Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Examples

Software Engineering

Typische Prüfungsaufgabe: Prozessmodelle vergleichen

Vergleichen Sie das Wasserfallmodell mit einem iterativ-inkrementellen Ansatz anhand folgender Kriterien:

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

Musterlösung:

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

Modellierungsumfang bestimmen

Folgende Fragen zur Bestimmung des notwendigen Modellierungsumfangs:

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

Prüfungsfrage zur Modellierung

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

Mögliche Antwort:

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

Architekturmuster

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);
        }
    }
}
```

Framework Design

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

KR und Beispiele für Einführung

Prozessmodelle vergleichen

Vorgehen bei der Analyse:

1. **Kriterien identifizieren:**
   - Zeit/Budget/Scope
   - Risikomanagement
   - Kundeneinbindung
   - Änderungsmanagement
2. **Modelle analysieren:**
   - Charakteristiken der Modelle verstehen
   - Stärken und Schwächen erkennen
   - Anwendungsszenarien berücksichtigen
3. **Gegenüberstellung:**
   - Direkte Vergleiche für jedes Kriterium
   - Unterschiede herausarbeiten
   - Mit Beispielen unterlegen

Prozessmodell-Analyse

Aufgabe: Vergleichen Sie das Wasserfallmodell mit einem iterativ-inkrementellen Ansatz anhand folgender Kriterien:

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

Musterlösung:

- **Wasserfall:**
  - Änderungen schwierig zu integrieren
  - Risiken erst spät erkennbar
  - Gut planbar durch feste Phasen
  - Kunde hauptsächlich am Anfang und Ende involviert
- **Iterativ-inkrementell:**
  - Flexibel bei Änderungen durch kurze Zyklen
  - Frühes Erkennen von Risiken durch regelmäßige Reviews
  - Planung pro Iteration, mehr Flexibilität
  - Kontinuierliches Kundenfeedback in jeder Iteration

Modellierungsumfang bestimmen

Analyseschritte:

1. **Projektkontext analysieren:**
   - Projektgröße und Komplexität
   - Anzahl beteiligter Stakeholder
   - Kritikalität des Systems
   - Domänenwissen im Team
2. **Faktoren bewerten:**
   - Risiko bei Fehlern
   - Änderungshäufigkeit
   - Dokumentationspflichten
   - Teamverteilung
3. **Umfang festlegen:**
   - Minimaler vs. maximaler Modellierungsumfang
   - Kosten-Nutzen Abwägung
   - Verfügbare Ressourcen

Modellierungsumfang: Beispielaufgabe

Aufgabe: Ein Softwaresystem soll die Verwaltung von Patientenakten in einer Arztpraxis unterstützen. Das System muss verschiedene gesetzliche Auflagen erfüllen. Bestimmen Sie den notwendigen Modellierungsumfang.
Analyse:

- **Hoher Modellierungsumfang notwendig wegen:**
  - Medizinische Domäne mit hoher Komplexität
  - Gesetzliche Anforderungen (Datenschutz, Dokumentation)
  - Kritische Daten und Prozesse
  - Verschiedene Stakeholder (Ärzte, Personal, Patienten)
- **Erforderliche Modelle:**
  - Detailliertes Domänenmodell
  - Vollständige Use Cases
  - Ausführliche Systemarchitektur
  - Sicherheits- und Datenschutzkonzepte
  - Prozessmodelle für kritische Abläufe

Begründung: Bei einem medizinischen System überwiegen die Vorteile einer ausführlichen Modellierung klar die Kosten:

- Fehler können schwerwiegende Folgen haben
- Nachträgliche Änderungen sind aufwändig
- Dokumentationspflichten müssen erfüllt werden
- Zertifizierungen erfordern genaue Modelle

Requirements vs. Technology Matrix

Analyse-Schritte:

1. **Einordnung Anforderungen:**
   - Known Requirements → Klare Spezifikation
   - Unknown Requirements → Agile Exploration
2. **Einordnung Technologie:**
   - Known Technology → Bewährte Tools/Methoden
   - Unknown Technology → Prototypen/Spikes
3. **Quadranten analysieren:**
   - Known/Known → Wasserfall möglich
   - Known/Unknown → Technische Prototypen
   - Unknown/Known → Agile Methoden
   - Unknown/Unknown → Extreme Prototyping

Requirements vs. Technology: Projektanalyse

Aufgabe: Analysieren Sie folgende Projekte und ordnen Sie sie in die Requirements/Technology Matrix ein:
Projekt 1: Online-Shop Update

- Bekannte E-Commerce Plattform
- Standard-Funktionalitäten
- Bewährte Technologien → **Known/Known Quadrant**

Projekt 2: KI-basierte Diagnose

- Neue Anwendungsdomäne
- Unklare Nutzeranforderungen
- Innovative KI-Technologie → **Unknown/Unknown Quadrant**

Projekt 3: Legacy-System Migration

- Klare Funktionsanforderungen
- Neue Cloud-Technologie
- Unbekannte Performance-Charakteristik → **Known/Unknown Quadrant**

Empfohlene Vorgehensweise:

- Projekt 1: Strukturierter Wasserfall-Ansatz
- Projekt 2: Extreme Prototyping mit kurzen Iterationen
- Projekt 3: Technische Prototypen, dann inkrementelle Migration

KR und Beispiele für Anforderungsanalyse

Use Case Erstellung

Vorgehen bei der Erstellung eines vollständigen Use Cases:

1. **Identifikation:**
   - Primärakteur bestimmen
   - Scope festlegen
   - Ebene definieren (Summary, User-Goal, Subfunction)
2. **Stakeholder und Interessen:**
   - Alle betroffenen Parteien identifizieren
   - Interessen pro Stakeholder beschreiben
   - Priorisierung der Interessen
3. **Bedingungen:**
   - Vorbedingungen definieren
   - Nachbedingungen (Erfolgsfall)
   - Minimalgarantien festlegen
4. **Standardablauf:**
   - Schritte durchnummerieren
   - Akteur-System Interaktion
   - Klare, aktive Formulierung
5. **Erweiterungen/Alternativen:**
   - Fehlerfälle identifizieren
   - Alternative Abläufe beschreiben
   - Auf Standardablauf referenzieren

Fully-dressed Use Case

Aufgabe: Schreiben Sie einen vollständigen Use Case für "Ticket buchenïn einem Hotelreservierungssystem.

Use Case: Hotelzimmer buchen
Scope: Hotelbuchungssystem
Level: User Goal
Primary Actor: Hotel-Gast
Stakeholder und Interessen:

- **Hotel-Gast:**
  - Schnelle, einfache Buchung
  - Bestätigung der Buchung
  - Korrekte Preisberechnung
- **Hotel:**
  - Korrekte Zimmerbelegung
  - Zahlungsgarantie
  - Vollständige Gästeinformationen

Vorbedingungen:

- Gast ist im System angemeldet
- Mindestens ein Zimmer verfügbar

Nachbedingungen:

- Buchung ist gespeichert
- Zimmer ist reserviert
- Bestätigung ist versendet

Standardablauf:

1. Gast wählt Reisedaten aus
2. System zeigt verfügbare Zimmer
3. Gast wählt Zimmer aus
4. System zeigt Buchungsdetails und Gesamtpreis
5. Gast gibt Zahlungsinformationen ein
6. System validiert Zahlungsdaten
7. System bestätigt Buchung
8. System sendet Buchungsbestätigung

Erweiterungen:

- 2a. Keine Zimmer verfügbar:
  1. System zeigt alternative Daten
  2. Gast wählt neue Daten oder bricht ab
- 6a. Zahlung fehlgeschlagen:
  1. System zeigt Fehlermeldung
  2. Gast kann neue Zahlungsdaten eingeben oder abbrechen

Systemsequenzdiagramme erstellen

Vorgehen für SSD:

1. **Akteure identifizieren:**
   - Primärakteur festlegen
   - System als Black Box
   - Zeitachse definieren
2. **Operationen definieren:**
   - Systemoperationen identifizieren
   - Parameter festlegen
   - Rückgabewerte bestimmen
3. **Ablauf modellieren:**
   - Nachrichten einzeichnen
   - Alternative Pfade markieren
   - Schleifen kennzeichnen
4. **Dokumentation:**
   - Beschriftungen prüfen
   - Alternatives ergänzen
   - Bezug zum Use Case herstellen

Systemsequenzdiagramm: Hotelbuchung

Aufgabe: Erstellen Sie ein SSD für den Use Case "Hotelzimmer buchen".

Systemoperationen:

```
// Verfuegbarkeit pruefen
checkAvailability(dates: DateRange): List<Room>

// Zimmer reservieren
bookRoom(roomId: RoomId,
         dates: DateRange): Reservation

// Zahlung durchfuehren
processPayment(reservationId: ReservationId,
               paymentInfo: PaymentDetails):
                   boolean

// Buchung bestaetigen
confirmBooking(reservationId: ReservationId):
    BookingConfirmation
```

Alternative Pfade:

- Keine Verfügbarkeit → Alternative Daten
- Zahlungsfehler → Neue Zahlungsdaten
- Systembuchungsfehler → Fehlermeldung

Contracts für Systemoperationen

Contract-Elemente:

1. **Operation:**
   - Name und Parameter
   - Rückgabetyp
   - Exceptions
2. **Vorbedingungen:**
   - Systemzustand
   - Gültige Parameter
   - Benutzerkontext
3. **Nachbedingungen:**
   - Zustandsänderungen
   - Objekterzeugung
   - Attributänderungen
   - Assoziationen

Contract: Hotelbuchung

Operation: bookRoom(roomId: RoomId, dates: DateRange): Reservation
Querverweis: UC "Hotelzimmer buchen"
Vorbedingungen:

- Benutzer ist authentifiziert
- Zimmer ist im spezifizierten Zeitraum verfügbar
- Zimmer existiert

Nachbedingungen:

- Reservierungs-Instanz wurde erstellt
- Reservierung ist mit Zimmer verknüpft
- Zimmer ist als reserviert markiert
- Reservierungszeitraum ist gesetzt
- Benutzer ist mit Reservierung verknüpft

Usability-Requirements analysieren

Analyseschritte:

1. **Benutzergruppen identifizieren:**
   - Primäre/sekundäre Nutzer
   - Erfahrungsniveau
   - Nutzungskontext
2. **Anforderungen nach ISO 9241-110:**
   - Aufgabenangemessenheit
   - Selbstbeschreibungsfähigkeit
   - Steuerbarkeit
   - Erwartungskonformität
   - Fehlertoleranz
   - Individualisierbarkeit
   - Lernförderlichkeit
3. **Messbare Kriterien definieren:**
   - Erfolgsrate bei Aufgaben
   - Bearbeitungszeit
   - Fehlerrate
   - Nutzerzufriedenheit

Usability-Analyse: Online-Banking

Aufgabe: Analysieren Sie die Usability-Anforderungen für eine Online-Banking App.

Analyse nach ISO 9241-110:

- **Aufgabenangemessenheit:**
  - Schneller Zugriff auf häufige Funktionen
  - Klare Übersicht über Kontostände
  - Effiziente Überweisungsprozesse
- **Selbstbeschreibungsfähigkeit:**
  - Klare Status-Anzeigen
  - Verständliche Fehlermeldungen
  - Hilfe-Funktion
- **Fehlertoleranz:**
  - Bestätigung bei kritischen Aktionen
  - Korrekturmöglichkeiten
  - Plausibilitätsprüfungen

Messbare Kriterien:

- Überweisung in < 60 Sekunden
- Fehlerrate $< 1\%$
- Nutzerzufriedenheit $> 4/5$

KR und Beispiele für Domänenmodellierung

Domänenmodell erstellen

1. Konzepte identifizieren

- **Substantive aus Text extrahieren:**
  - Physische oder virtuelle Objekte
  - Rollen und Akteure
  - Ereignisse und Transaktionen
  - Kataloge und Spezifikationen
- **Konzeptkategorien prüfen:**
  - Geschäftsobjekte
  - Container/Sammlungen
  - Beschreibungen/Spezifikationen
  - Orte/Standorte
  - Transaktionen/Ereignisse

2. Attribute zuordnen

- **Attributregeln:**
  - Nur atomare Werte
  - Keine abgeleiteten Attribute
  - Keine IDs als Attribute
  - Keine Referenzen als Attribute
- **Typische Attribute:**
  - Beschreibende Eigenschaften
  - Status und Zustände
  - Mengen und Werte
  - Zeitangaben

3. Beziehungen modellieren

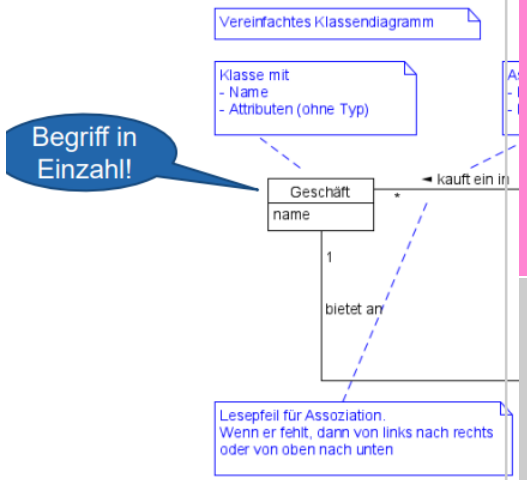
- **Assoziationstypen:**
  - Einfache Assoziation
  - Aggregation/Komposition
  - Vererbung
- **Multiplizitäten festlegen:**
  - 1:1, 1:n, n:m
  - Optionalität (0..1)
  - Mindest-/Maximalwerte

Domänenmodell: Restaurant-System

Aufgabentext aus SEP-Muster: Eine Restaurantkette möchte ihr Bestellsystem modernisieren. Gäste bestellen Speisen und Getränke, die von der Küche bzw. Bar zubereitet werden. Bestellungen werden pro Tisch gesammelt und später gemeinsam oder getrennt bezahlt.

Konzept-Analyse:

- **Physische Objekte:**
  - Tisch
  - Speisen/Getränke
- **Rollen:**
  - Gast
  - Servicepersonal
  - Koch/Barkeeper
- **Transaktionen:**
  - Bestellung
  - Bezahlung



Domänenmodell:

Begründung der Modellierungsentscheidungen:

- **Beschreibungsklassen:**
  - MenueItem für Speisen/Getränke (Trennung von konkreten Bestellpositionen)
  - Produktkategorien für Gruppierung
- **Aggregationen:**
  - Bestellung aggregiert Bestellpositionen
  - Tisch aggregiert Plätze
- **Beziehungen:**
  - Gast sitzt an Platz (1:1)
  - Bestellung gehört zu Tisch (n:1)
  - Position referenziert MenueItem (n:1)

Analysemuster anwenden

1. Beschreibungsklassen

- **Anwendung bei:**
  - Trennung von Typ und Instanz
  - Gemeinsame unveränderliche Eigenschaften
  - Mehrere gleichartige Objekte
- **Implementierung:**
  - Beschreibungsklasse für Typinformationen
  - Instanzklasse für konkrete Objekte
  - 1:n Beziehung zwischen beiden

2. Zustandsmodellierung

- **Anwendung bei:**
  - Komplexe Zustandsübergänge
  - Zustandsabhängiges Verhalten
  - Viele verschiedene Status
- **Implementierung:**
  - Abstrakte Zustandsklasse
  - Konkrete Zustände als Subklassen
  - Assoziationen zum Hauptobjekt

3. Rollen

- **Anwendung bei:**
  - Verschiedene Funktionen eines Objekts
  - Dynamische Rollenzuordnung
  - Unterschiedliche Verantwortlichkeiten
- **Implementierung:**
  - Rolleninterface oder abstrakte Klasse
  - Konkrete Rollenklassen
  - Assoziation zum Basisobjekt

Analysemuster: Bibliothekssystem

Aufgabentext: Modellieren Sie ein Bibliothekssystem mit Büchern, die in mehreren Exemplaren vorliegen können. Benutzer können Bücher ausleihen und reservieren.

Musterlösung mit Analysemuster:

- **Beschreibungsklassen:**
  - Book (Beschreibung: ISBN, Titel, Autor)
  - BookCopy (Instanz: Inventarnummer, Status)
- **Zustandsmodellierung:**
  - LendingState (abstrakt)
  - Available, Borrowed, Reserved als Subklassen
- **Rollen:**
  - Person (Basis)
  - Member, Librarian als Rollen

Begründung der Muster:

- Beschreibungsklassen trennen unveränderliche Buchdaten von Exemplaren
- Zustandsmuster ermöglicht komplexe Statusübergänge und -validierung
- Rollenmuster erlaubt verschiedene Berechtigungen und Funktionen

Typische Modellierungsfehler vermeiden

1. Konzeptuelle Fehler

- **Vermeiden:**
  - Technische statt fachliche Klassen
  - Prozesse als Klassen
  - Operationen im Domänenmodell
- **Richtig:**
  - Fachliche Konzepte modellieren
  - Prozesse durch Beziehungen
  - Nur Attribute im Modell

2. Strukturelle Fehler

- **Vermeiden:**
  - IDs als Attribute
  - Referenzen als Attribute
  - Redundante Attribute
- **Richtig:**
  - Assoziationen statt IDs
  - Abgeleitete Attribute weglassen
  - Informationen zentralisieren

Modellierungsfehler erkennen

Fehlerhaftes Modell:

- Klasse 'OrderManager' mit CRUD-Operationen
- Attribut 'customerID' statt Assoziation
- Klasse 'PaymentProcess' für Ablauf
- Operation 'calculateTotal()' in Order

Korrigiertes Modell:

- Klasse 'Order' mit fachlichen Attributen
- Assoziation zwischen Order und Customer
- Payment als eigenständiges Konzept
- Keine Operationen im Modell

Begründung:

- Technische Manager-Klassen gehören nicht ins Domänenmodell
- IDs werden durch Assoziationen ersetzt
- Prozesse werden durch Beziehungen und Status modelliert
- Operationen gehören ins Designmodell

KR und Beispiele für Softwarearchitektur und Design

Architekturanalyse durchführen

1. Requirements analysieren

- **Funktionale Anforderungen:**
  - Use Cases gruppieren
  - Systemgrenzen definieren
  - Schnittstellen identifizieren
- **Nicht-funktionale Anforderungen:**
  - FURPS+ Kategorien prüfen
  - ISO 25010 Qualitätsmerkmale
  - Priorisierung vornehmen
- **Randbedingungen:**
  - Technische Constraints
  - Organisatorische Vorgaben
  - Budget und Zeitrahmen

2. Architekturentscheidungen:

- **Variationspunkte analysieren:**
  - Veränderliche Komponenten
  - Austauschbare Teile
  - Erweiterungsmöglichkeiten
- **Entscheidungen dokumentieren:**
  - Problem beschreiben
  - Alternativen evaluieren
  - Entscheidung begründen

Architekturanalyse: Hotelsystem

Aus Muster-SEP: Analysieren Sie die Anforderungen zur Ansteuerung des automatischen Zimmerverschlusssystems (ZVS).

Analyse:

- **System-Schnittstelle:**
  - ZVS arbeitet autonom
  - Benötigt nur Karten-ID und Zimmer-Zuordnung
  - Hauslieferant für ZVS festgelegt
- **Architektur-Entscheidung:**
  - **Kein Variationspunkt nötig da:**
    - Gleicher Lieferant für alle Hotels
    - Schnittstelle stabil
    - Keine Änderungen geplant
  - **Trotzdem Adaption empfohlen:**
    - Technische Änderungen möglich
    - Adapter-Pattern einfach umsetzbar
    - Zukünftige Flexibilität

Layered Architecture Design

1. Schichten definieren

- **Presentation Layer:**
  - UI-Komponenten
  - Controller
  - View Models
- **Application Layer:**
  - Services
  - Use Case Implementation
  - Koordination
- **Domain Layer:**
  - Business Objects
  - Domain Logic
  - Domain Services
- **Infrastructure Layer:**
  - Persistence
  - External Services
  - Technical Services

2. Regeln definieren:

- Abhängigkeiten nur nach unten
- Schichten über Interfaces verbinden
- Domänenlogik isolieren

Schichtenarchitektur: SafeHome

Aus Muster-SEP: Entwerfen Sie die Architektur für ein Sicherheitssystem.

Schichtenmodell:

```java
// Presentation Layer
public class SecurityController {
    private SecurityService securityService;

    public void arm(String code) {
        securityService.armSystem(code);
    }
}

// Application Layer
public class SecurityService {
    private AlarmSystem alarmSystem;
    private CodeValidator validator;

    public void armSystem(String code) {
        if (validator.isValid(code)) {
            alarmSystem.arm();
        }
    }
}

// Domain Layer
public class AlarmSystem {
    private List<Sensor> sensors;
    private AlarmState state;

    public void arm() {
        validateSystemState();
        state = state.arm();
        activateSensors();
    }
}

// Infrastructure Layer
public class SensorRepository {
    public List<Sensor> getActiveSensors() {
        // DB access
    }
}
```

Architekturdokumentation erstellen

1. Überblick

- **Systemkontext:**
  - Externe Systeme
  - Akteure/Benutzer
  - Schnittstellen
- **Architekturziele:**
  - Qualitätsattribute
  - Randbedingungen
  - Designprinzipien

2. Entscheidungen

- **Architekturentscheidungen:**
  - Problem/Kontext
  - Alternativen
  - Begründung
- **Muster/Konzepte:**
  - Verwendete Patterns
  - Architekturstile
  - Prinzipien

Architekturdokumentation: SafeHome

Aus Muster-SEP: Dokumentieren Sie die Architekturentscheidungen für das Sicherheitssystem.

1. Systemkontext

- **Externe Systeme:**
  - Sensoren (Bewegung, Feuer, Wasser)
  - Aktoren (Alarm, Licht)
  - Mobile Clients
- **Schnittstellen:**
  - REST API für Remote-Zugriff
  - Hardware-Protokolle für Sensoren
  - Event-Bus für Benachrichtigungen

2. Architekturmuster

- **State Pattern:**
  - Für Systemzustände (Armed, Disarmed)
  - Zustandsübergänge kontrollieren
  - Zustandsspezifisches Verhalten
- **Observer Pattern:**
  - Für Sensor-Events
  - Benachrichtigung bei Alarmen
  - Lose Kopplung

Code-Beispiel:

```java
// State Pattern Implementation
public interface SystemState {
    void arm();
    void disarm(String code);
    void handleSensorTriggered(Sensor sensor);
}

public class ArmedState implements SystemState {
    private AlarmSystem system;

    @Override
    public void handleSensorTriggered(Sensor sensor) {
        if (sensor.requiresImmediateResponse()) {
            system.triggerAlarm();
        } else {
            system.startEntryTimer();
        }
    }
}

// Observer Pattern Implementation
public interface SensorListener {
    void onSensorTriggered(SensorEvent event);
}

public class AlarmSystem implements SensorListener {
    @Override
    public void onSensorTriggered(SensorEvent event) {
        currentState.handleSensorTriggered(
            event.getSensor());
    }
}
```

Qualitätsanforderungen prüfen

1. ISO 25010 Kategorien

- **Performance:**
  - Response Times
  - Throughput
  - Resource Usage
- **Reliability:**
  - Fault Tolerance
  - Recovery
  - Availability
- **Security:**
  - Authentication
  - Authorization
  - Data Protection

2. Szenario-basierte Bewertung

- **Szenario definieren:**
  - Auslöser/Stimulus
  - Umgebung/Kontext
  - Erwartete Reaktion
- **Architektur prüfen:**
  - Mechanismen identifizieren
  - Risiken bewerten
  - Maßnahmen definieren

Qualitätsszenarien: SafeHome

Performance-Szenario:

- **Stimulus:** Sensor meldet Einbruch
- **Umgebung:** System im Armed-Zustand
- **Response:** Alarm innerhalb 500ms
- **Messung:** Reaktionszeit < 500ms in 99.9

Verfügbarkeits-Szenario:

- **Stimulus:** Hardware-Komponente fällt aus
- **Umgebung:** Normalbetrieb
- **Response:** System bleibt funktionsfähig
- **Messung:** 99.99

Architekturmaßnahmen:

```java
// Performance Optimization
@Component
public class SensorEventProcessor {
    private BlockingQueue<SensorEvent> eventQueue;
    private ThreadPoolExecutor executor;

    @Async
    public void processSensorEvent(SensorEvent event) {
        eventQueue.offer(event);
        executor.execute(() -> processEvent(event));
    }
}

// High Availability
public class RedundantAlarmSystem {
    private List<AlarmDevice> alarmDevices;

    public void triggerAlarm() {
        for (AlarmDevice device : alarmDevices) {
            try {
                device.activate();
                return; // Success
            } catch (DeviceFailureException e) {
                // Try next device
                continue;
            }
        }
    }
}
```

KR und Beispiele für Architekturmuster

Architekturmuster auswählen

1. Pattern-Analyse

- **Problemkontext:**
  - Art der Anwendung
  - Verteilungsanforderungen
  - Qualitätsattribute
- **Pattern-Katalog durchsuchen:**
  - Layered Architecture
  - Client-Server
  - Pipe-Filter
  - Event-Bus
  - Master-Slave
- **Trade-offs evaluieren:**
  - Vorteile/Nachteile
  - Komplexität vs. Flexibilität
  - Implementierungsaufwand

2. Implementierung planen

- **Struktur:**
  - Komponenten definieren
  - Schnittstellen festlegen
  - Interaktionen beschreiben
- **Qualitätssicherung:**
  - Testbarkeit berücksichtigen
  - Performance-Aspekte
  - Wartbarkeit sicherstellen

Event-Bus Pattern: SafeHome

Aufgabe: Implementieren Sie das Event-Bus Pattern für die Sensor-Kommunikation im SafeHome System.

```java
// Event Klasse
public class SensorEvent {
    private String sensorId;
    private SensorType type;
    private LocalDateTime timestamp;
    private Map<String, Object> data;

    // Constructor und Getter
}

// Event Bus
public class SecurityEventBus {
    private Map<Class<?>, List<EventHandler>> handlers
        =
        new HashMap<>();

    public void publish(SensorEvent event) {
        List<EventHandler> eventHandlers =
            handlers.getOrDefault(
                event.getClass(),
                Collections.emptyList()
            );

        for (EventHandler handler : eventHandlers) {
            try {
                handler.handle(event);
            } catch (Exception e) {
                handleError(handler, event, e);
            }
        }
    }

    public void subscribe(Class<?> eventType,
                          EventHandler handler) {
        handlers.computeIfAbsent(
            eventType,
            k -> new ArrayList<>()
        ).add(handler);
    }
}

// Event Handler Implementation
@Component
public class MotionSensorHandler
    implements EventHandler<MotionEvent> {

    private AlarmSystem alarmSystem;

    @Override
    public void handle(MotionEvent event) {
        if (alarmSystem.isArmed()) {
            evaluateMotion(event);
        }
    }

    private void evaluateMotion(MotionEvent event) {
        if (event.getIntensity() > THRESHOLD) {
            alarmSystem.startEntryTimer();
        }
    }
}
```

Client-Server Pattern implementieren

1. Komponenten definieren

- **Server:**
  - Service-Interfaces
  - Request-Handling
  - Resource-Management
- **Client:**
  - Service-Proxies
  - Error-Handling
  - UI/Interaktion
- **Protokoll:**
  - Nachrichten-Format
  - Statushandling
  - Security

2. Implementierungsaspekte

- **Kommunikation:**
  - Synchron/Asynchron
  - Serialisierung
  - Fehlerbehandlung
- **Skalierung:**
  - Load Balancing
  - Caching
  - Connection Pooling

Client-Server: SafeHome Mobile App

Aufgabe: Implementieren Sie die Client-Server Architektur für die mobile SafeHome App.

```
// Server-Side Controller
@RestController
@RequestMapping("/api/v1/security")
public class SecurityController {
    private SecurityService securityService;

    @PostMapping("/arm")
    public ResponseEntity<SystemStatus> armSystem(
            @RequestBody ArmRequest request) {
        try {
            SystemStatus status =
                securityService.armSystem(
                    request.getCode());
            return ResponseEntity.ok(status);
        } catch (InvalidCodeException e) {
            return ResponseEntity
                .status(HttpStatus.UNAUTHORIZED)
                .build();
        }
    }

    @GetMapping("/status")
    public ResponseEntity<SystemStatus> getStatus() {
        return ResponseEntity.ok(
            securityService.getCurrentStatus());
    }
}

// Client-Side Service
public class SecurityClient {
    private final WebClient webClient;
    private final String baseUrl;

    public Mono<SystemStatus> armSystem(String code) {
        return webClient.post()
            .uri(baseUrl + "/arm")
            .body(new ArmRequest(code))
            .retrieve()
            .bodyToMono(SystemStatus.class)
            .timeout(Duration.ofSeconds(5))
            .retry(3)
            .onErrorResume(this::handleError);
    }

    private Mono<SystemStatus> handleError(Throwable
        error) {
        if (error instanceof TimeoutException) {
            return Mono.error(
                new ConnectionException("Timeout"));
        }
        return Mono.error(error);
    }
}

// Mobile App Component
public class SecurityViewModel {
    private SecurityClient securityClient;
    private MutableLiveData<SystemStatus> status =
        new MutableLiveData<>();

    public void armSystem(String code) {
        securityClient.armSystem(code)
            .subscribe(
                status::setValue,
                this::handleError
            );
    }
}
```

Master-Slave Pattern implementieren

1. Komponenten

- **Master:**
  - Aufgabenverteilung
  - Ergebnissammlung
  - Fehlerhandling
- **Slaves:**
  - Aufgabenausführung
  - Statusmeldungen
  - Autonome Arbeit

2. Koordination

- **Verteilung:**
  - Load Balancing
  - Resource Management
  - Failover
- **Monitoring:**
  - Health Checks
  - Performance Metrics
  - Error Tracking

Master-Slave: Sensor Processing

Aufgabe: Implementieren Sie eine Master-Slave Architektur für die Verarbeitung von Sensordaten.

```
// Master Component
public class SensorMaster {
    private List<SensorSlave> slaves;
    private Map<String, SensorData> results =
        new ConcurrentHashMap<>();

    public void processSensorData(List<SensorInput>
        inputs) {
        // Distribute work
        Map<SensorSlave, List<SensorInput>>
            distribution =
            distributeWork(inputs);

        // Start processing
        List<Future<SensorData>> futures =
            submitTasks(distribution);

        // Collect results
        collectResults(futures);
    }

    private Map<SensorSlave, List<SensorInput>>
            distributeWork(List<SensorInput> inputs) {
        Map<SensorSlave, List<SensorInput>>
            distribution =
            new HashMap<>();

        int slaveIndex = 0;
        for (SensorInput input : inputs) {
            SensorSlave slave = slaves.get(
                slaveIndex % slaves.size());
            distribution.computeIfAbsent(
                slave,
                k -> new ArrayList<>()
            ).add(input);
            slaveIndex++;
        }

        return distribution;
    }
}

// Slave Component
public class SensorSlave {
    private String slaveId;
    private SensorProcessor processor;

    public SensorData process(SensorInput input) {
        try {
            return processor.process(input);
        } catch (Exception e) {
            handleError(e);
            return SensorData.error(input, e);
        }
    }

    public HealthStatus checkHealth() {
        return new HealthStatus(
            slaveId,
            processor.getStatus(),
            getResourceMetrics()
        );
    }
}

// Coordinator
public class SensorCoordinator {
    private SensorMaster master;
```

KR und Beispiele für Architektur-Evaluation

Architektur-Evaluation durchführen

1. Evaluationskriterien definieren

- **Qualitätsattribute:**
  - Erfüllung der ISO 25010 Kriterien
  - Priorisierte Qualitätsziele
  - Messbare Metriken
- **Business-Ziele:**
  - Time-to-Market
  - Entwicklungskosten
  - Wartungsaufwand
- **Technische Ziele:**
  - Skalierbarkeit
  - Integrationsfähigkeit
  - Technologie-Stack

2. Evaluation durchführen

- **Szenarien prüfen:**
  - Use Case Realisierung
  - Qualitätsszenarien
  - Änderungsszenarien
- **Risiken analysieren:**
  - Technische Risiken
  - Abhängigkeiten
  - Komplexität

KR und Beispiele für Architektur-Evaluation

Architektur-Evaluation durchführen

1. Evaluationskriterien definieren

- **Qualitätsattribute:**
  - Erfüllung der ISO 25010 Kriterien
  - Priorisierte Qualitätsziele
  - Messbare Metriken
- **Business-Ziele:**
  - Time-to-Market
  - Entwicklungskosten
  - Wartungsaufwand
- **Technische Ziele:**
  - Skalierbarkeit
  - Integrationsfähigkeit
  - Technologie-Stack

2. Evaluation durchführen

- **Szenarien prüfen:**
  - Use Case Realisierung
  - Qualitätsszenarien
  - Änderungsszenarien
- **Risiken analysieren:**
  - Technische Risiken
  - Abhängigkeiten
  - Komplexität

Architektur-Evaluation: SafeHome

Aufgabe: Evaluieren Sie die Architektur des SafeHome Systems bezüglich Zuverlässigkeit und Skalierbarkeit.

1. Qualitätsszenarien

- **Zuverlässigkeit:**
  - **Szenario:** Sensorausfall
  - **Stimulus:** Sensor antwortet nicht
  - **Response:** System bleibt funktionsfähig
  - **Metrik:** 99.99
- **Skalierbarkeit:**
  - **Szenario:** Erhöhte Sensoranzahl
  - **Stimulus:** 100 neue Sensoren
  - **Response:** Performance stabil
  - **Metrik:** $<$ 500ms Reaktionszeit

2. Architektur-Review

```java
// Reliability Pattern Implementation
public class SensorManager {
    private Map<String, Sensor> sensors;
    private CircuitBreaker circuitBreaker;

    @Retry(maxAttempts = 3)
    public SensorStatus checkSensor(String sensorId) {
        return circuitBreaker.execute(() ->
            sensors.get(sensorId)
                .getStatus());
    }

    public void handleSensorFailure(String sensorId) {
        // Deactivate failed sensor
        sensors.get(sensorId).deactivate();

        // Notify system
        notifySystemStatus(
            String.format(
                "Sensor %s failed, system operating in
                    degraded mode",
                sensorId
            )
        );

        // Adjust monitoring
        updateMonitoringStrategy();
    }
}

// Scalability Pattern Implementation
public class SensorEventProcessor {
    private Queue<SensorEvent> eventQueue;
    private ThreadPoolExecutor executor;

    public SensorEventProcessor(int maxThreads) {
        this.eventQueue = new LinkedBlockingQueue<>();
        this.executor = new ThreadPoolExecutor(
            2, maxThreads,
            60L, TimeUnit.SECONDS,
            new LinkedBlockingQueue<>()
        );
    }

    public void processEvent(SensorEvent event) {
        executor.submit(() -> {
            try {
                handleEvent(event);
            } catch (Exception e) {
                handleProcessingError(event, e);
            }
        });
    }
```

Variationspunkte analysieren

1. Identifikation

- **Technische Variation:**
  - Datenbankanbindung
  - UI-Framework
  - Protokolle
- **Fachliche Variation:**
  - Geschäftsregeln
  - Workflows
  - Berechnungen
- **Konfiguration:**
  - Parameter
  - Grenzwerte
  - Features

2. Design

- **Pattern-Auswahl:**
  - Strategy Pattern
  - Factory Pattern
  - Plugin-System
- **Interface-Design:**
  - Abstraktion
  - Stabilität
  - Erweiterbarkeit

Variationspunkte: SafeHome

Aufgabe: Identifizieren und implementieren Sie Variationspunkte im SafeHome System.

1. Analyse

- **Sensor-Protokolle:**
  - Verschiedene Hersteller
  - Unterschiedliche Kommunikationsprotokolle
  - Erweiterbarkeit für neue Sensoren
- **Alarmierung:**
  - Multiple Benachrichtigungskanäle
  - Konfigurierbare Eskalation
  - Kundenspezifische Regeln

2. Implementation

```
// Sensor Protocol Variation
public interface SensorProtocol {
    void initialize();
    SensorData readData();
    void sendCommand(Command cmd);
}

public class ZigBeeSensor implements SensorProtocol {
    @Override
    public void initialize() {
        // ZigBee specific initialization
    }

    @Override
    public SensorData readData() {
        // Read from ZigBee sensor
        return new SensorData(/* ... */);
    }
}

// Notification Variation
public interface NotificationChannel {
    void sendAlert(AlertLevel level, String message);
    boolean isAvailable();
    Priority getPriority();
}

public class NotificationService {
    private List<NotificationChannel> channels;
    private NotificationConfig config;

    public void notify(Alert alert) {
        // Sort channels by priority
        List<NotificationChannel> availableChannels =
            channels.stream()
                .filter(NotificationChannel::isAvailable)
                .sorted(comparing(
                    NotificationChannel::getPriority))
                .collect(toList());

        // Try channels until successful
        for (NotificationChannel channel :
             availableChannels) {
            try {
                channel.sendAlert(
                    alert.getLevel(),
                    alert.getMessage()
                );
                return;
            } catch (NotificationException e) {
                // Try next channel
                continue;
            }
        }

        // No channel available
```

KR und Beispiele für Use Case Realization

Use Case Realization durchführen

1. Vorbereitung

- **Use Case analysieren:**
  - Standardablauf identifizieren
  - Wichtige Erweiterungen identifizieren
  - Systemoperationen aus SSD extrahieren
- **Domänenmodell prüfen:**
  - Benötigte Klassen identifizieren
  - Beziehungen validieren
  - Fehlende Konzepte ergänzen

2. Design

- **Controller bestimmen:**
  - Use Case oder Fassaden Controller
  - Verantwortlichkeiten definieren
  - Schnittstellen festlegen
- **Interaktionen modellieren:**
  - Sequenzdiagramm erstellen
  - GRASP Prinzipien anwenden
  - Patterns einsetzen

3. Implementation

- **Code strukturieren:**
  - Package-Struktur
  - Schichtenarchitektur
  - Abhängigkeiten
- **Tests erstellen:**
  - Unit Tests
  - Integrationstests
  - Use Case Tests

Use Case Realization: Forum

Use Case: Neue Diskussion erstellen

Systemoperationen:

- addNewDiscussion(title, content)
- getNumberOfContributions()

```java
// Controller
public class ForumController {
    private ForumService forumService;
    private AccessValidator accessValidator;

    public Discussion addNewDiscussion(
            String topicId,
            DiscussionRequest request) {
        // Validate access
        accessValidator.validateUserAccess(
            request.getUserId());

        // Create discussion
        return forumService.createDiscussion(
            topicId,
            request.getTitle(),
            request.getContent()
        );
    }
}

// Domain Model
public class Discussion {
    private String title;
    private String content;
    private User author;
    private List<Comment> comments;
    private LocalDateTime createdAt;

    public void addComment(Comment comment) {
        validateComment(comment);
        comments.add(comment);
    }

    public int getContributionCount() {
        return 1 + comments.size(); // Discussion +
            Comments
    }
}

// Service Layer
public class ForumService {
    private DiscussionRepository repository;
    private TopicRepository topicRepository;

    @Transactional
    public Discussion createDiscussion(
            String topicId,
            String title,
            String content) {
        // Find topic
        Topic topic = topicRepository
            .findById(topicId)
            .orElseThrow(TopicNotFoundException::new);

        // Create discussion
        Discussion discussion = new Discussion(
            title, content);

        // Add to topic
        topic.addDiscussion(discussion);

        // Save
        return repository.save(discussion);
    }
}
```

GRASP Prinzipien anwenden

1. Information Expert

- **Identifikation:**
  - Welche Klasse hat die Information?
  - Wo liegen die relevanten Daten?
  - Wer kennt die Berechnungsgrundlagen?
- **Anwendung:**
  - Methoden dort platzieren wo die Daten sind
  - Berechnungen in Expertenklasse
  - Delegieren wenn nötig

2. Low Coupling

- **Analyse:**
  - Abhängigkeiten identifizieren
  - Kritische Kopplungen erkennen
  - Alternatives Design prüfen
- **Maßnahmen:**
  - Interfaces einführen
  - Dependency Injection
  - Vermittler einsetzen

3. High Cohesion

- **Prüfung:**
  - Zusammengehörigkeit der Methoden
  - Fokus der Klasse
  - Aufgabenteilung
- **Verbesserung:**
  - Klassen aufteilen
  - Verantwortlichkeiten gruppieren
  - Hilfsklassen einführen

GRASP Anwendung: Online Shop

Use Case: Bestellung aufgeben

```java
// Information Expert: Order berechnet eigenen Gesamtbetrag
public class Order {
    private List<OrderLine> lines;
    private Customer customer;

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

// Low Coupling: Interface statt konkreter Implementierung
public interface PaymentGateway {
    PaymentResult processPayment(Money amount);
}

public class OrderService {
    private final PaymentGateway paymentGateway;

    public OrderResult createOrder(OrderRequest request) {
        Order order = createOrderFromRequest(request);

        PaymentResult payment =
            paymentGateway.processPayment(
                order.calculateTotal());

        if (payment.isSuccessful()) {
            return OrderResult.success(order);
        } else {
            return
                OrderResult.failed(payment.getReason());
        }
    }
}

// High Cohesion: Spezialisierte Services
public class OrderValidator {
    public void validate(Order order) {
        validateCustomer(order.getCustomer());
        validateOrderLines(order.getLines());
        validateDeliveryAddress(
            order.getDeliveryAddress());
    }
}

public class InventoryService {
    public void reserveStock(Order order) {
        for (OrderLine line : order.getLines()) {
            reserveItem(
                line.getProduct(),
                line.getQuantity());
        }
    }
}
```

Interaction Diagrams erstellen

1. Sequenzdiagramm

- **Elemente:**
  - Lebenslinien für Objekte
  - Nachrichten (synchron/asynchron)
  - Aktivierungsbalken
  - Alternative Abläufe
- **Best Practices:**
  - Von links nach rechts lesen
  - Wichtige Objekte links
  - Klare Beschriftungen
  - Rückgabewerte zeigen

2. Kommunikationsdiagramm

- **Elemente:**
  - Objekte als Rechtecke
  - Nummerierte Nachrichten
  - Assoziationen als Linien
- **Best Practices:**
  - Übersichtliche Anordnung
  - Klare Nummerierung
  - Wichtige Beziehungen hervorheben

Interaction Diagrams: Bestellprozess

Use Case: Bestellung aufgeben

Sequenzdiagramm Code:

```java
@RestController
public class OrderController {
    private OrderService orderService;
    private PaymentService paymentService;

    public OrderResponse createOrder(
            OrderRequest request) {
        // 1: Validiere Bestellung
        OrderValidator.validate(request);

        // 2: Erstelle Order
        Order order =
            orderService.createOrder(request);

        // 3: Prozessiere Zahlung
        PaymentResult payment =
            paymentService.processPayment(
                order.getId(),
                order.getTotal());

        // 4: Bestaetige Order
        if (payment.isSuccessful()) {
            order.confirm();
            orderService.save(order);
            return OrderResponse.success(order);
        } else {
            return OrderResponse.failed(
                payment.getReason());
        }
    }
}

public class OrderService {
    @Transactional
    public Order createOrder(OrderRequest request) {
        // 2.1: Create order entity
        Order order = new Order(
            request.getCustomerId());

        // 2.2: Add items
        for (OrderItemRequest item :
            request.getItems()) {
            Product product = productRepository
                .findById(item.getProductId())
                .orElseThrow(ProductNotFoundException::new);

            order.addItem(
                product,
                item.getQuantity());
        }

        // 2.3: Save order
        return orderRepository.save(order);
    }
}
```

Sequenzdiagramm Analyse:

- **Objekte:**
  - OrderController als Fassade
  - OrderService für Geschäftslogik
  - PaymentService für Zahlungen
  - Order als Domain Object
- **Verantwortlichkeiten:**
  - Controller: Koordination
  - Service: Transaktionslogik
  - Domain: Business Rules

KR und Beispiele für Design Patterns

Design Pattern Auswahl

1. Problemanalyse

- **Kontext verstehen:**
  - Art des Problems
  - Flexibilitätsanforderungen
  - Qualitätsattribute
- **Pattern Kategorien:**
  - Creational Patterns für Objekterzeugung
  - Structural Patterns für Beziehungen
  - Behavioral Patterns für Verhalten

2. Pattern auswählen

- **Kriterien:**
  - Passend zum Problem
  - Angemessene Komplexität
  - Kombination mit anderen Patterns
- **Trade-offs:**
  - Flexibilität vs. Komplexität
  - Performance vs. Erweiterbarkeit
  - Einfachheit vs. Wiederverwendbarkeit

Pattern-Analyse: Document Processing

Aufgabe: Ein System soll verschiedene Dokumenttypen (PDF, DOC, TXT) verarbeiten können.

Analyse:

- **Anforderungen:**
  - Unterschiedliche Parser pro Format
  - Erweiterbar für neue Formate
  - Einheitliche Verarbeitungsschnittstelle
- **Patterns:**
  - Factory Method für Parser-Erzeugung
  - Strategy für Verarbeitungsalgorithmen
  - Template Method für allgemeinen Prozess

Implementation:

```java
// Template Method Pattern
public abstract class DocumentProcessor {
    public final void processDocument(byte[] data) {
        Document doc = parseDocument(data);
        validateDocument(doc);
        processContent(doc);
        saveResults(doc);
    }

    protected abstract Document parseDocument(byte[]
        data);
    protected abstract void processContent(Document
        doc);

    // Hook methods mit Default-Implementation
    protected void validateDocument(Document doc) {
        if (doc.isEmpty()) {
            throw new EmptyDocumentException();
        }
    }

    protected void saveResults(Document doc) {
        // Default-Speicherung
    }
}

// Factory Method Pattern
public abstract class DocumentParserFactory {
    public abstract DocumentParser
        createParser(String fileType);
}

public class PDFParserFactory
        extends DocumentParserFactory {
    @Override
    public DocumentParser createParser(String
        fileType) {
        if ("pdf".equals(fileType)) {
            return new PDFParser();
        }
        throw new UnsupportedFormatException(fileType);
    }
}

// Strategy Pattern
public interface ProcessingStrategy {
    void process(Document doc);
}

public class TextExtractionStrategy
        implements ProcessingStrategy {
    @Override
    public void process(Document doc) {
        // Text aus Dokument extrahieren
    }
}
```

Pattern Implementierung

1. Strukturierung

- **Klassen definieren:**
  - Pattern-Rollen identifizieren
  - Interfaces festlegen
  - Beziehungen modellieren
- **Flexibilität einbauen:**
  - Erweiterungspunkte
  - Loose Coupling
  - Interface Segregation

2. Best Practices

- **Prinzipien beachten:**
  - SOLID Principles
  - GRASP Patterns
  - Clean Code
- **Testbarkeit:**
  - Unit Tests pro Pattern
  - Integrationstests
  - Edge Cases

Pattern Implementation: Event System

Aufgabe: Implementieren Sie ein Event-System mit Observer und Command Pattern.

```java
// Observer Pattern
public interface EventListener<T extends Event> {
    void onEvent(T event);
}

public class EventBus {
    private Map<Class<?>, 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.getOrDefault(
                event.getClass(),
                Collections.emptyList()
            );

        for (EventListener listener : eventListeners) {
            try {
                listener.onEvent(event);
            } catch (Exception e) {
                handleError(listener, event, e);
            }
        }
    }
}

// Command Pattern
public interface Command {
    void execute();
    void undo();
}

public class CreateOrderCommand implements Command {
    private OrderService orderService;
    private OrderRequest request;
    private Order createdOrder;

    @Override
    public void execute() {
        createdOrder =
             orderService.createOrder(request);
    }

    @Override
    public void undo() {
        if (createdOrder != null) {
            orderService.cancelOrder(createdOrder.getId());
        }
    }
}

public class CommandProcessor {
    private Deque<Command> executedCommands =
        new ArrayDeque<>();

    public void execute(Command command) {
        command.execute();
        executedCommands.push(command);
    }
```

Pattern Kombinationen

1. Analyse

- **Komplexe Anforderungen:**
  - Mehrere Probleme identifizieren
  - Abhängigkeiten erkennen
  - Interaktionen planen
- **Pattern-Auswahl:**
  - Komplementäre Patterns
  - Verschachtelungsmöglichkeiten
  - Integration planen

2. Implementation

- **Struktur:**
  - Klare Hierarchie
  - Definierte Schnittstellen
  - Saubere Integration
- **Qualität:**
  - Testbarkeit sicherstellen
  - Komplexität kontrollieren
  - Dokumentation erstellen

Pattern Kombination: Plugin System

Aufgabe: Implementieren Sie ein Plugin-System mit Factory, Strategy und Observer Pattern.

```java
// Plugin Interface (Strategy Pattern)
public interface Plugin {
    String getName();
    void initialize();
    void processData(Data data);
}

// Plugin Factory
public abstract class PluginFactory {
    public abstract Plugin createPlugin(
        String pluginType);

    protected void validatePlugin(Plugin plugin) {
        // Validation logic
    }
}

// Plugin Manager (Observer Pattern)
public class PluginManager {
    private List<PluginObserver> observers =
        new ArrayList<>();
    private Map<String, Plugin> activePlugins =
        new HashMap<>();

    public void registerPlugin(Plugin plugin) {
        plugin.initialize();
        activePlugins.put(plugin.getName(), plugin);
        notifyObservers(
            new PluginEvent(
                PluginEventType.REGISTERED,
                plugin
            )
        );
    }

    public void processData(Data data) {
        for (Plugin plugin : activePlugins.values()) {
            try {
                plugin.processData(data);
            } catch (Exception e) {
                handlePluginError(plugin, e);
            }
        }
    }

    private void notifyObservers(PluginEvent event) {
        for (PluginObserver observer : observers) {
            observer.onPluginEvent(event);
        }
    }
}

// Concrete Implementation
public class ImageProcessingPlugin implements Plugin {
    private ProcessingStrategy strategy;

    @Override
    public void processData(Data data) {
        if (data instanceof ImageData) {
            strategy.process((ImageData) data);
        }
    }
}

// Usage
public class Application {
    public void initializePlugins() {
        PluginManager manager = new PluginManager();
```

Weitere KR und Beispiele für Design Patterns

Chain of Responsibility implementieren

1. Handler Struktur

- **Handler Interface:**
  - Einheitliche Methode
  - Nächster Handler
  - Behandlungslogik
- **Konkrete Handler:**
  - Spezifische Logik
  - Weitergabekriterien
  - Fehlerbedingungen

2. Kette aufbauen

- **Reihenfolge:**
  - Prioritäten beachten
  - Abhängigkeiten prüfen
  - Standardhandler
- **Flexibilität:**
  - Dynamische Kette
  - Konfigurierbar
  - Erweiterbar

Weitere KR und Beispiele für Design Patterns

Chain of Responsibility implementieren

1. Handler Struktur

- **Handler Interface:**
  - Einheitliche Methode
  - Nächster Handler
  - Behandlungslogik
- **Konkrete Handler:**
  - Spezifische Logik
  - Weitergabekriterien
  - Fehlerbedingungen

2. Kette aufbauen

- **Reihenfolge:**
  - Prioritäten beachten
  - Abhängigkeiten prüfen
  - Standardhandler
- **Flexibilität:**
  - Dynamische Kette
  - Konfigurierbar
  - Erweiterbar

Chain of Responsibility: Authentication

Aufgabe: Implementieren Sie eine Authentifizierungskette mit verschiedenen Validierungen.

```java
// Handler Interface
public interface AuthHandler {
    void setNext(AuthHandler next);
    void handle(AuthRequest request)
        throws AuthException;
}

// Abstract Base Handler
public abstract class BaseAuthHandler
        implements AuthHandler {
    protected AuthHandler nextHandler;

    @Override
    public void setNext(AuthHandler next) {
        this.nextHandler = next;
    }

    protected void handleNext(AuthRequest request)
            throws AuthException {
        if (nextHandler != null) {
            nextHandler.handle(request);
        }
    }
}

// Concrete Handlers
public class TokenValidationHandler
        extends BaseAuthHandler {
    @Override
    public void handle(AuthRequest request)
            throws AuthException {
        String token = request.getToken();
        if (token == null || token.isEmpty()) {
            throw new AuthException("Missing token");
        }
        if (!isValidToken(token)) {
            throw new AuthException("Invalid token");
        }
        handleNext(request);
    }
}

public class RoleCheckHandler extends BaseAuthHandler {
    private Set<String> requiredRoles;

    @Override
    public void handle(AuthRequest request)
            throws AuthException {
        Set<String> userRoles = getUserRoles(
            request.getUserId());
        if (!userRoles.containsAll(requiredRoles)) {
            throw new AuthException(
                "Insufficient privileges");
        }
        handleNext(request);
    }
}

// Chain Builder
public class AuthChainBuilder {
    public AuthHandler buildChain() {
        AuthHandler tokenHandler =
            new TokenValidationHandler();
        AuthHandler roleHandler =
            new RoleCheckHandler(ADMIN_ROLES);
        AuthHandler ipHandler =
            new IPValidationHandler();

```

State Pattern implementieren

1. Zustände modellieren

- **State Interface:**
  - Gemeinsame Methoden
  - Zustandsübergänge
  - Kontextbezug
- **Konkrete Zustände:**
  - Zustandsspezifisches Verhalten
  - Übergangsbedingungen
  - Validierungen

2. Kontext implementieren

- **Zustandsverwaltung:**
  - Aktueller Zustand
  - Zustandswechsel
  - Historie
- **Delegation:**
  - Methodenweiterleitung
  - Zustandszugriff
  - Fehlerbehandlung

State Pattern: Document Workflow

Aufgabe: Implementieren Sie einen Dokumenten-Workflow mit verschiedenen Zuständen.

```
// State Interface
public interface DocumentState {
    void review(Document document);
    void approve(Document document);
    void reject(Document document);
    void publish(Document document);
}

// Concrete States
public class DraftState implements DocumentState {
    @Override
    public void review(Document document) {
        // Validiere Review-Berechtigung
        validateReviewPermission();

        // Pruefe Dokument-Vollstaendigkeit
        if (!document.isComplete()) {
            throw new IllegalStateException(
                "Document incomplete");
        }

        // Wechsel zu Review-Zustand
        document.setState(new ReviewState());
    }

    @Override
    public void approve(Document document) {
        throw new IllegalStateException(
            "Draft cannot be approved");
    }

    @Override
    public void reject(Document document) {
        throw new IllegalStateException(
            "Draft cannot be rejected");
    }

    @Override
    public void publish(Document document) {
        throw new IllegalStateException(
            "Draft cannot be published");
    }
}

public class ReviewState implements DocumentState {
    @Override
    public void approve(Document document) {
        validateApprovalPermission();
        document.setState(new ApprovedState());
    }

    @Override
    public void reject(Document document) {
        document.addComment(
            "Rejected in Review. Needs revision.");
        document.setState(new DraftState());
    }
}

// Context
public class Document {
    private DocumentState state;
    private String content;
    private List<String> comments;
    private User author;

    public Document() {
        this.state = new DraftState();
```

Pattern Testing

1. Unit Tests

- **Einzelne Komponenten:**
  - Pattern-Struktur testen
  - Verhaltensvalidierung
  - Edge Cases prüfen
- **Mocking:**
  - Abhängigkeiten mocken
  - Interaktionen verifizieren
  - Zustände simulieren

2. Integrationstests

- **Pattern-Kombination:**
  - Zusammenspiel prüfen
  - End-to-End Szenarien
  - Fehlerszenarien
- **Systemverhalten:**
  - Korrekte Integration
  - Performance-Impact
  - Ressourcenverbrauch

Pattern Testing: Document Workflow

Unit Tests für State Pattern:

```java
public class DocumentStateTest {
    private Document document;

    @BeforeEach
    void setUp() {
        document = new Document();
    }

    @Test
    void draftShouldTransitionToReviewWhenComplete() {
        // Given
        document.setContent("Test content");
        document.setAuthor(new User("author"));

        // When
        document.review();

        // Then
        assertInstanceOf(ReviewState.class,
            document.getState());
    }

    @Test
    void draftShouldNotAllowApproval() {
        // Given
        document = new Document(); // In Draft state

        // When/Then
        assertThrows(IllegalStateException.class,
            () -> document.approve());
    }

    @Test
    void reviewStateShouldAllowApprovalOrRejection() {
        // Given
        document.setContent("Test content");
        document.setAuthor(new User("author"));
        document.review(); // Move to Review state

        // When/Then
        assertDoesNotThrow(() -> document.approve());

        // Reset and test rejection
        document.review();
        assertDoesNotThrow(() -> document.reject());
    }
}

// Integration Test
public class DocumentWorkflowTest {
    private DocumentService service;
    private Document document;

    @Test
    void shouldCompleteWorkflow() {
        // Given
        document = createValidDocument();

        // When
        service.processDocument(document);

        // Then
        assertInstanceOf(PublishedState.class,
            document.getState());
        assertTrue(document.isPublished());
        assertFalse(document.getComments().isEmpty());
    }

    @Test
```

Weitere Pattern Beispiele

Observer Pattern implementieren

1. Struktur aufbauen

- **Subject Interface:**
  - Observer registrieren/entfernen
  - Benachrichtigungsmethode
  - Zustandsverwaltung
- **Observer Interface:**
  - Update-Methode
  - Parameterdefinition
  - Fehlerbehandlung

2. Implementation

- **Benachrichtigung:**
  - Thread-Sicherheit
  - Reihenfolge beachten
  - Performance optimieren
- **Fehlerbehandlung:**
  - Observer-Ausfall
  - Inkonsistenzen
  - Zyklische Updates

Observer Pattern: Monitoring System

Aufgabe: Implementieren Sie ein Monitoring-System mit Observer Pattern.

```java
// Subject Interface
public interface MonitoringSubject {
    void addObserver(MonitoringObserver observer);
    void removeObserver(MonitoringObserver observer);
    void notifyObservers(SystemStatus status);
}

// Observer Interface
public interface MonitoringObserver {
    void update(SystemStatus status);
}

// Concrete Subject
public class SystemMonitor implements
     MonitoringSubject {
    private List<MonitoringObserver> observers =
        Collections.synchronizedList(new
             ArrayList<>());
    private SystemStatus currentStatus;

    @Override
    public void addObserver(MonitoringObserver
         observer) {
        observers.add(observer);
        // Send current status to new observer
        if (currentStatus != null) {
            observer.update(currentStatus);
        }
    }

    @Override
    public void notifyObservers(SystemStatus status) {
        this.currentStatus = status;
        observers.forEach(observer -> {
            try {
                observer.update(status);
            } catch (Exception e) {
                handleObserverError(observer, e);
            }
        });
    }

    public void checkSystem() {
        SystemStatus status = calculateSystemStatus();
        if (statusChanged(status)) {
            notifyObservers(status);
        }
    }
}

// Concrete Observers
public class AlertSystem implements MonitoringObserver
     {
    @Override
    public void update(SystemStatus status) {
        if (status.isCritical()) {
            sendAlerts(status);
        }
    }

    private void sendAlerts(SystemStatus status) {
        // Send SMS, Email, etc.
    }
}

public class DashboardUpdater implements
     MonitoringObserver {
    private Dashboard dashboard;
```

Template Method implementieren

1. Basisklasse definieren

- **Template Methode:**
  - Algorithmus-Skelett
  - final deklarieren
  - Schrittabfolge
- **Abstrakte Schritte:**
  - Muss-Implementierungen
  - Parameter/Rückgaben
  - Dokumentation
- **Hook Methoden:**
  - Optionale Schritte
  - Default-Implementation
  - Erweiterungspunkte

Template Method: Report Generation

Aufgabe: Implementieren Sie einen flexiblen Report-Generator mit Template Method.

```java
// Abstract Base Class
public abstract class ReportGenerator {

    // Template Method
    public final Report generateReport(ReportData
        data) {
        validateData(data);
        Report report = new Report();

        try {
            // 1. Header
            report.setHeader(createHeader(data));

            // 2. Content
            List<ReportSection> sections =
                processData(data);
            report.setSections(sections);

            // 3. Optional Customization
            if (shouldCustomize()) {
                customizeReport(report);
            }

            // 4. Footer
            report.setFooter(createFooter(data));

            // 5. Optional Validation
            validateReport(report);

            return report;
        } catch (Exception e) {
            handleError(e);
            throw new ReportGenerationException(e);
        }
    }

    // Abstract Methods (must implement)
    protected abstract ReportHeader createHeader(
        ReportData data);
    protected abstract List<ReportSection> processData(
        ReportData data);
    protected abstract ReportFooter createFooter(
        ReportData data);

    // Hook Methods (optional override)
    protected boolean shouldCustomize() {
        return false;
    }

    protected void customizeReport(Report report) {
        // Default empty implementation
    }

    protected void validateReport(Report report) {
        // Default validation
        if (report.getSections().isEmpty()) {
            throw new EmptyReportException();
        }
    }

    protected void validateData(ReportData data) {
        if (data == null) {
            throw new InvalidDataException("No data");
        }
    }
}

// Concrete Implementation
```

Strategy Pattern implementieren

1. Strategie Interface

- **Methode definieren:**
  - Klare Signatur
  - Parameter/Rückgaben
  - Dokumentation
- **Kontext festlegen:**
  - Strategiewechsel
  - Zustandshaltung
  - Fehlerbehandlung

2. Implementation

- **Strategien:**
  - Konkrete Algorithmen
  - Unabhängige Logik
  - Validierung
- **Konfiguration:**
  - Strategieauswahl
  - Parameter
  - Fallback

Strategy Pattern: Payment Processing

Aufgabe: Implementieren Sie ein flexibles Zahlungssystem mit verschiedenen Strategien.

```
// Strategy Interface
public interface PaymentStrategy {
    PaymentResult process(PaymentRequest request);
}

// Concrete Strategies
public class CreditCardPayment implements
     PaymentStrategy {
    private CreditCardValidator validator;
    private PaymentGateway gateway;

    @Override
    public PaymentResult process(PaymentRequest
         request) {
        // Validate credit card
        CreditCardDetails card =
             request.getCardDetails();
        if (!validator.isValid(card)) {
            return PaymentResult.failed(
                "Invalid card details");
        }

        // Process payment
        try {
            TransactionResult result =
                gateway.processCard(
                    card,
                    request.getAmount()
                );

            return PaymentResult.success(
                result.getTransactionId());
        } catch (GatewayException e) {
            return PaymentResult.failed(
                e.getMessage());
        }
    }
}

public class PayPalPayment implements PaymentStrategy {
    private PayPalClient paypalClient;

    @Override
    public PaymentResult process(PaymentRequest
         request) {
        try {
            String paymentId =
                 paypalClient.createPayment(
                request.getAmount(),
                request.getCurrency());

            PayPalResponse response =
                paypalClient.executePayment(paymentId);

            return PaymentResult.success(
                response.getTransactionId());
        } catch (PayPalException e) {
            return PaymentResult.failed(
                "PayPal error: " + e.getMessage());
        }
    }
}

// Context
public class PaymentProcessor {
    private Map<PaymentType, PaymentStrategy>
         strategies =
        new EnumMap<>(PaymentType.class);
```

Factory und Composite Pattern Beispiele

Factory Pattern implementieren

1. Factory Method

- **Creator definieren:**
  - Abstrakte Factory-Methode
  - Gemeinsame Logik
  - Erweiterungspunkte
- **Produkte:**
  - Gemeinsames Interface
  - Konkrete Implementierungen
  - Produktfamilien

2. Abstract Factory

- **Factory Interface:**
  - Produktfamilien definieren
  - Erstellungsmethoden
  - Abhängigkeiten
- **Implementierung:**
  - Konkrete Factories
  - Produktkombinationen
  - Konfiguration

Factory Pattern: GUI Components

Aufgabe: Implementieren Sie eine Factory für GUI-Komponenten mit verschiedenen Themes.

```java
// Product Interfaces
public interface Button {
    void render();
    void handleClick();
}

public interface TextField {
    void render();
    void handleInput(String text);
}

// Abstract Factory
public interface ComponentFactory {
    Button createButton();
    TextField createTextField();
}

// Concrete Products for Light Theme
public class LightButton implements Button {
    @Override
    public void render() {
        // Render light themed button
    }

    @Override
    public void handleClick() {
        // Handle click with light theme feedback
    }
}

public class LightTextField implements TextField {
    @Override
    public void render() {
        // Render light themed text field
    }

    @Override
    public void handleInput(String text) {
        // Handle input with light theme styling
    }
}

// Concrete Factory for Light Theme
public class LightThemeFactory implements
     ComponentFactory {
    @Override
    public Button createButton() {
        return new LightButton();
    }

    @Override
    public TextField createTextField() {
        return new LightTextField();
    }
}

// Dark Theme Implementation
public class DarkThemeFactory implements
     ComponentFactory {
    @Override
    public Button createButton() {
        return new DarkButton();
    }

    @Override
    public TextField createTextField() {
        return new DarkTextField();
    }
```

Composite Pattern implementieren

1. Komponenten-Hierarchie

- **Component Interface:**
  - Gemeinsame Operationen
  - Kind-Management
  - Traversierung
- **Leaf Klassen:**
  - Atomare Operationen
  - Keine Kinder
  - Spezifisches Verhalten
- **Composite Klassen:**
  - Kinderverwaltung
  - Operation-Delegation
  - Aggregation

2. Implementierungsaspekte

- **Kinderverwaltung:**
  - Hinzufügen/Entfernen
  - Validierung
  - Traversierung
- **Operation-Ausführung:**
  - Delegation an Kinder
  - Ergebnisaggregation
  - Fehlerbehandlung

Composite Pattern: File System

Aufgabe: Implementieren Sie eine Verzeichnisstruktur mit dem Composite Pattern.

```java
// Component Interface
public interface FileSystemItem {
    String getName();
    long getSize();
    void accept(FileSystemVisitor visitor);
}

// Leaf
public class File implements FileSystemItem {
    private String name;
    private long size;

    @Override
    public String getName() {
        return name;
    }

    @Override
    public long getSize() {
        return size;
    }

    @Override
    public void accept(FileSystemVisitor visitor) {
        visitor.visitFile(this);
    }
}

// Composite
public class Directory implements FileSystemItem {
    private String name;
    private List<FileSystemItem> children =
        new ArrayList<>();

    @Override
    public String getName() {
        return name;
    }

    @Override
    public long getSize() {
        return children.stream()
            .mapToLong(FileSystemItem::getSize)
            .sum();
    }

    public void addItem(FileSystemItem item) {
        children.add(item);
    }

    public void removeItem(FileSystemItem item) {
        children.remove(item);
    }

    public List<FileSystemItem> getChildren() {
        return Collections.unmodifiableList(children);
    }

    @Override
    public void accept(FileSystemVisitor visitor) {
        visitor.visitDirectory(this);
        children.forEach(child ->
            child.accept(visitor));
    }
}

// Visitor Interface for Operations
public interface FileSystemVisitor {
```

Adapter und Bridge Pattern Beispiele

Adapter Pattern implementieren

1. Analyse

- **Inkompatible Schnittstellen:**
  - Ziel-Interface identifizieren
  - Adaptee-Interface analysieren
  - Unterschiede dokumentieren
- **Adapter-Typ wählen:**
  - Klassen-Adapter (Vererbung)
  - Objekt-Adapter (Komposition)
  - Two-Way Adapter

2. Implementation

- **Methodenabbildung:**
  - Parameter-Konvertierung
  - Rückgabewert-Anpassung
  - Fehlerbehandlung
- **Zusätzliche Features:**
  - Caching
  - Logging
  - Performance-Optimierung

Adapter Pattern

Szenario: Altbestand an Drittanbieter-Bibliothek integrieren

```
// Bestehende Schnittstelle
interface ModernPrinter {
    void printDocument(String content);
}

// Alte Drittanbieter-Klasse
class LegacyPrinter {
    public void print(String[] pages) {
        for(String page : pages) {
            System.out.println(page);
        }
    }
}

// Adapter
class PrinterAdapter implements ModernPrinter {
    private LegacyPrinter legacyPrinter;

    public PrinterAdapter(LegacyPrinter
        printer) {
        this.legacyPrinter = printer;
    }

    public void printDocument(String content) {
        String[] pages = content.split("\n");
        legacyPrinter.print(pages);
    }
}
```

Simple Factory

Szenario: Erzeugung von verschiedenen Datenbankverbindungen

```
class DatabaseFactory {
    public static Database
        createDatabase(String type) {
        switch(type) {
            case "MySQL":
                return new MySQLDatabase();
            case "PostgreSQL":
                return new PostgreSQLDatabase();
            default:
                throw new
                    IllegalArgumentException("Unknow
                    DB type");
        }
    }
}

// Verwendung
Database db =
    DatabaseFactory.createDatabase("MySQL");
```

Singleton

Szenario: Globale Konfigurationsverwaltung

```
public class Configuration {
    private static Configuration instance;
    private Map<String, String> config;

    private Configuration() {
        config = new HashMap<>();
    }

    public static Configuration getInstance() {
        if(instance == null) {
            instance = new Configuration();
        }
        return instance;
    }
}
```

Dependency Injection

Szenario: Flexible Logger-Implementation

```
interface Logger {
    void log(String message);
}

class FileLogger implements Logger {
    public void log(String message) {
        // Log to file
    }
}

class UserService {
    private final Logger logger;

    public UserService(Logger logger) { //
        Dependency Injection
        this.logger = logger;
    }
}
```

Proxy

Szenario: Verzögertes Laden eines großen Bildes

```
interface Image {
    void display();
}

class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading " +
            filename);
    }

    public void display() {
        System.out.println("Displaying " +
            filename);
    }
}

class ImageProxy implements Image {
    private RealImage realImage;
    private String filename;

    public ImageProxy(String filename) {
        this.filename = filename;
    }

    public void display() {
        if(realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

Chain of Responsibility

Szenario: Authentifizierungskette

```
abstract class AuthHandler {
    protected AuthHandler next;

    public void setNext(AuthHandler next) {
        this.next = next;
    }

    public abstract boolean handle(String
        username, String password);
}

class LocalAuthHandler extends AuthHandler {
    public boolean handle(String username,
        String password) {
        if(checkLocalDB(username, password)) {
            return true;
        }
        return next != null ?
            next.handle(username, password) :
            false;
    }
}

class LDAPAuthHandler extends AuthHandler {
    public boolean handle(String username,
        String password) {
        if(checkLDAP(username, password)) {
            return true;
        }
        return next != null ?
            next.handle(username, password) :
            false;
    }
}
```

Decorator

Szenario: Dynamische Erweiterung eines Text-Editors

```
interface TextComponent {
    String render();
}

class SimpleText implements TextComponent {
    private String text;

    public SimpleText(String text) {
        this.text = text;
    }

    public String render() {
        return text;
    }
}

class BoldDecorator implements TextComponent {
    private TextComponent component;

    public BoldDecorator(TextComponent
        component) {
        this.component = component;
    }

    public String render() {
        return "<b>" + component.render() +
            "</b>";
    }
}
```

Observer

Szenario: News-Benachrichtigungssystem

```java
interface NewsObserver {
    void update(String news);
}

class NewsAgency {
    private List<NewsObserver> observers = new
        ArrayList<>();

    public void addObserver(NewsObserver
        observer) {
        observers.add(observer);
    }

    public void notifyObservers(String news) {
        for(NewsObserver observer : observers) {
            observer.update(news);
        }
    }
}

class NewsChannel implements NewsObserver {
    private String name;

    public NewsChannel(String name) {
        this.name = name;
    }

    public void update(String news) {
        System.out.println(name + " received: "
            + news);
    }
}
```

Strategy

Szenario: Verschiedene Zahlungsmethoden

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

class CreditCardPayment implements
    PaymentStrategy {
    private String cardNumber;

    public void pay(int amount) {
        System.out.println("Paid " + amount + "
            using Credit Card");
    }
}

class PayPalPayment implements PaymentStrategy {
    private String email;

    public void pay(int amount) {
        System.out.println("Paid " + amount + "
            using PayPal");
    }
}
```

Strategy Pattern Implementation

```java
public interface SortStrategy {
    void sort(List<String> data);
}

public class QuickSort implements SortStrategy {
    public void sort(List<String> data) {
        // Implementierung
    }
}

public class Context {
    private SortStrategy strategy;

    public void setStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void executeStrategy(List<String> data) {
        strategy.sort(data);
    }
}
```

Composite

Szenario: Dateisystem-Struktur

```java
interface FileSystemComponent {
    void list(String prefix);
}

class File implements FileSystemComponent {
    private String name;

    public void list(String prefix) {
        System.out.println(prefix + name);
    }
}

class Directory implements FileSystemComponent {
    private String name;
    private List<FileSystemComponent> children
        = new ArrayList<>();

    public void add(FileSystemComponent
        component) {
        children.add(component);
    }

    public void list(String prefix) {
        System.out.println(prefix + name);
        for(FileSystemComponent child :
            children) {
            child.list(prefix + "  ");
        }
    }
}
```

State

Szenario: Verkaufsautomat

```
interface VendingMachineState {
    void insertCoin();
    void ejectCoin();
    void selectProduct();
    void dispense();
}

class HasCoinState implements
    VendingMachineState {
    private VendingMachine machine;

    public void selectProduct() {
        System.out.println("Product selected");
        machine.setState(machine.getSoldState());
    }

    public void insertCoin() {
        System.out.println("Already have coin");
    }
}

class VendingMachine {
    private VendingMachineState currentState;

    public void setState(VendingMachineState
        state) {
        this.currentState = state;
    }

    public void insertCoin() {
        currentState.insertCoin();
    }
}
```

Visitor

Szenario: Dokumentstruktur mit verschiedenen Operationen

```
interface DocumentElement {
    void accept(Visitor visitor);
}

interface Visitor {
    void visit(Paragraph paragraph);
    void visit(Heading heading);
}

class HTMLVisitor implements Visitor {
    public void visit(Paragraph p) {
        System.out.println("<p>" + p.getText()
            + "</p>");
    }

    public void visit(Heading h) {
        System.out.println("<h1>" + h.getText()
            + "</h1>");
    }
}
```

Facade

Szenario: Vereinfachte Multimedia-Bibliothek

```
class MultimediaFacade {
    private AudioSystem audio;
    private VideoSystem video;
    private SubtitleSystem subtitles;

    public void playMovie(String movie) {
        audio.initialize();
        video.initialize();
        subtitles.load(movie);
        video.play(movie);
        audio.play();
    }
}
```

Abstract Factory

Szenario: GUI-Elemente für verschiedene Betriebssysteme

```
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

class WindowsFactory implements GUIFactory {
    public Button createButton() {
        return new WindowsButton();
    }

    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}

class MacFactory implements GUIFactory {
    public Button createButton() {
        return new MacButton();
    }

    public Checkbox createCheckbox() {
        return new MacCheckbox();
    }
}
```

Factory Method Implementation

Aufgabe: Implementieren Sie eine Factory für verschiedene Dokumenttypen (PDF, Word, Text)

Lösung:

```
// Interface fuer Produkte
interface Document {
    void open();
    void save();
}

// Konkrete Produkte
class PdfDocument implements Document {
    public void open() { /* ... */ }
    public void save() { /* ... */ }
}

// Factory Method Pattern
abstract class DocumentCreator {
    abstract Document createDocument();

    // Template Method
    final void processDocument() {
        Document doc = createDocument();
        doc.open();
        doc.save();
    }
}

// Konkrete Factory
class PdfDocumentCreator extends
    DocumentCreator {
    Document createDocument() {
        return new PdfDocument();
    }
}
```

Observer Pattern Implementation

Aufgabe: Implementieren Sie ein Benachrichtigungssystem für Aktienkurse

Lösung:

```
interface StockObserver {
    void update(String stock, double price);
}

class StockMarket {
    private List<StockObserver> observers = new
        ArrayList<>();

    public void attach(StockObserver observer) {
        observers.add(observer);
    }

    public void notifyObservers(String stock,
        double price) {
        for(StockObserver observer : observers)
            {
            observer.update(stock, price);
        }
    }
}

class StockDisplay implements StockObserver {
    public void update(String stock, double
        price) {
        System.out.println("Stock: " + stock +
                           " updated to " +
                               price);
    }
}
```

Kochrezepte und Übungen

Implementation, Refactoring and Testing

Test-Driven Development (TDD)

Schritte:

1. **Red:** Test schreiben der fehlschlägt
   - Testfall definieren
   - Erwartetes Verhalten spezifizieren
   - Test implementieren
2. **Green:** Minimale Implementation
   - Nur das Nötigste implementieren
   - Test soll grün werden
   - Keine Optimierungen
3. **Refactor:** Code verbessern
   - Code aufräumen
   - Duplizierung entfernen
   - Tests müssen grün bleiben

Typische Refactoring Patterns

Methoden:

- **Extract Method:**
  - Code in neue Methode auslagern
  - Gemeinsame Funktionalität zusammenfassen
  - Lesbarkeit verbessern
- **Move Method:**
  - Methode in andere Klasse verschieben
  - Näher an verwendeten Daten
  - Kohäsion verbessern
- **Replace Conditional with Polymorphism:**
  - Switch/if durch Vererbung ersetzen
  - Flexibilität erhöhen
  - Wartbarkeit verbessern

Typische Prüfungsaufgabe: Code Refactoring

Aufgabe: Refactoren Sie den folgenden Code unter Anwendung geeigneter Patterns

Vorher:

```
public class Order {
    private List<OrderItem> items;
    private double totalAmount;
    private String status;

    public void calculateTotal() {
        totalAmount = 0;
        for(OrderItem item : items) {
            totalAmount += item.getPrice() *
                item.getQuantity();
            // Komplexe Rabattberechnung
            if(item.getQuantity() > 10) {
                totalAmount *= 0.9;  // 10%
                    Rabatt
            }
            if(totalAmount > 1000) {
                totalAmount *= 0.95; // 5%
                    Rabatt
            }
        }
    }

    public void processOrder() {
        calculateTotal();
        // Komplexe Statusberechnung
        if(totalAmount < 100) {
            status = "SMALL_ORDER";
        } else if(totalAmount < 1000) {
            status = "MEDIUM_ORDER";
        } else {
            status = "LARGE_ORDER";
        }
        // Weitere 20 Zeilen Status-Logik...
    }
}
```

Nachher:

```
public class Order {
    private List<OrderItem> items;
    private OrderStatus status;
    private DiscountStrategy discountStrategy;

    public Money calculateTotal() {
        Money total = items.stream()
            .map(OrderItem::getSubtotal)
            .reduce(Money.ZERO, Money::add);

        return
            discountStrategy.applyDiscount(total);
    }

    public void processOrder() {
        Money total = calculateTotal();
        status = OrderStatus.forAmount(total);
        status.process(this);
    }
```

Unit Testing Best Practices

1. **Test-Struktur (AAA):**
   - **Arrange:** Testdaten vorbereiten
   - **Act:** Testobjekt ausführen
   - **Assert:** Ergebnis prüfen
2. **Namenskonventionen:**
   - methodName_testCase_expectedResult
   - should_doSomething_when_condition
   - given_when_then Format
3. **Coverage:**
   - Happy Path testen
   - Edge Cases abdecken
   - Fehlerfälle prüfen

Typische Prüfungsaufgabe: Unit Tests

Aufgabe: Schreiben Sie Unit Tests für eine Warenkorb-Komponente

```java
public class ShoppingCartTest {
    private ShoppingCart cart;
    private Product testProduct;

    @BeforeEach
    void setUp() {
        cart = new ShoppingCart();
        testProduct = new Product("Test",
            Money.of(10));
    }

    @Test
    void shouldCalculateTotal_whenEmpty() {
        assertEquals(Money.ZERO,
            cart.getTotal());
    }

    @Test
    void shouldCalculateTotal_withOneItem() {
        cart.addItem(testProduct, 1);
        assertEquals(Money.of(10),
            cart.getTotal());
    }

    @Test
    void
        shouldApplyQuantityDiscount_whenOver10Items()
        {
        cart.addItem(testProduct, 11);
        Money expected = Money.of(10 * 11 *
            0.9);
        assertEquals(expected, cart.getTotal());
    }

    @Test
    void
        shouldThrowException_whenNegativeQuantity()
        {
        assertThrows(IllegalArgumentException.class,
            () -> cart.addItem(testProduct,
                -1));
    }
}
```

Code Review

Review Checklist:

1. **Funktionalität:**
   - Anforderungen erfüllt?
   - Edge Cases behandelt?
   - Fehlerbehandlung korrekt?
2. **Code Qualität:**
   - Clean Code Prinzipien
   - SOLID Prinzipien
   - Naming Conventions
3. **Tests:**
   - Testabdeckung ausreichend?
   - Tests aussagekräftig?
   - Testfälle vollständig?
4. **Best Practices:**
   - Design Patterns korrekt?
   - Logging vorhanden?
   - Dokumentation aktuell?

Code Review Szenario

Aufgabe: Führen Sie ein Code Review für folgende Implementierung durch:

```java
// Original Code
public class DataManager {
    private static DataManager instance;
    private Connection conn;

    private DataManager() {
        try {
            conn =
                DriverManager.getConnection("db_url")
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    public static DataManager getInstance() {
        if(instance == null) {
            instance = new DataManager();
        }
        return instance;
    }

    public void saveData(String data) {
        try {
            Statement stmt =
                conn.createStatement();
            stmt.execute("INSERT INTO data
                VALUES('" + data + "')");
        } catch(Exception e) {
            System.out.println("Error: " +
                e.getMessage());
        }
    }
}
```

Review Feedback:

- **Probleme:**
  - Singleton nicht thread-safe
  - SQL Injection Gefahr
  - Schlechte Exception-Behandlung
  - Keine Ressourcen-Freigabe
  - Keine Konfigurierbarkeit
- **Verbesserungsvorschläge:**
  - Dependency Injection statt Singleton
  - Prepared Statements verwenden
  - Proper Exception Handling
  - Try-with-resources für Statements
  - Konfiguration externalisieren

Verbesserte Version:

```java
@Component
public class DataManager implements
    AutoCloseable {
    private final DataSource dataSource;
    private final Logger logger =
        LoggerFactory.getLogger(DataManager.class);

    public DataManager(DataSource dataSource) {
```

Kochrezepte und Übungen

Verteilte Systeme

Verteiltes System entwerfen

1. Architekturanalyse

- Anforderungen analysieren
  - Skalierbarkeit
  - Verfügbarkeit
  - Konsistenz
- Kommunikationsmuster bestimmen
  - Synchron vs. Asynchron
  - Request-Response vs. Messaging
  - Push vs. Pull
- Fehlerszenarien identifizieren
  - Netzwerkfehler
  - Server-Ausfälle
  - Datenverlust

2. Design

- Architekturmuster wählen
  - Client-Server
  - Peer-to-Peer
  - Event-basiert
- Verteilungsstruktur festlegen
  - Service-Grenzen
  - Datenverteilung
  - Load Balancing
- Konsistenzmodell definieren
  - Strong vs. Eventual Consistency
  - ACID vs. BASE
  - CAP Trade-offs

3. Implementation

- Kommunikation implementieren
  - Protokolle definieren
  - Serialisierung festlegen
  - Error Handling
- Monitoring einrichten
  - Logging
  - Metriken
  - Alarme
- Testing
  - Unit Tests
  - Integration Tests
  - Chaos Testing

Remote Procedure Call (RPC) implementieren

1. Interface definieren

- Services spezifizieren
- Parameter definieren
- Rückgabewerte festlegen
- Fehlerbehandlung planen

2. Stub/Skeleton generieren

- IDL verwenden
- Marshalling implementieren
- Proxies erstellen

3. Netzwerkkommunikation

- Protokoll wählen
- Serialisierung implementieren
- Timeouts einbauen
- Retry-Logik

4. Fehlerbehandlung

- Netzwerkfehler behandeln
- Timeout-Handling
- Circuit Breaker einbauen
- Fallback-Strategien

Typische Prüfungsaufgabe: Verteiltes System

Aufgabe: Entwerfen Sie ein verteiltes Chat-System

Anforderungen:

- Unterstützung für 100.000 gleichzeitige Nutzer
- Nachrichtenhistorie speichern
- Offline-Nachrichten möglich
- Maximale Latenz 500ms

Lösung:

```
// Message Broker Interface
public interface MessageBroker {
    void publish(String topic, Message message);
    void subscribe(String topic, MessageHandler
        handler);
}

// Chat Service
@Service
public class ChatService {
    private final MessageBroker broker;
    private final MessageRepository repository;

    public void sendMessage(ChatMessage
        message) {
        // Persist message
        repository.save(message);

        // Publish to online users
        broker.publish(
            "chat." + message.getRoomId(),
            message
        );

        // Handle offline users
        message.getRecipients().stream()
            .filter(user -> !userIsOnline(user))
            .forEach(user ->
                queueOfflineMessage(user,
                    message));
    }

    public void joinRoom(String userId, String
        roomId) {
        broker.subscribe("chat." + roomId,
            message -> deliverToUser(userId,
                message));
    }

    private void queueOfflineMessage(
            String userId, ChatMessage message)
                {
        broker.publish(
            "offline." + userId,
            message
        );
    }
}

// Load Balancer Configuration
@Configuration
public class LoadBalancerConfig {
```

Message Oriented Middleware

Aufgabe: Implementieren Sie ein Nachrichtensystem mit JMS

```java
// Message Producer
@Component
public class OrderProducer {
    @Autowired
    private JmsTemplate jmsTemplate;

    public void sendOrder(Order order) {
        try {
            jmsTemplate.convertAndSend("orders",
                order, message -> {
                message.setStringProperty(
                    "orderType",
                    order.getType().toString()
                );
                return message;
            });
        } catch (JmsException e) {
            handleMessageError(order, e);
        }
    }
}

// Message Consumer
@Component
public class OrderConsumer {
    @JmsListener(
        destination = "orders",
        selector = "orderType = 'PREMIUM'"
    )
    public void processPremiumOrder(Order
        order) {
        try {
            // Process order with high priority
            processOrderWithPriority(order);
        } catch (Exception e) {
            // Dead Letter Queue
            handleFailedOrder(order, e);
        }
    }

    @JmsListener(
        destination = "orders",
        selector = "orderType = 'STANDARD'"
    )
    public void processStandardOrder(Order
        order) {
        // Process normal order
    }
}

// Error Handling
@Component
public class DeadLetterQueueHandler {
    @JmsListener(destination = "DLQ")
    public void handleFailedMessages(Message
        failedMessage) {
        // Analyze failure
        // Retry with backoff
```

Fehlerbehandlung in verteilten Systemen

1. **Netzwerkfehler**
   - Timeouts implementieren
   - Retry-Strategien definieren
   - Circuit Breaker einsetzen
2. **Dateninkonsistenzen**
   - Eventual Consistency
   - Konfliktauflösung
   - Versioning
3. **Ausfallsicherheit**
   - Redundanz einbauen
   - Fallback-Strategien
   - Graceful Degradation
4. **Monitoring**
   - Logging
   - Metriken sammeln
   - Alerting einrichten

Circuit Breaker Implementation

Aufgabe: Implementieren Sie einen Circuit Breaker für einen Microservice

```java
public class CircuitBreaker {
    private final long timeout;
    private final int failureThreshold;
    private final long resetTimeout;

    private AtomicInteger failures = new
        AtomicInteger();
    private AtomicReference<State> state =
        new AtomicReference<>(State.CLOSED);
    private AtomicLong lastFailureTime = new
        AtomicLong();

    public enum State {
        CLOSED, OPEN, HALF_OPEN
    }

    public <T> T execute(Supplier<T> action)
        throws Exception {
        if (shouldExecute()) {
            try {
                T result = action.get();
                reset();
                return result;
            } catch (Exception e) {
                handleFailure();
                throw e;
            }
        }
        throw new
            CircuitBreakerException("Circuit
            open");
    }

    private boolean shouldExecute() {
        State currentState = state.get();
        if (currentState == State.CLOSED) {
            return true;
        }
        if (currentState == State.OPEN) {
            if (hasResetTimeoutExpired()) {
                state.compareAndSet(State.OPEN,
                    State.HALF_OPEN);
                return true;
            }
            return false;
        }
        return true; // HALF_OPEN
    }

    private void handleFailure() {
        lastFailureTime.set(System.currentTimeMillis
        if (failures.incrementAndGet() >=
            failureThreshold) {
            state.set(State.OPEN);
        }
    }
```

Service Discovery implementieren

1. **Registry Service**
   - Service-Registrierung
   - Health Checking
   - Load Balancing
2. **Service Registration**
   - Startup Registration
   - Heartbeat Mechanism
   - Graceful Shutdown
3. **Service Discovery**
   - Cache Management
   - Failure Detection
   - Load Balancing

Service Discovery mit Spring Cloud

Aufgabe: Implementieren Sie Service Discovery für Microservices

```java
// Eureka Server
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {
    public static void main(String[] args) {
        SpringApplication.run(
            ServiceRegistryApplication.class,
            args
        );
    }
}

// Service Registration
@SpringBootApplication
@EnableDiscoveryClient
public class UserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(
            UserServiceApplication.class,
            args
        );
    }
}

// Service Discovery
@Service
public class UserServiceClient {
    @Autowired
    private DiscoveryClient discoveryClient;

    @Autowired
    private RestTemplate restTemplate;

    public UserDTO getUser(String id) {
        // Get service instance
        List<ServiceInstance> instances =
            discoveryClient.getInstances("user-service");

        if (instances.isEmpty()) {
            throw new ServiceNotFoundException(
                "user-service not available");
        }

        // Load balance
        ServiceInstance instance =
            loadBalance(instances);

        // Make request
        return restTemplate.getForObject(
            instance.getUri() + "/api/users/" +
                id,
            UserDTO.class
        );
    }

    private ServiceInstance loadBalance(
            List<ServiceInstance> instances) {
        // Simple round-robin
```