Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Einführung und Überblick

Software Engineering Software Engineering ist eine systematische und strukturierte Entwicklung von Software mit dem Ziel:

- Hohe Qualität und Zuverlässigkeit
- Systematische Fehlervermeidung
- Risikominimierung
- Planbarkeit und Steuerung

Softwareentwicklungsprozesse **Dimensionen der Komplexität:**

- Requirements (bekannt - unbekannt)
- Technology (bekannt - unbekannt)
- Skills/Experience (vorhanden - fehlend)

Kernprozesse:

- **Primärprozesse:**
  - Anforderungsanalyse
  - Architektur/Design
  - Implementierung
  - Test und Qualitätssicherung
  - Wartung/Betrieb
- **Unterstützungsprozesse:**
  - Projektmanagement
  - Konfigurationsmanagement
  - Qualitätsmanagement

Entwicklungsprozess-Modelle

- **Code and Fix:**
  - Einfach und agil
  - Keine strukturierte Planung
  - Schwer wartbar
  - Nur für kleine Projekte
- **Wasserfallmodell:**
  - Sequentielle Phasen
  - Gute Planbarkeit
  - Wenig Flexibilität
  - Spätes Feedback
- **Iterativ-inkrementell:**
  - Schrittweise Entwicklung
  - Regelmäßiges Feedback
  - Risikoorientierte Planung
  - Basis für agile Modelle

Modelle in der Softwareentwicklung Modelle dienen als Abstraktion und Kommunikationsmittel:

- **Zweck:**
  - Verstehen eines Systems
  - Kommunikation zwischen Stakeholdern
  - Spezifikation von Anforderungen
  - Dokumentation von Design
- **Arten:**
  - Anforderungsmodelle
  - Architekturmodelle
  - Entwurfsmodelle
  - Testmodelle

Unified Modeling Language (UML) Standardsprache für grafische Modellierung:

- **Verwendung als:**
  - Sketch: Informelle Kommunikation
  - Blueprint: Detaillierte Spezifikation
  - Programming Language: Ausführbare Modelle

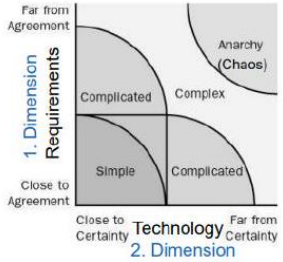
Einführung und Überblick

Software Engineering

- Systematischer Ansatz für Entwicklung, Betrieb und Wartung von Software
- Ziel: Qualitativ hochwertige Software durch strukturierte Prozesse
- Kernprozesse: Anforderungen, Design, Implementierung, Test, Wartung

Dimensionen Software-Entwicklungs-Probleme

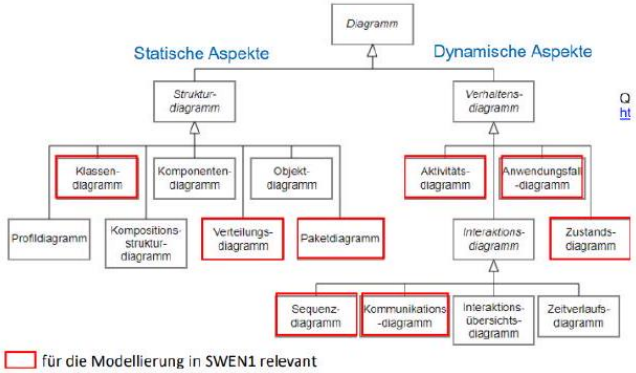
- Requirements (Bekannt - Unbekannt)
- Technology (Bekannt - Unbekannt)
- Skills/Experience (Vorhanden - Nicht vorhanden)



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

Modelle in der Softwareentwicklung

- Modelle als Abstraktionen der Realität
- Dienen Kommunikation und Dokumentation
- Typen:
  - Anforderungsmodelle
  - Architektur- und Designmodelle
  - Implementierungsmodelle
  - Testmodelle



Prozessmodelle

Code and Fix:

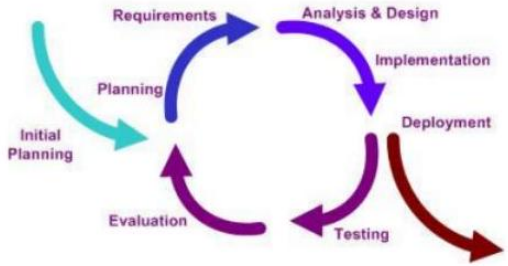
- Ad-hoc Entwicklung ohne formalen Prozess
- Schnell aber schwer wartbar

Wasserfallmodell:

- Sequenzielle Phasen
- Gut planbar aber unflexibel

Iterativ-inkrementell:

- Schrittweise Entwicklung in Iterationen
- Flexibel und risikominimierend
- Basis für agile Entwicklung



UML (Unified Modeling Language)

- Standardsprache für Softwaremodellierung
- Verwendung als:
  - Sketch: Informelle Kommunikation
  - Blueprint: Detaillierte Spezifikation
  - Programming Language: Ausführbare Modelle

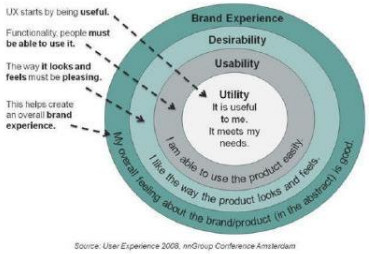
Anforderungsanalyse

Usability und User Experience

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)



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 Nutzererfahrung

Usability-Evaluation durchführen

1. **Vorbereitung**
   - Testziele definieren
   - Testpersonen auswählen
   - Testaufgaben erstellen
2. **Durchführung**
   - Beobachtung der Nutzer
   - Protokollierung von Problemen
   - Zeitmessung der Aufgaben
3. **Auswertung**
   - Probleme klassifizieren
   - Schweregrad bestimmen
   - Verbesserungen vorschlagen
4. **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:**
  - Einfacher Einstieg
  - Unterstützung beim Lernen

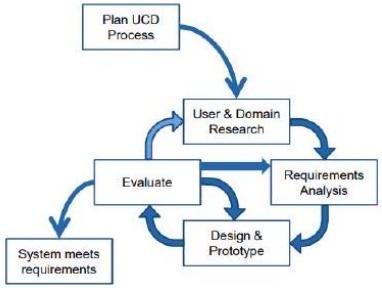
User-Centered Design

UCD (User-Centered Desigr

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

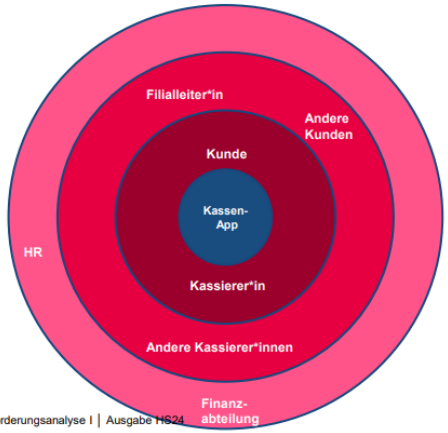


Wichtige Artefakte

- 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.



UCD Artefakte erstellen

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

UCD Prozess-Phasen

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

User & Domain Research

1. **Zielgruppe identifizieren**
   - Wer sind die Benutzer?
   - Was sind ihre Aufgaben/Ziele?
   - Wie sieht ihre Arbeitsumgebung aus?
   - Welche Sprache/Begriffe verwenden sie?
2. **Daten sammeln durch**
   - Contextual Inquiry
   - Interviews
   - Beobachtung
   - Fokusgruppen
   - Nutzungsauswertung
3. **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

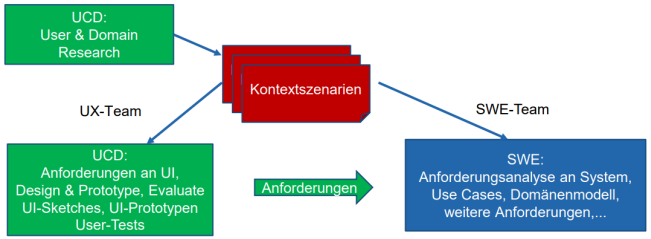
Frustrationen:

- 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
- ïst-einBeziehung

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:

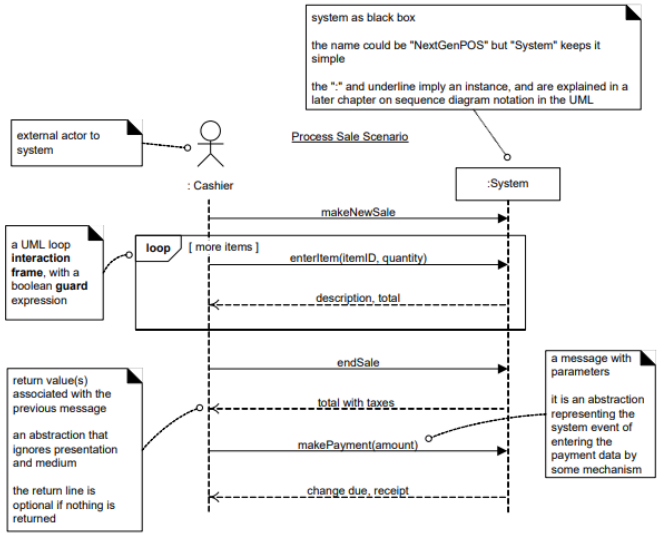
- Technische Implementierungsdetails
- Fehlende Akteurperspektive
- Keine Alternativen/Fehlerbehandlung
- Unklarer Nutzen/Ziel

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:

```
// Gut - klar und domaenenorientiert
createOrder(customer: CustomerId): OrderId
addOrderItem(orderId: OrderId,
             product: ProductId,
             quantity: int)

// Schlecht - zu technisch/implementierungsnah
insertIntoOrderTable(customerData: Map)
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:

- Kartenvalidierung
- 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:

```
// Kontostand pruefen
checkBalance(): Money

// Ueberweisung initiieren
initiateTransfer(recipient: String,
                 iban: String,
                 amount: Money,
                 purpose: String): TransferId

// TAN anfordern
requestTAN(transferId: TransferId): void

// Ueberweisung bestaetigen
confirmTransfer(transferId: TransferId,
                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 bestellenïn 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:

```
// Request credit card approval
requestApproval(cardNum: String,
                expiryDate: Date,
                amount: Money): Boolean

// Post transaction to accounting
postTransaction(transactionData:
    TransactionData)
```

Wichtige Aspekte:

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

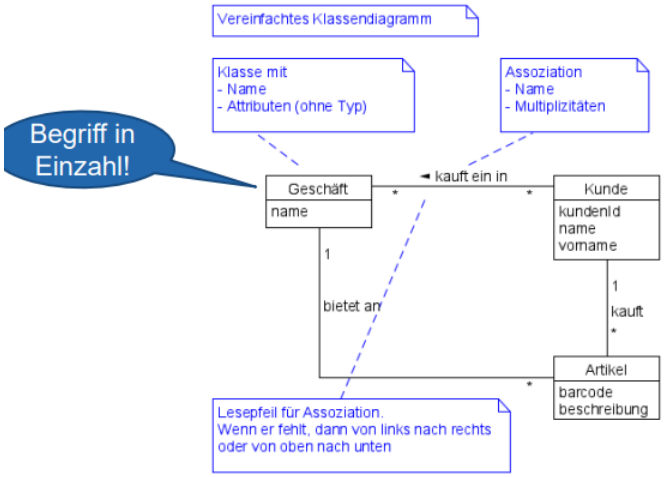
Sequenzdiagramm: TO BE ADDED

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



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

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

Analysemuster im Domänenmodell

Analysemuster im Überblick

Details siehe nächste Seite

Bewährte Strukturen für wiederkehrende Modellierungssituationen:

- **Beschreibungsklassen:** Trennung von Typ und Instanz
- **Generalisierung:** ïst-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 ïst-einBeziehungen
   - 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

Prüfungsaufgabe: Konzeptidentifikation

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

Identifizierte Konzepte: Buch (Beschreibungsklasse), Exemplar (Physisches Objekt), Mitglied (Rolle), Ausleihe (Transaktion), Mahnung (Artefakt)

Begründung:

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

Review eines Domänenmodells

Checkliste für die Überprüfung:

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

Typische Modellierungsfehler vermeiden

- **Keine Softwareklassen modellieren**
  - Manager-Klassen vermeiden
  - Keine technischen Helper-Klassen
- **Keine Operationen modellieren**
  - Fokus auf Struktur, nicht Verhalten
  - Keine CRUD-Operationen
- **Richtige Abstraktionsebene wählen**
  - Nicht zu detailliert 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: String, orderId: Integer
- **Richtig:** Direkte Assoziation zwischen Kunde und Bestellung

Fehler 3: Implementierungsdetails

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

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

Detaillierte Analysemuster

1. Beschreibungsklassen

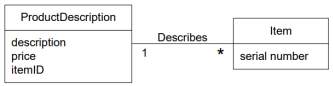
Trennt die Beschreibung eines Typs von seinen konkreten Instanzen.

Anwendung:

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

Beispielstruktur:

- ProductDescription (Typ)
  - price, description, itemID
- Product (Instanz)
  - serialNumber



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

Regeln:

- 100% Regel: Jede Instanz der Spezialisierung ist auch Instanz der Generalisierung
- 'IS-A' Beziehung
- Gemeinsame Attribute/Assoziationen als Grund für Generalisierung
- Gemeinsame Eigenschaften in Basisklasse
- Spezifische Eigenschaften in Unterklassen

Beispiele:

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

Generalisierung im Online-Shop

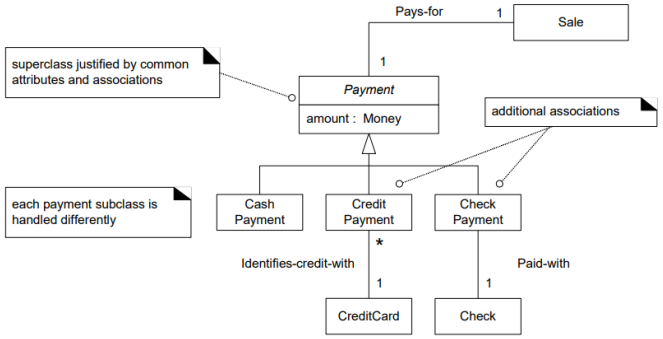
Szenario: Versch. Zahlungsarten

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 der Teile.

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

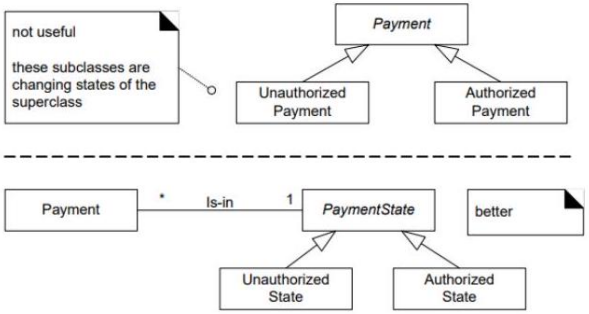
Modelliert Zustände als eigene Konzepthierarchie statt als Attribut.

Vorteile:

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

Beispielstruktur:

- OrderState (abstrakt)
  - New, InProgress, Completed
- Order ïst inÖrderState



Zustandsmodellierung: Ticketsystem

Szenario: Support-Tickets mit verschiedenen Status

Falsche Modellierung:

```
class Ticket {
    enum Status {NEW, OPEN, IN_PROGRESS,
        RESOLVED, CLOSED}
    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

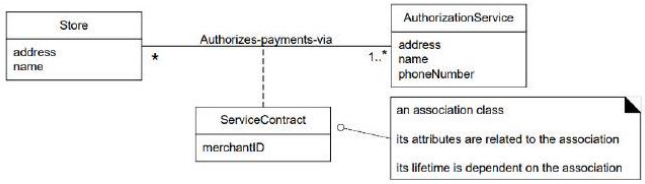
Modelliert Attribute einer Beziehung zwischen Konzepten, eigene Klasse für die Assoziation

Einsatz wenn:

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

Notation:

- Gestrichelte Linie zur Assoziation
- Klasse enthält beziehungsspezifische 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 beziehungsspezifisch

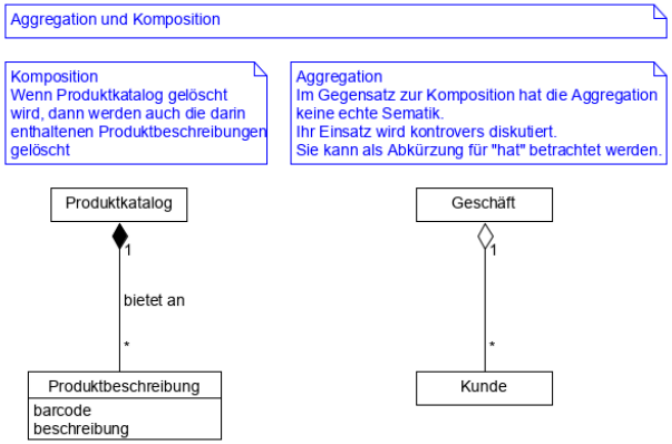
7. Wertobjekte

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

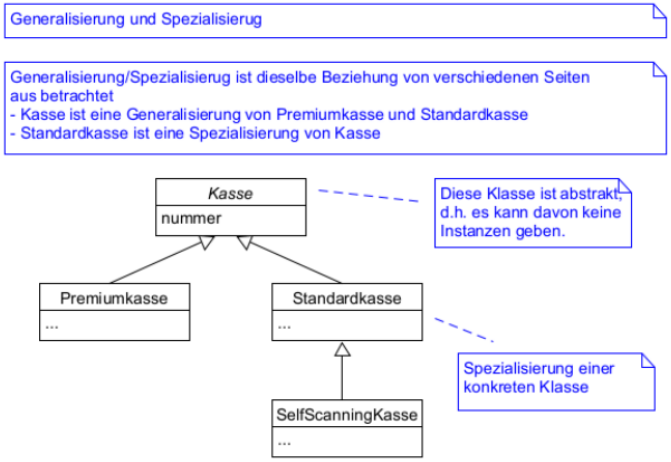
Mehr Info und Beispiele

Optionale Elemente im Domänenmodell

- Optional: Aggregationen/Kompositionen



- Optional: Generalisierung/Spezialisierung



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:

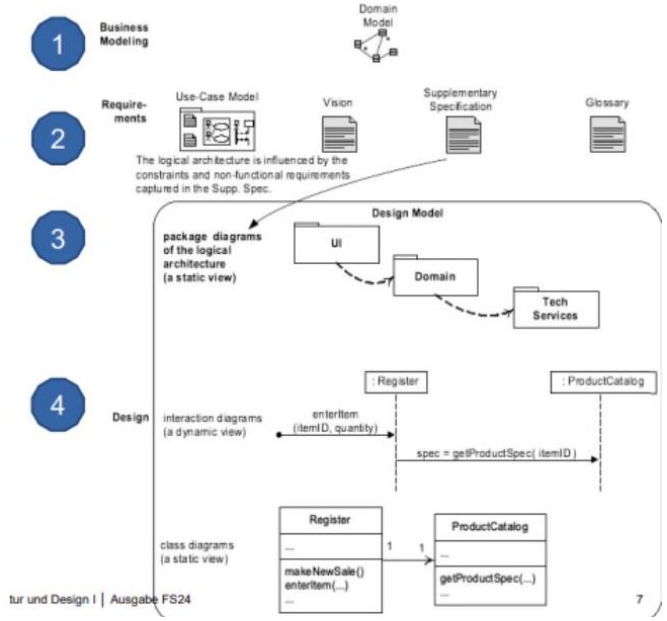
- **Beschreibungsklassen:**
  - Flugverbindung vs. konkreter Flug
  - Hotelkategorie vs. konkretes Zimmer
  - Aktivitätstyp vs. konkrete Durchführung
- **Zustände:**
  - Buchungszustände: angefragt, bestätigt, storniert
  - Zahlungszustände: offen, teilbezahlt, vollständig
- **Rollen:**
  - Person als: Kunde, Reiseleiter, Kontaktperson
- **Wertobjekte:**
  - Geldbetrag mit Währung
  - Zeitraum für Reisedauer

Softwarearchitektur und Design

Grundlagen der Softwarearchitektur

Grundlagen und Überblick

- **Business Analyse:**
  - Domänenmodell und Kontextdiagramm
  - Requirements (funktional und nicht-funktional)
  - Vision und Stakeholder
- **Architektur:**
  - Logische Struktur des Systems
  - Technische Konzeption
  - Qualitätsanforderungen
- **Entwicklung:**
  - Use Case / User Story Realisierung
  - Design-Klassendiagramm (DCD)
  - Implementierung und Tests



Softwarearchitektur

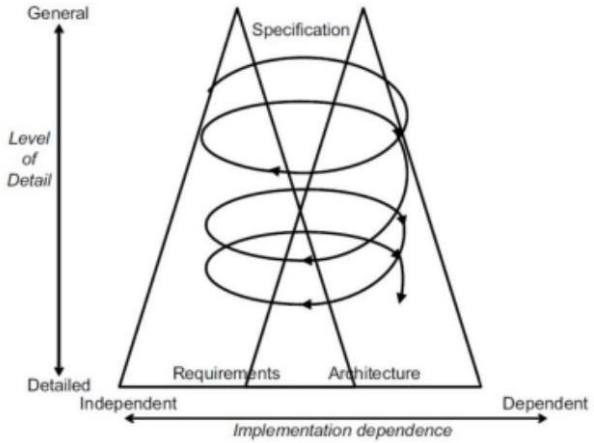
Die Architektur eines Softwaresystems definiert:

- **Grundlegende Entscheidungen:**
  - Programmiersprachen und Plattformen
  - Aufteilung in Teilsysteme und Komponenten
  - Schnittstellen zwischen Komponenten
- **Strukturelle Aspekte:**
  - Verantwortlichkeiten der Teilsysteme
  - Abhängigkeiten zwischen Komponenten
  - Einsatz von Basis-Technologien/Frameworks
- **Qualitätsaspekte:**
  - Erfüllung nicht-funktionaler Anforderungen
  - Maßnahmen für Performance, Skalierbarkeit etc.
  - Fehlertoleranz und Ausfallsicherheit

Architekturanalyse

erfolgt iterativ mit den Anforderungen (Twin Peaks Model):

- **Anforderungsanalyse:**
  - Analyse funktionaler und nicht-funktionaler Anforderungen
  - Prüfung der Qualität und Stabilität der Anforderungen
  - Identifikation von Lücken und impliziten Anforderungen
- **Architekturentscheidungen:**
  - Abstimmung mit Stakeholdern
  - Berücksichtigung von Randbedingungen
  - Vorausschauende Planung für zukünftige Änderungen



Qualitätsanforderungen

ISO 25010:

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

FURPS+:

- **F**unctionality (Funktionalität)
- **U**sability (Benutzerfreundlichkeit)
- **R**eliability (Zuverlässigkeit)
- **P**erformance (Leistung)
- **S**upportability (Wartbarkeit)
- **+**: Implementation, Interface, Operations, Packaging, Legal

Architekturdesign

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

Schnittstellen

Module kommunizieren über definierte Schnittstellen:

- **Exportierte Schnittstellen:**
  - Definieren angebotene Funktionalität
  - Vertraglich garantierte Leistungen
  - Einzige nach außen sichtbare Information
- **Importierte Schnittstellen:**
  - Von anderen Modulen benötigte Funktionalität
  - Definieren Abhängigkeiten
  - Basis für Kopplung
  - Sollten minimiert werden (Low Coupling)

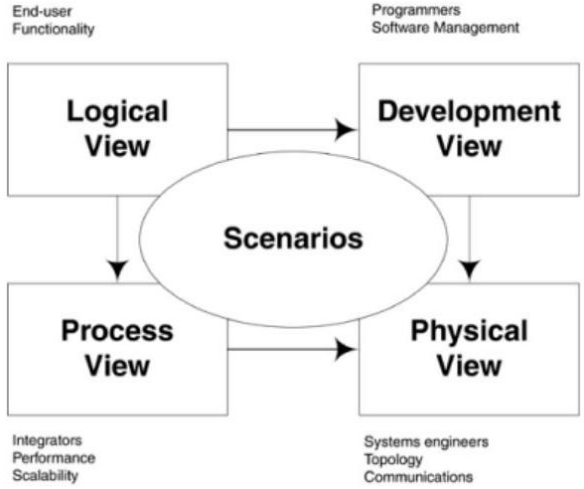
Architektursichten (4+1 View Model)

Verschiedene Perspektiven auf die Architektur:

- **Logical View:** End-User, Functionality
  - Funktionalität des Systems
  - Schichten, Subsysteme, Pakete
  - Klassen und Schnittstellen
- **Process View:** Integrators, Performance, Scalability
  - Laufzeitverhalten
  - Prozesse und Threads
  - Performance und Skalierung
- **Development View:** Programmers, Software Management
  - Implementierungsstruktur
  - Quellcode-Organisation
  - Build und Deployment
- **Physical View:** System Engineers, Topology, Communications
  - Hardware-Topologie
  - Verteilung der Software
  - Netzwerkkommunikation

+1: Scenarios:

- Wichtige Use Cases
- Validierung der Architektur
- Integration der anderen Views



Architekturprinzipien Grundlegende Prinzipien für gute Architektur:

Separation of Concerns:
- Trennung von Verantwortlichkeiten
- Klare Modulgrenzen
- Reduzierte Komplexität

Information Hiding:
- Kapselung von Implementierungsdetails
- Definierte Schnittstellen
- Änderbarkeit ohne Seiteneffekte

Loose Coupling:
- Minimale Abhängigkeiten
- Austauschbarkeit
- Unabhängige Entwicklung

Qualitätskriterien und deren Umsetzung

Strategien zur Erfüllung von Qualitätsanforderungen:

Performance:
- Effiziente Ressourcennutzung (Resource Pooling, Caching)
- Optimierte Verarbeitung (Parallelisierung, Lazy Loading)

Skalierbarkeit:
- Dynamische Anpassung (horizontale/vertikale Skalierung)
- Effiziente Lastverteilung (Load Balancing, Partitionierung)

Wartbarkeit:
- Klare Strukturen (Separation of Concerns, Modularisierung)
- Verbesserte Codequalität (Information Hiding, Standardisierung)

Zuverlässigkeit:
- Fehlerresistenz (Redundanz, Fehlertoleranz)
- Prävention und Wiederherstellung (Monitoring, Backup/Recovery)

Verfügbarkeit:
- Ausfallschutz (Redundanz, Failover-Mechanismen)
- Überwachung/Stabilisierung (Health Monitoring, Circuit Breaker)

Modularität:
- Gut definierte Grenzen (klare Modulgrenzen, hohe Kohäsion)
- Minimale Abhängigkeiten zwischen Modulen

Testbarkeit:
- Einfachheit von Tests (Isolation, Mockbarkeit)
- Automatisierung und Skalierung von Tests

Änderbarkeit:
- Anpassungsfähigkeit (Lokalisierung, Erweiterbarkeit)
- Sicherstellung der Kompatibilität (Backward Compatibility)

Erweiterbarkeit:
- Flexible Architekturen (offene Schnittstellen, Plugin-Systeme)
- Serviceorientierung für modulare Erweiterungen

Architekturprozess und Best Practices

Best Practices im Architekturentwurf

1. Analyse und Planung
- Anforderungen priorisieren
- Qualitätsziele definieren
- Constraints identifizieren
- Stakeholder einbinden

2. Design-Prinzipien
- Separation of Concerns
- Single Responsibility
- Information Hiding
- Don't Repeat Yourself (DRY)

3. Strukturierung
- Klare Schichtenarchitektur
- Definierte Schnittstellen
- Lose Kopplung
- Hohe Kohäsion

4. Dokumentation
- Architekturentscheidungen
- Begründungen
- Alternativen
- Trade-offs

Gesamter Architekturprozess

Architekturprozess-Komponenten

Architekturanalyse:
- Erster Schritt im Architekturprozess
- Analyse der funktionalen und nicht-funktionalen Anforderungen
- Identifikation von Qualitätszielen
- Parallel zur Anforderungserhebung (Twin Peaks)

Architektur-Entscheidungen:
- Konkrete Beschlüsse basierend auf der Analyse
- Technologiewahl und Strukturierung
- Dokumentation und Begründung
- Einschließlich verworfener Alternativen

Architektur-Entwurf:
- Praktischer Gestaltungsprozess
- Anwendung von Architekturmustern
- Umsetzung von Qualitätsanforderungen
- Erstellung konkreter Artefakte

Architektur-Review:
- Systematische Überprüfung
- Meist durch externe Experten
- Prüfung der Anforderungserfüllung
- Identifikation von Schwachstellen

Architektur-Evaluation:
- Bewertung anhand definierter Kriterien
- Quantitative und qualitative Analyse
- Szenario-basierte Prüfung
- Bewertung von Qualitätsattributen

Gesamter Architekturprozess

1. Initiale Phase
- Architekturanalyse durchführen
- Grundlegende Entscheidungen treffen
- Ersten Entwurf erstellen

2. Iterative Verfeinerung
- Review durchführen
- Evaluation vornehmen
- Anpassungen basierend auf Feedback

3. Kontinuierliche Verbesserung
- Regelmäßige Reviews
- Neue Anforderungen einarbeiten
- Technische Schulden adressieren

4. Dokumentation
- Entscheidungen festhalten
- Architektur dokumentieren
- Änderungen nachverfolgen

5. Qualitätssicherung
- Architektur-Konformität prüfen
- Performance-Tests durchführen
- Sicherheitsaudits durchführen

Gesamter Architekturprozess

Architekturanalyse

1. Anforderungen sammeln
- Funktionale Anforderungen gruppieren
- Nicht-funktionale Anforderungen identifizieren
- Randbedingungen dokumentieren

2. Qualitätsziele definieren
- Messbare Kriterien festlegen
- Priorisierung vornehmen
- Trade-offs identifizieren

3. Einflussfaktoren analysieren
- Technische Faktoren
- Organisatorische Faktoren
- Wirtschaftliche Faktoren

Architekturanalyse

Architektur-Entscheidungen

1. Alternativen identifizieren
- Mögliche Lösungen sammeln
- Vor- und Nachteile analysieren
- Machbarkeit prüfen

2. Bewertungskriterien
- Erfüllung der Anforderungen
- Technische Umsetzbarkeit
- Kosten und Aufwand

3. Entscheidung dokumentieren
- Begründung
- Konsequenzen
- Verworfene Alternativen

Architektur-Entscheidungen dokumentieren

1. Entscheidung festhalten
- Dokumentation der getroffenen Architekturentscheidungen
- Begründungen und Alternativen
- Auswirkungen und Konsequenzen

2. Strukturierte Dokumentation
- Einheitliches Format für alle Entscheidungen
- Verwendung von Templates
- Nachvollziehbare Historie der Entscheidungen

3. Kommunikation
- Regelmäßige Updates an Stakeholder
- Transparenz über getroffene Entscheidungen
- Einbindung des gesamten Teams

4. Review und Anpassung
- Regelmäßige Überprüfung der Entscheidungen
- Anpassung bei geänderten Rahmenbedingungen
- Lessons Learned dokumentieren

Architektur-Entscheidungen

Architekturentwurf

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

Architekturentwurf

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

Architektur-Review

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

Beispiele Architekturentwurf

Typische Prüfungsaufgabe: Architekturanalyse und Entscheidungen

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

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

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

```java
// Performance-optimierte Produktabfrage
@Cacheable(value = "products")
public ProductDTO getProduct(String id) {
    ProductDTO product = cache.get(id);
    if (product == null) {
        product = repository.findById(id)
                            .map(this::toDTO)
                            .orElseThrow();
        cache.put(id, product);
    }
    return product;
}
```

Architekturmuster und Clean Code

Architekturmuster

Grundlegende Architekturmuster für Software-Systeme:

- **Layered Pattern:**
  - Strukturierung in horizontale Schichten
  - Klare Trennung der Verantwortlichkeiten
  - Abhängigkeiten nur nach unten
- **Client-Server Pattern:**
  - Verteilung von Diensten
  - Zentralisierte Ressourcen
  - Mehrere Clients pro Server
- **Master-Slave Pattern:**
  - Verteilung von Aufgaben
  - Zentrale Koordination
  - Parallelverarbeitung
- **Pipe-Filter Pattern:**
  - Datenstromverarbeitung
  - Verkettung von Operationen
  - Wiederverwendbare Filter
- **Broker Pattern:**
  - Vermittlung zwischen Komponenten
  - Entkopplung von Diensten
  - Zentrale Koordination
- **Event-Bus Pattern:**
  - Asynchrone Kommunikation
  - Publisher-Subscriber Modell
  - Lose Kopplung

Event-Bus Pattern

Implementierung eines Event-Bus Systems:

1. Event-Bus

- Publisher-Subscriber Mechanismus implementieren
- Event-Routing einrichten
- Event-Persistenz berücksichtigen
- Ordering und Filtering ermöglichen

2. Publisher

- Event-Typen definieren
- Event-Publikation implementieren
- Transaktionshandling berücksichtigen
- Fehlerbehandlung vorsehen

3. Subscriber

- Event-Handler implementieren
- Idempotenz sicherstellen
- Fehlertoleranz einbauen
- Dead-Letter-Queue vorsehen

Event-Bus Implementation

```java
// Event Bus
public class EventBus {
    private Map<Class<?>, List<EventSubscriber>>
        subscribers = new HashMap<>();

    public void publish(Event event) {
        List<EventSubscriber> eventSubscribers =
            subscribers
            .getOrDefault(event.getClass(),
                Collections.emptyList());

        for (EventSubscriber subscriber :
            eventSubscribers) {
            try {
                subscriber.onEvent(event);
            } catch (Exception e) {
                handleSubscriberError(subscriber,
                    event, e);
            }
        }
    }
    public void subscribe(Class<?> eventType,
        EventSubscriber subscriber) {
        subscribers.computeIfAbsent(eventType, k ->
            new ArrayList<>())
                .add(subscriber);
    }
}
// Publisher
public class OrderService {
    private EventBus eventBus;

    public void createOrder(OrderRequest request) {
        Order order = orderRepository.save(
            new Order(request));

        eventBus.publish(new OrderCreatedEvent(
            order.getId(),
            order.getCustomerId(),
            order.getTotalAmount(),
            LocalDateTime.now()
        ));
    }
}
// Subscriber
@Component
public class NotificationService implements
    EventSubscriber {
    private ProcessedEventRepository processedEvents;

    @Override
    public void onEvent(Event event) {
        if (!(event instanceof OrderCreatedEvent))
            return;

        String eventId = event.getId();
        if (processedEvents.exists(eventId)) return;

        try {
            sendNotification((OrderCreatedEvent)
                event);
            processedEvents.save(eventId);
        } catch (Exception e) {
            sendToDeadLetterQueue(event, e);
        }
    }
}
```

Layered Pattern

Anwendung des Schichtenmusters:

1. Schichten identifizieren

- Präsentationsschicht (UI)
- Anwendungsschicht (Application Logic)
- Geschäftslogikschicht (Domain Logic)
- Datenzugriffsschicht (Data Access)

2. Regeln definieren

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

3. Schnittstellen festlegen

- Klare Service-Interfaces pro Schicht
- Dependency Injection für lose Kopplung
- Daten-DTOs für Schichtübergänge

Layered Pattern Implementation

```java
// Presentation Layer
public class CustomerController {
    private CustomerService service;

    public CustomerDTO getCustomer(String id) {
        Customer customer = service.findById(id);
        return CustomerDTO.from(customer);
    }
}

// Application Layer
public class CustomerService {
    private CustomerRepository repository;

    public Customer findById(String id) {
        validateId(id);
        return repository.findById(id)
        .orElseThrow(CustomerNotFoundException::new);
    }
}

// Domain Layer
public class Customer {
    private CustomerId id;
    private String name;
    private Address address;

    public void updateAddress(Address newAddress) {
        validateAddress(newAddress);
        this.address = newAddress;
    }
}

// Persistence Layer
public class CustomerRepository {
    private JpaRepository<Customer, CustomerId>
        jpaRepo;

    public Optional<Customer> findById(String id) {
        return jpaRepo.findById(new CustomerId(id));
    }
}
```

Client-Server Pattern

Implementierung einer Client-Server Architektur:

1. Server-Design
- Services definieren
- Schnittstellen dokumentieren
- Sicherheitsaspekte berücksichtigen
- Skalierbarkeit einplanen

2. Client-Design
- Client-Typen festlegen (Thin/Rich/Web)
- Fehlerbehandlung implementieren
- Offline-Fähigkeit berücksichtigen
- Caching-Strategie entwickeln

3. Kommunikation
- Protokoll wählen (REST, GraphQL, etc.)
- API-Versionierung einplanen
- Rate Limiting implementieren
- Monitoring einrichten

Client-Server Implementation

```
// Server-Side
@RestController
@RequestMapping("/api/v1")
public class ProductController {
    private final ProductService service;

    @GetMapping("/products/{id}")
    public ProductDTO getProduct(@PathVariable String
         id) {
        return service.findById(id)
            .map(ProductDTO::from)
            .orElseThrow(ProductNotFoundException::new);
    }
}

// Client-Side
public class ProductClient {
    private final WebClient webClient;

    public Mono<Product> getProduct(String id) {
        return webClient.get()
                        .uri("/products/{id}", id)
                        .retrieve()
                        .bodyToMono(ProductDTO.class)
                        .map(ProductDTO::toDomain)
                        .onErrorMap(this::handleError);
    }
}
```

Master-Slave Pattern

Implementierung einer Master-Slave Architektur:

1. Master-Komponente
- Aufgabenverteilung implementieren
- Slave-Management einrichten
- Fehlertoleranz berücksichtigen
- Load Balancing implementieren

2. Slave-Komponenten
- Aufgabenbearbeitung implementieren
- Statusmeldungen einrichten
- Fehlerbehandlung implementieren
- Recovery-Mechanismen vorsehen

3. Koordination
- Kommunikationsprotokoll definieren
- Synchronisation implementieren
- Monitoring einrichten
- Failover-Strategien entwickeln

Master-Slave Implementation

```
// Master
public class TaskMaster {
    private List<SlaveWorker> workers;
    private Queue<Task> taskQueue;

    public void distributeTask(Task task) {
        SlaveWorker worker = selectAvailableWorker();
        worker.assignTask(task);
        monitorProgress(worker);
    }

    private void handleWorkerFailure(SlaveWorker
         worker) {
        Task failedTask = worker.getCurrentTask();
        reassignTask(failedTask,
             getNextAvailableWorker());
    }
}

// Slave
public class SlaveWorker {
    private TaskProcessor processor;
    private WorkerStatus status;

    public void assignTask(Task task) {
        status = WorkerStatus.BUSY;
        try {
            Result result = processor.process(task);
            reportSuccess(result);
        } catch (Exception e) {
            reportFailure(e);
        }
        status = WorkerStatus.IDLE;
    }
}
```

Pipe-Filter Pattern

Implementierung einer Pipe-Filter Architektur:

1. Filter-Design
- Atomare Transformationen definieren
- Ein- und Ausgabeformat festlegen
- Fehlerbehandlung implementieren
- Unabhängige Verarbeitung sicherstellen

2. Pipe-Design
- Datentransfer implementieren
- Pufferung einrichten
- Threading-Modell festlegen
- Backpressure berücksichtigen

3. Pipeline-Konfiguration
- Filter-Reihenfolge definieren
- Verzweigungen ermöglichen
- Monitoring einrichten
- Fehlerszenarien behandeln

Pipe-Filter Implementation

```
// Filter Interface
public interface Filter<I, O> {
    O process(I input);
}

// Concrete Filter
public class ValidationFilter implements Filter<Data,
     Data> {
    @Override
    public Data process(Data input) {
        if (!isValid(input)) {
            throw new ValidationException("Invalid
                 data");
        }
        return input;
    }
}

// Pipe Implementation
public class Pipeline<I, O> {
    private List<Filter> filters = new ArrayList<>();

    public void addFilter(Filter filter) {
        filters.add(filter);
    }

    public O process(I input) {
        Object current = input;
        for (Filter filter : filters) {
            current = filter.process(current);
        }
        return (O) current;
    }
}

// Usage
Pipeline<RawData, ProcessedData> pipeline = new
     Pipeline<>();
pipeline.addFilter(new ValidationFilter());
pipeline.addFilter(new TransformationFilter());
pipeline.addFilter(new EnrichmentFilter());
ProcessedData result = pipeline.process(rawData);
```

Broker Pattern

Implementierung eines Broker Systems:

1. Broker-Komponente

- Service-Registry implementieren
- Request-Routing einrichten
- Load Balancing implementieren
- Service-Discovery ermöglichen

2. Service-Provider

- Service-Interface definieren
- Bei Broker registrieren
- Health Checks implementieren
- Fehlerbehandlung vorsehen

3. Service-Consumer

- Service-Lookup implementieren
- Fehlertoleranz einbauen
- Caching-Strategie entwickeln
- Retry-Mechanismen vorsehen

Broker Implementation

```java
public class ServiceBroker {    // Broker
    private Map<String, List<ServiceProvider>>
        services = new HashMap<>();

    public void register(ServiceProvider provider) {
        String serviceType = provider.getServiceType();
        services.computeIfAbsent(serviceType, k -> new
            ArrayList<>())
                .add(provider);
    }
    public ServiceProvider getProvider(String
        serviceType) {
        List<ServiceProvider> providers =
            services.get(serviceType);
        if (providers == null || providers.isEmpty()) {
            throw new
                ServiceNotFoundException(serviceType);
        }
        return selectProvider(providers); // Load
            balancing
    }
}
public class ServiceProvider {  // Service Provider
    private String serviceType;
    private String endpoint;
    private HealthCheck healthCheck;
    public Response handleRequest(Request request) {
        try {
            validateRequest(request);
            return processRequest(request);
        } catch (Exception e) {
            return handleError(e);
        }
    }
}
public class ServiceConsumer {  // Service Consumer
    private ServiceBroker broker;
    private Cache cache;
    public Response callService(String serviceType,
        Request request) {
        Response cached = cache.get(request);
        if (cached != null) return cached;
        return RetryTemplate.execute(() -> {
            ServiceProvider provider =
                broker.getProvider(serviceType);
            Response response =
                provider.handleRequest(request);
            cache.put(request, response);
            return response;
        });
    }
}
```

Clean Architecture

Clean Architecture

Architektur-Prinzipien nach Robert C. Martin:

Hauptprinzipien:

- Unabhängigkeit von Frameworks
- Unabhängigkeit von UI
- Unabhängigkeit von Datenbank
- Testbarkeit ohne externe Systeme

Schichten (von innen nach außen):

- **Entities:**
  - Zentrale Geschäftsregeln
  - Unternehmensweit gültig
  - Höchste Stabilität
- **Use Cases:**
  - Anwendungsspezifische Geschäftsregeln
  - Orchestrierung der Entities
  - Anwendungslogik
- **Interface Adapters:**
  - Konvertierung von Daten
  - Präsentation und Controller
  - Gateway-Implementierungen
- **Frameworks & Drivers:**
  - UI-Framework
  - Datenbank
  - Externe Schnittstellen

Clean Architecture

Implementierung der Clean Architecture:

1. Schichten definieren

- **Entities:**
  - Zentrale Geschäftsobjekte identifizieren
  - Geschäftsregeln definieren
  - Unabhängig von externen Frameworks
- **Use Cases:**
  - Anwendungsfälle implementieren
  - Geschäftslogik orchestrieren
  - Input/Output Boundaries definieren
- **Interface Adapters:**
  - Controller implementieren
  - Presenter erstellen
  - Gateways definieren
- **Frameworks & Drivers:**
  - UI-Framework einbinden
  - Datenbankzugriff implementieren
  - Externe Services anbinden

2. Dependency Rule beachten

- Abhängigkeiten nur nach innen
- Interfaces für Richtungsumkehr
- DTOs für Datentransfer

3. Clean Architecture Testing

- Unit Tests für Entities
- Use Case Tests ohne externe Systeme
- Integrationstests für Adapter
- End-to-End Tests für das Gesamtsystem

Clean Architecture Implementation

```java
// Entity Layer
public class Order {
    private OrderId id;
    private CustomerId customerId;
    private Money totalAmount;
    private OrderStatus status;

    public void confirm() {
        validateStateForConfirmation();
        this.status = OrderStatus.CONFIRMED;
    }

    private void validateStateForConfirmation() {
        if (status != OrderStatus.PENDING) {
            throw new InvalidOrderStateException();
        }
    }
}

// Use Case Layer
public class CreateOrderUseCase implements CreateOrder
    {
    private final OrderRepository orderRepository;
    private final PaymentGateway paymentGateway;

    @Override
    public CreateOrderResponse
        execute(CreateOrderRequest request) {
        // Business logic
        Order order = new Order(
            request.getCustomerId(),
            request.getAmount()
        );

        // Business rules validation
        validateOrder(order);

        // Store order
        orderRepository.save(order);

        // Process payment
        PaymentResult payment = paymentGateway.process(
            order.getId(),
            order.getTotalAmount()
        );

        // Return response
        return new CreateOrderResponse(
            order.getId(),
            payment.getTransactionId()
        );
    }
}

// Interface Adapters Layer
@RestController
public class OrderController {
    private final CreateOrder createOrderUseCase;

    @PostMapping("/orders")
    public ResponseEntity<OrderResponse> createOrder(
            @RequestBody OrderRequest request) {
        // Convert request to use case input
        CreateOrderRequest useCaseRequest =
            mapToUseCaseRequest(request);

        // Execute use case
        CreateOrderResponse response =
```

```
            createOrderUseCase.execute(useCaseRequest);

        // Convert and return response
        return ResponseEntity.ok(
            mapToApiResponse(response));
    }
}

// Frameworks & Drivers Layer
@Repository
public class JpaOrderRepository implements
     OrderRepository {
    private final JpaOrderEntityRepository jpaRepo;

    @Override
    public void save(Order order) {
        OrderEntity entity = mapToEntity(order);
        jpaRepo.save(entity);
    }

    @Override
    public Optional<Order> findById(OrderId id) {
        return jpaRepo.findById(id.getValue())
                      .map(this::mapToDomain);
    }
}
```

Clean Architecture Best Practices

1. Separation of Concerns
- Jede Schicht hat klare Verantwortlichkeit
- Geschäftslogik unabhängig von Frameworks
- Klare Grenzen zwischen Schichten

2. Dependency Management
- Dependency Injection verwenden
- Interfaces für Richtungsumkehr
- Minimale externe Abhängigkeiten

3. Testbarkeit
- Unabhängiges Testen der Schichten
- Mocking von externen Abhängigkeiten
- Automatisierte Tests auf allen Ebenen

4. Code Organisation
- Package by Feature
- Klare Modulstruktur
- Explizite Abhängigkeiten

Objektorientiertes Design

GRASP Prinzipien

General Responsibility Assignment Software Patterns - Grundlegende Prinzipien für die Zuweisung von Verantwortlichkeiten:

Information Expert:
- Zuständigkeit basierend auf Information
- Klasse mit relevanten Daten übernimmt Aufgabe
- Fördert Kapselung und Kohäsion

Creator:
- Verantwortung für Objekterstellung
- Basierend auf Beziehungen (enthält, aggregiert)
- Starke Verwendungsbeziehung

Controller:
- Koordination von Systemoperationen
- Erste Anlaufstelle nach UI
- Fassade für Subsystem

Low Coupling:
- Minimale Abhängigkeiten
- Erhöht Wiederverwendbarkeit
- Erleichtert Änderungen

High Cohesion:
- Fokussierte Verantwortlichkeiten
- Zusammengehörige Funktionalität
- Wartbare Klassen

Polymorphism:
- Verhaltensänderungen durch Vererbung
- Type-dependent behavior
- Alternative zu if/else Ketten

Pure Fabrication:
- Hilfsklassen für besseres Design
- Keine direkte Domänenentsprechung
- Unterstützt High Cohesion

Indirection:
- Vermittler für lose Kopplung
- Intermediate object
- Reduziert direkte Abhängigkeiten

Protected Variations:
- Kapselung von Änderungen
- Interface für Variation Points
- Stabilität bei Änderungen

GRASP Anwendung

```
public class Order {        // Information Expert
    private List<OrderLine> lines;

    public Money calculateTotal() {
        return lines.stream()
                    .map(OrderLine::getSubtotal)
                    .reduce(Money.ZERO, Money::add);
    }
}
// Creator
public class Order {
    public void addProduct(Product product, int
         quantity) {
        OrderLine line = new OrderLine(product,
             quantity);
        lines.add(line);
    }
}
// Controller
public class OrderController {
    private OrderService service;

    public OrderResponse createOrder(OrderRequest
         request) {
        Order order = service.createOrder(request);
        return OrderResponse.from(order);
    }
}
// Low Coupling & High Cohesion
public interface PaymentGateway {
    PaymentResult processPayment(Money amount);
}
public class StripePaymentGateway implements
     PaymentGateway {
    public PaymentResult processPayment(Money amount) {
        // Stripe-spezifische Implementation
    }
}
// Polymorphism
public interface DiscountStrategy {
    Money calculateDiscount(Order order);
}
public class VolumeDiscountStrategy implements
     DiscountStrategy {
    public Money calculateDiscount(Order order) {
        // Mengenrabatt Berechnung
    }
}
// Pure Fabrication
public class OrderNumberGenerator {
    public String generateOrderNumber() {
        // Generiert eindeutige Bestellnummer
    }
}
// Indirection
public class PaymentService {
    private PaymentGateway gateway;
    public PaymentResult processPayment(Order order) {
        return
            gateway.processPayment(order.getTotal());
    }
}
// Protected Variations
public interface OrderRepository {
    void save(Order order);
    Optional<Order> findById(OrderId id);
}
```

Grundlagen des RDD

Design basierend auf Verantwortlichkeiten und Kollaborationen:

Verantwortlichkeiten:

- **Doing:**
  - Aktionen ausführen
  - Berechnungen durchführen
  - Andere Objekte steuern
  - Aktivitäten koordinieren
- **Knowing:**
  - Eigene Daten kennen
  - Verwandte Objekte kennen
  - Berechnete Informationen
  - Private enkapsulierte Daten

Kollaborationen:

- Klare Rollen definieren
- Aufgaben verteilen
- Interfaces abstimmen
- Verantwortlichkeiten zuweisen

RDD Anwendung

1. Verantwortlichkeiten identifizieren

- Systemoperationen analysieren
- Notwendige Aktionen auflisten
- Benötigte Daten identifizieren
- Abhängigkeiten erkennen

2. Rollen definieren

- Klassen nach Verantwortlichkeiten gruppieren
- Schnittstellen festlegen
- Kollaborationen planen
- GRASP Prinzipien anwenden

3. Implementierung

- Interfaces definieren
- Klassen implementieren
- Kollaborationen umsetzen
- Tests schreiben

Best Practices im RDD

1. Klare Verantwortlichkeiten

- Eine Hauptverantwortung pro Klasse
- Logisch zusammenhängende Aufgaben
- Überschaubare Klassengröße

2. Effektive Kollaborationen

- Minimale Abhängigkeiten
- Klare Schnittstellen
- Wiederverwendbare Komponenten

3. Wartbarkeit

- Testbare Komponenten
- Dokumentierte Verantwortlichkeiten
- Erweiterbare Struktur

RDD Implementierung

```java
// Doing Responsibility: Bestellungsverarbeitung
public class OrderProcessor {
    private final InventoryService inventory;
    private final PaymentService payment;

    public OrderResult processOrder(Order order) {
        // Koordination der Verarbeitung
        validateOrder(order);
        reserveInventory(order);
        processPayment(order);
        return createResult(order);
    }

    private void validateOrder(Order order) {
        if (order.isEmpty()) {
            throw new EmptyOrderException();
        }
    }
}

// Knowing Responsibility: Bestellungsinformationen
public class Order {
    private OrderId id;
    private CustomerId customerId;
    private List<OrderLine> lines;
    private OrderStatus status;

    // Kennt seine eigenen Daten
    public Money calculateTotal() {
        return lines.stream()
                    .map(OrderLine::getSubtotal)
                    .reduce(Money.ZERO, Money::add);
    }

    public boolean isEmpty() {
        return lines.isEmpty();
    }
}

// Kollaboration zwischen Objekten
public class OrderService {
    private final OrderProcessor processor;
    private final OrderRepository repository;
    private final OrderNotifier notifier;

    public OrderResult createOrder(OrderRequest
         request) {
        // Orchestrierung der Kollaboration
        Order order = createOrderFromRequest(request);
        OrderResult result =
             processor.processOrder(order);
        repository.save(order);
        notifier.notifyCustomer(order);
        return result;
    }
}
```

Design Patterns im Architekturkontext

Architektur-relevante Patterns:

- **Structural Patterns:**
  - Facade für Subsystem-Zugriff
  - Adapter für System-Integration
  - Proxy für Remote-Zugriff
  - Bridge für Implementierungs-Entkopplung
- **Behavioral Patterns:**
  - Observer für Event-Handling
  - Command für Service-Aufrufe
  - Strategy für austauschbare Algorithmen
  - Template Method für Framework-Hooks
- **Creational Patterns:**
  - Factory Method für Komponenten-Erstellung
  - Abstract Factory für Familien von Komponenten
  - Builder für komplexe Objektkonstruktion
  - Singleton für zentrale Dienste

Architektur Pattern Implementation

```java
public class OrderFacade {  // Facade Pattern fuer
     Subsystem
    private OrderService orderService;
    private PaymentService paymentService;
    private ShippingService shippingService;
    // Vereinfachter Zugriff auf Subsystem
    public OrderResult processOrder(OrderRequest
         request) {
        Order order =
             orderService.createOrder(request);
        Payment payment =
             paymentService.processPayment(order);
        Shipment shipment =
             shippingService.arrangeShipment(order);
        return new OrderResult(order, payment,
             shipment);
    }
}
// Adapter Pattern fuer Legacy-System
public class LegacySystemAdapter implements
     ModernSystemInterface {
    private LegacySystem legacySystem;
    @Override
    public ModernResponse process(ModernRequest
         request) {
        LegacyRequest legacyRequest =
             convertRequest(request);
        LegacyResponse legacyResponse =
            legacySystem.processRequest(legacyRequest);
        return convertResponse(legacyResponse);
    }
}
// Strategy Pattern fuer verschiedene Implementierungen
public interface PaymentStrategy {
    PaymentResult processPayment(Order order);
}
public class StripePaymentStrategy implements
     PaymentStrategy {
    @Override   // Stripe-spezifische Implementation
    public PaymentResult processPayment(Order order) {
    }
}
public class PayPalPaymentStrategy implements
     PaymentStrategy {
    @Override   // PayPal-spezifische Implementation
    public PaymentResult processPayment(Order order) {
    }
}
```

Häufige Anti-Patterns

1. Big Ball of Mud
- Keine klare Struktur
- Vermischung von Zuständigkeiten
- Schwer wartbar und erweiterbar

2. Golden Hammer
- Ein Pattern/Tool für alle Probleme
- Ignorieren besserer Alternativen
- Übermäßige Komplexität

3. Spaghetti Code
- Unstrukturierter Code
- Keine klaren Abhängigkeiten
- Schwer zu verstehen und zu ändern

4. God Class
- Zu viele Verantwortlichkeiten
- Verletzt Single Responsibility
- Schwer zu testen und zu warten

5. Lava Flow
- Veralteter, ungenutzter Code
- Niemand traut sich zu löschen
- Erhöht Komplexität unnötig

Anti-Pattern Erkennung und Vermeidung

1. Code-Review Checkliste
- Single Responsibility prüfen
- Abhängigkeiten analysieren
- Testbarkeit bewerten
- Dokumentation prüfen

2. Refactoring-Strategien
- Klassen aufteilen
- Verantwortlichkeiten extrahieren
- Interfaces einführen
- Tests schreiben

3. Präventive Maßnahmen
- Design Reviews durchführen
- Architektur-Guidelines definieren
- Code-Qualität messen
- Kontinuierliches Refactoring

Anti-Pattern Refactoring

```
// Before: God Class
public class OrderManager {
    private Connection dbConnection;

    public void createOrder(OrderData data) {
        // Validation
        if (data.getCustomerId() == null) throw new
            Error("No customer");
        if (data.getItems().isEmpty()) throw new
            Error("No items");

        // Database operations
        String sql = "INSERT INTO orders...";
        // ... 50 lines of SQL and JDBC code ...

        // Payment processing
        // ... 100 lines of payment processing ...

        // Email notification
        // ... 50 lines of email sending code ...
    }
}

// After: Separated Responsibilities
public class OrderService {
    private final OrderValidator validator;
    private final OrderRepository repository;
    private final PaymentService paymentService;
    private final NotificationService
        notificationService;

    public OrderResult createOrder(OrderRequest
        request) {
        // Validate
        validator.validate(request);

        // Create and save order
        Order order = new Order(request);
        repository.save(order);

        // Process payment
        PaymentResult payment =
            paymentService.processPayment(order);

        // Notify customer
        notificationService.sendOrderConfirmation(order);

        return new OrderResult(order, payment);
    }
}

public class OrderValidator {
    public void validate(OrderRequest request) {
        if (request.getCustomerId() == null) {
            throw new ValidationException("No
                customer");
        }
        if (request.getItems().isEmpty()) {
            throw new ValidationException("No items");
        }
    }
}
```

UML-Modellierung

Grundlagen der UML-Modellierung

UML (Unified Modeling Language) wird im Design auf zwei Arten verwendet:

Statische Modelle:
- Struktur des Systems
- Klassendiagramme, Paketdiagramme
- Fokus auf Pakete, Klassen, Attribute
- Keine Methodenimplementierung

Dynamische Modelle:
- Verhalten des Systems
- Sequenz-, Zustands-, Aktivitätsdiagramme
- Fokus auf Logik und Verhalten
- Methodenimplementierung

UML Diagrammtypen

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

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

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

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

UML Diagrammauswahl

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

Klassendiagramm: E-Commerce System

```java
public interface OrderRepository {
    Optional<Order> findById(OrderId id);
    void save(Order order);
}

public class Order {
    private OrderId id;
    private Customer customer;
    private List<OrderLine> orderLines;
    private OrderStatus status;

    public Money calculateTotal() {
        return orderLines.stream()
                         .map(OrderLine::getSubTotal)
                         .reduce(Money.ZERO,
                              Money::add);
    }
}

public class OrderLine {
    private Product product;
    private int quantity;
    private Money price;

    public Money getSubTotal() {
        return price.multiply(quantity);
    }
}
```

Sequenzdiagramm: Bestellprozess

```java
public class OrderService {
    private final OrderRepository orderRepo;
    private final PaymentService paymentService;

    public OrderConfirmation processOrder(OrderRequest
         request) {
        // Validiere Bestellung
        validateOrder(request);

        // Erstelle Order
        Order order = createOrder(request);
        orderRepo.save(order);

        // Prozessiere Zahlung
        PaymentResult result = paymentService
            .processPayment(order.getId(),
                 order.getTotal());

        // Bestaetige Bestellung
        if (result.isSuccessful()) {
            order.confirm();
            orderRepo.save(order);
            return new OrderConfirmation(order);
        }

        throw new PaymentFailedException();
    }
}
```

Zustandsdiagramm: Bestellstatus

```java
public interface OrderState {
    void process(Order order);
    void cancel(Order order);
    void ship(Order order);
}

public class NewOrderState implements OrderState {
    @Override
    public void process(Order order) {
        validateOrder(order);
        order.setState(new ProcessingState());
    }
    @Override
    public void cancel(Order order) {
        order.setState(new CancelledState());
    }
    @Override
    public void ship(Order order) {
        throw new IllegalStateException(
            "Cannot ship new order");
    }
}

public class Order {
    private OrderState state;

    public void process() {
        state.process(this);
    }
    void setState(OrderState newState) {
        this.state = newState;
    }
}
```

Aktivitätsdiagramm: Bestellabwicklung

```java
public class OrderProcessor {
    public void processOrder(Order order) {
        // Parallele Verarbeitung
        CompletableFuture.allOf(
            validateInventory(order),
            validatePayment(order)
        ).thenRun(() -> {
            if (order.isValid()) {
                fulfillOrder(order);
            } else {
                handleValidationFailure(order);
            }
        });
    }
    private CompletableFuture<Void> validateInventory(
            Order order) {
        return CompletableFuture.runAsync(() -> {
            order.getItems().forEach(item -> {
                if
                    (!inventoryService.isAvailable(item))
                    {
                    throw new
                        OutOfStockException(item);
                }
            });
        });
    }
}
```

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

```
public class ShoppingCart {
    private List<CartItem> items;
    private CheckoutService checkoutService;

    public Order checkout() {
        // 1: validateItems()
        validateItems();

        // 2: calculateTotal()
        Money total = calculateTotal();

        // 3: createOrder(items, total)
        Order order = checkoutService.createOrder(
            items, total);

        // 4: clearCart()
        items.clear();

        return order;
    }
}
```

Verteilungsdiagramm

Elemente:

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

Verteilungsdiagramm: Microservice-Architektur

```
@Configuration
public class ServiceConfig {
    @Value("${service.host}")
    private String serviceHost;

    @Value("${service.port}")
    private int servicePort;

    @Bean
    public ServiceRegistry registry() {
        return ServiceRegistry.builder()
            .host(serviceHost)
            .port(servicePort)
            .healthCheck("/health")
            .build();
    }

    @Bean
    public LoadBalancer loadBalancer(
            ServiceRegistry registry) {
        return new RoundRobinLoadBalancer(registry);
    }
}
```

Best Practices für UML-Modellierung

1. Allgemeine Richtlinien

- Nur relevante Details zeigen
- Konsistente Notation verwenden
- Diagramme dokumentieren
- Lesbarkeit priorisieren

2. Diagrammspezifische Richtlinien

- Klassendiagramm: Wichtige Beziehungen hervorheben
- Sequenzdiagramm: Kritische Interaktionen zeigen
- Zustandsdiagramm: Komplexe Zustände hierarchisch strukturieren
- Aktivitätsdiagramm: Parallelität klar darstellen

3. Tooling

- UML-Tool auswählen
- Versionskontrolle einsetzen
- Templates definieren
- Review-Prozess etablieren

Use Case Realization

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

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

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

Vorgehen bei der Use Case Realization

1. Vorbereitung:

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

2. Analyse:

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

3. Realisierung:

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

Use Case Realization 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

Diagramme und Modelle in der Use Case Realization

UML im Implementierungsprozess

UML dient als:

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

System Sequence Diagrams (SSD)

System Sequence Diagrams (SSD) erstellen

Interaction Diagrams in der Use Case Realization

Sequenzdiagramm für enterItem():

```
:Register -> :ProductCatalog: getDescription(itemId)
:ProductCatalog --> :Register: desc
:Register -> currentSale: makeLineItem(desc, quantity)
currentSale -> :SalesLineItem: create(desc, quantity)
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

Design Class Diagram (DCD)

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

Design Class Diagram

GRASP in der Use Case Realization

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

GRASP Prinzipien in der Use Case Realization

GRASP-basierte Implementation

Szenario: Implementierung einer Bestellverwaltung mit GRASP-Prinzipien

Information Expert:

```
public class Order {
    private List<OrderItem> items = new ArrayList<>();

    public BigDecimal calculateTotal() {
        return items.stream()
                    .map(OrderItem::getSubtotal)
                    .reduce(BigDecimal.ZERO,
                         BigDecimal::add);
    }
}
```

Creator:

```
public class Order {
    public OrderItem createOrderItem(Product product,
         int quantity) {
        OrderItem item = new OrderItem(product,
             quantity);
        items.add(item);
        return item;
    }
}
```

Controller:

```
public class OrderController {
    private OrderService orderService;

    public OrderDTO createOrder(String customerId) {
        Order order =
             orderService.initializeOrder(customerId);
        return OrderMapper.toDTO(order);
    }
}
```

Implementierung und Prüfung

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

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:

```
public class OrderController {
    private OrderService orderService;
    private Order currentOrder;

    public void createOrder() {
        currentOrder = orderService.createOrder();
    }

    public void addItem(String productId, int
         quantity) {
        currentOrder.addItem(productId, quantity);
    }

    public void submitOrder() {
        orderService.submitOrder(currentOrder);
    }
}
```

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

Typische Implementierungsfehler

1. Verletzung von GRASP-Prinzipien

```
// Falsch: Information Expert verletzt
public class Register {
    public BigDecimal calculateTotal(Sale sale) {
        // Register berechnet Total statt Sale
        return sale.getItems().stream()
                   .map(item -> item.getPrice())
                   .reduce(BigDecimal.ZERO,
                        BigDecimal::add);
    }
}
// Richtig: Sale ist Information Expert
public class Sale {
    public BigDecimal getTotal() {
        return items.stream()
                    .map(item -> item.getSubtotal())
                    .reduce(BigDecimal.ZERO,
                         BigDecimal::add);
    }
}
```

2. Architekturverletzungen

```
// Falsch: Domain-Objekt mit UI-Abhaengigkeit
public class Sale {
    private JFrame frame;
    public void complete() {
        // Domain-Logik vermischt mit UI
        frame.showMessage("Sale completed");
    }
}
// Richtig: Trennung der Schichten
public class Sale {
    public void complete() {
        // Reine Domain-Logik
        this.status = SaleStatus.COMPLETED;
        this.completionTime = LocalDateTime.now();
    }
}
```

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

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:

```
// Vor Refactoring
public class Sale {
    public void complete() {
        BigDecimal total = BigDecimal.ZERO;
        for(SaleLineItem item : items) {
            total = total.add(item.getPrice()
                    .multiply(new
                        BigDecimal(item.getQuantity())));
        }
        this.total = total;
        this.status = "COMPLETED";
        // ... weitere 20 Zeilen Code
    }
}
// Nach Refactoring
public class Sale {
    public void complete() {
        calculateTotal();
        updateStatus();
        generateReceipt();
        notifyInventory();
    }
    private void calculateTotal() {
        this.total = items.stream()
            .map(SaleLineItem::getSubtotal)
            .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
    // ... weitere private Methoden
}
```

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

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:

```
public class CartController {
    private CartValidator validator;
    private CheckoutService checkoutService;

    public CheckoutResult prepareCheckout(String
         cartId) {
        ShoppingCart cart = findCart(cartId);
        // Validierung (Information Expert)
        ValidationResult result =
             validator.validate(cart);
        if (!result.isValid()) {
            throw new
                 ValidationException(result.getErrors());
        }
        // Checkout vorbereiten
        String checkoutUrl =
             checkoutService.initiate(cart);
        return new CheckoutResult(checkoutUrl);
    }
}
```

4. Tests:

```
@Test
public void shouldPrepareCheckoutForValidCart() {
    // Given
    ShoppingCart cart = createValidCart();
    when(validator.validate(cart))
        .thenReturn(ValidationResult.valid());
    when(checkoutService.initiate(cart))
        .thenReturn("https://checkout/123");
    // When
    CheckoutResult result =
         controller.prepareCheckout(cart.getId());
    // Then
    assertNotNull(result.getCheckoutUrl());
    verify(checkoutService).initiate(cart);
}
```

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

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

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

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

Anwendung und Implementation der Design Patterns

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

Pattern-Analyse für Prüfung

Systematisches Vorgehen:

1. **Problem identifizieren und analysieren**
   - Art des Problems identifizieren
   - Anforderungen klar definieren
   - Welche Flexibilität wird benötigt?
   - Welche Einschränkungen gibt es?
   - Kontext verstehen
2. **Pattern auswählen und evaluieren**
   - Passende Patterns suchen
   - Welche Patterns lösen ähnliche Probleme?
   - Wie unterscheiden sich die Patterns?
   - Welche Trade-offs gibt es? (Komplexität, Flexibilität)
3. **Lösung skizzieren**
   - Klassenstruktur entwerfen
   - Beziehungen/Schnittstellen definieren
   - Vor- und Nachteile nennen
   - Anpassungen für spezifisches Problem

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

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 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

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:

```java
// Abstract Factory
public interface DocumentFactory {
    Document createDocument();
    Storage createStorage();
}
// Strategy
public interface ProcessingStrategy {
    void process(Document doc);
}
// Command
public interface DocumentCommand {
    void execute();
    void undo();
}
// Adapter
public class CloudStorageAdapter implements Storage {
    private CloudService service;

    public void save(Document doc) {
        service.upload(doc.getBytes());
    }
}
```

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

Exam-Style Pattern Recognition

Aufgabe: Analysieren Sie den folgenden Code und identifizieren Sie verwendete Patterns:

```java
public interface PaymentStrategy {
    void pay(int amount);
}

public class CreditCardPayment implements
     PaymentStrategy {
    private String cardNum;

    public void pay(int amount) {
        // Process credit card payment
    }
}

public class PayPalPayment implements PaymentStrategy {
    private String email;

    public void pay(int amount) {
        // Process PayPal payment
    }
}

public class ShoppingCart {
    private PaymentStrategy paymentMethod;

    public void setPaymentMethod(PaymentStrategy
         method) {
        this.paymentMethod = method;
    }

    public void checkout(int amount) {
        paymentMethod.pay(amount);
    }
}
```

Lösung:

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

Implementation, Refactoring und Testing

Von Design zu Code

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

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

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 (Externes Verhalten bleibt gleich)
- Verbesserung der Codequalität und interner Struktur
- Ziel: Bessere Wartbarkeit und Erweiterbarkeit

Code Smells

Anzeichen für mögliche Probleme im Code:
- **Duplizierter Code:** Gleicher Code an mehreren Stellen
- **Lange Methoden:** Methoden mit zu vielen Verantwortlichkeiten
- **Große Klassen:** Klassen mit vielen Instanzvariablen
- **Auffällig ähnliche Unterklassen:** Potential für Abstraktion
- **Hohe Kopplung:** Zu viele Abhängigkeiten zwischen Klassen
- **Keine Interfaces:** Mangelnde Abstraktion

Laufzeit-Optimierung

Grundregeln:
- Zuerst messen, dann optimieren
- Performance-Profile nutzen
- Bottlenecks identifizieren

Häufige Probleme:
- Datenbank-Zugriffe
- Ineffiziente Algorithmen
- Speicherlecks

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

Refactoring Patterns

1. Extract Method
- Code in eigene Methode auslagern
- Verbessert Lesbarkeit und Wiederverwendbarkeit
- Reduziert Duplikation

2. Move Method/Field
- Methode/Feld in andere Klasse verschieben
- Verbessert Kohäsion
- Folgt Information Expert

3. Extract Class
- Teil einer Klasse in neue Klasse auslagern
- Trennt Verantwortlichkeiten
- Erhöht Kohäsion

4. Rename Method/Class/Variable
- Bessere Namen für besseres Verständnis
- Dokumentiert Zweck
- Erleichtert Wartung

Typische Prüfungsaufgabe: Code Smells

Analysieren Sie folgenden Code auf Code Smells:

Problematischer Code:
- Klasse ÜserManager"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

Refactoring: Extract Method **Vor dem Refactoring:**

```java
public void printOwing() {
    // print banner
    System.out.println("*****************");
    System.out.println("**Customer Owes**");
    System.out.println("*****************");

    // calculate total
    double total = 0;
    for (Order order : orders) {
        total += order.getAmount();
    }

    // print details
    System.out.println("name: " + name);
    System.out.println("amount: " + total);
}
```

Nach dem Refactoring:

```java
public void printOwing() {
    printBanner();
    double total = calculateTotal();
    printDetails(total);
}

private void printBanner() {
    System.out.println("*****************");
    System.out.println("**Customer Owes**");
    System.out.println("*****************");
}

private double calculateTotal() {
    double total = 0;
    for (Order order : orders) {
        total += order.getAmount();
    }
    return total;
}

private void printDetails(double total) {
    System.out.println("name: " + name);
    System.out.println("amount: " + total);
}
```

Testing

Testarten (Nach Sicht)

- **Funktionaler Test (Black-Box Verfahren):**
  - Test aus Benutzersicht
  - Ohne Codekenntnis (Kenntnis der internen Implementation)
  - Fokus auf Input/Output
- **Strukturbezogener Test (White-Box Verfahren):**
  - Test mit Codekenntnis (Kenntnis der Implementation)
  - Code Coverage Pfadtests

Teststufen (Nach Umfang)

- **1. Unit Tests** Einzelne Komponenten
  - Isolation durch Mocks/Stubs
  - Schnelle Ausführung und Automatisierte Tests
- **2. Integrationstests** Zusammenspiel mehrer Komponenten
  - Fokus auf Schnittstellen
  - Kann externe Systeme einbeziehen
- **3. Systemtests** Gesamtsystem testen
  - End-to-End Szenarien
  - Auch nicht-funktionale Anforderungen
- **4. Abnahmetests** Gegen Kundenanforderungen
  - Oft manuelle Tests
  - User Acceptance Testing (UAT)

Wichtige Testbegriffe

- **Testling/Testobjekt:** Das zu testende Element
- **Fehler:** Fehler des Entwicklers bei der Implementation
- **Fehlerwirkung/Bug:** Abweichung vom spezifizierten Verhalten
- **Testfall:** Spezifische Testkonfiguration mit Testdaten
- **Testtreiber:** Programm zur Testausführung

Testmerkmale **Was wird getestet?**

- Eine Einheit/Klasse (Unit-Test)
- Zusammenarbeit mehrerer Klassen
- Die gesamte Applikationslogik (ohne UI)
- Die gesamte Anwendung (über UI)

Wie wird getestet?

- Dynamisch: Testfall wird ausgeführt
  - Black-Box Test
  - White-Box Test
- Statisch: Quelltext wird analysiert
  - Walkthrough
  - Review
  - Inspektion

Wann wird der Test geschrieben?

- Vor dem Implementieren (Test-Driven Development)
- Nach dem Implementieren

Wer testet?

- Entwickler
- Tester, Qualitätssicherungsabteilung
- Kunde, Endbenutzer

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

Common Testing Pitfalls

1. Flaky Tests

```java
// Schlecht: Test koennte fehlschlagen
@Test
public void testAsyncOperation() throws Exception {
    service.startAsyncOperation();
    Thread.sleep(1000); // Fragil!
    assertTrue(service.isOperationComplete());
}
// Besser: Explizites Warten
@Test
public void testAsyncOperation() {
    service.startAsyncOperation();
    await()
        .atMost(5, TimeUnit.SECONDS)
        .until(() -> service.isOperationComplete());
    assertTrue(service.isOperationComplete());
}
```

2. Zu viele Mocks

```java
// Schlecht: Uebermassiges Mocking
@Test
public void testOverMocked() {
    Database db = mock(Database.class);
    Logger logger = mock(Logger.class);
    EmailService email = mock(EmailService.class);
    UserService users = mock(UserService.class);

    // Test wird unuebersichtlich und fragil
}

// Besser: Sociable Unit Testing
@Test
public void testSociable() {
    // Nur externe Abhaengigkeiten mocken
    EmailService email = mock(EmailService.class);
    UserService service = new UserService(email);

    User user = service.createUser("test@test.com");
    assertNotNull(user.getId());
}
```

Testing Frameworks and Tools

1. Unit Testing

- **JUnit:**
  - Assertions für Vergleiche
  - Test Lifecycle (@Before, @After)
  - Test Suites organisieren
- **Mockito:**
  - Mocking von Abhängigkeiten
  - Verifikation von Aufrufen
  - Stubbing von Verhalten

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

Test Coverage Analysis

1. Line Coverage

```
public class PaymentProcessor {
    public boolean processPayment(Payment payment) {
        if (payment == null) {          // Covered
            return false;               // Covered
        }

        if (payment.getAmount() <= 0) { // Not covered
            return false;               // Not covered
        }

        return doProcessPayment(payment); // Covered
    }
}
```

2. Branch Coverage

```
@Test
public void testPaymentProcessing() {
    // Test 1: null payment
    assertFalse(processor.processPayment(null));

    // Test 2: valid payment
    Payment payment = new Payment(100.00);
    assertTrue(processor.processPayment(payment));

    // Missing: Test fuer payment.getAmount() <= 0
}
```

Integration Testing

1. Teststrategie:

- Bottom-up Integration
- Top-down Integration
- Big Bang Integration

2. Testumfang:

- Komponenteninteraktion
- Schnittstellentests
- Datenfluss

3. Testaufbau:

```
@SpringBootTest
public class OrderIntegrationTest {
    @Autowired
    private OrderService orderService;

    @Autowired
    private CustomerService customerService;

    @Test
    public void shouldProcessOrderEndToEnd() {
        // Setup test data
        Customer customer =
            customerService.createCustomer(
            "Test Customer", "test@example.com");

        Order order = new Order(customer);
        order.addItem(new OrderItem("Test Product",
            29.99, 1));

        // Execute process
        OrderResult result =
            orderService.processOrder(order);

        // Verify results
        assertTrue(result.isSuccess());
        assertEquals("COMPLETED", result.getStatus());
        // Verify database state
        Order savedOrder =
            orderService.findOrder(result.getOrderId());
        assertEquals("COMPLETED",
            savedOrder.getStatus());
    }
}
```

Continuous Integration Testing

1. Build Pipeline

- Compile und Unit Tests
- Integration Tests
- Code Quality Checks
- Security Scans

2. Test Automatisierung

- Smoke Tests
- Regression Tests
- Performance Tests

3. Reporting

- Test Coverage Reports
- Performance Metrics
- Error Logs

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

Performance Testing

1. Last Tests (Load Testing)

- Normales Benutzerverhalten simulieren
- Durchschnittliche Antwortzeiten messen
- Ressourcenverbrauch überwachen

2. Stress Tests

- System an Grenzen bringen
- Verhalten bei Überlast prüfen
- Wiederherstellung testen

3. Endurance Tests

- Langzeitverhalten prüfen
- Memory Leaks aufdecken
- Ressourcenverbrauch über Zeit

Beispiel JMeter Test Plan:

```
// Thread Group Configuration
Number of Threads (Users): 100
Ramp-up Period: 60 seconds
Loop Count: 10

// HTTP Request
GET /api/products
Headers:
 Content-Type: application/json
 Authorization: Bearer ${token}

// Response Assertions
Response Code: 200
Response Time: < 500ms
```

Verteilte Systeme

Verteiltes System

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

- Autonome Knoten und Komponenten
- Netzwerkverbindung
- Erscheint dem Benutzer wie ein einzelnes, kohärentes 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

Kommunikationsmodelle

1. Synchrone Kommunikation

- Synchroner entfernter Dienstaufruf → blockierend
- Sender wartet auf Ergebnis der Methode send
- Typisch für Request-Response Pattern

2. Asynchrone Kommunikation

- Asynchroner entfernter Serviceaufruf → nicht blockierend
- Sender kann direkt weitermachen
- Senden und Empfangen zeitlich versetzt
- Keine Blockierung des Prozesses

Prüfungsaufgabe: Kommunikationsanalyse

Szenario: Ein verteiltes System soll große Datenmengen verarbeiten. Analysieren Sie die Vor- und Nachteile synchroner vs. asynchroner Kommunikation.

Synchrone Kommunikation:

- **Vorteile:**
  - Einfaches Programmiermodell
  - Direktes Feedback
  - Garantierte Reihenfolge
- **Nachteile:**
  - Blockierung von Ressourcen
  - Schlechte Skalierbarkeit
  - Anfällig für Timeouts

Asynchrone Kommunikation:

- **Vorteile:**
  - Bessere Ressourcennutzung
  - Höhere Skalierbarkeit
  - Entkopplung von Systemen
- **Nachteile:**
  - Komplexere Implementierung
  - Schwierigere Fehlerbehandlung
  - Reihenfolge nicht garantiert

Kommunikation zwischen Client und Server

Grundlagen:

- Services sind über URLs aufrufbar: protokoll://<server>:<port>/<pfad_des_service>
- Kommunikation über TCP oder UDP
- Socket = Programmierschnittstelle zu Kommunikationskanal
- IP-Socket-Adresse besteht aus IP-Adresse + Portnummer

Ablauf: study THIN :)

Lebenszyklus von Serverbausteinen

Ein Serverbaustein durchläuft verschiedene Zustände:

- Wird zur Laufzeit von Server instanziiert
- Zustandsübergänge abhängig von Bausteintyp und Implementierung
- Anzahl und Benennung der Zustände variiert je nach Middleware

Heterogenität

Mehrere Ebenen der Heterogenität müssen berücksichtigt werden:

1. Hardware und Betriebssysteme

- Unterschiedliche Speicherung der Daten ("Little Endian"vs "Big Endian")
- Verschiedene Zeichensätze (ASCII, EBCDIC, Unicode)

2. Überwindung der Heterogenität

- Einheitliche Transportsyntax (ASN.1, XDR, HTML, XML, JSON)
- Middleware-Technologien mit standardisierten Ansätzen
- Marshalling/Unmarshalling über generierten Code

Middleware

Middleware ist eine Softwareschicht, die standardisierte Kommunikations- und andere Dienste über ein API bereitstellt.

Middleware-Kategorien:

- **Anwendungsorientiert:**
  - Java Enterprise Edition (Jakarta EE)
  - Spring-Framework
  - .NET Enterprise Services
- **Kommunikationsorientiert:**
  - Remote Procedure Call (RPC)
  - Remote Method Invocation (RMI)
  - CORBA (Common Object Request Broker Architecture)
  - REST, WebSocket
  - RESTful APIs, GraphQL
- **Nachrichtenorientiert:**
  - Message Oriented Middleware (MOM)
  - Java Messaging Service (JMS)
  - Message Broker (MQTT, RabbitMQ, Kafka)

Marshalling/Unmarshalling

Umwandlung von Daten für die Übermittlung:

1. Tag-basierte Transfersyntax

- ASN.1 mit BER (Basic Encoding Rules)
- TLV-Kodierung (Type, Length, Value)

2. Tag-freie Transfersyntax

- Sun ONC XDR, CORBA CDR
- Beschreibung durch Stellung in Nachricht
- Aufbau beiden Seiten bekannt

3. Moderne Formate

- XML (Tag-basiert)
- JSON (Tag-basiert, sprachunabhängig)

Entwurf verteilter Systeme

1. Systemanalyse

- Anforderungen identifizieren
- Verteilungsaspekte analysieren
- Konsistenzanforderungen definieren
- **Funktional:**
  - Kernfunktionalitäten
  - Datenmodell
  - Schnittstellen
- **Nicht-funktional:**
  - Skalierbarkeit
  - Verfügbarkeit
  - Latenz

2. Architekturentscheidungen

- Architekturstil wählen
- Kommunikationsmuster festlegen
- Fehlertoleranzstrategie definieren
- **Kommunikation:**
  - Synchron vs. Asynchron
  - Push vs. Pull
  - Protokollwahl
- **Datenmanagement:**
  - Sharding
  - Replikation
  - Caching

3. Technologieauswahl/Implementierungsaspekte

- Middleware evaluieren
- Protokolle bestimmen
- Werkzeuge auswählen
- **Middleware:**
  - REST
  - gRPC
  - Apache Kafka
- **Implementierung:**
  - Nebenläufigkeit
  - Fehlersituationen
  - Parameterübergabe

Architekturmodelle / Architekturstile

Architekturmodelle

Heute finden vor allem folgende Architekturmodelle ihren Einsatz:

1. Client/Server

- Kurzlebiger Client-Prozess kommuniziert mit langlebigem Server-Prozess
- Beispiel: Web-Applikation

2. Peer-to-Peer

- Gleichberechtigte Peer-Prozesse
- Informationsaustausch nur bei Bedarf
- Beispiel: Blockchain

3. Event Systems (Publish-Subscribe)

- Event-Sources-Prozesse und Event-Sinks-Prozesse
- Asynchroner Informationsaustausch
- Beispiel: E-Mail-System

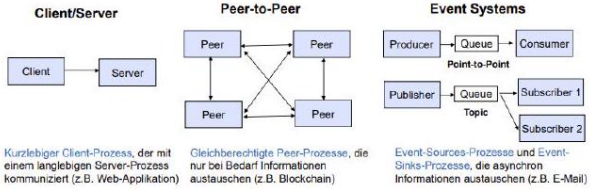


Abbildung 20: Architekturmodelle

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

Prüfungsaufgabe: Systemanalyse

Aufgabe: Für ein Online-Banking-System sollen kritische Aspekte analysiert werden.

1. Anforderungen identifizieren:

- Hohe Verfügbarkeit (24/7)
- Strikte Konsistenz bei Finanztransaktionen
- Skalierbarkeit für viele Nutzer
- Sicherheit und Verschlüsselung

2. Architekturentscheidungen:

- **Client-Server mit Multi-Tier:**
  - Präsentationsschicht (Web/Mobile)
  - Geschäftslogik
  - Datenbankschicht
- **Synchrone Kommunikation:**
  - Direkte Bestätigung von Transaktionen
  - Garantierte Reihenfolge

3. Technische Maßnahmen:

- Verteilte Datenbank mit Two-Phase Commit
- Load Balancing für Hochverfügbarkeit
- SSL/TLS für Verschlüsselung
- Session Management für Authentifizierung

Typische Prüfungsaufgabe: CAP-Theorem

Aufgabe: Analysieren Sie für ein verteiltes Datenbanksystem die Auswirkungen des CAP-Theorems.

CAP-Theorem Komponenten:

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

Analyse der Trade-offs:

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

Design Pattern Auswahl

1. Remote Proxy Pattern

- Grundlegendes Pattern für Zugriff auf Services
- Proxy wird durch Dependency Injection (DI) instanziiert
- Client- und serverseitiger Stub (SSkeleton")

2. Data Transfer Object (DTO)

- Bündelt mehrere Daten für einen einzelnen Programmaufruf
- Reduziert Anzahl der Remote-Zugriffe
- Typischerweise ïmmutable"(nur getter-Methoden)

3. Service Locator

- Zentrale Registrierung von Services
- Ermöglicht dynamisches Auffinden von Diensten
- Unterstützt Load Balancing

Prüfungsaufgabe: Architekturmuster-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 Architekturmuster:

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

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

Implementierung verteilter Systeme

1. Nebenläufigkeit

- Iterative oder parallele Serverbausteine
- Threadpooling für gleichzeitige Bedienung mehrerer Clients
- Dispatcher verteilt Requests auf Threads
- Einfaches sequentielles Programmiermodell für Entwickler

2. Fehlersituationen

- Request geht verloren
- Server-Response geht verloren
- Server stürzt während Ausführung ab
- Server braucht zu lange für Bearbeitung
- Client stürzt vor Ankunft des Ergebnisses ab

3. Parameterübergabe

- Call-by-value: Wert wird übergeben
- Call-by-reference: Verweis auf Variable wird übergeben
- Call-by-copy/restore: Aufrufer arbeitet mit Kopie

Implementierungsaspekte

Zustandsverwaltung:

- **Stateful vs. Stateless:**
  - Stateless Server bevorzugt
  - Zustand in Datenbank oder Client
  - Bessere Skalierbarkeit
- **Session Management:**
  - Verteilte Sessions
  - Session Clustering
  - Sticky Sessions

Garbage Collection:

- Verteiltes Reference-Counting
- Leases für temporäre Ressourcen
- Zusammenarbeit mit lokalem GC

Lastverteilung und Skalierung

1. Load Balancing

- Lastverteiler für multiple Serverinstanzen
- DNS-basiertes Request-Routing
- Session-Sticky Load Balancing

2. Hochverfügbarkeit

- Server-Cluster
- Failover-Mechanismen
- Session-Replikation

3. Skalierungsmethoden

- **Horizontal:** Mehr Rechner hinzufügen
- **Vertikal:** Ressourcen pro Rechner erhöhen
- **Funktional:** Services aufteilen

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)

Best Practices für Persistenz

1. Architektur-Ebene

- Trennung von Concerns
  - Repository für Datenzugriff
  - Service für Geschäftslogik
  - DTO für Datentransfer
- Transaktionsmanagement
  - Auf Service-Ebene
  - Atomare Operationen
  - Konsistente Daten

2. Entity Design

- Immutable wenn möglich
- Validierung durch Bean Validation
- Geschäftsregeln in Entity-Klassen

3. Performance Optimierung

- Caching Strategien
- Batch Processing
- Query Optimierung

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)
- **Besondere Herausforderungen:**
  - Unterschiedliche Repräsentationsformen
  - Flache Tabellenstruktur vs. Objekthierarchien
  - Verschiedene Konzepte für Beziehungen

Prüfungsaufgabe: O/R-Mapping Analyse

Szenario: Ein Universitätssystem verwaltet Studenten, Kurse und Noten. Studenten können mehrere Kurse belegen, ein Kurs hat mehrere Studenten.

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

Lösung:

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

JDBC - Java Database Connectivity

JDBC Grundlagen

JDBC ist die standardisierte Schnittstelle für Datenbankzugriffe in Java:

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

JDBC Architektur

Hauptkomponenten:

- **JDBC API:**
  - Interfaces und Klassen für DB-Zugriff
  - Im java.sql Package
- **JDBC Driver Manager:**
  - Verwaltet Datenbanktreiber
  - Erstellt Verbindungen
- **JDBC Drivers:**
  - Herstellerspezifische Implementierungen
  - Übersetzen JDBC-Aufrufe in DB-Protokoll

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

JDBC Basisoperationen

1. Verbindungsaufbau

```
Connection con = DriverManager.getConnection(
    "jdbc:postgresql://server/db",
    "user", "password");
```

2. Statement erstellen und ausführen

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
    "SELECT * FROM users");
```

3. Ergebnisse verarbeiten

```
while (rs.next()) {
    String name = rs.getString("name");
    int age = rs.getInt("age");
}
```

4. Ressourcen freigeben

```
rs.close();
stmt.close();
con.close();
```

Java Persistence API (JPA)

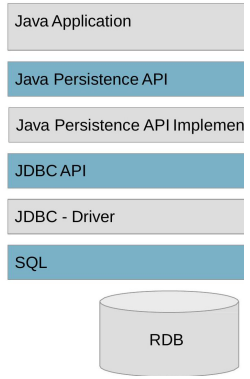
JPA Grundkonzepte

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

Design Patterns für Persistenz

Persistenz Design Patterns

Drei grundlegende Ansätze für die Persistenzschicht:

- **Active Record (Anti-Pattern):**
  - Jede Entität verwaltet eigene Persistenz
  - Vermischung von Fachlichkeit und Technik in einer Klasse
  - Wrapper für Datenbankzeilen, spiegelt DB-Struktur direkt
  - Schlechte Testbarkeit der Domänenlogik
  - Schlechte Wartbarkeit und Erweiterbarkeit
- **Data Access Object (DAO):**
  - Kapselung des Datenbankzugriffs
  - Trennung von Fachlichkeit und Technik
  - Domänenklasse hat hohe Kohäsion
  - Gute Testbarkeit durch Mocking
  - Bevorzugtes Design ohne O/R-Mapper
- **Repository (DDD):**
  - Abstraktionsschicht über Data-Mapper
  - Zentralisierung von Datenbankabfragen
  - Komplexere Implementierung, aber Unterstützung für Komplexere Abfragen
  - Domänenorientierte Schnittstelle
  - Häufig in Kombination mit Spring Data

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

Analyse der Persistenzanforderungen

1. Funktionale Anforderungen
- Datenmodell und Beziehungen
- Abfrageanforderungen
- Transaktionsverhalten

2. Nicht-funktionale Anforderungen
- Performance
- Skalierbarkeit
- Wartbarkeit
- Konsistenz

3. Technische Randbedingungen
- Vorhandene Systeme
- Datenbanktyp
- Entwicklungsumgebung

4. Designentscheidungen
- Persistenzstrategie wählen
- Framework-Auswahl
- Architekturmuster festlegen

Common Pitfalls in Persistence Implementation

1. JPA Fallstricke

N+1 Problem:
- Symptom: Für jedes Objekt wird eine zusätzliche Query ausgeführt
- Lösung: Join Fetch oder Eager Loading strategisch einsetzen

LazyInitializationException:
- Symptom: Zugriff auf lazy geladene Referenz außerhalb der Session
- Lösung: Transaktionen richtig abgrenzen oder Fetch Join verwenden

Bidirektionale Beziehungen:
- Symptom: Inkonsistente Objektzustände
- Lösung: Helper-Methoden für Beziehungspflege

2. Performance Probleme

Zu viele Queries:
- Problem: Ineffiziente Datenbankzugriffe
- Lösung: Batch Processing, Bulk Operations

Memory Leaks:
- Problem: Große Result Sets
- Lösung: Pagination, Streaming Results

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

Implementation der Patterns

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

DAO Pattern mit JDBC

Implementierungsstruktur:
- **Interface:**
  - Definiert CRUD-Operationen
  - Definiert spezifische Abfragen
- **Domänenklasse:**
  - Reine Geschäftslogik
  - Keine Persistenzlogik
- **Implementierung:**
  - Verwendet JDBC für DB-Zugriff
  - Implementiert O/R-Mapping
  - Behandelt Fehler

DAO Implementation — 1. Interface definieren:

```java
public interface UserDAO {
    void insert(User user);
    User findById(long id);
    List<User> findAll();
    void update(User user);
    void delete(User user);
}
```

2. JDBC-Implementierung:

```java
public class JdbcUserDAO implements UserDAO {
    private Connection getConnection() {
        // Verbindungsaufbau
    }
    public User findById(long id) {
        Connection conn = getConnection();
        try {
            PreparedStatement ps =
                conn.prepareStatement(
                "SELECT * FROM users WHERE id = ?");
            ps.setLong(1, id);
            ResultSet rs = ps.executeQuery();
            if (rs.next()) {
                return mapUser(rs);
            }
            return null;
        } finally {
            conn.close();
        }
    }
    private User mapUser(ResultSet rs) {
        User user = new User();
        user.setId(rs.getLong("id"));
        user.setName(rs.getString("name"));
        return user;
    }
}
```

Typische Prüfungsaufgabe: Parent-Child Beziehung

Szenario: Implementieren Sie eine bidirektionale One-to-Many Beziehung zwischen Department und Employee.

1. Entity-Klassen:

```java
@Entity
public class Department {
    @Id @GeneratedValue
    private Long id;
    private String name;

    @OneToMany(mappedBy = "department",
               cascade = CascadeType.ALL,
               orphanRemoval = true)
    private List<Employee> employees = new
        ArrayList<>();

    // Helper Methoden
    public void addEmployee(Employee employee) {
        employees.add(employee);
        employee.setDepartment(this);
    }

    public void removeEmployee(Employee employee) {
        employees.remove(employee);
        employee.setDepartment(null);
    }
}

@Entity
public class Employee {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "department_id")
    private Department department;
}
```

2. Repository Interface:

```java
public interface DepartmentRepository
    extends JpaRepository<Department, Long> {

    @Query("SELECT d FROM Department d " +
           "LEFT JOIN FETCH d.employees " +
           "WHERE d.id = :id")
    Optional<Department> findByIdWithEmployees(
        @Param("id") Long id);
}
```

3. Service Layer:

```java
@Service
@Transactional
public class DepartmentService {
    @Autowired
    private DepartmentRepository repository;

    public void addEmployeeToDepartment(
            Long deptId, Employee employee) {
        Department dept = repository
            .findById(deptId)
            .orElseThrow(() ->
                new EntityNotFoundException());
        dept.addEmployee(employee);
        repository.save(dept);
    }
}
```

Praktische Implementierungsbeispiele

1. Optimistisches Locking:

```java
@Entity
public class Account {
    @Version
    private Long version;

    private BigDecimal balance;

    public void withdraw(BigDecimal amount) {
        if (balance.compareTo(amount) < 0) {
            throw new InsufficientFundsException();
        }
        balance = balance.subtract(amount);
    }
}
```

2. Entity Lifecycle Callbacks:

```java
@Entity
public class AuditedEntity {
    @PrePersist
    void onCreate() {
        this.createdDate = LocalDateTime.now();
    }

    @PreUpdate
    void onUpdate() {
        this.lastModified = LocalDateTime.now();
    }
}
```

3. Custom Repository Implementation:

```java
@Repository
public class CustomOrderRepositoryImpl
    implements CustomOrderRepository {

    @PersistenceContext
    private EntityManager em;

    public List<Order> findComplexOrders(
            OrderCriteria criteria) {
        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Order> query =
            cb.createQuery(Order.class);
        Root<Order> root = query.from(Order.class);

        // Complex criteria building
        Predicate[] predicates = buildPredicates(
            cb, root, criteria);
        query.where(predicates);

        return em.createQuery(query)
                 .getResultList();
    }
}
```

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

Analyse von Framework-Anforderungen

1. Fachliche Analyse

- **Core Features:**
  - Zentrale Funktionalität
  - Gemeinsame Abstraktionen
  - Standardverhalten
- **Variationspunkte:**
  - Kundenspezifische Anpassungen
  - Optionale Features
  - Erweiterungsmöglichkeiten

2. Technische Analyse

- **Architektur-Entscheidungen:**
  - Erweiterungsmechanismen
  - Integration in bestehende Systeme
  - Schnittstellen-Design
- **Qualitätsanforderungen:**
  - Performance
  - Wartbarkeit
  - Testbarkeit

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

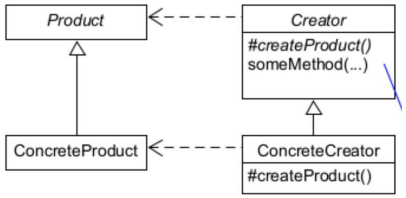
Design Patterns in Frameworks

Factory Method

Problem: Flexible Objekterzeugung in wiederverwendbarer Klasse

Lösung:

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

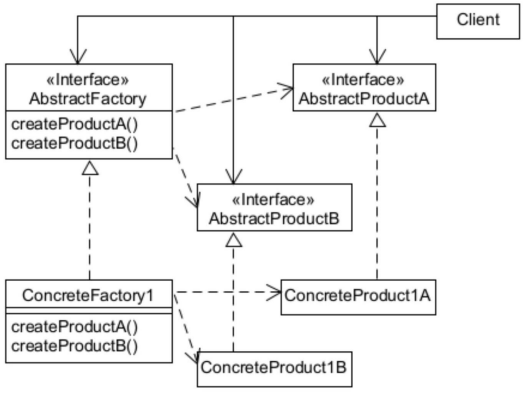


Abstract Factory

Problem: Erzeugung verschiedener, zusammengehörender Objekte ohne Kenntnis konkreter Klassen

Lösung:

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

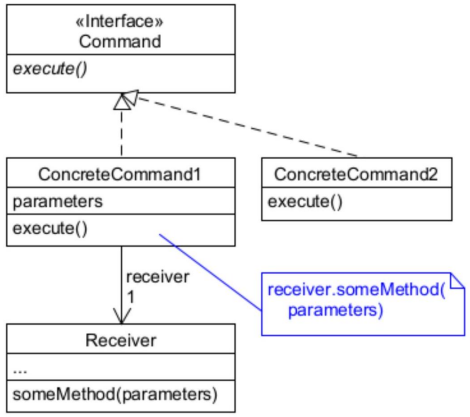


Command

Problem: Aktionen für späteren Gebrauch speichern und verwalten

Lösung:

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

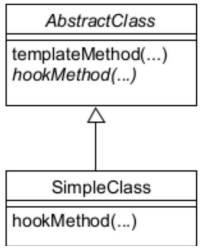


Template Method

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"



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

Moderne Framework Mechanismen

Annotation-basierte Konfiguration

Moderne Frameworks nutzen Annotationen für:

- Dependency Injection
- Konfiguration
- Interface-Implementation
- Funktionalitätserweiterung

Annotations als Steuerungsmechanismus

Vorteile von Annotations:

- Keine harte Abhängigkeit zum Framework (geringere Kopplung zur API)
- Annotation wird stillschweigend entfernt wenn nicht gefunden
- Geeignet für Domänenlogik ohne technische Abhängigkeiten
- Deklarativer Programmierstil
- Reduzierung von Boilerplate-Code

Nachteil von Annotations: Kann zu längeren Startzeiten führen

Auswertung von Annotations:

- **Startzeitpunkt:**
  - Framework wird mit Anwendung gestartet
  - Sucht Anwendungsklassen auf dem Klassenpfad
  - Untersucht Annotationen
- **Mögliche Framework-Aktionen:**
  - Dependency Injection in Anwendungsobjekte
  - Automatische Interface-Implementierung
  - Funktionalität zu Klassen hinzufügen

Aspekt-orientierte Programmierung in Frameworks

Querschnittliche Belange (Cross-Cutting Concerns):

- Logging
- Sicherheit
- Transaktionsmanagement
- Performance Monitoring

Implementation mit Annotations:

```java
@Aspect
public class LoggingAspect {
    @Around("@annotation(Logged)")
    public Object logMethod(
            ProceedingJoinPoint joinPoint)
            throws Throwable {

        String methodName =
            joinPoint.getSignature().getName();
        Logger.info("Entering " + methodName);

        try {
            Object result = joinPoint.proceed();
            Logger.info("Exiting " + methodName);
            return result;
        } catch (Exception e) {
            Logger.error("Error in " + methodName, e);
            throw e;
        }
    }
}
// Usage in Framework Client Code
@Logged
public void businessMethod() {
    // Method implementation
}
```

Java Mechanismen für Framework-Implementation

1. Zeitpunkte für Code-Generierung

- **Compile-Zeit:**
  - AnnotationProcessor
  - Quellcode oder Bytecode generieren
- **Laufzeit:**
  - Beim Laden der Klassen
  - Framework-Classloader
  - Bytecode-Modifikation

2. Implementierungstechniken

- **Code Generation:**
  - Quellcode hinzufügen
  - Bytecode modifizieren
- **Proxy Generation:**
  - java.lang.reflect.Proxy
  - Interface-Implementation

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

Prüfungsaufgabe: Framework Design Entscheidungen

Szenario: Sie sollen für eine Firma ein Framework zum Verarbeiten von Datenexporten entwickeln. Die Firma arbeitet mit verschiedenen Datenformaten (CSV, Excel, XML) und möchte das Framework später einfach um weitere Formate erweitern können.

Aufgabenstellung:

1. Identifizieren Sie die Variationspunkte
2. Wählen Sie geeignete Design Patterns
3. Skizzieren Sie die Framework-Architektur

Musterlösung:

- **Variationspunkte:**
  - Format-Erkennung
  - Datei-Parser
  - Daten-Transformation
  - Export-Ziele
- **Design Patterns:**
  - Abstract Factory für Parser-Erzeugung
  - Strategy für unterschiedliche Parse-Algorithmen
  - Template Method für generellen Export-Workflow
  - Chain of Responsibility für Format-Erkennung
- **Framework-Architektur:**
  - Core API mit Interfaces
  - Plugin-System für neue Formate
  - Event-System für Export-Status
  - Konfigurationsschicht

Framework Design Pattern Kombination

Aufgabe: Analysieren Sie die Kombination verschiedener Design Patterns in einem Framework.

Muster-Framework: Event-Processing Framework mit folgenden Patterns:

1. Template Method

- Definiert Workflow für Event-Verarbeitung
- Hook-Methoden für:
  - Event-Validierung
  - Event-Transformation
  - Event-Persistierung

2. Chain of Responsibility

- Event-Handler-Kette
- Flexible Verarbeitungsreihenfolge
- Dynamische Handler-Registration

3. Command

- Kapselung von Event-Handling-Logik
- Queuing von Events
- Undo/Redo Funktionalität

4. Observer

- Benachrichtigung über Event-Status
- Lose Kopplung zwischen Komponenten
- Flexible Registration von Listeners

Pattern-Interaktion:

- Template Method definiert Grundstruktur
- Chain of Responsibility organisiert Handler
- Command kapselt konkrete Aktionen
- Observer informiert über Ergebnisse

Prüfungsaufgabe: Framework Testing

Szenario: Ein Framework soll gründlich getestet werden. Entwickeln Sie eine Teststrategie.

Testebenen:

- **Unit Tests:**
  - Einzelne Komponenten
  - Mock-Objekte für Dependencies
  - Edge Cases
- **Integration Tests:**
  - Zusammenspiel der Komponenten
  - Plugin-Mechanismen
  - Event-Handling
- **System Tests:**
  - End-to-End Szenarien
  - Performance Tests
  - Load Tests

Besondere Aspekte:

- Extension Points testen
- Verschiedene Konfigurationen
- Backward Compatibility
- Error Handling

Framework-Extensions entwickeln

1. Extension Points identifizieren

- Core-Funktionalität analysieren
- Variationspunkte bestimmen
- Interface-Hierarchie planen

2. Extension Mechanismen

- **Interface-basiert:**

```
public interface Plugin {
    void initialize();
    void shutdown();
    String getName();
}
```

- **Annotation-basiert:**

```
@Extension
public class CustomPlugin {
    @Initialize
    public void setup() { ... }

    @Shutdown
    public void cleanup() { ... }
}
```

3. Discovery Mechanism

```
public class ExtensionLoader {
    public List<Plugin> loadPlugins() {
        ServiceLoader<Plugin> loader =
            ServiceLoader.load(Plugin.class);
        return StreamSupport
            .stream(loader.spliterator(), false)
            .collect(Collectors.toList());
    }
}
```

Prüfungsaufgabe: Framework Evolution

Ausgangslage: Ein bestehendes Framework soll um neue Funktionalität erweitert werden, ohne bestehende Clients zu beeinträchtigen.

Analyse der Optionen:

- **Annotation-basierte Erweiterung:**
  - Vorteile:
    - Keine Änderung bestehender Interfaces
    - Optionale Funktionalität
    - Deklarativer Ansatz
  - Nachteile:
    - Komplexere Verarbeitung
    - Mögliche Performance-Einbußen
    - Schwieriger zu debuggen
- **Interface-basierte Erweiterung:**
  - Vorteile:
    - Klare Kontrakte
    - Compile-time Checks
    - Einfache Dokumentation
  - Nachteile:
    - Änderungen an Interfaces nötig
    - Adapter für alte Clients
    - Höherer Implementierungsaufwand

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

Framework Integration Case Study

Szenario: Integration eines Logging-Frameworks in eine bestehende Anwendung

Anforderungen:

- Minimale Änderungen am bestehenden Code
- Konfigurierbare Log-Level
- Verschiedene Log-Ausgaben (Konsole, File, DB)
- Performance-Monitoring

Lösung mit Framework Patterns:

```
// Logger Interface
public interface Logger {
    void debug(String message);
    void info(String message);
    void error(String message, Throwable t);
}
// Abstract Factory fuer Logger
public interface LoggerFactory {
    Logger getLogger(Class<?> clazz);
}
// Decorator fuer Performance Monitoring
public class PerformanceLogger implements Logger {
    private final Logger delegate;
    private final MetricsCollector metrics;

    @Override
    public void info(String message) {
        long start = System.nanoTime();
        try {
            delegate.info(message);
        } finally {
            long duration = System.nanoTime() - start;
            metrics.recordLogDuration(duration);
        }
    }
    // Other methods...
}
// Framework Configuration
@Configuration
public class LoggingConfig {
    @Bean
    public LoggerFactory loggerFactory(
            MetricsCollector metrics) {
        return clazz -> {
            Logger baseLogger = // create base logger
            return new PerformanceLogger(
                baseLogger, metrics);
        };
    }
}
```

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

complete examples

Framework Design: Validation Framework

Anforderungen: Ein Framework für Validierung von Geschäftsobjekten soll entwickelt werden.

Design:

```java
// Validation Annotations
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface NotNull {
    String message() default "Value cannot be null";
}

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Length {
    int min() default 0;
    int max() default Integer.MAX_VALUE;
    String message() default "Length must be between
        {min} and {max}";
}

// Business Object
public class Customer {
    @NotNull
    private String id;

    @NotNull
    @Length(min = 2, max = 50)
    private String name;

    // getters and setters
}

// Validator Interface
public interface Validator<T> {
    ValidationResult validate(T object);
}

// Framework Implementation
public class ValidationFramework {
    public static <T> ValidationResult validate(T
        object) {
        Class<?> clazz = object.getClass();
        ValidationResult result = new
            ValidationResult();

        for (Field field : clazz.getDeclaredFields()) {
            validateField(object, field, result);
        }

        return result;
    }
}
```

Verwendung:

```java
Customer customer = new Customer();
customer.setName("J"); // too short

ValidationResult result =
    ValidationFramework.validate(customer);
if (!result.isValid()) {
    System.out.println(result.getErrors());
}
```

Framework Design Pattern: Event System

Anforderung: Ein Framework soll Benutzern ermöglichen, auf verschiedene Events zu reagieren.

Implementation:

```java
// Event Base Class
public abstract class Event {
    private final LocalDateTime timestamp;

    protected Event() {
        this.timestamp = LocalDateTime.now();
    }

    public LocalDateTime getTimestamp() {
        return timestamp;
    }
}

// Concrete Event
public class UserCreatedEvent extends Event {
    private final String userId;

    public UserCreatedEvent(String userId) {
        this.userId = userId;
    }
}

// Event Listener Interface
public interface EventListener<T extends Event> {
    void onEvent(T event);
}

// Event Bus
public class EventBus {
    private Map<Class<? extends Event>,
                List<EventListener>> listeners = new
                    HashMap<>();

    public <T extends Event> void register(
            Class<T> eventType,
            EventListener<T> listener) {
        listeners.computeIfAbsent(eventType,
            k -> new ArrayList<>()).add(listener);
    }

    public void publish(Event event) {
        List<EventListener> eventListeners =
            listeners.get(event.getClass());
        if (eventListeners != null) {
            eventListeners.forEach(
                listener -> listener.onEvent(event));
        }
    }
}
```

Framework Nutzung:

```java
// Framework Usage
EventBus eventBus = new EventBus();

// Register Listener
eventBus.register(UserCreatedEvent.class,
    event -> System.out.println("User created: "
        + event.getUserId()));

// Publish Event
eventBus.publish(new UserCreatedEvent("user123"));
```