - Testbarkeit gewährleisten

## Von Design zu Code

### Implementierungsstrategien

#### 1. Bottom-Up Entwicklung:

- Implementierung beginnt mit Basisbausteinen
- Schrittweise Integration zu größeren Komponenten
- **Vorteile:** Gründlich, solide Basis
- **Nachteile:** Spätes Feedback

#### 2. Agile Entwicklung:

- Inkrementelle Entwicklung in Sprints
- Kontinuierliche Integration und Auslieferung
- **Vorteile:** Flexibilität, schnelles Feedback
- **Nachteile:** Mögliche Restrukturierung nötig

### Entwicklungsansätze

#### Code-Driven Development (CDD):

- Direkte Implementierung der Klassen
- Nachträgliches Testing

#### Test-Driven Development (TDD):

- Tests vor Implementation
- Red-Green-Refactor Zyklus

#### Behavior-Driven Development (BDD):

- Testbeschreibung aus Anwendersicht
- Gherkin-Syntax für Szenarios

### Clean Code

#### 1. Code-Guidelines:

- Einheitliche Formatierung
- Klare Namenskonventionen
- Dokumentationsrichtlinien

#### 2. Fehlerbehandlung:

- Exceptions statt Fehlercodes
- Sinnvolle Error Messages
- Logging-Strategie

#### 3. Namensgebung:

- Aussagekräftige Namen
- Konsistente Begriffe
- Domain-Driven Naming

### Laufzeit-Optimierung

#### Grundregeln:

- Zuerst messen, dann optimieren
- Performance-Profilen nutzen
- Bottlenecks identifizieren

#### Häufige Probleme:

- Datenbank-Zugriffe
- Ineffiziente Algorithmen
- Speicherlecks

**Prüfungsaufgabe:** Entwicklungsansätze vergleichen **Szenario:** Ein Team soll eine neue Webanwendung entwickeln. Diskutieren Sie die Vor- und Nachteile von TDD gegenüber CDD für dieses Projekt.

#### Musterlösung:

- **TDD Vorteile:**
  - Testbare Architektur von Anfang an
  - Frühe Fehlererkennung
  - Dokumentation durch Tests
  - Sicherheit bei Refactoring
- **TDD Nachteile:**
  - Initial höherer Zeitaufwand
  - Lernkurve für das Team
  - Schwierig bei unklaren Anforderungen
- **Empfehlung:**
  - TDD für kritische Kernkomponenten
  - CDD für Prototypen und UI
  - Hybridansatz je nach Modulkritikalität

## Refactoring

### Refactoring Grundlagen

Strukturierte Verbesserung des Codes ohne Änderung des externen Verhaltens:

- Kleine, kontrollierte Schritte
- Erhaltung der Funktionalität
- Verbesserung der Codequalität

### Refactoring Durchführung

#### 1. Code Smells identifizieren:

- Duplizierter Code
- Lange Methoden
- Große Klassen
- Hohe Kopplung

#### 2. Refactoring durchführen:

- Tests sicherstellen
- Änderungen vornehmen
- Tests ausführen

#### 3. Patterns anwenden:

- Extract Method
- Move Method
- Rename
- Introduce Variable

### Code Review Durchführung 1. Vorbereitung

- Code-Guidelines bereitstellen
- Checkliste erstellen
- Scope definieren

#### 2. Review durchführen

- Lesbarkeit prüfen
- Naming Conventions
- Architekturkonformität
- Testabdeckung

#### 3. Feedback geben

- Konstruktiv formulieren
- Priorisieren
- Lösungen vorschlagen

**Typische Prüfungsaufgabe:** Code Smells **Analysieren Sie folgenden Code auf Code Smells:**

#### Problematischer Code:

- Klasse UserManager mit 1000 Zeilen
- Methode processData mit 200 Zeilen
- Variable data wird in 15 Methoden verwendet
- Duplizierte Validierungslogik in mehreren Klassen

#### Identifizierte Smells:

- **God Class:** UserManager zu groß
- **Long Method:** processData zu komplex
- **Global Variable:** data zu weit verbreitet
- **Duplicate Code:** Validierungslogik

#### Refactoring-Vorschläge:

- Aufteilen in spezialisierte Klassen
- Extract Method für processData
- Einführen einer Validierungsklasse
- Dependency Injection für data

## Testing

### Testarten

#### Nach Sicht:

- **Black-Box:** Funktionaler Test ohne Codekenntnis
- **White-Box:** Strukturbbezogener Test mit Codekenntnis

#### Nach Umfang:

- **Unit-Tests:** Einzelne Komponenten
- **Integrationstests:** Zusammenspiel
- **Systemtests:** Gesamtsystem
- **Akzeptanztests:** Kundenanforderungen

### Testentwicklung

#### 1. Testfall definieren:

- Vorbedingungen festlegen
- Testdaten vorbereiten
- Erwartetes Ergebnis definieren

#### 2. Test implementieren:

- Setup vorbereiten
- Testlogik schreiben
- Assertions definieren

#### 3. Test ausführen:

- Automatisiert ausführen
- Ergebnisse prüfen
- Dokumentation erstellen



**Prüfungsaufgabe: Teststrategie** **Szenario:** Ein Onlineshop-System soll getestet werden. Entwickeln Sie eine Teststrategie.

**Lösung:**

- **Unit Tests:**
  - Warenkorb-Berechnungen
  - Preis-Kalkulationen
  - Validierungsfunktionen
- **Integrationstests:**
  - Bestellprozess
  - Zahlungsabwicklung
  - Lagerverwaltung
- **System Tests:**
  - Performance unter Last
  - Sicherheitsaspekte
  - Datenbankinteraktionen
- **Akzeptanztests:**
  - Benutzerszenarien
  - Geschäftsprozesse
  - Reporting

**Testabdeckung optimieren** **1. Analyse der Testabdeckung**

- Code Coverage messen
- Kritische Pfade identifizieren
- Lücken dokumentieren

**2. Priorisierung**

- Geschäftskritische Funktionen
- Fehleranfällige Bereiche
- Komplexe Algorithmen

**3. Ergänzung der Tests**

- Randfall-Tests
- Negativtests
- Performance-Tests

**4. Wartung**

- Regelmäßige Überprüfung
- Anpassung an Änderungen
- Entfernung veralteter Tests

**Prüfungsaufgabe: Testfälle entwerfen** **Aufgabe:** Entwickeln Sie Testfälle für eine Methode zur Validierung einer Email-Adresse.

**Testfälle:**

- **Positive Tests:**
  - Standard Email (user@domain.com)
  - Subdomain (user@sub.domain.com)
  - Mit Punkten (first.last@domain.com)
- **Negative Tests:**
  - Fehlende @ (userdomain.com)
  - Mehrere @ (user@@domain.com)
  - Ungültige Zeichen (user#@domain.com)
- **Randfälle:**
  - Leerer String
  - Nur Whitespace
  - Sehr lange Adressen

## Verteilte Systeme

### Verteiltes System

Ein Netzwerk aus autonomen Computern und Softwarekomponenten, die als einheitliches System erscheinen:

- Autonome Knoten und Komponenten
- Netzwerkverbindung
- Erscheint als ein System

### Charakteristika verteilter Systeme

Typische Merkmale moderner verteilter Systeme:

- **Skalierbarkeit:** Oft sehr große Systeme
- **Datenorientierung:** Zentrale Datenbanken
- **Interaktivität:** GUI und Batch-Verarbeitung
- **Nebenläufigkeit:** Parallele Benutzerinteraktionen
- **Konsistenz:** Hohe Anforderungen an Datenkonsistenz

### Grundlegende Konzepte

#### 1. Kommunikation:

- Remote Procedure Calls (RPC)
- Message Queuing
- Publish-Subscribe-Systeme

#### 2. Fehlertoleranz:

- Replikation von Komponenten
- Failover-Mechanismen
- Fehlererkennung und -behandlung

#### 3. Fehlersemantik:

- Konsistenzgarantien
- Recovery-Verfahren
- Kompensationsmechanismen

### Architekturmuster

Grundlegende Architekturstile für verteilte Systeme:

- **Client-Server:** Zentraler Server, multiple Clients
- **Peer-to-Peer:** Gleichberechtigte Knoten
- **Publish-Subscribe:** Event-basierte Kommunikation

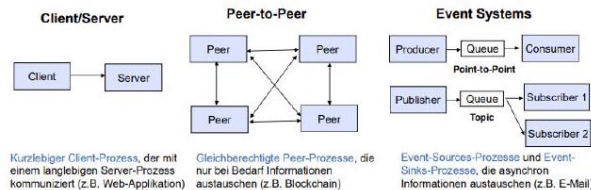


Abbildung 20: Architekturmodelle

**Prüfungsaufgabe: Architekturstil-Analyse** **Szenario:** Ein Messaging-System soll entwickelt werden, das folgende Anforderungen erfüllt:

- Hohe Skalierbarkeit
- Keine zentrale Komponente (Single Point of Failure)
- Direkter Nachrichtenaustausch zwischen Nutzern
- Offline-Fähigkeit

**Analysieren Sie die Architekturstile:**

#### 1. Client-Server

- **Vorteile:**
  - Zentrale Verwaltung
  - Einfache Konsistenzsicherung
- **Nachteile:**
  - Single Point of Failure
  - Skalierungsprobleme

#### 2. Peer-to-Peer

- **Vorteile:**
  - Keine zentrale Komponente
  - Direkte Kommunikation
  - Gute Skalierbarkeit
- **Nachteile:**
  - Komplexe Konsistenzsicherung
  - Schwierige Verwaltung

**Empfehlung:** Peer-to-Peer mit hybriden Elementen

### Verteilungsprobleme analysieren 1. Probleme identifizieren

- **Netzwerk:**
  - Latenz
  - Bandbreite
  - Ausfälle
- **Daten:**
  - Konsistenz
  - Replikation
  - Synchronisation
- **System:**
  - Skalierung
  - Verfügbarkeit
  - Wartbarkeit

#### 2. Lösungsstrategien entwickeln

- **Netzwerk:**
  - Caching
  - Compression
  - Redundanz
- **Daten:**
  - Eventual Consistency
  - Master-Slave Replikation
  - Konfliktauflösung
- **System:**
  - Load Balancing
  - Service Discovery
  - Circuit Breaker

**Typische Prüfungsaufgabe: CAP-Theorem** **Aufgabe:** Analysieren Sie für ein verteiltes Datenbanksystem die Auswirkungen des CAP-Theorems.

**CAP-Theorem Komponenten:**

- **Consistency:** Alle Knoten sehen dieselben Daten
- **Availability:** Jede Anfrage erhält eine Antwort
- **Partition Tolerance:** System funktioniert trotz Netzwerkausfällen

**Analyse der Trade-offs:**

- **CA-System:**
  - Hohe Konsistenz und Verfügbarkeit
  - Keine Netzwerkpartitionierung möglich
  - Beispiel: Traditionelle RDBMS
- **CP-System:**
  - Konsistenz und Partitionstoleranz
  - Eingeschränkte Verfügbarkeit
  - Beispiel: MongoDB
- **AP-System:**
  - Verfügbarkeit und Partitionstoleranz
  - Eventual Consistency
  - Beispiel: Cassandra

### Verteilte System-Design 1. Anforderungsanalyse

- **Funktional:**
  - Kernfunktionalitäten
  - Datenmodell
  - Schnittstellen
- **Nicht-funktional:**
  - Skalierbarkeit
  - Verfügbarkeit
  - Latenz
- 2. **Architekturentscheidungen**
  - **Kommunikation:**
    - Synchron vs. Asynchron
    - Push vs. Pull
    - Protokollwahl
  - **Datenmanagement:**
    - Sharding
    - Replikation
    - Caching
- 3. **Implementierungsaspekte**
  - **Fehlerbehandlung:**
    - Retry-Strategien
    - Fallbacks
    - Monitoring
  - **Sicherheit:**
    - Authentifizierung
    - Verschlüsselung
    - Autorisierung

## Entwurf verteilter Systeme

### 1. Systemanalyse

- Anforderungen identifizieren
- Verteilungsaspekte analysieren
- Konsistenzanforderungen definieren

### 2. Architekturentscheidungen

- Architekturstil wählen
- Kommunikationsmuster festlegen
- Fehlertoleranzstrategie definieren

### 3. Technologieauswahl

- Middleware evaluieren
- Protokolle bestimmen
- Werkzeuge auswählen

## Middleware-Technologien

Gängige Technologien für verteilte Systeme:

- **Message Broker:**
  - Apache Kafka
  - RabbitMQ
- **RPC Frameworks:**
  - gRPC
  - CORBA
- **Web Services:**
  - RESTful APIs
  - GraphQL

## Typische Fehlerquellen

### 1. Netzwerkfehler

- Verbindungsabbrüche
- Timeouts
- Partitionierung

### 2. Konsistenzprobleme

- Race Conditions
- Veraltete Daten
- Lost Updates

### 3. Skalierungsprobleme

- Lastverteilung
- Resource-Management
- Bottlenecks

### Lösungsstrategien:

- Circuit Breaker Pattern
- Retry mit Exponential Backoff
- Idempotente Operationen
- Optimistic Locking

## Persistenz

### Persistenz Grundlagen

Persistenz bezeichnet die dauerhafte Speicherung von Daten über das Programmende hinaus:

- Speicherung in Datenbankmanagementsystemen (DBMS)
- Haupttypen:
  - Relationale Datenbanksysteme (RDBMS)
  - NoSQL-Datenbanken (ohne fixes Schema)
- O/R-Mapping (Object Relational Mapping)
  - Abbildung zwischen Objekten und Datensätzen
  - Überwindung des Strukturbruchs (Impedance Mismatch)

### O/R-Mismatch

Der Strukturbruch zwischen objektorientierter und relationaler Welt:

- **Typen-Systeme:**
  - Unterschiedliche NULL-Behandlung
  - Datum/Zeit-Darstellung
- **Beziehungen:**
  - Richtung der Beziehungen
  - Mehrfachbeziehungen
  - Vererbung
- **Identität:**
  - OO: Implizite Objektidentität
  - DB: Explizite Identität (Primary Key)

**Prüfungsaufgabe: O/R-Mapping Analyse** **Szenario:** Ein Universitätssystem verwaltet Studenten, Kurse und Noten. Studenten können mehrere Kurse belegen, ein Kurs hat mehrere Studenten.

**Aufgabe:** Analysieren Sie die O/R-Mapping Herausforderungen dieser Domain.

**Lösung:**

- **Beziehungen:**
  - Many-to-Many zwischen Student und Kurs
  - Zusätzliche Attribute in der Beziehung (Noten)
  - Bidirektionale Navigation erforderlich
- **Vererbung:**
  - Person -> Student/Dozent
  - Verschiedene Mapping-Strategien möglich
- **Komplexe Daten:**
  - Adressdaten als Wertobjekte
  - Zeiträume für Kursbelegung

## JDBC - Java Database Connectivity

### JDBC Grundlagen

JDBC ist die standardisierte Schnittstelle für Datenbankzugriffe in Java:

- Seit JDK 1.1 (1997)
- Plattformunabhängig
- Datenbankunabhängig
- Aktuelle Version: 4.2

**JDBC Verwendung** Grundlegende Schritte für Datenbankzugriff:

1. JDBC-Treiber installieren und laden
2. Verbindung zur Datenbank aufbauen
3. SQL-Statements ausführen
4. Ergebnisse verarbeiten
5. Transaktion abschließen (Commit/Rollback)
6. Verbindung schließen

## Design Patterns für Persistenz

### Persistenz Design Patterns

Drei grundlegende Ansätze für die Persistenzschicht:

- **Active Record (Anti-Pattern):**
  - Entität verwaltet eigene Persistenz
  - Vermischung von Fachlichkeit und Technik
  - Schlechte Testbarkeit
- **Data Access Object (DAO):**
  - Kapselung des Datenbankzugriffs
  - Trennung von Fachlichkeit und Technik
  - Gute Testbarkeit durch Mocking
- **Repository (DDD):**
  - Abstraktionsschicht über Data-Mapper
  - Zentralisierung von Datenbankabfragen
  - Komplexere Implementierung

### Persistenzstrategie wählen 1. Anforderungen analysieren

- **Funktional:**
    - Datenmodell-Komplexität
    - Abfrageanforderungen
    - Transaktionsverhalten
  - **Nicht-funktional:**
    - Performance
    - Skalierbarkeit
    - Wartbarkeit
- 2. Technologien evaluieren**
- **JDBC:**
    - Direkte Kontrolle
    - Hohe Performance
    - Hoher Implementierungsaufwand
  - **JPA:**
    - Standardisiert
    - Produktiv
    - Lernkurve
  - **NoSQL:**
    - Flexibles Schema
    - Hohe Skalierbarkeit
    - Spezielle Anwendungsfälle

**Prüfungsaufgabe: Design Pattern Vergleich** **Aufgabe:** Vergleichen Sie Active Record, DAO und Repository Pattern.

**Analysematrix:**

- **Active Record:**
  - **Vorteile:**
    - \* Einfache Implementierung
    - \* Schnell zu entwickeln
  - **Nachteile:**
    - \* Keine Trennung der Belange
    - \* Schlechte Testbarkeit
    - \* Vermischung von Fachlogik und Persistenz
- **DAO:**
  - **Vorteile:**
    - \* Klare Trennung der Belange
    - \* Gute Testbarkeit
    - \* Austauschbare Implementierung
  - **Nachteile:**
    - \* Mehr Initialaufwand
    - \* Zusätzliche Abstraktionsebene
- **Repository:**
  - **Vorteile:**
    - \* Domänenorientierte Schnittstelle
    - \* Zentrale Abfragelogik
    - \* DDD-konform
  - **Nachteile:**
    - \* Komplexere Implementierung
    - \* Höhere Lernkurve

### DAO Implementation

Schritte zur Implementierung eines DAOs:

1. Interface definieren:
  - CRUD-Methoden (Create, Read, Update, Delete)
  - Spezifische Suchmethoden
2. Domänenklasse erstellen:
  - Nur fachliche Attribute
  - Keine Persistenzlogik
3. DAO-Implementierung:
  - Datenbankzugriff kapseln
  - O/R-Mapping implementieren
  - Transaktionshandling

## Java Persistence API (JPA)

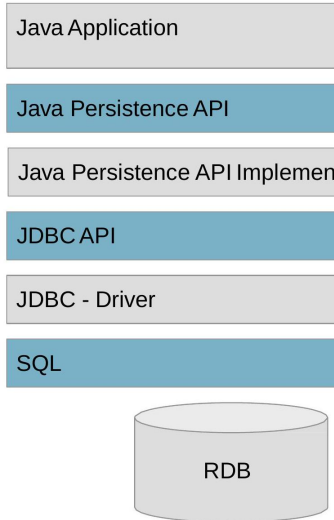
### JPA Grundkonzepte

JPA ist der Java-Standard für O/R-Mapping:

- **Entity-Klassen:**
  - Plain Old Java Objects (POJOs)
  - Annotation @Entity
  - Keine JPA-spezifischen Abhängigkeiten
- **Referenzen:**
  - Eager/Lazy Loading
  - Automatisches Nachladen
- **Provider:**
  - Hibernate
  - EclipseLink
  - OpenJPA

## JPA Technologie-Stack

- Java Application
- Java Persistence API
- JPA Provider (Hibernate, EclipseLink, etc.)
- JDBC Driver
- Relationale Datenbank



## JPA Entity Erstellung

1. Entity-Klasse definieren:
  - @Entity Annotation
  - ID-Feld mit @Id markieren
2. Beziehungen definieren:
  - @OneToMany, @ManyToOne etc.
  - Navigationsrichtung festlegen
3. Validierung hinzufügen:
  - @NotNull, @Size etc.
  - Geschäftsregeln

## JPA Entity Design 1. Grundstruktur

- **Basisanforderungen:**
  - Default Constructor
  - Serializable (optional)
  - Getter/Setter
- **Identifikation:**
  - Primary Key Strategie
  - Natural vs. Surrogate Key
- 2. **Beziehungen**
  - **Kardinalität:**
    - OneToOne
    - OneToMany/ManyToOne
    - ManyToMany
  - **Richtung:**
    - Unidirektional
    - Bidirektional
  - **Lifecycle:**
    - Cascade-Operationen
    - Orphan Removal
- 3. **Optimierungen**
  - **Lazy Loading:**
    - Fetch-Strategien
    - Join Fetching
  - **Caching:**
    - First-Level Cache
    - Second-Level Cache

## Repository Pattern

### Repository Pattern

Das Repository Pattern bietet eine zusätzliche Abstraktionsschicht über der Data-Mapper-Schicht:

- Zentralisierung von Datenbankabfragen
- Domänenorientierte Schnittstelle
- Unterstützung komplexer Abfragen
- Häufig in Kombination mit Spring Data

Spring Data unterstützt die automatische Generierung von Repository-Implementierungen basierend auf Methodennamen. Dies reduziert den Implementierungsaufwand erheblich.



## Framework Design

### Framework Grundlagen

Ein Framework ist ein Programmiergerüst mit folgenden Eigenschaften:

- Bietet wiederverwendbare Funktionalität
- Definiert Erweiterungs- und Anpassungspunkte
- Verwendet Design Patterns
- Enthält keinen applikationsspezifischen Code
- Gibt Rahmen für anwendungsspezifischen Code vor
- Klassen arbeiten eng zusammen (vs. reine Bibliothek)

### Framework Entwicklung

Die Entwicklung eines Frameworks erfordert:

- Höhere Zuverlässigkeit als normale Software
- Tiefergehende Analyse der Erweiterungspunkte
- Hoher Architektur- und Designaufwand
- Sorgfältige Planung der Schnittstellen

Kritische Betrachtung

Herausforderungen beim Framework-Einsatz:

- Frameworks tendieren zu wachsender Funktionalität
- Gefahr von inkonsistentem Design
- Funktionale Überschneidungen möglich
- Hoher Einarbeitungsaufwand
- Schwierige SScheidung nach Integration
- Trade-off zwischen Abhängigkeit und Nutzen

### Framework Design Principles 1. Abstraktionsebenen definieren

- **Core API:**
  - Zentrale Interfaces
  - Hauptfunktionalität
  - Erweiterungspunkte
- **Extensions:**
  - Plugin-Mechanismen
  - Callback-Interfaces
  - Event-Systeme
- **Implementierung:**
  - Standard-Implementierungen
  - Utility-Klassen
  - Helper-Funktionen
- 2. **Erweiterungsmechanismen**
- **Interface-basiert:**
  - Klare Verträge
  - Lose Kopplung
  - Einfache Erweiterung
- **Annotations:**
  - Deklarative Konfiguration
  - Metadaten-getrieben
  - Runtime-Processing
- **Composition:**
  - Plugin-System
  - Service-Loader
  - Dependency Injection

**Prüfungsaufgabe: Framework-Analyse** **Szenario:** Ein Framework für die Verarbeitung verschiedener Dokumentformate (PDF, DOC, TXT) soll entwickelt werden.

**Aufgabe:** Analysieren Sie die Design-Entscheidungen.

**Lösung:**

- **Erweiterungspunkte:**
  - Dokumenttyp-Erkennung
  - Parser für Formate
  - Konvertierungslogik
- **Design Patterns:**
  - Factory für Parser-Erzeugung
  - Strategy für Verarbeitungsalgorithmen
  - Template Method für Konvertierung
- **Schnittstellen:**
  - DocumentParser Interface
  - ConversionStrategy Interface
  - DocumentMetadata Klasse

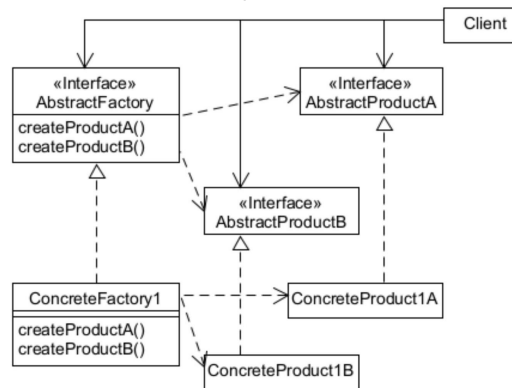
## Design Patterns in Frameworks

### Abstract Factory

**Problem:** Erzeugung verschiedener, zusammengehörender Objekte ohne Kenntnis konkreter Klassen

**Lösung:**

- AbstractFactory-Interface definieren
- Pro Produkt eine create-Methode
- Konkrete Factories implementieren Interface

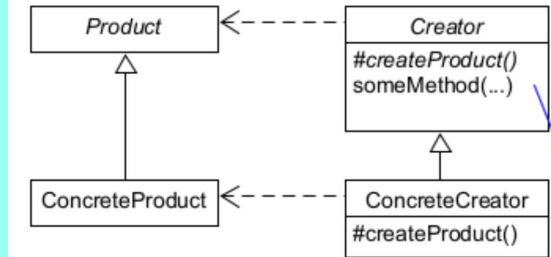


### Factory Method

**Problem:** Flexible Objekterzeugung in wiederverwendbarer Klasse

**Lösung:**

- Abstrakte Factory-Methode in Creator-Klasse
- Konkrete Subklassen überschreiben Methode
- Parallele Vererbungshierarchien

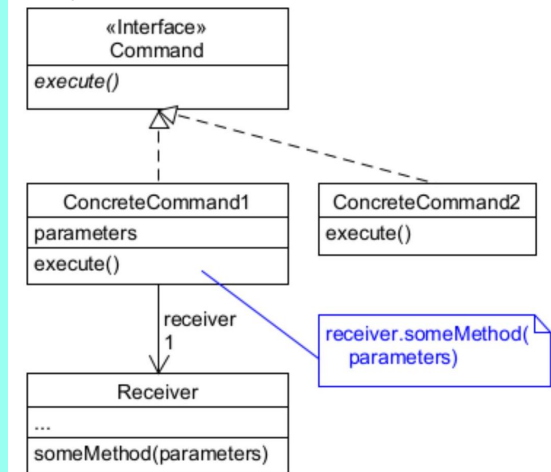


### Command

**Problem:** Aktionen für späteren Gebrauch speichern und verwalten

**Lösung:**

- Command-Interface definieren
- Konkrete Commands implementieren
- Parameter für Ausführung speichern
- Optional: Undo-Funktionalität

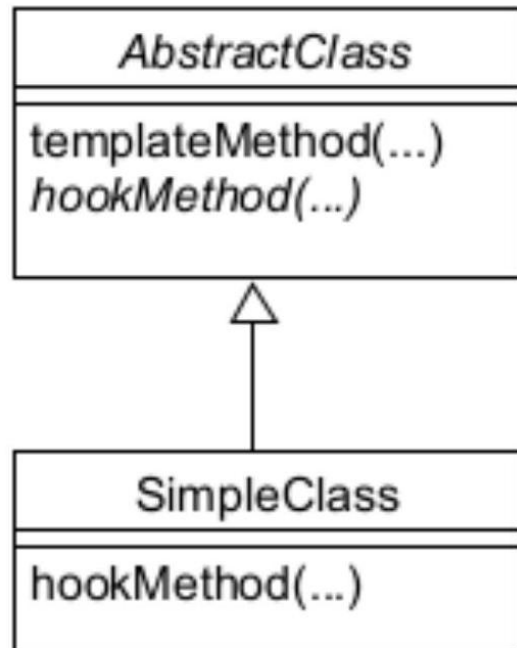


## Template Method

**Problem:** Algorithmus mit anpassbaren Teilschritten

**Lösung:**

- Template Method in abstrakter Klasse
- Hook-Methoden für variable Teile
- Hollywood Principle: "Don't call us, we'll call you"



## Moderne Framework Patterns

### Annotation-basierte Konfiguration

Moderne Frameworks nutzen Annotationen für:

- Dependency Injection
- Konfiguration
- Interface-Implementation
- Funktionalitätserweiterung

### Framework Integration

#### 1. Convention over Configuration

- Namenskonventionen einhalten
- Standard-Verhalten nutzen
- Nur Ausnahmen konfigurieren

#### 2. Dependency Injection

- Abhängigkeiten deklarieren
- Framework übernimmt Injection
- Constructor- oder Setter-Injection

#### 3. Interface-basierte Entwicklung

- Interfaces definieren
- Framework generiert Implementation
- Methodennamen als Spezifikation

Annotation-basierte Frameworks bieten:

- Geringere Kopplung zur Framework-API
- Deklarativen Programmierstil
- Reduzierte Boilerplate-Code
- Kann aber zu längeren Startzeiten führen

**Framework Design Pattern Anwendung** **Aufgabe:** Implementieren Sie ein Plugin-System mit verschiedenen Design Patterns.

**Analyse der Pattern-Kombination:**

- **Abstract Factory:**
  - Plugin-Familie erzeugen
  - Zusammengehörige Komponenten
  - Austauschbare Implementierungen
- **Template Method:**
  - Plugin-Lifecycle definieren
  - Standardablauf vorgeben
  - Erweiterungspunkte bieten
- **Command:**
  - Plugin-Aktionen kapseln
  - Asynchrone Ausführung
  - Undo-Funktionalität

### Framework Evaluation 1. Qualitätskriterien

- **Usability:**
  - Intuitive API
  - Gute Dokumentation
  - Beispiele/Templates
- **Flexibilität:**
  - Erweiterbarkeit
  - Konfigurierbarkeit
  - Modularität
- **Wartbarkeit:**
  - Klare Struktur
  - Testbarkeit
  - Versionierung

### 2. Risikobewertung

- **Technisch:**
  - Kompatibilität
  - Performance
  - Skalierbarkeit
- **Organisatorisch:**
  - Learning Curve
  - Support/Community
  - Zukunftssicherheit

**Typische Prüfungsaufgabe: Framework Migration** **Szenario:** Ein bestehendes System soll von einem proprietären Framework auf ein Standard-Framework migriert werden.

**Aufgabenstellung:**

- Analysieren Sie die Herausforderungen
- Entwickeln Sie eine Migrationsstrategie
- Bewerten Sie Risiken

**Lösungsansatz:**

- **Analyse:**
  - Framework-Abhängigkeiten identifizieren
  - Geschäftskritische Funktionen isolieren
  - Testabdeckung prüfen
- **Strategie:**
  - Adapter für Framework-Bridging
  - Schrittweise Migration
  - Parallelbetrieb ermöglichen
- **Risikominimierung:**
  - Automated Testing
  - Feature Toggles
  - Rollback-Möglichkeit