

## Einführung und Überblick

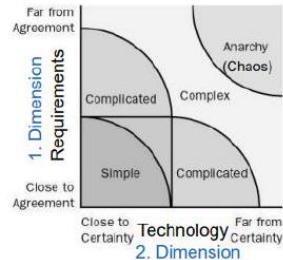
## Software Engineering

- **Disziplinen:**  
Anforderungen, Architektur, Implementierung, Test und Wartung
- **Ziel:**  
Strukturierte Prozesse für Qualität, Risiko- & Fehlerminimierung

## Softwareentwicklungsprozesse

## Dimensionen Software-Entwicklungs-Probleme

- Requirements (Bekannt - Unbekannt)
- Technology (Bekannt - Unbekannt)
- Skills/Experience (Vorhanden - Nicht vorhanden)



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



## Kernprozesse

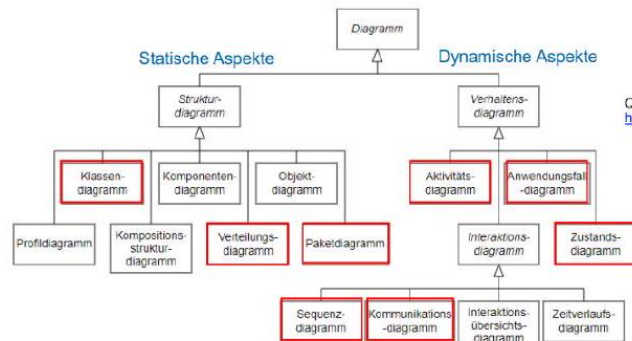
- Anforderungserhebung
- Systemdesign/technische Konzeption
- Implementierung
- Softwaretest
- Softwareeinführung
- Wartung/Pflege

## Unterstützungsprozesse

- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

## 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.



## Agile Softwareentwicklungsprozesse 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

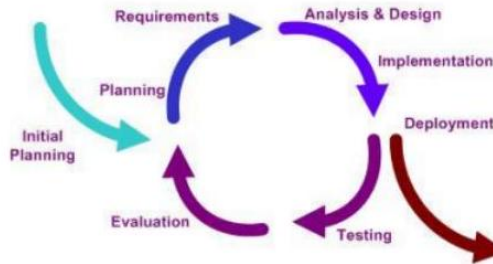
## Wasserfallmodell

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

## 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

## 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).

## 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 Engineerings

## Modellierung

## 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, 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

## 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, programminglanguage)

## Basically

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

**Usability und User Experience** drei Säulen der Benutzererfahrung:

- **Usability (Gebrauchstauglichkeit):** Grundlegende Nutzbarkeit des Systems
- **User Experience:** Usability + Desirability (Attraktivität)
- **Customer Experience:** UX + Brand Experience (Markenwahrnehmung)



Source: User Experience 2008, nGroup Conference Amsterdam

**Wichtige Aspekte:** Benutzer und seine Ziele/Aufgaben, Kontext der Nutzung, Softwaresystem (inkl. UI)

### Usability-Dimensionen nach ISO 9241

- **Effektivität:**
  - Der Benutzer kann alle Aufgaben vollständig erfüllen
  - Gewünschte Genauigkeit wird erreicht
  - Ziele werden im vorgegebenen Kontext erreicht
- **Effizienz:**
  - Minimaler Aufwand für:
    - Mentale Belastung
    - Physische Anstrengung
    - Zeitlicher Aufwand
    - Ressourceneinsatz
- **Zufriedenheit:**
  - Minimum: Keine Verärgerung
  - Standard: Zufriedenheit
  - Optimal: Begeisterung
  - Subjektive Benutzererfahrung

### Usability-Evaluation durchführen

- Vorbereitung**
  - Testziele definieren
  - Testpersonen auswählen
  - Testaufgaben erstellen
- Durchführung**
  - Beobachtung der Nutzer
  - Protokollierung von Problemen
  - Zeitmessung der Aufgaben
- Auswertung**
  - Probleme klassifizieren
  - Schweregrad bestimmen
  - Verbesserungen vorschlagen
- Dokumentation**
  - Ergebnisse zusammenfassen
  - Empfehlungen formulieren
  - Maßnahmen priorisieren

### ISO 9241-110: Usability-Anforderungen

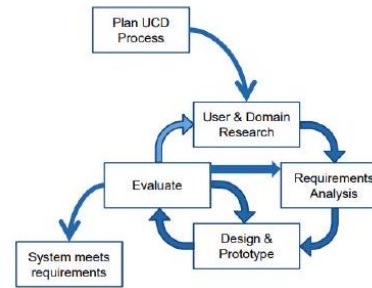
- **Aufgabenangemessenheit:**
  - Funktionalität unterstützt Arbeitsaufgaben
  - Keine unnötige Komplexität
- **Selbstbeschreibungsfähigkeit:**
  - Verständliche Benutzerführung
  - Klare Statusanzeigen
- **Steuerbarkeit:**
  - Benutzer kontrolliert Ablauf
  - Geschwindigkeit anpassbar
- **Erwartungskonformität:**
  - Konsistentes Verhalten
  - Bekannte Konventionen
- **Fehlertoleranz:**
  - Fehler vermeiden
  - Fehlerkorrektur ermöglichen
- **Individualisierbarkeit:**
  - Anpassung an Benutzergruppen
  - Flexible Nutzung
- **Lernförderlichkeit:**
  - Unterstützung beim Lernen

### UCD (User-Centered Design)

Ein iterativer Prozess zur nutzerzentrierten Entwicklung, der die Bedürfnisse, Wünsche und Einschränkungen der Benutzer in jeder Phase des Design-Prozesses berücksichtigt.

**Hauptziele:**

- Benutzerfreundlichkeit
- Effektive Nutzung
- Hohe Akzeptanz

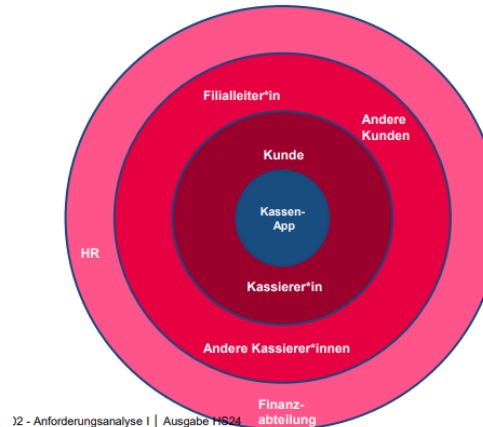


### Wichtige Artefakte

- 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 Stakeholder im Umfeld der Problemdomäne.



12 - Anforderungsanalyse I | Ausgabe HS24

### UCD Artefakte erstellen

- Personas**
  - Daten aus User Research sammeln
  - Gemeinsame Merkmale identifizieren
  - Repräsentative Person definieren
  - Details ausarbeiten:
    - Demografische Daten
    - Ziele und Motivation
    - Fähigkeiten/Kenntnisse
    - Frustrationspunkte
- Usage-Szenarien**
  - Kontext beschreiben
  - Akteure identifizieren
  - Ablauf definieren
  - Probleme/Lösungen darstellen
- Mentales Modell**
  - Nutzerverständnis dokumentieren
  - Konzepte und Beziehungen visualisieren
  - Mit Fachmodell abgleichen

### UCD Prozess-Phasen

- User & Domain Research** (see KR)
- Requirements Analysis** (see KR)
- Design & Prototype**
  - Interaktionskonzept entwickeln
  - Wireframes erstellen
  - Prototypen bauen
  - Design iterativ verbessern
- Evaluate**
  - Mit Benutzern testen
  - Feedback sammeln
  - Probleme identifizieren
  - Verbesserungen einarbeiten

### User & Domain Research

- Zielgruppe identifizieren**
  - Wer sind die Benutzer?
  - Was sind ihre Aufgaben/Ziele?
  - Wie sieht ihre Arbeitsumgebung aus?
  - Welche Sprache/Begriffe verwenden sie?
- Daten sammeln durch**
  - Contextual Inquiry
  - Interviews
  - Beobachtung
  - Fokusgruppen
  - Nutzungsauswertung
- Ergebnisse dokumentieren in**
  - Personas
  - Usage-Szenarien
  - Mentales Modell

### Requirements Analysis

- Benutzeranforderungen ableiten
- Kontextszenarien erstellen
- UI-Skizzen entwickeln
- Use Cases definieren

### Stakeholder identifizieren

- Benutzer
- Auftraggeber
- Weitere Interessengruppen

### Anforderungsquellen analysieren

- Interviews und Workshops
- Existierende Dokumente
- Beobachtung der Arbeitsabläufe

### Anforderungen dokumentieren

- Funktionale Anforderungen (Use Cases)
- Nicht-funktionale Anforderungen
- Randbedingungen

### Anforderungen validieren

- Review mit Stakeholdern
- Priorisierung
- Machbarkeitsanalyse

### Usage-Szenario: Online-Banking

**Kontext:** Sarah möchte eine Überweisung tätigen

**Aktuelles Szenario:** Sarah loggt sich in ihr Online-Banking ein. Sie sucht nach der letzten Überweisung an ihren Vermieter, um die Kontodetails zu finden. Nach mehreren Klicks findet sie die Information und kopiert die IBAN. Sie öffnet das Überweisungsformular und fügt die Daten ein. Beim Absenden erscheint eine Fehlermeldung, weil sie vergessen hat, den Verwendungszweck einzutragen.

**Probleme:**

- Umständliche Suche nach Kontodetails
- Fehleranfällige manuelle Dateneingabe
- Späte Validierung der Eingaben

**Verbessertes Szenario:** Sarah wählt aus einer Liste ihrer häufigen Empfänger ihren Vermieter aus. Das System füllt automatisch alle bekannten Daten ein. Fehlende Pflichtfelder sind deutlich markiert. Sarah ergänzt den Verwendungszweck und sendet die Überweisung ab.

Weitere Beispiele z.B. Persona erstellen auf nächster Seite

## Persona erstellen

**Aufgabe:** Erstellen Sie eine Persona für ein Online-Banking-System.

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

- **Hintergrund:**
  - Arbeitet Vollzeit in IT-Firma
  - Technik-affin, aber keine Expertin
  - Nutzt Smartphone für die meisten Aufgaben
- **Ziele:**
  - Schnelle Überweisungen zwischen Konten
  - Überblick über Ausgaben
  - Sichere Authentifizierung
- **Frustrationen:**
  - Komplexe Menüführung
  - Lange Ladezeiten
  - Mehrfache Login-Prozesse

## Persona für E-Learning-System

**Thomas Weber, 19, Informatik-Student**

**Hintergrund:**

- Erstsemester-Student
- Arbeitet nebenbei 10h/Woche
- Pendelt zur Universität (1h pro Weg)

**Technische Fähigkeiten:**

- Versiert im Umgang mit Computern
- Nutzt hauptsächlich Smartphone für Online-Aktivitäten
- Kennt gängige Learning-Management-Systeme

**Ziele:**

- Effizientes Lernen trotz Zeitdruck
- Flexible Zugriffsmöglichkeiten auf Lernmaterialien
- Gute Prüfungsvorbereitung

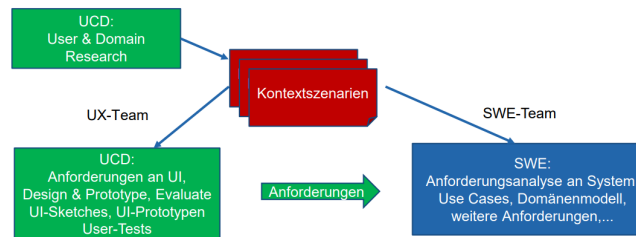
**Frustrationen:**

- Unübersichtliche Kursstrukturen
- Fehlende Mobile-Optimierung
- Schwierige Navigation zwischen Materialien

## Requirements Engineering

### Requirements (Anforderungen)

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



**Charakteristiken:**

- Können explizit oder implizit sein
- Sind fast nie im Vorneherein vollständig bekannt
- Müssen mit allen Stakeholdern erarbeitet werden
- Entwickeln sich während des Projekts
- Müssen verifizierbar und messbar sein

**Herkunft:**

- Benutzer (Ziele, Bedürfnisse, Kontext)
- Weitere Stakeholder (Management, IT, etc.)
- Regulatorien, Gesetze, Normen

### Arten von Anforderungen

**Funktionale Anforderungen:**

- Beschreiben, WAS das System tun soll
- Werden in Use Cases dokumentiert
- Müssen konkret und testbar sein

**Nicht-funktionale Anforderungen (ISO 25010):**

- Performance Efficiency
  - Time Behaviour
  - Resource Utilization
  - Capacity
- Compatibility
  - Co-existence
  - Interoperability
- Usability (siehe oben)
- Reliability
  - Maturity
  - Availability
  - Fault Tolerance
  - Recoverability
- Security
- Maintainability
- Portability

**Randbedingungen:**

- Technische Einschränkungen
- Rechtliche Vorgaben
- Budgetäre Grenzen
- Zeitliche Limitationen

## Anforderungen erheben und dokumentieren

### 1. Erhebung

1. Stakeholder identifizieren
2. Informationsquellen erschließen
3. Erhebungstechniken anwenden:
  - Interviews
  - Workshops
  - Beobachtung
  - Dokumentenanalyse

### 2. Analyse und Dokumentation

1. Anforderungen klassifizieren
2. Abhängigkeiten identifizieren
3. Konflikte auflösen
4. Priorisieren

### 3. Validierung

1. Vollständigkeit prüfen
2. Konsistenz sicherstellen
3. Machbarkeit bewerten
4. Mit Stakeholdern abstimmen

### Anforderungsanalyse: Onlineshop

**Ausgangssituation:** Ein traditioneller Buchladen möchte einen Onlineshop entwickeln.

**Funktionale Anforderungen:**

- Produktkatalog durchsuchen
- Warenkorb verwalten
- Bestellung aufgeben
- Kundenkonto verwalten

**Nicht-funktionale Anforderungen:**

- Performance:
  - Seitenaufbau < 2 Sekunden
  - Suche < 1 Sekunde
- Sicherheit:
  - HTTPS-Verschlüsselung
  - Zwei-Faktor-Authentifizierung
- Usability:
  - Responsive Design
  - Max. 3 Klicks zur Bestellung

**Randbedingungen:**

- DSGVO-Konformität
- Integration mit bestehendem ERP
- Budget: 100.000 EUR
- Launch in 6 Monaten

## Use Cases

### Use Case (Anwendungsfall)

Ein Use Case beschreibt eine konkrete Interaktion zwischen Akteur und System mit folgenden Eigenschaften:

#### Grundprinzipien:

- Aus Sicht des Akteurs beschrieben
- Aktiv formuliert (Verb + Objekt)
- Konkreter Nutzen für Akteur
- Mehr als eine einzelne Interaktion
- Essentieller Stil (Logik statt Implementierung)

#### Qualitätskriterien:

- Boss-Test: Sinnvolle Arbeitseinheit
- EBP-Test: Elementary Business Process
- Size-Test: Mehrere Interaktionen

### Use Case Erstellung

Schritte zur Erstellung eines vollständigen Use Cases:

1. **Identifikation:** siehe **Use Case Identifikation**
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

### Use Case Identifikation

1. **Systemgrenzen definieren**
  - Was gehört zum System?
  - Was ist externe Umgebung?
2. **Akteure identifizieren**
3. **Ziele ermitteln**
  - Geschäftsziele
  - Benutzerziele
  - Systemziele

### Akteure in Use Cases

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

### Use Case Beziehungen

#### Include-Beziehung:

- Ein UC schließt einen anderen UC ein
- Wiederverwendung von Funktionalität
- Obligatorische Beziehung

#### Extend-Beziehung:

- Optionale Erweiterung eines UC
- Unter bestimmten Bedingungen
- Ursprünglicher UC bleibt unverändert

#### Generalisierung:

- Spezialisierung von Akteuren/UCs
- Vererbung von Eigenschaften
- ist-ein-Beziehung

### Use Case Granularität

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

### Fully-dressed Use Case erstellen

1. **Grundinformationen**
  - Aussagekräftiger Name (aktiv)
  - Umfang (Scope)
  - Ebene (Level)
  - Primärakteur
2. **Stakeholder und Interessen**
  - Alle beteiligten Parteien
  - Deren spezifische Interessen
3. **Vor- und Nachbedingungen**
  - Was muss vorher erfüllt sein?
  - Was ist nachher garantiert?
4. **Standardablauf**
  - Nummerierte Schritte
  - Akteur-System-Interaktion
  - Klare Erfolgskriterien
5. **Erweiterungen**
  - Alternative Abläufe
  - Fehlerszenarien
  - Verzweigungen

### 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.

**Casual Use Case UC: Verkauf abwickeln** Der Umfang des Use Cases ist das Kassensystem. Der Primärakteur ist der Kassier. Der Stakeholder ist der Kunde, der eine schnelle Abwicklung wünscht, und das Geschäft, das eine korrekte Abrechnung benötigt. Die Vorbedingung ist, dass die Kasse geöffnet ist.

Der Standardablauf ist wie folgt: Kassier startet neuen Verkauf und System initialisiert neue Transaktion. Kassier erfasst Produkte und System zeigt Zwischensumme. Kassier schliesst Verkauf ab und System zeigt Gesamtbetrag. Kunde bezahlt und 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

**Fully-dressed Use Case 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

### 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."

### Prüfungsaufgabe: Use Case Analyse

**Aufgabe:** Analysieren Sie folgenden Use Case und verbessern Sie ihn.

**Ursprünglicher Use Case:** "Der User loggt sich ein. Das System überprüft seine Credentials in der Datenbank. Bei erfolgreicher Validierung wird die Startseite angezeigt. Der User klickt auf 'Profil bearbeiten' und das System speichert die Änderungen in der Datenbank."

#### Probleme:

- Technische Implementierungsdetails
- Fehlende Akteurperspektive
- Keine Alternativen/Fehlerbehandlung
- Unklarer Nutzen/Ziel

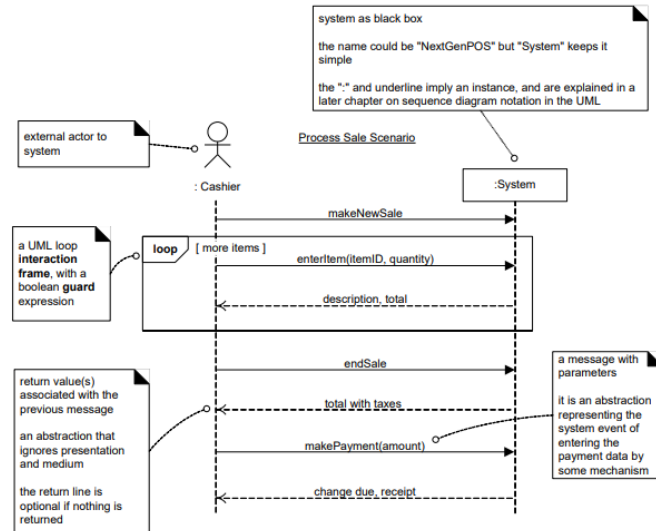
**Verbesserter Use Case:** "Profilinformationen aktualisieren"

- **Primärakteur:** Registrierter Benutzer
- **Vorbedingung:** Benutzer ist authentifiziert
- **Standardablauf:**
  1. Benutzer wählt Profilbearbeitung
  2. System zeigt aktuelle Profildaten
  3. Benutzer ändert gewünschte Informationen
  4. System prüft Änderungen
  5. System bestätigt erfolgreiche Aktualisierung
- **Erweiterungen:**
  - 4a: Ungültige Eingaben
  - 4b: Verbindungsfehler



## Systemsequenzdiagramm (SSD)

Formalisierte Darstellung der System-Interaktionen:



## System Sequence Diagram

Ein SSD visualisiert die Interaktion zwischen Akteur und System auf einer höheren Abstraktionsebene:

## Hauptmerkmale:

- Zeigt Input/Output-Events
- Identifiziert Systemoperationen
- Bildet Basis für API-Design
- Abstrahiert von UI-Details

## Notationselemente:

- Akteur und System als Lebenslinien
- Methodenaufrufe als durchgezogene Pfeile
- Rückgabewerte als gestrichelte Pfeile
- Parameter für benötigte Informationen

## System Sequence Diagram erstellen

## 1. Vorbereitung

- Use Case als Grundlage wählen
- Standardablauf identifizieren
- Akteur und System festlegen

## 2. Methodenaufrufe definieren

- Aussagekräftige Namen wählen
- Notwendige Parameter bestimmen
- Rückgabewerte festlegen

## 3. Zeitliche Abfolge

- Sequenz der Aufrufe modellieren
- Abhängigkeiten beachten
- Kontrollstrukturen einbauen (alt, loop, etc.)

## 4. Externe Systeme

- Bei Bedarf weitere Akteure einbinden
- Schnittstellen definieren
- Kommunikationsfluss darstellen

## Systemoperationen definieren

## Namenskonventionen:

- Verben für Aktionen
- Substantive für Entitäten
- Präzise, aber nicht technisch

## Parameter:

- Nur notwendige Information
- Domänenorientierte Typen
- Sinnvolle Standardwerte

## Rückgabewerte:

- Eindeutige Bestätigungen
- Relevante Geschäftsobjekte
- Fehlerindikationen

## Beispiele guter Operationen:

```

1 // Gut - klar und domänenorientiert
2 createOrder(customer: CustomerId): OrderId
3 addOrderItem(orderId: OrderId,
4               product: ProductId,
5               quantity: int)
6
7 // Schlecht - zu technisch/implementierungsnah
8 insertIntoOrderTable(customerData: Map)
9 updateOrderItemList(items: ArrayList)

```

## Contracts für Systemoperationen

Ein Contract definiert die Vor- und Nachbedingungen einer Systemoperation:

## 1. Struktur

- Name und Parameter
- Querverweis zum Use Case
- Vorbedingungen
- Nachbedingungen

## 2. Vorbedingungen

- Systemzustand vor Aufruf
- Notwendige Initialisierungen
- Gültige Parameter

## 3. Nachbedingungen

- Erstellte/gelöschte Instanzen
- Geänderte Attribute
- Neue/gelöschte Assoziationen

## Contract für enterItem()

**Operation:** enterItem(itemId: ItemID, quantity: int)

**Querverweis:** UC "Process Sale"

## Vorbedingungen:

- Verkauf ist gestartet
- ItemID existiert im System

## Nachbedingungen:

- SalesLineItem-Instanz wurde erstellt
- Verknüpfung mit aktueller Sale-Instanz
- quantity wurde gesetzt
- Verknüpfung mit ProductDescription

## 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()

Sequenzdiagramm: **TO BE ADDED**

### SSD: Online-Banking Überweisung

**Use Case:** Überweisung durchführen

**Systemoperationen:**

```
1 // Kontostand pruefen
2 checkBalance(): Money
3
4 // Ueberweisung initiieren
5 initiateTransfer(recipient: String,
6                 iban: String,
7                 amount: Money,
8                 purpose: String): TransferId
9
10 // TAN anfordern
11 requestTAN(transferId: TransferId): void
12
13 // Ueberweisung bestaetigen
14 confirmTransfer(transferId: TransferId,
15                tan: String): Boolean
```

**Wichtige Aspekte:**

- Validierung vor Ausführung
- Zweistufige Bestätigung
- Klare Rückmeldungen
- Fehlerbehandlung

**Sequenzdiagramm:** TO BE ADDED

### SSD: Typische Prüfungsaufgabe

**Aufgabe:** Erstellen Sie ein SSD für den Use Case "Produkt bestellen" in einem Webshop.

**Analyse:**

- Identifiziere Hauptaktionen:
  - Warenkorb verwalten
  - Bestellung aufgeben
  - Zahlung durchführen
- Definiere Systemoperationen:
  - addToCart(productId, quantity)
  - showCart(): CartContents
  - checkout(shippingAddress, paymentMethod)
  - confirmOrder(): OrderId
- Berücksichtige Rückgabewerte:
  - Bestätigungen
  - Zwischensummen
  - Fehlermeldungen

**Sequenzdiagramm:** TO BE ADDED

### SSD: Integration mit externen Systemen

**Use Case:** Kreditkartenzahlung durchführen

**Beteiligte Systeme:**

- Verkaufssystem (SuD)
- Kreditkarten-Autorisierungssystem
- Buchhaltungssystem

**Systemoperationen:**

```
1 // Request credit card approval
2 requestApproval(cardNum: String,
3                 expiryDate: Date,
4                 amount: Money): Boolean
5
6 // Post transaction to accounting
7 postTransaction(transactionData:
8                 TransactionData)
```

**Wichtige Aspekte:**

- Asynchrone Kommunikation
- Fehlerbehandlung über mehrere Systeme
- Transaktionsmanagement
- Logging und Nachvollziehbarkeit

**Sequenzdiagramm:** TO BE ADDED

## 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

images/domÄdnenmodell.png

- Optional: Aggregationen/Kompositionen

images/aggregation\_komposition.png

### 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

### Domänenmodell Erstellung

#### Schritt 1: Konzepte identifizieren

- Substantive aus Anforderungen extrahieren
- Kategorien prüfen:
  - Physische Objekte (Produkte, Geräte)
  - Kataloge und Listen
  - Container (Warenkorb, Lager)
  - Externe Systeme
  - Rollen von Personen
  - Artefakte (Pläne, Dokumente, Verträge)
  - Zahlungsinstrumente (Transaktionen)
  - **Wichtig:** Keine Softwareklassen modellieren!

- Irrelevante Konzepte ausschließen
- Synonyme vereinheitlichen

#### Schritt 2: Attribute definieren

- Nur wichtige/zentrale Attribute
- Typische Kategorien:
  - Transaktionsdaten
  - Teil-Ganzes Beziehungen
  - Beschreibungen
  - Verwendungszwecke
- **Wichtig:** Beziehungen als Assoziationen, nicht als Attribute!
- Keine technischen IDs
- Keine abgeleiteten Werte

#### Schritt 3: Beziehungen modellieren

- Assoziationen zwischen Konzepten identifizieren
- Multiplizitäten festlegen
- Art der Beziehung bestimmen
- Richtung der Assoziation falls nötig
- Rollen an Assoziationsenden benennen

### 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 (Beschreibungs-kategorie), 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 Überblick** Details siehe nächste Seite
- Bewährte Strukturen für wiederkehrende Modellierungssituationen:
- **Beschreibungsklassen:** Trennung von Typ und Instanz
  - **Generalisierung:** ist-ein-Beziehungen
  - **Komposition:** Starke Teil-Ganzes Beziehung
  - **Zustände:** Eigene Zustandshierarchie
  - **Rollen:** Verschiedene Funktionen eines Konzepts
  - **Assoziationsklasse:** Attribute einer Beziehung

- Musterauswahl und Kombination**
- Systematisches Vorgehen bei der Anwendung von Analysemustern:
1. **Analyse der Situation**
    - Konzepte und Beziehungen identifizieren, Attribute zuordnen
    - Probleme im einfachen Modell erkennen (Bei Kombination)

2. **Passende 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
3. **Musterauswahl**
  - Vor- und Nachteile abwägen
  - Komplexität vs. Nutzen bewerten
4. **Muster anwenden**
  - Struktur des Musters übernehmen, an Kontext anpassen
  - Konsistenz und fachliche Korrektheit sicherstellen
5. **Muster kombinieren**
  - An Kontext anpassen und mit bestehenden Elementen verbinden
  - Überschneidungen identifizieren und Konflikte auflösen
  - Gesamtmodell harmonisieren
  - Konsistenz und fachliche Korrektheit sicherstellen

**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 oder abstrakt
  - Fachliche Begriffe verwenden
- **Assoziationen statt Attribute**
  - Beziehungen als Assoziationen modellieren
  - Keine Objekt-IDs als Attribute

**Typische Modellierungsfehler**

**Fehler 1: Technische statt fachliche Klassen**

- **Falsch:** CustomerManager, OrderController, DatabaseHandler
- **Richtig:** Kunde, Bestellung, Produkt

**Fehler 2: IDs als Attribute statt Assoziationen**

- **Falsch:** customerId, orderId: Integer
- **Richtig:** Direkte Assoziation zwischen Kunde und Bestellung

**Fehler 3: Implementierungsdetails**

- **Falsch:** saveToDatabase(), validateInput(), createPDF()
- **Richtig:** Keine Operationen im Domänenmodell

**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 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

**Vollständige Beispiele Domänenmodell**

**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

**1. Beschreibungsklassen**

Trennt die Beschreibung eines Typs von seinen konkreten Instanzen.

**Anwendung:**

- Bei mehreren gleichartigen Objekten
- Gemeinsame unveränderliche Eigenschaften
- Vermeidung von Redundanz

**Beispielstruktur:**

- ProductDescription (Typ)
  - name, price, description
- Product (Instanz)
  - serialNumber, location

**Beschreibungsklassen in der Praxis**

**Szenario:** Bibliothekssystem

**Problem:** Ein Buch kann mehrere physische Exemplare haben, die alle dieselben Grunddaten (Titel, Autor, ISBN) aber unterschiedliche Zustände (ausgeliehen, verfügbar) haben.

**Lösung:**

- **Book** (Beschreibungsklasse)
  - title, author, isbn, publisher
- **BookCopy** (Instanzklasse)
  - inventoryNumber, status, location
- Assoziation: BookCopy "beschrieben durch"Book

**2. Generalisierung/Spezialisierung**

Modelliert ist-ein-Beziehungen zwischen Konzepten.

**Regeln:**

- 100% Regel: Jede Instanz der Spezialisierung ist auch Instanz der Generalisierung
- Gemeinsame Eigenschaften in Basisklasse
- Spezifische Eigenschaften in Unterklassen

**Beispiele:**

- Person → Student, Dozent
- Zahlung → Barzahlung, Kreditkartenzahlung
- Dokument → Rechnung, Lieferschein



### Generalisierung im Online-Shop

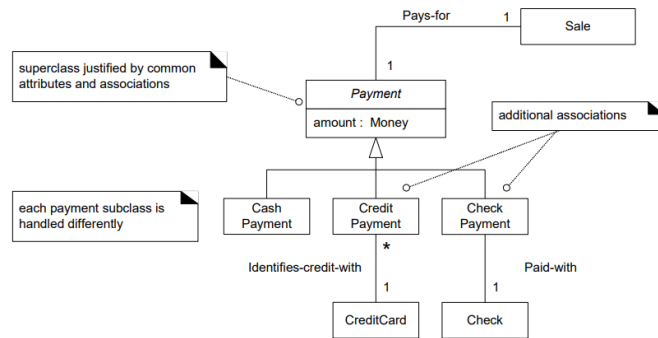
**Szenario:** Verschiedene Zahlungsarten

**Struktur:**

- **Payment** (abstrakt)
  - amount, date, status
- **CashPayment**
  - receivedAmount, changeAmount
- **CreditCardPayment**
  - cardType, authorizationCode

**Begründung:**

- Gemeinsame Attribute in Payment
- Spezifische Attribute in Unterklassen
- Jede Zahlung ist genau ein Typ



### 3. Komposition

Modelliert eine starke Teil-Ganzes Beziehung mit Existenzabhängigkeit.

**Eigenschaften:**

- Teile können nicht ohne Ganzes existieren
- Teil gehört zu genau einem Ganzen
- Löschen des Ganzen löscht alle Teile

**Notation:**

- Ausgefüllte Raute am "GanzesEnde"
- Multiplizität am "TeilEnde"

### Komposition im Bestellsystem

**Szenario:** Bestellung mit Bestellpositionen

**Struktur:**

- **Order**
  - orderDate, status
- **OrderItem**
  - quantity, price
- Komposition von Order zu OrderItem (1 zu \*)

**Begründung:**

- OrderItems existieren nur im Kontext einer Order
- Löschen der Order löscht alle OrderItems
- Ein OrderItem gehört zu genau einer Order

### 4. Zustandsmodellierung

Modelliert Zustände als eigene Konzeptionshierarchie statt als Attribut.

**Vorteile:**

- Klare Strukturierung der Zustände
- Erweiterbarkeit durch neue Zustandsklassen
- Vermeidung von if/else Kaskaden
- Zustandsspezifisches Verhalten möglich

**Beispielstruktur:**

- **OrderState** (abstrakt)
  - New, InProgress, Completed
- **Order** ist inOrderState

### Zustandsmodellierung: Ticketsystem

**Szenario:** Support-Tickets mit verschiedenen Status

**Falsche Modellierung:**

```
1 class Ticket {
2     enum Status {NEW, OPEN, IN_PROGRESS,
3                 RESOLVED, CLOSED}
4     private Status status;
```

**Bessere Modellierung:**

- **TicketState** (abstrakt)
  - timestamp, changedBy
- Konkrete Zustände:
  - NewState: assignedTo
  - OpenState: priority
  - InProgressState: estimatedCompletion
  - ResolvedState: solution
  - ClosedState: closureReason

### 5. Rollen

Modelliert verschiedene Funktionen eines Konzepts.

**Varianten:**

- Rollen als eigene Konzepte
- Rollen als Assoziationsenden
- Rollen durch Generalisierung

**Anwendung:**

- Bei verschiedenen Verantwortlichkeiten
- Wenn Rollen wechseln können
- Bei unterschiedlichen Beziehungen

### Rollenmuster: Universitätssystem

**Szenario:** Person kann gleichzeitig Student und Tutor sein

**Variante 1: Rollen als Konzepte**

- **Person**
  - name, birthDate, address
- **StudentRole**
  - matriculationNumber, program
- **TutorRole**
  - department, hourlyRate

**Variante 2: Generalisierung**

- Person als Basisklasse
- Student und Tutor als Spezialisierungen
- Problem: Mehrfachrollen schwierig

### 6. Assoziationsklassen

Modelliert Attribute einer Beziehung zwischen Konzepten.

**Einsatz wenn:**

- Attribute zur Beziehung gehören
- Beziehung eigene Identität hat
- Mehrere Beziehungen möglich sind

**Notation:**

- Gestrichelte Linie zur Assoziation
- Klasse enthält beziehungspezifische Attribute

### Assoziationsklasse: Kursbuchungssystem

**Szenario:** Studenten können sich für Kurse einschreiben

**Struktur:**

- **Student** und **Course** als Hauptkonzepte
- **Enrollment** als Assoziationsklasse:
  - enrollmentDate
  - grade
  - attendance
  - status

**Begründung:**

- Noten gehören zur Einschreibung
- Student kann mehrere Kurse belegen
- Kurs hat mehrere Studenten
- Einschreibungsdaten sind beziehungspezifisch

### 1. Beschreibungsklassen

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

### 2. Generalisierung/Spezialisierung

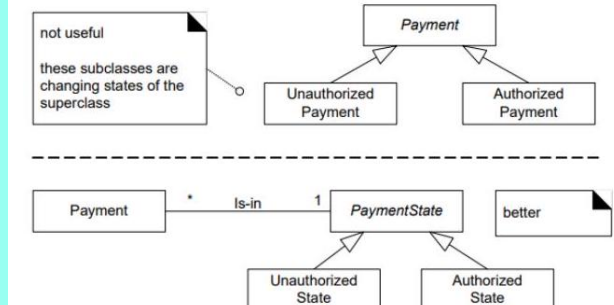
- 100% Regel: Alle Instanzen der Spezialisierung sind auch Instanzen der Generalisierung
- IS-A-Beziehung
- Gemeinsame Attribute/Assoziationen als Grund für Generalisierung

### 3. Komposition

- Starke Teil-Ganzes Beziehung
- Existenzabhängigkeit der Teile

### 4. Zustandsmodellierung

- Zustände als eigene Hierarchie
- Vermeidet problematische Vererbung

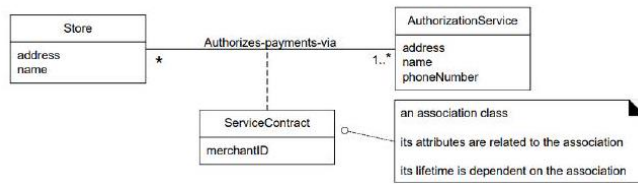


## 5. Rollen

- Unterschiedliche Rollen eines Konzepts
- Als eigene Konzepte oder Assoziationen

## 6. Assoziationsklassen

- Attribute einer Beziehung
- Eigene Klasse für die Assoziation



## 7. Wertobjekte

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

## Architekturmuster (Patterns)

### Architekturmuster Übersicht

Grundlegende Architekturmuster für verteilte Systeme:

- **Layered Pattern:** Schichtenarchitektur
- **Client-Server Pattern:** Verteilte Dienste
- **Master-Slave Pattern:** Verteilte Verarbeitung
- **Pipe-Filter Pattern:** Datenstromverarbeitung
- **Broker Pattern:** Vermittler zwischen Endpunkten
- **Event-Bus Pattern:** Nachrichtenverteilung
- **MVC Pattern:** Trennung von Daten und Darstellung

#### Layered Pattern

Struktur:

```
1 // Presentation Layer
2 public class CustomerUI {
3     private CustomerService service;
4     public void showCustomerDetails(int id) {
5         Customer customer = service.getCustomer(id);
6         // display logic
7     }
8 }
9
10 // Business Layer
11 public class CustomerService {
12     private CustomerRepository repository;
13     public Customer getCustomer(int id) {
14         return repository.findById(id);
15     }
16 }
17
18 // Data Layer
19 public class CustomerRepository {
20     public Customer findById(int id) {
21         // database access
22         return customer;
23     }
24 }
```

Vorteile:

- Klare Trennung der Verantwortlichkeiten
- Austauschbarkeit einzelner Schichten
- Einfachere Wartung und Tests

### Clean Architecture

Prinzipien nach Robert C. Martin:

- **Unabhängigkeit von Frameworks**
  - Framework als Tool, nicht als Einschränkung
  - Geschäftslogik unabhängig von UI/DB
- **Testbarkeit**
  - Business Rules ohne externe Systeme testbar
  - Keine DB/UI für Tests notwendig
- **Unabhängigkeit von UI**
  - UI austauschbar ohne Business Logic Änderung
  - Web, Desktop, Mobile möglich
- **Unabhängigkeit von Datenbank**
  - DB-System austauschbar
  - Business Rules unabhängig von Datenpersistenz

Schichten von außen nach innen:

1. Frameworks & Drivers (UI, DB, External Interfaces)
2. Interface Adapters (Controllers, Presenters)
3. Application Business Rules (Use Cases)
4. Enterprise Business Rules (Entities)

### Clean Architecture Implementation

Strukturbeispiel für einen Online-Shop:

```
1 // Enterprise Business Rules (Entities)
2 public class Order {
3     private List<OrderItem> items;
4     private OrderStatus status;
5
6     public Money calculateTotal() {
7         return items.stream()
8             .map(OrderItem::getSubtotal)
9             .reduce(Money.ZERO, Money::add);
10    }
11 }
12
13 // Application Business Rules (Use Cases)
14 public class CreateOrderUseCase {
15     private OrderRepository repository;
16     private PaymentGateway paymentGateway;
17
18     public OrderId execute(CreateOrderCommand command)
19     {
20         Order order = new Order(command.getItems());
21         PaymentResult result = paymentGateway.process(
22             order.calculateTotal());
23         if (result.isSuccessful()) {
24             return repository.save(order);
25         }
26         throw new PaymentFailedException();
27     }
28 }
29
30 // Interface Adapters
31 public class OrderController {
32     private CreateOrderUseCase createOrderUseCase;
33
34     public OrderResponse createOrder(OrderRequest
35         request) {
36         CreateOrderCommand command =
37             mapToCommand(request);
38         OrderId id =
39             createOrderUseCase.execute(command);
40         return new OrderResponse(id);
41     }
42 }
```

### Microservices Architektur

Grundprinzipien:

- Unabhängig deploybare Services
- Lose Kopplung
- Eigene Datenhaltung pro Service
- REST/Message-basierte Kommunikation

Vorteile:

- Bessere Skalierbarkeit
- Unabhängige Entwicklung
- Technologiefreiheit
- Robustheit

Herausforderungen:

- Verteilte Transaktionen
- Service Discovery
- Datenkonvergenz
- Monitoring

## Microservice Design

### Service für Benutzerprofile:

```
1 @RestController
2 @RequestMapping("/api/users")
3 public class UserProfileController {
4     private final UserService userService;
5
6     @GetMapping("/{id}")
7     public UserProfileDTO getProfile(@PathVariable
8         String id) {
9         UserProfile profile = userService.findById(id);
10        return UserProfileDTO.from(profile);
11    }
12
13    @PutMapping("/{id}")
14    public ResponseEntity<Void> updateProfile(
15        @PathVariable String id,
16        @RequestBody UpdateProfileCommand command)
17    {
18        userService.updateProfile(id, command);
19        return ResponseEntity.ok().build();
20    }
21 }
22 // Event fuer andere Services
23 public class UserProfileUpdatedEvent {
24     private final String userId;
25     private final String newEmail;
26     private final LocalDateTime timestamp;
27
28     // Konstruktor und Getter
29 }
```

### Microservices Design Prinzipien

#### 1. Service Boundaries

- Nach Business Capabilities trennen
- Bounded Context (DDD) beachten
- Datenhoheit festlegen

#### 2. Service Kommunikation

- Synchron vs. Asynchron
- Event-Driven Design
- API Gateway Pattern

#### 3. Datenmanagement

- Database per Service
- Event Sourcing
- CQRS Pattern

#### 4. Resilience

- Circuit Breaker
- Bulkhead Pattern
- Fallback Mechanismen

## Design Patterns in der Architektur

### Model-View-Controller (MVC)

Trennt Anwendung in drei Hauptkomponenten:

- **Model:** Geschäftslogik und Daten
- **View:** Darstellung der Daten
- **Controller:** Steuerung und Koordination

```
1 // Model
2 public class CustomerModel {
3     private String name;
4     private List<Order> orders;
5
6     public void addOrder(Order order) {
7         orders.add(order);
8         notifyViews();
9     }
10 }
11
12 // View
13 public class CustomerView {
14     private CustomerModel model;
15
16     public void displayCustomerInfo() {
17         System.out.println("Customer: " +
18             model.getName());
19         System.out.println("Orders: " +
20             model.getOrders().size());
21     }
22 }
23
24 // Controller
25 public class CustomerController {
26     private CustomerModel model;
27     private CustomerView view;
28
29     public void createOrder(OrderData data) {
30         Order order = new Order(data);
31         model.addOrder(order);
32         view.displayCustomerInfo();
33     }
34 }
```

### Event-Driven Architecture (EDA)

Basiert auf der Produktion, Erkennung und Reaktion auf Events:

#### Komponenten:

- Event Producer
- Event Channel
- Event Consumer
- Event Processor

```
1 // Event Definition
2 public class OrderCreatedEvent {
3     private final String orderId;
4     private final LocalDateTime timestamp;
5     private final BigDecimal totalAmount;
6 }
7
8 // Event Producer
9 public class OrderService {
10     private EventBus eventBus;
11
12     public void createOrder(OrderData data) {
13         Order order = orderRepository.save(data);
14         OrderCreatedEvent event = new
15             OrderCreatedEvent(
16                 order.getId(),
17                 LocalDateTime.now(),
18                 order.getTotalAmount());
19         eventBus.publish(event);
20     }
21 }
22
23 // Event Consumer
24 @EventListener
25 public class InvoiceGenerator {
26     public void handleOrderCreated(OrderCreatedEvent
27         event) {
28         generateInvoice(event.getOrderId());
29     }
30 }
```

### Architektur-Dokumentation

#### 1. Überblick

- Systemkontext
- Hauptkomponenten
- Technologie-Stack

#### 2. Architektur-Entscheidungen

- Begründungen
- Alternativen
- Trade-offs

#### 3. Technische Konzepte

- Persistenz
- Sicherheit
- Integration
- Deployment

#### 4. Qualitätsszenarien

- Performance
- Skalierbarkeit
- Verfügbarkeit
- Wartbarkeit

## Architektur-Dokumentation: REST API

### API-Design und Dokumentation:

```
1 @RestController
2 @RequestMapping("/api/v1/orders")
3 public class OrderController {
4
5     @GetMapping("/{id}")
6     @Operation(summary = "Get order by ID",
7               description = "Returns detailed order
8               information")
9     @ApiResponse(responseCode = "200",
10                description = "Order found"),
11     @ApiResponse(responseCode = "404",
12                description = "Order not found")
13 })
14 public OrderDTO getOrder(@PathVariable String id) {
15     return orderService.findById(id)
16         .map(OrderDTO::from)
17         .orElseThrow(OrderNotFoundException::new);
18 }
19 }
```

#### Qualitätsszenarien:

- Response Time < 200ms (95. Perzentil)
- Verfügbarkeit 99.9
- Maximal 1000 req/s pro Instance
- Automatische Skalierung ab 70

## Integrationsmuster

### Integration Patterns

Muster für die Integration von Systemen und Services:

#### Hauptkategorien:

- **File Transfer:** Datenaustausch über Dateien
- **Shared Database:** Gemeinsame Datenbasis
- **Remote Procedure Call:** Direkter Methodenaufruf
- **Messaging:** Nachrichtenbasierte Kommunikation

## Messaging Pattern Implementation

### Message Producer und Consumer:

```
1 // Message Definition
2 public class OrderMessage {
3     private String orderId;
4     private String customerId;
5     private BigDecimal amount;
6     private OrderStatus status;
7 }
8
9 // Message Producer
10 public class OrderProducer {
11     private MessageQueue messageQueue;
12
13     public void sendOrderCreated(Order order) {
14         OrderMessage message = new OrderMessage(
15             order.getId(),
16             order.getCustomerId(),
17             order.getAmount(),
18             OrderStatus.CREATED
19         );
20         messageQueue.send("orders", message);
21     }
22 }
23
24 // Message Consumer
25 public class OrderProcessor {
26     @MessageListener(queue = "orders")
27     public void processOrder(OrderMessage message) {
28         if (message.getStatus() ==
29             OrderStatus.CREATED) {
30             processNewOrder(message);
31         }
32     }
33
34     private void processNewOrder(OrderMessage message) {
35         {
36             // Verarbeitung der Bestellung
37             validateOrder(message);
38             updateInventory(message);
39             notifyCustomer(message);
40         }
41     }
42 }
```

## API Gateway Pattern

Zentraler Einstiegspunkt für Client-Anfragen:

#### Verantwortlichkeiten:

- Routing von Anfragen
- Authentifizierung/Autorisierung
- Last-Verteilung
- Caching
- Monitoring
- API-Versionierung

```
1 @Component
2 public class ApiGateway {
3     private final AuthService authService;
4     private final ServiceRegistry registry;
5
6     @GetMapping("/api/v1/**")
7     public ResponseEntity<Object> routeRequest(
8         HttpServletRequest request,
9         @RequestHeader("Authorization") String
10         token) {
11
12         // Authentifizierung
13         if (!authService.validateToken(token)) {
14             return ResponseEntity.status(401).build();
15         }
16
17         // Service Discovery
18         String serviceName =
19             extractServiceName(request);
20         ServiceInstance instance =
21             registry.getInstance(serviceName);
22
23         // Request Weiterleitung
24         return forwardRequest(instance, request);
25     }
26 }
```

### API Design Best Practices

#### 1. Ressourcen-Orientierung

- Klare Ressourcen-Namen
- Hierarchische Struktur
- Korrekte HTTP-Methoden

#### 2. Versionierung

- Explizite Versions-Nummer
- Abwärtskompatibilität
- Migrations-Strategie

#### 3. Fehlerbehandlung

- Standardisierte Fehler-Formate
- Aussagekräftige Fehlermeldungen
- Korrekte HTTP-Status-Codes

#### 4. Dokumentation

- OpenAPI/Swagger
- Beispiele und Use Cases
- Fehlerszenarien



## REST API Design

### Ressourcen-Design für E-Commerce System:

```
1 @RestController
2 @RequestMapping("/api/v1")
3 public class ProductController {
4
5     // Collection Resource
6     @GetMapping("/products")
7     public PagedResponse<ProductDTO> getProducts(
8         @RequestParam(defaultValue = "0") int page,
9         @RequestParam(defaultValue = "20") int
10            size) {
11         return productService.findAll(page, size);
12     }
13
14     // Single Resource
15     @GetMapping("/products/{id}")
16     public ProductDTO getProduct(@PathVariable String
17        id) {
18         return productService.findById(id);
19     }
20
21     // Sub-Resource Collection
22     @GetMapping("/products/{id}/reviews")
23     public List<ReviewDTO> getProductReviews(
24         @PathVariable String id) {
25         return reviewService.findByProductId(id);
26     }
27
28     // Error Handling
29     @ExceptionHandler(ProductNotFoundException.class)
30     public ResponseEntity<ErrorResponse>
31         handleNotFound(
32             ProductNotFoundException ex) {
33         ErrorResponse error = new ErrorResponse(
34             "PRODUCT_NOT_FOUND",
35             ex.getMessage());
36         return ResponseEntity.status(404).body(error);
37     }
38 }
```

## Architekturstile und Patterns

### Schichtenarchitektur (Layered Architecture)

Strukturierung eines Systems in horizontale Schichten:

#### Typische Schichten:

- Präsentationsschicht (UI)
- Anwendungsschicht (Application)
- Geschäftslogikschicht (Domain)
- Datenzugriffsschicht (Persistence)

#### Regeln:

- Kommunikation nur mit angrenzenden Schichten
- Abhängigkeiten nur nach unten
- Jede Schicht kapselt ihre Implementierung

## Schichtenarchitektur Implementation

### Beispiel einer typischen Schichtenstruktur:

```
1 // Presentation Layer
2 public class CustomerController {
3     private CustomerService service;
4
5     public CustomerDTO getCustomer(String id) {
6         Customer customer = service.findById(id);
7         return CustomerDTO.from(customer);
8     }
9 }
10
11 // Application Layer
12 public class CustomerService {
13     private CustomerRepository repository;
14
15     public Customer findById(String id) {
16         validateId(id);
17         return repository.findById(id)
18             .orElseThrow(CustomerNotFoundException::new);
19     }
20 }
21
22 // Domain Layer
23 public class Customer {
24     private CustomerId id;
25     private String name;
26     private Address address;
27
28     public void updateAddress(Address newAddress) {
29         validateAddress(newAddress);
30         this.address = newAddress;
31     }
32 }
33
34 // Persistence Layer
35 public class CustomerRepository {
36     private JpaRepository<Customer, CustomerId>
37        jpaRepo;
38
39     public Optional<Customer> findById(String id) {
40         return jpaRepo.findById(new CustomerId(id));
41     }
42 }
```

### Client-Server Architektur

Verteilung von Funktionalitäten zwischen Client und Server:

#### Charakteristiken:

- Klare Trennung von Zuständigkeiten
- Zentralisierte Ressourcenverwaltung
- Skalierbarkeit durch Server-Erweiterung
- Verschiedene Client-Typen möglich

#### Varianten:

- Thin Client: Minimale Client-Logik
- Rich Client: Komplexe Client-Funktionalität
- Web Client: Browser-basiert
- Mobile Client: Für mobile Geräte optimiert

## Architektur-Evaluation

Systematische Bewertung einer Softwarearchitektur:

### 1. Qualitätsattribute identifizieren

- Performance
- Skalierbarkeit
- Wartbarkeit
- Sicherheit

### 2. Szenarien entwickeln

- Normale Nutzung
- Grenzfälle
- Fehlerfälle
- Wartungsszenarien

### 3. Architektur analysieren

- Strukturanalyse
- Verhaltensanalyse
- Trade-off Analyse

### 4. Risiken identifizieren

- Technische Risiken
- Geschäftsrisiken
- Architekturrisiken

### Architektur-Evaluation: Performance

Szenario: Online-Shop während Black Friday

#### Analyse:

- **Last-Annahmen:**
  - 10.000 gleichzeitige Nutzer
  - 1.000 Bestellungen pro Minute
  - 100.000 Produktaufrufe pro Minute
- **Architektur-Maßnahmen:**
  - Caching-Strategie für Produkte
  - Load Balancing für Anfragen
  - Asynchrone Bestellverarbeitung
  - Datenbank-Replikation
- **Monitoring:**
  - Response-Zeiten
  - Server-Auslastung
  - Cache-Hit-Rate
  - Fehlerraten

```
1 // Performance-optimierte Produktabfrage
2 @Cacheable(value = "products")
3 public ProductDTO getProduct(String id) {
4     ProductDTO product = cache.get(id);
5     if (product == null) {
6         product = repository.findById(id)
7             .map(this::toDTO)
8             .orElseThrow();
9         cache.put(id, product);
10    }
11    return product;
12 }
```

**Architektur-Taktiken**

Grundlegende Strategien zur Erfüllung von Qualitätsanforderungen:

**Performance:**

- Resource Pooling
- Caching
- Parallelisierung
- Lazy Loading/Evaluation

**Verfügbarkeit:**

- Redundanz
- Health Monitoring
- Failover Mechanismen
- Circuit Breaker

**Wartbarkeit:**

- Separation of Concerns
- Information Hiding
- Interface-basierte Kommunikation
- Standardisierung

**Performance-Optimierung****Implementation von Caching:**

```

1 @Service
2 public class ProductService {
3     private final Cache<String, Product> cache;
4     private final ProductRepository repository;
5
6     public Product getProduct(String id) {
7         return cache.get(id, key -> {
8             Product product = repository
9                 .findById(key)
10                .orElseThrow();
11
12            // Warm up frequently accessed data
13            product.getCategories().size();
14            product.getReviews().size();
15
16            return product;
17        });
18    }
19
20    @CacheEvict(value = "products")
21    public void updateProduct(Product product) {
22        repository.save(product);
23    }
24 }

```

**Architektur-Review**

Systematische Überprüfung der Architektur:

**1. Vorbereitung**

- Architektur-Dokumentation sichten
- Qualitätsanforderungen prüfen
- Review-Team zusammenstellen
- Checklisten erstellen

**2. Durchführung**

- Architektur-Walkthrough
- Szenario-basierte Evaluation
- Risiko-Analyse
- Trade-off Analyse

**3. Nachbereitung**

- Ergebnisse dokumentieren
- Maßnahmen ableiten
- Priorisierung vornehmen
- Follow-up planen

**Architektur-Review Checkliste****Qualitätskriterien:**

- **Modularität:**
  - Klare Modulgrenze
  - Minimale Abhängigkeiten
  - Hohe Kohäsion
- **Testbarkeit:**
  - Isolation von Komponenten
  - Mockbarkeit von Abhängigkeiten
  - Testautomatisierung
- **Änderbarkeit:**
  - Lokalisierung von Änderungen
  - Erweiterbarkeit
  - Backward Compatibility

```

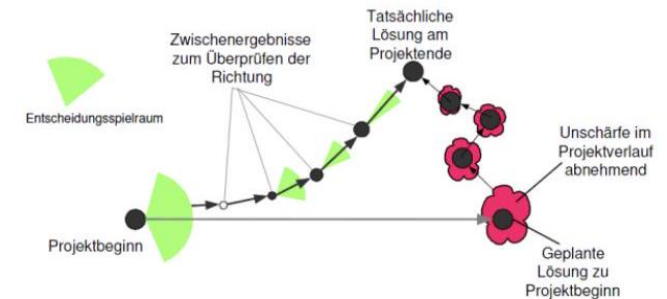
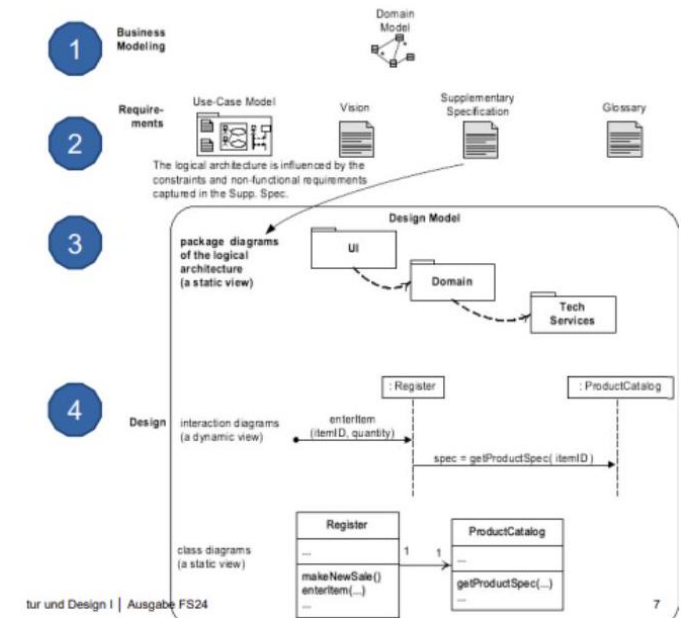
1 // Beispiel fuer gute Testbarkeit
2 public class OrderService {
3     private final OrderRepository repository;
4     private final PaymentGateway paymentGateway;
5
6     // Dependency Injection ermoeeglicht einfaches
7     // Mocking
8     public OrderService(
9         OrderRepository repository,
10        PaymentGateway paymentGateway) {
11         this.repository = repository;
12         this.paymentGateway = paymentGateway;
13     }
14
15     // Klare Methoden-Verantwortlichkeiten
16     public OrderResult createOrder(OrderRequest
17         request) {
18         validateRequest(request);
19         Order order = createOrderEntity(request);
20         PaymentResult payment = processPayment(order);
21         return createOrderResult(order, payment);
22     }
23 }

```

**Überblick Softwareentwicklung**

Die Entwicklung von Software erfolgt in verschiedenen Ebenen:

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



### Softwarearchitektur

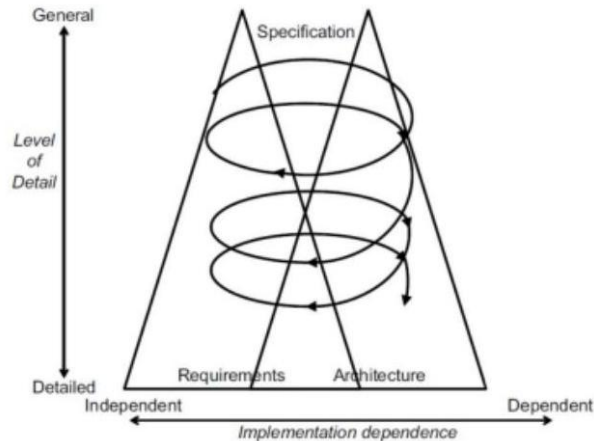
Die Architektur definiert:

- 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:

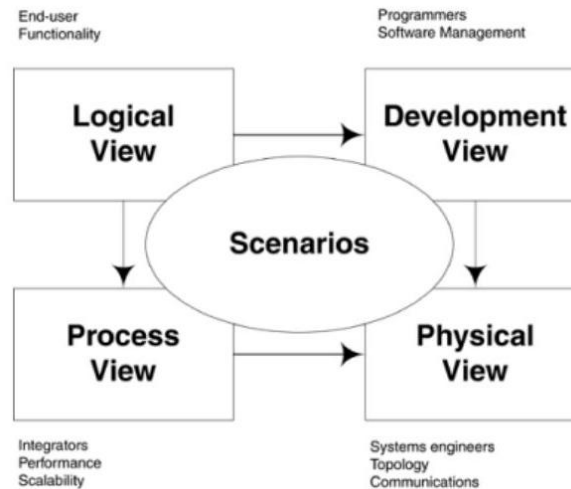
- **Kohäsion:** Innerer Zusammenhang
- **Kopplung:** Externe Abhängigkeiten

#### Eigenschaften:

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

### Architektursichten

Das N+1 View Model beschreibt verschiedene Perspektiven:



### Architekturentwurf

#### Schritte:

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

#### Qualitätskriterien:

- Änderbarkeit
- Wartbarkeit
- Erweiterbarkeit
- Testbarkeit

### 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

### 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

### 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:

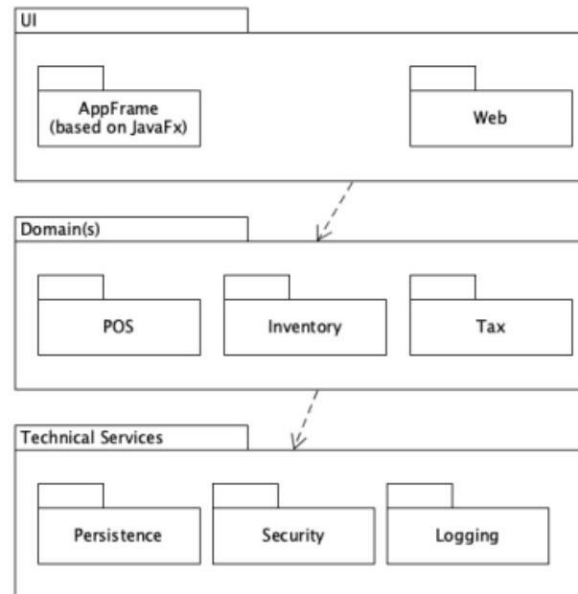
- Zustände und Übergänge
- Start- und Endzustände
- Composite States
- Historie und Parallelität

#### 4. Aktivitätsdiagramm:

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

### 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

### Responsibility Driven Design (RDD)

Design basierend auf Verantwortlichkeiten:

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

**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

### 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

### 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

## UML-Modellierung

### UML Diagrammtypen Übersicht

UML bietet verschiedene Diagrammtypen für statische und dynamische Modellierung:

#### Statische Modelle:

- Klassendiagramm
- Paketdiagramm
- Komponentendiagramm
- Verteilungsdiagramm

#### Dynamische Modelle:

- Sequenzdiagramm
- Kommunikationsdiagramm
- Zustandsdiagramm
- Aktivitätsdiagramm

## Klassendiagramm

### Hauptelemente:

- **Klassen:**
  - Name der Klasse
  - Attribute mit Sichtbarkeit
  - Operationen mit Parametern
- **Beziehungen:**
  - Assoziation (normaler Pfeil)
  - Vererbung (geschlossener Pfeil)
  - Implementierung (gestrichelter Pfeil)
  - Aggregation (leere Raute)
  - Komposition (gefüllte Raute)
- **Interfaces:**
  - Stereotyp «interface»
  - Nur Methodensignaturen
  - Implementierungsbeziehung

### Klassendiagramm: E-Commerce System

#### Domänenmodell mit wichtigen Beziehungen:

```
1 public interface OrderRepository {
2     Optional<Order> findById(OrderId id);
3     void save(Order order);
4 }
5
6 public class Order {
7     private OrderId id;
8     private Customer customer;
9     private List<OrderLine> orderLines;
10    private OrderStatus status;
11
12    public Money calculateTotal() {
13        return orderLines.stream()
14            .map(OrderLine::getSubTotal)
15            .reduce(Money.ZERO,
16                Money::add);
17    }
18 }
19
20 public class OrderLine {
21     private Product product;
22     private int quantity;
23     private Money price;
24
25     public Money getSubTotal() {
26         return price.multiply(quantity);
27     }
28 }
```

## Sequenzdiagramm

### Notationselemente:

- **Lebenslinien:**
  - Objekte als Rechtecke
  - Vertikale gestrichelte Linie
  - Aktivierungsbalken für Ausführung
- **Nachrichten:**
  - Synchron (durchgezogener Pfeil)
  - Asynchron (offener Pfeil)
  - Antwort (gestrichelter Pfeil)
  - Parameter und Rückgabewerte
- **Kontrollelemente:**
  - alt (Alternative)
  - loop (Schleife)
  - opt (Optional)
  - par (Parallel)

### Sequenzdiagramm: Bestellprozess

#### Interaktion zwischen Komponenten:

```
1 public class OrderService {
2     private final OrderRepository orderRepo;
3     private final PaymentService paymentService;
4
5     public OrderConfirmation processOrder(OrderRequest
6         request) {
7         // Validiere Bestellung
8         validateOrder(request);
9
10        // Erstelle Order
11        Order order = createOrder(request);
12        orderRepo.save(order);
13
14        // Prozessiere Zahlung
15        PaymentResult result = paymentService
16            .processPayment(order.getId(),
17                order.getTotal());
18
19        // Bestaetige Bestellung
20        if (result.isSuccessful()) {
21            order.confirm();
22            orderRepo.save(order);
23            return new OrderConfirmation(order);
24        }
25        throw new PaymentFailedException();
26    }
27 }
```

## Zustandsdiagramm

### Notationselemente:

- **Zustände:**
  - Startzustand (gefüllter Kreis)
  - Endzustand (Kreis mit Punkt)
  - Einfache Zustände (Rechteck)
  - Zusammengesetzte Zustände
- **Transitionen:**
  - Event [Guard] / Action
  - Interne Transitionen
  - Selbsttransitionen
- **Spezielle Elemente:**
  - History State (H)
  - Deep History (H\*)
  - Entry/Exit Points
  - Choice Points

### Zustandsdiagramm: Bestellstatus

#### Implementation eines State Patterns:

```
1 public interface OrderState {
2     void process(Order order);
3     void cancel(Order order);
4     void ship(Order order);
5 }
6
7 public class NewOrderState implements OrderState {
8     @Override
9     public void process(Order order) {
10        validateOrder(order);
11        order.setState(new ProcessingState());
12    }
13
14    @Override
15    public void cancel(Order order) {
16        order.setState(new CancelledState());
17    }
18
19    @Override
20    public void ship(Order order) {
21        throw new IllegalStateException(
22            "Cannot ship new order");
23    }
24 }
25
26 public class Order {
27     private OrderState state;
28
29     public void process() {
30        state.process(this);
31    }
32
33    void setState(OrderState newState) {
34        this.state = newState;
35    }
36 }
```



## Aktivitätsdiagramm

### Hauptelemente:

- **Aktionen:**
  - Atomare Aktionen
  - Call Behavior Action
  - Send/Receive Signal
- **Kontrollfluss:**
  - Verzweigungen (Diamond)
  - Parallelisierung (Balken)
  - Join/Merge Nodes
- **Strukturierung:**
  - Activity Partitions (Swimlanes)
  - Structured Activity Nodes
  - Interruptible Regions

## Aktivitätsdiagramm: Bestellabwicklung

### Implementation eines Geschäftsprozesses:

```
1 public class OrderProcessor {
2     public void processOrder(Order order) {
3         // Parallele Verarbeitung
4         CompletableFuture.allOf(
5             validateInventory(order),
6             validatePayment(order)
7         ).thenRun(() -> {
8             if (order.isValid()) {
9                 fulfillOrder(order);
10            } else {
11                handleValidationFailure(order);
12            }
13        });
14    }
15
16    private CompletableFuture<Void> validateInventory(
17        Order order) {
18        return CompletableFuture.runAsync(() -> {
19            order.getItems().forEach(item -> {
20                if
21                    (!inventoryService.isAvailable(item))
22                {
23                    throw new
24                        OutOfStockException(item);
25                }
26            });
27        });
28    }
29 }
```

## Verteilungsdiagramm

### Elemente:

- **Nodes:**
  - Device Nodes
  - Execution Environment
  - Artifacts
- **Verbindungen:**
  - Kommunikationspfade
  - Protokolle
  - Multiplizitäten
- **Deployment:**
  - Deployment Specifications
  - Manifestationen

## Verteilungsdiagramm: Microservice-Architektur

### Deployment-Konfiguration:

```
1 @Configuration
2 public class ServiceConfig {
3     @Value("${service.host}")
4     private String serviceHost;
5
6     @Value("${service.port}")
7     private int servicePort;
8
9     @Bean
10    public ServiceRegistry registry() {
11        return ServiceRegistry.builder()
12            .host(serviceHost)
13            .port(servicePort)
14            .healthCheck("/health")
15            .build();
16    }
17
18    @Bean
19    public LoadBalancer loadBalancer(
20        ServiceRegistry registry) {
21        return new RoundRobinLoadBalancer(registry);
22    }
23 }
```

## Kommunikationsdiagramm

### Hauptelemente:

- **Objekte:**
  - Als Rechtecke dargestellt
  - Mit Objektname und Klasse
  - Verbunden durch Links
- **Nachrichten:**
  - Nummerierte Sequenz
  - Synchrone/Asynchrone Aufrufe
  - Parameter und Rückgabewerte
- **Steuerungselemente:**
  - Bedingte Nachrichten [condition]
  - Iterationen \*
  - Parallele Ausführung ||

## Kommunikationsdiagramm: Shopping Cart

### Objektinteraktionen beim Checkout:

```
1 public class ShoppingCart {
2     private List<CartItem> items;
3     private CheckoutService checkoutService;
4
5     public Order checkout() {
6         // 1: validateItems()
7         validateItems();
8
9         // 2: calculateTotal()
10        Money total = calculateTotal();
11
12        // 3: createOrder(items, total)
13        Order order = checkoutService.createOrder(
14            items, total);
15
16        // 4: clearCart()
17        items.clear();
18
19        return order;
20    }
21 }
```

## Paketdiagramm

### Elemente:

- **Pakete:**
  - Gruppierung von Modellelementen
  - Hierarchische Strukturierung
  - Namensräume
- **Abhängigkeiten:**
  - Import/Export von Elementen
  - «use» Beziehungen
  - Zugriffsrechte

## UML Diagrammauswahl

Entscheidungshilfen für die Wahl des passenden Diagrammtyps:

### 1. Statische Struktur

- Klassendiagramm für Typen und Beziehungen
- Paketdiagramm für Modularisierung
- Komponentendiagramm für Bausteinsicht
- Verteilungsdiagramm für physische Verteilung

### 2. Dynamisches Verhalten

- Sequenzdiagramm für zeitliche Abläufe
- Kommunikationsdiagramm für Objektkollaborationen
- Zustandsdiagramm für Objektlebenszyklen
- Aktivitätsdiagramm für Geschäftsprozesse

### 3. Verwendungszweck

- Analyse: Konzeptuelle Modellierung
- Design: Detaillierte Spezifikation
- Implementation: Code-nahe Darstellung
- Dokumentation: Architekturübersicht

## UML in der Praxis

### Beispiel eines kompletten Designs:

```
1 // Paketstruktur
2 package com.example.shop;
3
4 // Domain Model
5 public class Product {
6     private ProductId id;
7     private String name;
8     private Money price;
9     private Category category;
10 }
11
12 // Service Layer
13 @Service
14 public class ProductService {
15     private final ProductRepository repository;
16     private final PriceCalculator calculator;
17
18     public Product updatePrice(
19         ProductId id, Money newPrice) {
20         Product product = repository.findById(id)
21             .orElseThrow(ProductNotFoundException::new);
22
23         Money calculatedPrice = calculator
24             .calculateFinalPrice(newPrice);
25
26         product.updatePrice(calculatedPrice);
27         return repository.save(product);
28     }
29 }
30
31 // Controller Layer
32 @RestController
33 @RequestMapping("/api/products")
34 public class ProductController {
35     private final ProductService service;
36
37     @PutMapping("/{id}/price")
38     public ProductDTO updatePrice(
39         @PathVariable ProductId id,
40         @RequestBody PriceUpdateRequest request) {
41         Product product = service.updatePrice(
42             id, request.getNewPrice());
43         return ProductDTO.from(product);
44     }
45 }
```

## Use Case Realization

- Was ist Use Case Realization?** Use Case Realization beschreibt die Umsetzung von Use Cases in konkreten Code. Dies umfasst:
- Mapping von Analyse-Artefakten auf Design-Artefakte
  - Implementierung der Systemoperationen aus dem SSD
  - Erstellung von testbarem und wartbarem Code
  - Dokumentation der Design-Entscheidungen

### Vorgehen bei der Use Case Realization 1. Vorbereitung

- Use Case auswählen, offene Fragen klären, SSD ableiten
- Systemoperation auswählen
- Operation Contract erstellen/überlegen/lesen
- Aktuellen Code/Dokumentation analysieren
  - DCD überprüfen/aktualisieren
  - Vergleich mit Domänenmodell durchführen
  - Allenfalls bereits jetzt neue Software-Klassen erstellen
- Falls notwendig, Refactorings durchführen

### 2. Design

- Controller Klasse bestimmen/identifizieren
- Zu verändernde Klassen festlegen
- Weg zu diesen Klassen festlegen:
  - Parameter zur Wegwahl definieren
  - Notwendige Klassen erstellen
  - Verantwortlichkeiten nach GRASP zuweisen
  - Varianten evaluieren nach Low Coupling/High Cohesion

### 3. Implementation

- Veränderungen gemäß Systemvertrag implementieren
- Review bezüglich High Cohesion und Architekturkonformität
- Tests erstellen und durchführen

**GRASP Prinzipien in der Use Case Realization** GRASP (General Responsibility Assignment Software Patterns) dient als Leitfaden für die Zuweisung von Verantwortlichkeiten:

#### Zentrale Prinzipien:

- Information Expert: Verantwortlichkeit dort, wo die Information ist
- Creator: Objekte werden von eng verbundenen anderen Objekten erstellt
- Controller: Erste Anlaufstelle für Systemoperationen
- Low Coupling: Minimale Abhängigkeiten zwischen Klassen
- High Cohesion: Fokussierte Verantwortlichkeiten pro Klasse

## Use Case Realization am Beispiel "Process Sale" 1. Systemoperationen aus SSD:

- makeNewSale()
  - enterItem(itemId, quantity)
  - endSale()
  - makePayment(amount)
- ### 2. Operation Contract für makeNewSale():
- **Vorbedingung:** Register ist betriebsbereit
  - **Nachbedingungen:**
    - Neue Sale-Instanz s ist erstellt
    - s ist die neue aktuelle Sale Instanz von Register
- ### 3. Implementation mit GRASP:

```
1 // Controller Pattern
2 public class Register {
3     private Sale currentSale;
4     private ProductCatalog catalog;
5
6     public void makeNewSale() {
7         currentSale = new Sale(); // Creator Pattern
8     }
9
10    public void enterItem(String itemId, int quantity)
11    {
12        ProductDescription desc =
13            catalog.getDescription(itemId);
14        currentSale.makeLineItem(desc, quantity); //
15            Information Expert
16    }
17 }
```

## Design Class Diagram (DCD) erstellen 1. Analyse Artefakte prüfen

- Domänenmodell als Ausgangsbasis verwenden
  - Use Cases und Contracts analysieren
  - Systemoperationen identifizieren
- ### 2. Technische Klassen ergänzen
- Controller Klasse(n) definieren
  - Service Layer bei Bedarf einführen
  - Persistence Layer bei Bedarf ergänzen
- ### 3. Methoden und Attribute
- Aus Systemoperationen ableiten
  - Getter/Setter nach Bedarf
  - Datentypen festlegen
  - Sichtbarkeiten bestimmen
- ### 4. Beziehungen definieren
- Assoziationen aus Domänenmodell übernehmen
  - Navigierbarkeit festlegen
  - Multiplizitäten definieren

## Interaction Diagrams in der Use Case Realization Sequenzdiagramm für enterItem():

```
1 :Register -> :ProductCatalog: getDescription(itemId)
2 :ProductCatalog --> :Register: desc
3 :Register -> currentSale: makeLineItem(desc, quantity)
4 currentSale -> :SalesLineItem: create(desc, quantity)
5 currentSale -> lineItems: add(sl)
```

### Begründung der Interaktionen:

- Register als Controller empfängt Systemoperation
- ProductCatalog als Information Expert für Produkte
- Sale als Creator für SalesLineItem
- Sale als Container verwaltet seine LineItems

## Typische Prüfungsaufgaben 1. Use Case Realization dokumentieren

- System Sequence Diagram erstellen
- Operation Contracts definieren
- Design Class Diagram zeichnen
- GRASP Prinzipien begründen
- Sequenzdiagramm für wichtige Operationen

## 2. Implementation analysieren

- GRASP Verletzungen identifizieren
- Verbesserungen vorschlagen
- Alternative Designs diskutieren

## 3. Architektur evaluieren

- Schichtenarchitektur prüfen
- Kopplung analysieren
- Kohäsion bewerten

**Verantwortlichkeiten (Responsibilities)** Im objektorientierten Design unterscheiden wir zwei Arten von Verantwortlichkeiten:

#### Doing-Verantwortlichkeiten:

- Selbst etwas tun
- Aktionen anderer Objekte anstoßen
- Aktivitäten anderer Objekte kontrollieren

#### Knowing-Verantwortlichkeiten:

- Private eingekapselte Daten kennen
- Verwandte Objekte kennen
- Dinge berechnen/ableiten können

**Typische Prüfungsaufgabe mit Musterlösung** **Aufgabe:** Implementieren Sie die Use Case Realization für "Benutzer registrieren"

### 1. System Sequence Diagram

- registerUser(email, password)
- confirmRegistration(token)

### 2. Operation Contract registerUser()

- **Vorbedingungen:** Email nicht registriert
- **Nachbedingungen:**
  - User-Instanz erstellt
  - Bestätigungstoken generiert
  - Bestätigungsmail versendet

### 3. Implementation mit GRASP:

```
1 public class UserController {
2     private UserService userService;
3     private EmailService emailService;
4
5     public void registerUser(String email, String
6         password) {
7         // Controller empfaengt Systemoperation
8         User user = userService.createUser(email,
9             password);
10        String token = userService.generateToken();
11        emailService.sendConfirmation(email, token);
12    }
13 }
```

### Testing in Use Case Realization 1. Unit Tests

- Isolierte Tests für einzelne Klassen
- Mocking von Abhängigkeiten
- Tests für Standardfälle und Ausnahmen
- ATRIP-Prinzipien beachten:
  - Automatic: Tests müssen automatisch ausführbar sein
  - Thorough: Vollständige Testabdeckung wichtiger Funktionen
  - Repeatable: Tests müssen reproduzierbar sein
  - Independent: Tests dürfen sich nicht gegenseitig beeinflussen
  - Professional: Tests müssen wartbar und lesbar sein

### 2. Beispiel Unit Test:

```
1 @Test
2 public void shouldCalculateTotalForSale() {
3     // Given
4     Sale sale = new Sale();
5     ProductDescription product = new
6         ProductDescription("p1", "Book", 29.99);
7     sale.makeLineItem(product, 2);
8
9     // When
10    BigDecimal total = sale.getTotal();
11
12    // Then
13    assertEquals(new BigDecimal("59.98"), total);
14 }
```

**Architekturbezogene Aspekte** Bei der Use Case Realization müssen folgende architektonische Aspekte beachtet werden:

#### Schichtenarchitektur:

- Presentation Layer (UI)
- Application Layer (Use Cases)
- Domain Layer (Business Logic)
- Infrastructure Layer (Persistence, External Services)

#### Abhängigkeitsregeln:

- Abhängigkeiten nur nach unten
- Interfaces für externe Services
- Dependency Injection für lose Kopplung

#### Cross-Cutting Concerns:

- Logging
- Security
- Transaction Management
- Exception Handling

### Typische Implementierungsfehler 1. Verletzung von GRASP-Prinzipien

```
1 // Falsch: Information Expert verletzt
2 public class Register {
3     public BigDecimal calculateTotal(Sale sale) {
4         // Register berechnet Total statt Sale
5         return sale.getItems().stream()
6             .map(item -> item.getPrice())
7             .reduce(BigDecimal.ZERO,
8                 BigDecimal::add);
9     }
10 }
11 // Richtig: Sale ist Information Expert
12 public class Sale {
13     public BigDecimal getTotal() {
14         return items.stream()
15             .map(item -> item.getSubtotal())
16             .reduce(BigDecimal.ZERO,
17                 BigDecimal::add);
18     }
19 }
```

### 2. Architekturverletzungen

```
1 // Falsch: Domain-Objekt mit UI-Abhängigkeit
2 public class Sale {
3     private JFrame frame;
4     public void complete() {
5         // Domain-Logik vermischt mit UI
6         frame.showMessageDialog("Sale completed");
7     }
8 }
9 // Richtig: Trennung der Schichten
10 public class Sale {
11     public void complete() {
12         // Reine Domain-Logik
13         this.status = SaleStatus.COMPLETED;
14         this.completionTime = LocalDateTime.now();
15     }
16 }
17 }
```

### Refactoring in Use Case Realization 1. Code Smells erkennen

- Duplizierter Code
- Lange Methoden
- Große Klassen
- Feature Envy
- Data Class

### 2. Refactoring durchführen

- Extract Method
- Move Method
- Extract Class
- Introduce Parameter Object
- Replace Conditional with Polymorphism

### 3. Beispiel Refactoring:

```
1 // Vor Refactoring
2 public class Sale {
3     public void complete() {
4         BigDecimal total = BigDecimal.ZERO;
5         for(SaleLineItem item : items) {
6             total = total.add(item.getPrice())
7                 .multiply(new
8                     BigDecimal(item.getQuantity()));
9         }
10        this.total = total;
11        this.status = "COMPLETED";
12        // ... weitere 20 Zeilen Code
13    }
14 }
15 // Nach Refactoring
16 public class Sale {
17     public void complete() {
18         calculateTotal();
19         updateStatus();
20         generateReceipt();
21         notifyInventory();
22     }
23
24     private void calculateTotal() {
25         this.total = items.stream()
26             .map(SaleLineItem::getSubtotal)
27             .reduce(BigDecimal.ZERO, BigDecimal::add);
28     }
29     // ... weitere private Methoden
30 }
```

**Vollständiges Use Case Realization Beispiel**    **Use Case:** Warenkorb zum Check-out freigeben

### 1. System Sequence Diagram

- validateCart()
- prepareCheckout()
- getCheckoutURL()

### 2. Design Class Diagram

- CartController
- ShoppingCart
- CartValidator
- CheckoutService

### 3. Implementation:

```
1 public class CartController {
2     private CartValidator validator;
3     private CheckoutService checkoutService;
4
5     public CheckoutResult prepareCheckout(String
6         cartId) {
7         ShoppingCart cart = findCart(cartId);
8
9         // Validierung (Information Expert)
10        ValidationResult result =
11            validator.validate(cart);
12        if (!result.isValid()) {
13            throw new
14                ValidationException(result.getErrors());
15        }
16
17        // Checkout vorbereiten
18        String checkoutUrl =
19            checkoutService.initiate(cart);
20        return new CheckoutResult(checkoutUrl);
21    }
22 }
```

### 4. Tests:

```
1 @Test
2 public void shouldPrepareCheckoutForValidCart() {
3     // Given
4     ShoppingCart cart = createValidCart();
5     when(validator.validate(cart))
6         .thenReturn(ValidationResult.valid());
7     when(checkoutService.initiate(cart))
8         .thenReturn("https://checkout/123");
9
10    // When
11    CheckoutResult result =
12        controller.prepareCheckout(cart.getId());
13
14    // Then
15    assertNotNull(result.getCheckoutUrl());
16    verify(checkoutService).initiate(cart);
17 }
```

**GRASP-basierte Implementation**    **Szenario:** Implementierung einer Bestellverwaltung mit GRASP-Prinzipien

### Information Expert:

```
1 public class Order {
2     private List<OrderItem> items = new ArrayList<>();
3
4     public BigDecimal calculateTotal() {
5         return items.stream()
6             .map(OrderItem::getSubtotal)
7             .reduce(BigDecimal.ZERO,
8                 BigDecimal::add);
9     }
10 }
```

### Creator:

```
1 public class Order {
2     public OrderItem createOrderItem(Product product,
3         int quantity) {
4         OrderItem item = new OrderItem(product,
5             quantity);
6         items.add(item);
7         return item;
8     }
9 }
```

### Controller:

```
1 public class OrderController {
2     private OrderService orderService;
3
4     public OrderDTO createOrder(String customerId) {
5         Order order =
6             orderService.initializeOrder(customerId);
7         return OrderMapper.toDTO(order);
8     }
9 }
```

### Use Case Realization Dokumentation    1. Analysephase

- Use Case und Systemoperationen dokumentieren
- Domänenmodell-Ausschnitt zeigen
- Relevante Anforderungen auflisten

### 2. Design

- Design Class Diagram erstellen
- Sequenzdiagramme für komplexe Abläufe
- GRASP-Prinzipien begründen

### 3. Implementation

- Code-Struktur dokumentieren
- Wichtige Algorithmen erläutern
- Test-Strategie beschreiben

**Design Decisions Documentation**    **Use Case:** Warenkorb verwalten

### Design Entscheidungen:

- **Pattern: Repository**
  - Grund: Abstraktion der Datenpersistenz
  - Alternative: DAO Pattern verworfen
- **Pattern: Factory**
  - Grund: Komplexe Objekterstellung
  - Vorteil: Testbarkeit verbessert

### Implementation:

```
1 public class CartFactory {
2     public ShoppingCart createCart(String customerId) {
3         ShoppingCart cart = new
4             ShoppingCart(customerId);
5         cart.setCreatedAt(LocalDateTime.now());
6         cart.setStatus(CartStatus.NEW);
7         return cart;
8     }
9 }
```

### Testing in Use Case Realization

#### 1. Unit Tests

- Einzelne Klassen testen
- Mocks für Abhängigkeiten
- Edge Cases abdecken

#### 2. Integration Tests

- Zusammenspiel der Komponenten
- Use Case Szenarien testen
- Fehlerszenarien prüfen

### Beispiel Test:

```
1 @Test
2 public void shouldCalculateCartTotal() {
3     // Given
4     ShoppingCart cart = new
5         ShoppingCart("customer123");
6     cart.addItem(new Product("p1", "Book", 29.99), 2);
7     cart.addItem(new Product("p2", "DVD", 19.99), 1);
8
9     // When
10    BigDecimal total = cart.calculateTotal();
11
12    // Then
13    assertEquals(new BigDecimal("79.97"), total);
14 }
```



## Kompletter Use Case Realization Prozess    Use Case: Benutzerregistrierung

### 1. System Sequence Diagram

### 2. Operation Contract

- **Operation:** registerUser(userDTO)
- **Vorbedingungen:** Benutzer nicht registriert
- **Nachbedingungen:**
  - Benutzer erstellt
  - Bestätigungsmail versendet

### 3. Implementation:

```
1 public class UserRegistrationService {
2     private UserRepository userRepo;
3     private EmailService emailService;
4
5     public User registerUser(UserDTO dto) {
6         // Validate input
7         validateUserInput(dto);
8
9         // Create user
10        User user = new User();
11        user.setEmail(dto.getEmail());
12        user.setPassword(passwordEncoder.encode(dto.getPassword()));
13
14        // Save user
15        user = userRepo.save(user);
16
17        // Send confirmation
18        emailService.sendConfirmation(user.getEmail());
19
20        return user;
21    }
22 }
```

## Use Case Realisation old

### 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:

- Aktuelle 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

### Factory Method Pattern Problem:

- Erzeugung von Objekten soll flexibel sein
- Entscheidung über konkrete Klasse erst zur Laufzeit
- Basis für Frameworks/Libraries

### Lösung:

- Abstrakte Methode zur Objekterzeugung
- Subklassen bestimmen konkreten Typ
- Template Method Pattern für Algorithmus

### Beispiel:

```
1 public abstract class DocumentCreator {
2     // Factory Method
3     protected abstract Document createDocument();
4
5     // Template Method using factory method
6     public final void openDocument() {
7         Document doc = createDocument();
8         doc.open();
9     }
10 }
11
12 public class PDFCreator extends DocumentCreator {
13     protected Document createDocument() {
14         return new PDFDocument();
15     }
16 }
```

### Command Pattern Problem:

- Aktionen/Requests als Objekte kapseln
- Entkopplung von Sender und Empfänger
- Unterstützung für Undo/Redo

### Lösung:

- Command Interface mit execute() Methode
- Konkrete Commands für spezifische Aktionen
- Invoker ruft Commands auf
- Optional: Undo/Redo Funktionalität

### Beispiel:

```
1 public interface Command {
2     void execute();
3     void undo();
4 }
5
6 public class SaveCommand implements Command {
7     private Document doc;
8
9     public SaveCommand(Document doc) {
10         this.doc = doc;
11     }
12
13     public void execute() {
14         doc.save();
15     }
16
17     public void undo() {
18         doc.revertToLastVersion();
19     }
20 }
```

### Template Method Pattern Problem:

- Algorithmus-Struktur fest, aber Schritte variabel
- Code-Duplizierung vermeiden
- Erweiterbarkeit gewährleisten

### Lösung:

- Abstrakte Basisklasse definiert Algorithmus-Skelett
- Hook Methods für variable Schritte
- Konkrete Klassen implementieren Hook Methods

### Beispiel:

```
1 public abstract class DataMiner {
2     // Template method
3     public final void mine() {
4         openFile();
5         extractData();
6         parseData();
7         analyzeData();
8         sendReport();
9         closeFile();
10    }
11
12    abstract void extractData();
13    abstract void parseData();
14
15    // Hook method with default implementation
16    protected void analyzeData() {
17        // Default analysis
18    }
19 }
```

### Pattern Implementierung Schritte zur Pattern-Implementierung:

#### 1. Analyse

- Pattern-Struktur verstehen
- Anpassungen identifizieren
- Schnittstellen definieren

#### 2. Design

- Klassenstruktur erstellen
- Beziehungen definieren
- Methodensignaturen festlegen

#### 3. Implementation

- Code-Struktur aufbauen
- Schnittstellen implementieren
- Tests erstellen

#### 4. Review

- Pattern-Konformität prüfen
- Komplexität bewerten
- Testabdeckung sicherstellen

### Pattern Vergleichstabelle Creational Patterns:

- **Abstract Factory:** Familien verwandter Objekte
- **Factory Method:** Objekterzeugung durch Subklassen
- **Singleton:** Genau eine Instanz
- **Builder:** Komplexe Objektkonstruktion
- **Prototype:** Klonen existierender Objekte

### Structural Patterns:

- **Adapter:** Inkompatible Interfaces verbinden
- **Bridge:** Abstraktion von Implementation trennen
- **Composite:** Baumstrukturen einheitlich behandeln
- **Decorator:** Dynamisch Funktionalität erweitern
- **Facade:** Subsystem vereinfachen
- **Proxy:** Zugriffskontrolle auf Objekte

### Behavioral Patterns:

- **Chain of Responsibility:** Anfragen durch Handler-Kette
- **Command:** Requests als Objekte
- **Observer:** Abhängigkeiten bei Zustandsänderungen
- **Strategy:** Austauschbare Algorithmen
- **Template Method:** Skelett eines Algorithmus
- **State:** Zustandsabhängiges Verhalten

### Prüfungsvorbereitung Design Patterns 1. Kernkonzepte für jedes Pattern:

- Problem verstehen
- Lösung erkennen
- Struktur beschreiben können
- Anwendungsfälle kennen

### 2. Typische Prüfungsfragen:

- Pattern für gegebenes Problem identifizieren
- Unterschiede zwischen ähnlichen Patterns erklären
- UML-Diagramm für Pattern-Implementierung zeichnen
- Code-Beispiel analysieren und Pattern erkennen

### 3. Übungsaufgaben lösen:

- Pattern-Strukturen skizzieren
- Implementierungsbeispiele schreiben
- Pattern-Kombinationen üben
- Refactoring mit Patterns durchführen

**Komplexe Pattern-Anwendung Szenario:** Dokumentverarbeitungssystem

**Anforderungen:**

- Verschiedene Dokumenttypen (PDF, DOC, TXT)
- Unterschiedliche Speicherorte (Lokal, Cloud)
- Verschiedene Verarbeitungsoperationen
- Erweiterbarkeit für neue Formate

**Pattern-Kombination:**

- **Abstract Factory:** Dokumenterzeugung
- **Strategy:** Verarbeitungsalgorithmen
- **Adapter:** Speicheranbindung
- **Command:** Operationen

**Implementierung:**

```
1 // Abstract Factory
2 public interface DocumentFactory {
3     Document createDocument();
4     Storage createStorage();
5 }
6
7 // Strategy
8 public interface ProcessingStrategy {
9     void process(Document doc);
10 }
11
12 // Command
13 public interface DocumentCommand {
14     void execute();
15     void undo();
16 }
17
18 // Adapter
19 public class CloudStorageAdapter implements Storage {
20     private CloudService service;
21
22     public void save(Document doc) {
23         service.upload(doc.getBytes());
24     }
25 }
```

**Pattern Anti-Patterns 1. Überflüssige Pattern-Anwendung:**

- Komplexität ohne Nutzen
- Pattern als Selbstzweck
- Übertriebene Abstraktion

**2. Falsche Pattern-Wahl:**

- Pattern passt nicht zum Problem
- Bessere Alternativen ignoriert
- Performance-Probleme

**3. Schlechte Implementation:**

- Pattern-Struktur nicht eingehalten
- Schlechte Namensgebung
- Inkonsistente Verwendung

[continue with more content or specific focus area?]

**Exam-Style Pattern Recognition Aufgabe:** Analysieren Sie den folgenden Code und identifizieren Sie verwendete Patterns:

```
1 public interface PaymentStrategy {
2     void pay(int amount);
3 }
4
5 public class CreditCardPayment implements
6     PaymentStrategy {
7     private String cardNum;
8
9     public void pay(int amount) {
10         // Process credit card payment
11     }
12 }
13
14 public class PayPalPayment implements PaymentStrategy {
15     private String email;
16
17     public void pay(int amount) {
18         // Process PayPal payment
19     }
20 }
21
22 public class ShoppingCart {
23     private PaymentStrategy paymentMethod;
24
25     public void setPaymentMethod(PaymentStrategy
26         method) {
27         this.paymentMethod = method;
28     }
29
30     public void checkout(int amount) {
31         paymentMethod.pay(amount);
32     }
33 }
```

**Lösung:**

- Identifiziertes Pattern: Strategy
- Begründung:
  - Interface für austauschbare Algorithmen
  - Konkrete Strategien für verschiedene Zahlungsmethoden
  - Kontext (ShoppingCart) verwendet Strategy-Interface
- Vorteile:
  - Neue Zahlungsmethoden einfach hinzufügbär
  - Lose Kopplung zwischen Kontext und Algorithmus
  - Zahlungsmethode zur Laufzeit änderbar

## Design Patterns old

### 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



## Unit Testing Best Practices FIRST Prinzipien:

- **Fast:** Tests müssen schnell ausführbar sein
- **Independent:** Tests dürfen nicht voneinander abhängen
- **Repeatable:** Gleiche Ergebnisse bei mehrfacher Ausführung
- **Self-validating:** Automatische Ergebnisüberprüfung
- **Timely:** Tests werden zeitnah geschrieben

### Beispiel Implementierung:

```
1 public class OrderServiceTest {
2     private OrderService orderService;
3     private OrderRepository mockRepo;
4
5     @Before
6     public void setup() {
7         mockRepo = mock(OrderRepository.class);
8         orderService = new OrderService(mockRepo);
9     }
10
11     @Test
12     public void shouldCalculateOrderTotal() {
13         // Arrange
14         Order order = new Order();
15         order.addItem(new OrderItem("Book", 29.99, 2));
16         order.addItem(new OrderItem("DVD", 19.99, 1));
17
18         // Act
19         BigDecimal total =
20             orderService.calculateTotal(order);
21
22         // Assert
23         assertEquals(new BigDecimal("79.97"), total);
24     }
25
26     @Test(expected = ValidationException.class)
27     public void shouldThrowExceptionForEmptyOrder() {
28         // Arrange
29         Order order = new Order();
30
31         // Act & Assert
32         orderService.calculateTotal(order);
33     }
34 }
```

## Integration Testing 1. Teststrategie:

- Bottom-up Integration
  - Top-down Integration
  - Big Bang Integration
- ### 2. Testumfang:
- Komponenteninteraktion
  - Schnittstellentests
  - Datenfluss

### 3. Testaufbau:

```
1 @SpringBootTest
2 public class OrderIntegrationTest {
3     @Autowired
4     private OrderService orderService;
5
6     @Autowired
7     private CustomerService customerService;
8
9     @Test
10    public void shouldProcessOrderEndToEnd() {
11        // Setup test data
12        Customer customer =
13            customerService.createCustomer(
14                "Test Customer", "test@example.com");
15
16        Order order = new Order(customer);
17        order.addItem(new OrderItem("Test Product",
18            29.99, 1));
19
20        // Execute process
21        OrderResult result =
22            orderService.processOrder(order);
23
24        // Verify results
25        assertTrue(result.isSuccess());
26        assertEquals("COMPLETED", result.getStatus());
27        // Verify database state
28        Order savedOrder =
29            orderService.findOrder(result.getOrderId());
30        assertEquals("COMPLETED",
31            savedOrder.getStatus());
32    }
33 }
```

## Refactoring Patterns 1. Extract Class:

```
1 // Vorher
2 public class Order {
3     private List<OrderItem> items;
4     private BigDecimal total;
5     private BigDecimal tax;
6     private BigDecimal shipping;
7
8     public BigDecimal calculateTotal() {
9         total = items.stream()
10             .map(OrderItem::getSubtotal)
11             .reduce(BigDecimal.ZERO, BigDecimal::add);
12         tax = total.multiply(TAX_RATE);
13         shipping = calculateShipping();
14         return total.add(tax).add(shipping);
15     }
16 }
17
18 // Nachher
19 public class Order {
20     private List<OrderItem> items;
21     private PriceCalculator priceCalculator;
22
23     public BigDecimal calculateTotal() {
24         return priceCalculator.calculateTotal(items);
25     }
26 }
27
28 public class PriceCalculator {
29     public BigDecimal calculateTotal(List<OrderItem>
30         items) {
31         BigDecimal subtotal = calculateSubtotal(items);
32         BigDecimal tax = calculateTax(subtotal);
33         BigDecimal shipping = calculateShipping(items);
34         return subtotal.add(tax).add(shipping);
35     }
36 }
```

## System Testing 1. Funktionale Tests:

- End-to-End Szenarien
- Geschäftsprozesse
- Use Case Tests

### 2. Nicht-funktionale Tests:

- Performance Tests
- Last Tests
- Sicherheitstests
- Usability Tests

### 3. Dokumentation:

- Testfälle
- Testdaten
- Testergebnisse
- Fehlerberichte

[continue with more content or specific focus area?]

## 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

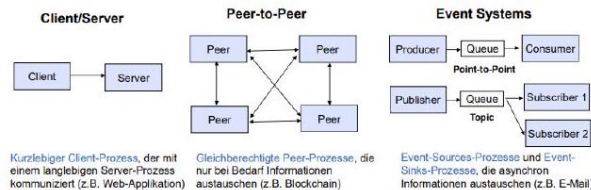


Abbildung 20: Architekturmodelle

**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 Netzerkausfällen

**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 Universitäts-system 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

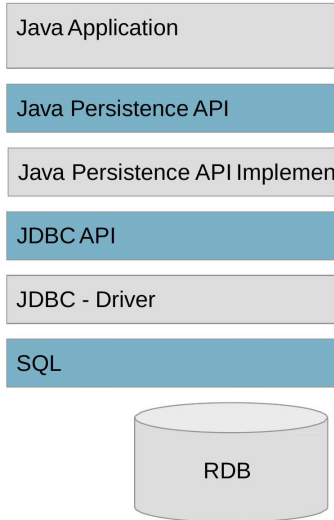
JPA ist der Java-Standard für O/R-Mapping:

- **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

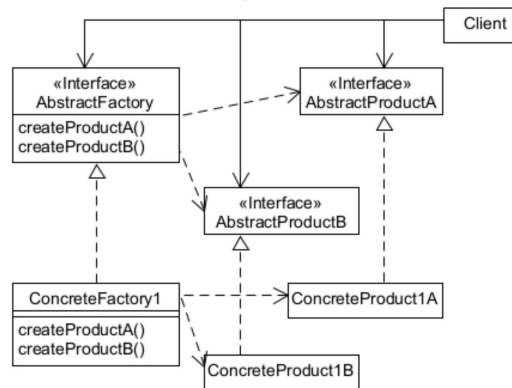
## Design Patterns in Frameworks

### Abstract Factory

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

**Lösung:**

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

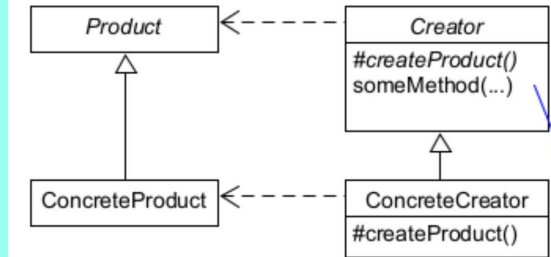


### Factory Method

**Problem:** Flexible Objekterzeugung in wiederverwendbarer Klasse

**Lösung:**

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

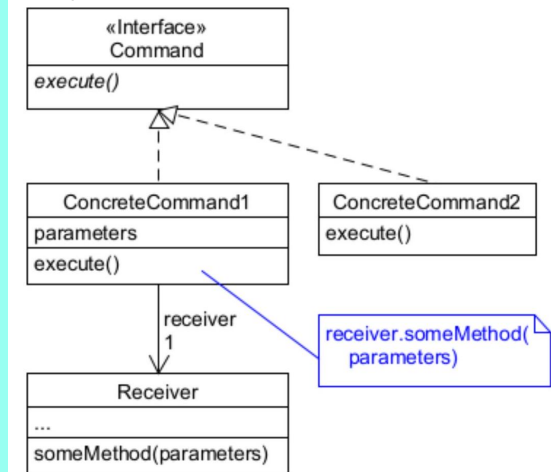


### Command

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

**Lösung:**

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

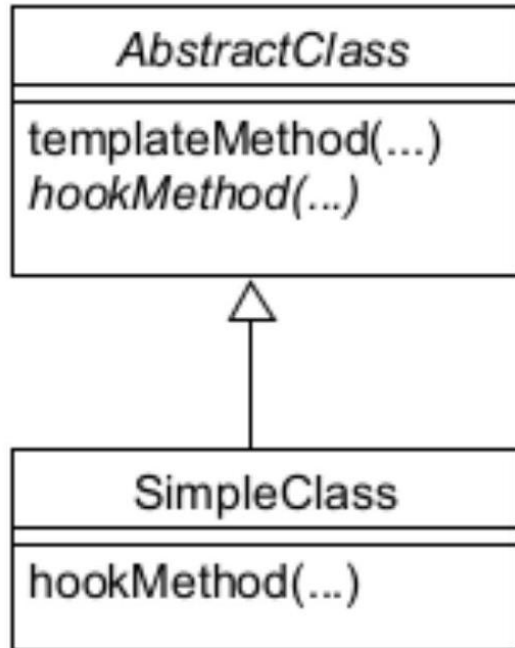


## Template Method

**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