

## 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

### 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

### Fully-dressed Use Case

**Aufgabe:** Schreiben Sie einen vollständigen Use Case für "Ticket buchen in 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:

```
1 // Verfügbbarkeit prüfen
2 checkAvailability(dates: DateRange): List<Room>
3
4 // Zimmer reservieren
5 bookRoom(roomId: RoomId,
6           dates: DateRange): Reservation
7
8 // Zahlung durchführen
9 processPayment(reservationId: ReservationId,
10               paymentInfo: PaymentDetails):
11               boolean
12
13 // Buchung bestaetigen
14 confirmBooking(reservationId: ReservationId):
15               BookingConfirmation
```

#### Alternative Pfade:

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

### Contracts für Systemoperationen

#### Contract-Elemente:

1. **Operation:**
  - Name und Parameter
  - Rückgabebetyp
  - 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

### 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:**
  - Menütem 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 Menütem (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

## 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 Zimmerverschlussystems (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:**

```
1 // Presentation Layer
2 public class SecurityController {
3     private SecurityService securityService;
4
5     public void arm(String code) {
6         securityService.armSystem(code);
7     }
8 }
9
10 // Application Layer
11 public class SecurityService {
12     private AlarmSystem alarmSystem;
13     private CodeValidator validator;
14
15     public void armSystem(String code) {
16         if (validator.isValid(code)) {
17             alarmSystem.arm();
18         }
19     }
20 }
21
22 // Domain Layer
23 public class AlarmSystem {
24     private List<Sensor> sensors;
25     private AlarmState state;
26
27     public void arm() {
28         validateSystemState();
29         state = state.arm();
30         activateSensors();
31     }
32 }
33
34 // Infrastructure Layer
35 public class SensorRepository {
36     public List<Sensor> getActiveSensors() {
37         // DB access
38     }
39 }
```

## 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:

```
1 // State Pattern Implementation
2 public interface SystemState {
3     void arm();
4     void disarm(String code);
5     void handleSensorTriggered(Sensor sensor);
6 }
7
8 public class ArmedState implements SystemState {
9     private AlarmSystem system;
10
11     @Override
12     public void handleSensorTriggered(Sensor sensor) {
13         if (sensor.requiresImmediateResponse()) {
14             system.triggerAlarm();
15         } else {
16             system.startEntryTimer();
17         }
18     }
19 }
20
21 // Observer Pattern Implementation
22 public interface SensorListener {
23     void onSensorTriggered(SensorEvent event);
24 }
25
26 public class AlarmSystem implements SensorListener {
27     @Override
28     public void onSensorTriggered(SensorEvent event) {
29         currentState.handleSensorTriggered(
30             event.getSensor());
31     }
32 }
```

## 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:

```
1 // Performance Optimization
2 @Component
3 public class SensorEventProcessor {
4     private BlockingQueue<SensorEvent> eventQueue;
5     private ThreadPoolExecutor executor;
6
7     @Async
8     public void processSensorEvent(SensorEvent event) {
9         eventQueue.offer(event);
10        executor.execute(() -> processEvent(event));
11    }
12 }
13
14 // High Availability
15 public class RedundantAlarmSystem {
16     private List<AlarmDevice> alarmDevices;
17
18     public void triggerAlarm() {
19         for (AlarmDevice device : alarmDevices) {
20             try {
21                 device.activate();
22                 return; // Success
23             } catch (DeviceFailureException e) {
24                 // Try next device
25                 continue;
26             }
27         }
28     }
29 }
```

## 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.

```

1 // Event Klasse
2 public class SensorEvent {
3     private String sensorId;
4     private SensorType type;
5     private LocalDateTime timestamp;
6     private Map<String, Object> data;
7
8     // Constructor und Getter
9 }
10
11 // Event Bus
12 public class SecurityEventBus {
13     private Map<Class<?>, List<EventHandler>>> handlers
14         =
15         new HashMap<>();
16
17     public void publish(SensorEvent event) {
18         List<EventHandler> eventHandlers =
19             handlers.getOrDefault(
20                 event.getClass(),
21                 Collections.emptyList()
22             );
23
24         for (EventHandler handler : eventHandlers){
25             try {
26                 handler.handle(event);
27             } catch (Exception e) {
28                 handleError(handler, event, e);
29             }
30         }
31
32         public void subscribe(Class<?> eventType,
33                             EventHandler handler) {
34             handlers.computeIfAbsent(
35                 eventType,
36                 k -> new ArrayList<>()
37             ).add(handler);
38         }
39     }
40
41 // Event Handler Implementation
42 @Component
43 public class MotionSensorHandler
44     implements EventHandler<MotionEvent> {
45
46     private AlarmSystem alarmSystem;
47
48     @Override
49     public void handle(MotionEvent event) {
50         if (alarmSystem.isArmed()) {
51             evaluateMotion(event);
52         }
53     }
54
55     private void evaluateMotion(MotionEvent event) {
56         if (event.getIntensity() > THRESHOLD) {
57             alarmSystem.startEntryTimer();
58         }
59     }
60 }

```

## 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.

```

1 // Server-Side Controller
2 @RestController
3 @RequestMapping("/api/v1/security")
4 public class SecurityController {
5     private SecurityService securityService;
6 }

```



```

7  @PostMapping("/arm")
8  public ResponseEntity<SystemStatus> armSystem(
9      @RequestBody ArmRequest request) {
10      try {
11          SystemStatus status =
12              securityService.armSystem(
13                  request.getCode());
14          return ResponseEntity.ok(status);
15      } catch (InvalidCodeException e) {
16          return ResponseEntity
17              .status(HttpStatus.UNAUTHORIZED)
18              .build();
19      }
20  }
21
22  @GetMapping("/status")
23  public ResponseEntity<SystemStatus> getStatus() {
24      return ResponseEntity.ok(
25          securityService.getCurrentStatus());
26  }
27  }
28
29  // Client-Side Service
30  public class SecurityClient {
31      private final WebClient webClient;
32      private final String baseUrl;
33
34      public Mono<SystemStatus> armSystem(String code) {
35          return webClient.post()
36              .uri(baseUrl + "/arm")
37              .body(new ArmRequest(code))
38              .retrieve()
39              .bodyToMono(SystemStatus.class)
40              .timeout(Duration.ofSeconds(5))
41              .retry(3)
42              .onErrorResume(this::handleError);
43      }
44
45      private Mono<SystemStatus> handleError(Throwable
46          error) {
47          if (error instanceof TimeoutException) {
48              return Mono.error(
49                  new ConnectionException("Timeout"));
50          }
51          return Mono.error(error);
52      }
53  }
54
55  // Mobile App Component
56  public class SecurityViewModel {
57      private SecurityClient securityClient;
58      private MutableLiveData<SystemStatus> status =
59          new MutableLiveData<>();
60
61      public void armSystem(String code) {
62          securityClient.armSystem(code)
63              .subscribe(
64                  status::setValue,
65                  this::handleError
66              );
67      }
68  }

```

## 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.

```

1  // Master Component
2  public class SensorMaster {
3      private List<SensorSlave> slaves;
4      private Map<String, SensorData> results =
5          new ConcurrentHashMap<>();
6
7      public void processSensorData(List<SensorInput>
8          inputs) {
9          // Distribute work
10         Map<SensorSlave, List<SensorInput>>
11             distribution =
12                 distributeWork(inputs);
13
14         // Start processing
15         List<Future<SensorData>> futures =
16             submitTasks(distribution);
17
18         // Collect results
19         collectResults(futures);
20     }
21
22     private Map<SensorSlave, List<SensorInput>>
23         distributeWork(List<SensorInput> inputs) {
24         Map<SensorSlave, List<SensorInput>>
25             distribution =
26                 new HashMap<>();
27
28         int slaveIndex = 0;
29         for (SensorInput input : inputs) {
30             SensorSlave slave = slaves.get(
31                 slaveIndex % slaves.size());
32             distribution.computeIfAbsent(
33                 slave,
34                 k -> new ArrayList<>()
35             ).add(input);
36             slaveIndex++;
37         }
38         return distribution;
39     }
40
41     // Slave Component
42     public class SensorSlave {
43         private String slaveId;
44         private SensorProcessor processor;
45
46         public SensorData process(SensorInput input) {

```

```

46     try {
47         return processor.process(input);
48     } catch (Exception e) {
49         handleError(e);
50         return SensorData.error(input, e);
51     }
52 }
53
54 public HealthStatus checkHealth() {
55     return new HealthStatus(
56         slaveId,
57         processor.getStatus(),
58         getResourceMetrics()
59     );
60 }
61 }
62
63 // Coordinator
64 public class SensorCoordinator {
65     private SensorMaster master;
66     private HealthMonitor monitor;
67
68     @Scheduled(fixedRate = 5000)
69     public void checkSlaveHealth() {
70         List<HealthStatus> statuses =
71             master.getSlaveStatuses();
72         monitor.updateHealth(statuses);
73
74         // Handle unhealthy slaves
75         List<String> unhealthySlaves =
76             findUnhealthySlaves(statuses);
77         if (!unhealthySlaves.isEmpty()) {
78             reconfigureMaster(unhealthySlaves);
79         }
80     }
81 }

```

## 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

```

1 // Reliability Pattern Implementation
2 public class SensorManager {
3     private Map<String, Sensor> sensors;
4     private CircuitBreaker circuitBreaker;
5
6     @Retry(maxAttempts = 3)
7     public SensorStatus checkSensor(String sensorId) {
8         return circuitBreaker.execute(() ->
9             sensors.get(sensorId)
10                .getStatus());
11     }
12
13     public void handleSensorFailure(String sensorId) {
14         // Deactivate failed sensor
15         sensors.get(sensorId).deactivate();
16
17         // Notify system
18         notifySystemStatus(
19             String.format(
20                 "Sensor %s failed, system operating in
21                 degraded mode",
22                 sensorId

```



```

22     )
23   );
24
25   // Adjust monitoring
26   updateMonitoringStrategy();
27 }
28 }
29
30 // Scalability Pattern Implementation
31 public class SensorEventProcessor {
32     private Queue<SensorEvent> eventQueue;
33     private ThreadPoolExecutor executor;
34
35     public SensorEventProcessor(int maxThreads) {
36         this.eventQueue = new LinkedBlockingQueue<>();
37         this.executor = new ThreadPoolExecutor(
38             2, maxThreads,
39             60L, TimeUnit.SECONDS,
40             new LinkedBlockingQueue<>()
41         );
42     }
43
44     public void processEvent(SensorEvent event) {
45         executor.submit(() -> {
46             try {
47                 handleEvent(event);
48             } catch (Exception e) {
49                 handleProcessingError(event, e);
50             }
51         });
52     }
53
54     private void handleEvent(SensorEvent event) {
55         // Process event based on type
56         switch (event.getType()) {
57             case MOTION:
58                 processMotionEvent((MotionEvent)
59                     event);
60                 break;
61             case FIRE:
62                 processFireEvent((FireEvent) event);
63                 break;
64             // ... other event types
65         }
66     }

```

### 3. Evaluation Ergebnisse

#### • Zuverlässigkeit:

- + Circuit Breaker verhindert Kaskadeneffekte
- + Retry-Mechanismus für Stabilität
- + Degraded Mode für Teilausfälle
- - Komplexe Fehlerbehandlung nötig

#### • Skalierbarkeit:

- + Asynchrone Event-Verarbeitung
- + Dynamischer Thread-Pool
- + Event-Queue als Buffer
- - Monitoring Overhead bei vielen Sensoren

### 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

[ breakable]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

```

1 // Sensor Protocol Variation
2 public interface SensorProtocol {
3     void initialize();
4     SensorData readData();
5     void sendCommand(Command cmd);
6 }
7
8 public class ZigBeeSensor implements SensorProtocol {
9     @Override
10    public void initialize() {
11        // ZigBee specific initialization
12    }
13
14    @Override
15    public SensorData readData() {
16        // Read from ZigBee sensor
17        return new SensorData(/* ... */);
18    }
19 }
20
21 // Notification Variation
22 public interface NotificationChannel {
23     void sendAlert(AlertLevel level, String message);
24     boolean isAvailable();
25     Priority getPriority();
26 }
27
28 public class NotificationService {
29     private List<NotificationChannel> channels;
30     private NotificationConfig config;
31
32     public void notify(Alert alert) {
33        // Sort channels by priority
34        List<NotificationChannel> availableChannels =
35            channels.stream()
36                .filter(NotificationChannel::isAvailable)
37                .sorted(comparing(
38                    NotificationChannel::getPriority))
39                .collect(toList());

```

```
40
41 // Try channels until successful
42 for (NotificationChannel channel :
43     availableChannels) {
44     try {
45         channel.sendAlert(
46             alert.getLevel(),
47             alert.getMessage()
48         );
49         return;
50     } catch (NotificationException e) {
51         // Try next channel
52         continue;
53     }
54
55 // No channel available
56 handleNotificationFailure(alert);
57 }
58 }
```

### 3. Bewertung

- **Vorteile:**
  - Neue Sensor-Protokolle einfach integrierbar
  - Flexible Alarmierungskonfiguration
  - Ausfallsicherheit durch Multiple Channels
- **Nachteile:**
  - Erhöhte Komplexität
  - Testing Overhead
  - Configuration Management

## 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()

```

1 // Controller
2 public class ForumController {
3     private ForumService forumService;
4     private AccessValidator accessValidator;
5
6     public Discussion addNewDiscussion(
7         String topicId,
8         DiscussionRequest request) {
9         // Validate access
10        accessValidator.validateUserAccess(
11            request.getUserId());
12
13        // Create discussion
14        return forumService.createDiscussion(
15            topicId,
16            request.getTitle(),
17            request.getContent()
18        );
19    }
20 }
21
22 // Domain Model
23 public class Discussion {
24     private String title;
25     private String content;
26     private User author;
27     private List<Comment> comments;

```

```

28     private LocalDateTime createdAt;
29
30     public void addComment(Comment comment) {
31         validateComment(comment);
32         comments.add(comment);
33     }
34
35     public int getContributionCount() {
36         return 1 + comments.size(); // Discussion +
37         Comments
38     }
39 }
40 // Service Layer
41 public class ForumService {
42     private DiscussionRepository repository;
43     private TopicRepository topicRepository;
44
45     @Transactional
46     public Discussion createDiscussion(
47         String topicId,
48         String title,
49         String content) {
50         // Find topic
51         Topic topic = topicRepository
52             .findById(topicId)
53             .orElseThrow(TopicNotFoundException::new);
54
55         // Create discussion
56         Discussion discussion = new Discussion(
57             title, content);
58
59         // Add to topic
60         topic.addDiscussion(discussion);
61
62         // Save
63         return repository.save(discussion);
64     }
65 }
66
67 // Unit Test
68 public class ForumControllerTest {
69     @Mock
70     private ForumService forumService;
71
72     @Mock
73     private AccessValidator accessValidator;
74
75     @InjectMocks
76     private ForumController controller;
77
78     @Test
79     public void shouldCreateNewDiscussion() {
80         // Given
81         DiscussionRequest request = new
82             DiscussionRequest(
83                 "userId", "title", "content");
84
85         // When
86         controller.addNewDiscussion("topicId",
87             request);
88
89         // Then
90         verify(accessValidator)
91             .validateUserAccess("userId");
92         verify(forumService)
93             .createDiscussion(
94                 "topicId", "title", "content");
95     }
96 }

```

## 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

```

1 // Information Expert: Order berechnet eigenen
2 // Gesamtbetrag
3 public class Order {
4     private List<OrderLine> lines;
5     private Customer customer;
6
7     public Money calculateTotal() {
8         return lines.stream()
9             .map(OrderLine::getSubtotal)
10            .reduce(Money.ZERO, Money::add);
11    }
12 }
13
14 // Low Coupling: Interface statt konkreter
15 // Implementierung
16 public interface PaymentGateway {
17     PaymentResult processPayment(Money amount);
18 }
19
20 public class OrderService {
21     private final PaymentGateway paymentGateway;
22
23     public OrderResult createOrder(OrderRequest
24         request) {
25         Order order = createOrderFromRequest(request);
26
27         PaymentResult payment =
28             paymentGateway.processPayment(
29                 order.calculateTotal());
30     }
31 }

```

```

28         if (payment.isSuccessful()) {
29             return OrderResult.success(order);
30         } else {
31             return
32                 OrderResult.failed(payment.getReason());
33         }
34     }
35 }
36 // High Cohesion: Spezialisierte Services
37 public class OrderValidator {
38     public void validate(Order order) {
39         validateCustomer(order.getCustomer());
40         validateOrderLines(order.getLines());
41         validateDeliveryAddress(
42             order.getDeliveryAddress());
43     }
44 }
45
46 public class InventoryService {
47     public void reserveStock(Order order) {
48         for (OrderLine line : order.getLines()) {
49             reserveItem(
50                 line.getProduct(),
51                 line.getQuantity());
52         }
53     }
54 }

```

## 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:

```

1 @RestController
2 public class OrderController {
3     private OrderService orderService;
4     private PaymentService paymentService;
5
6     public OrderResponse createOrder(
7         OrderRequest request) {
8         // 1: Validiere Bestellung
9         OrderValidator.validate(request);

```

```

10
11         // 2: Erstelle Order
12         Order order =
13             orderService.createOrder(request);
14
15         // 3: Prozessiere Zahlung
16         PaymentResult payment =
17             paymentService.processPayment(
18                 order.getId(),
19                 order.getTotal());
20
21         // 4: Bestätige Order
22         if (payment.isSuccessful()) {
23             order.confirm();
24             orderService.save(order);
25             return OrderResponse.success(order);
26         } else {
27             return OrderResponse.failed(
28                 payment.getReason());
29         }
30     }
31 }
32
33 public class OrderService {
34     @Transactional
35     public Order createOrder(OrderRequest request) {
36         // 2.1: Create order entity
37         Order order = new Order(
38             request.getCustomerId());
39
40         // 2.2: Add items
41         for (OrderItemRequest item :
42             request.getItems()) {
43             Product product = productRepository
44                 .findById(item.getProductId())
45                 .orElseThrow(ProductNotFoundException::new);
46
47             order.addItem(
48                 product,
49                 item.getQuantity());
50         }
51         // 2.3: Save order
52         return orderRepository.save(order);
53     }

```

## Sequenzdiagramm Analyse:

- **Objekte:**
  - OrderController als Fassade
  - OrderService für Geschäftslogik
  - PaymentService für Zahlungen
  - Order als Domain Object
- **Verantwortlichkeiten:**
  - Controller: Koordination
  - Service: Transaktionslogik
  - Domain: Business Rules
- **GRASP Patterns:**
  - Controller Pattern
  - Information Expert
  - Creator Pattern

## 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:

```
1 // Template Method Pattern
2 public abstract class DocumentProcessor {
3     public final void processDocument(byte[] data) {
4         Document doc = parseDocument(data);
5         validateDocument(doc);
6         processContent(doc);
7         saveResults(doc);
8     }
9
10    protected abstract Document parseDocument(byte[] data);
11    protected abstract void processContent(Document doc);
12
13    // Hook methods mit Default-Implementation
14    protected void validateDocument(Document doc) {
15        if (doc.isEmpty()) {
16            throw new EmptyDocumentException();
17        }
18    }
19
20    protected void saveResults(Document doc) {
21        // Default-Speicherung
22    }
23 }
24
25 // Factory Method Pattern
26 public abstract class DocumentParserFactory {
```

```
27     public abstract DocumentParser
28         createParser(String fileType);
29 }
30
31 public class PDFParserFactory
32     extends DocumentParserFactory {
33     @Override
34     public DocumentParser createParser(String
35         fileType) {
36         if ("pdf".equals(fileType)) {
37             return new PDFParser();
38         }
39         throw new UnsupportedOperationException(fileType);
40     }
41 }
42
43 // Strategy Pattern
44 public interface ProcessingStrategy {
45     void process(Document doc);
46 }
47
48 public class TextExtractionStrategy
49     implements ProcessingStrategy {
50     @Override
51     public void process(Document doc) {
52         // Text aus Dokument extrahieren
53     }
54 }
55
56 public class MetadataExtractionStrategy
57     implements ProcessingStrategy {
58     @Override
59     public void process(Document doc) {
60         // Metadaten extrahieren
61     }
62 }
```

### 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.

```
1 // Observer Pattern
2 public interface EventListener<T extends Event> {
3     void onEvent(T event);
4 }
```

```
4 }
5
6 public class EventBus {
7     private Map<Class<?>, List<EventListener>>
8         listeners = new HashMap<>();
9
10    public <T extends Event> void register(
11        Class<T> eventType,
12        EventListener<T> listener) {
13        listeners.computeIfAbsent(
14            eventType,
15            k -> new ArrayList<>()
16        ).add(listener);
17    }
18
19    public void publish(Event event) {
20        List<EventListener> eventListeners =
21            listeners.getOrDefault(
22                event.getClass(),
23                Collections.emptyList()
24            );
25
26        for (EventListener listener : eventListeners) {
27            try {
28                listener.onEvent(event);
29            } catch (Exception e) {
30                handleError(listener, event, e);
31            }
32        }
33    }
34 }
35
36 // Command Pattern
37 public interface Command {
38     void execute();
39     void undo();
40 }
41
42 public class CreateOrderCommand implements Command {
43     private OrderService orderService;
44     private OrderRequest request;
45     private Order createdOrder;
46
47     @Override
48     public void execute() {
49         createdOrder =
50             orderService.createOrder(request);
51     }
52
53     @Override
54     public void undo() {
55         if (createdOrder != null) {
56             orderService.cancelOrder(createdOrder.getId());
57         }
58     }
59 }
60
61 public class CommandProcessor {
62     private Deque<Command> executedCommands =
63         new ArrayDeque<>();
64
65     public void execute(Command command) {
66         command.execute();
67         executedCommands.push(command);
68     }
69
70     public void undo() {
71         if (!executedCommands.isEmpty()) {
72             Command command = executedCommands.pop();
73             command.undo();
74         }
75     }
76 }
```

```

73     }
74 }
75 }

```

#### Testen:

```

1 public class EventBusTest {
2     private EventBus eventBus;
3     private TestEventListener listener;
4
5     @BeforeEach
6     void setUp() {
7         eventBus = new EventBus();
8         listener = new TestEventListener();
9     }
10
11     @Test
12     void shouldDeliverEventToRegisteredListener() {
13         // Given
14         TestEvent event = new TestEvent("test");
15         eventBus.register(TestEvent.class, listener);
16
17         // When
18         eventBus.publish(event);
19
20         // Then
21         assertEquals(1, listener.getEventCount());
22         assertEquals(event, listener.getLastEvent());
23     }
24 }
25
26 public class CommandProcessorTest {
27     private CommandProcessor processor;
28     private OrderService orderService;
29
30     @Test
31     void shouldExecuteAndUndoCommand() {
32         // Given
33         OrderRequest request = new OrderRequest();
34         Command command = new CreateOrderCommand(
35             orderService, request);
36
37         // When
38         processor.execute(command);
39         processor.undo();
40
41         // Then
42         verify(orderService).createOrder(request);
43         verify(orderService).cancelOrder(any());
44     }
45 }

```

## 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.

```

1 // Plugin Interface (Strategy Pattern)
2 public interface Plugin {
3     String getName();
4     void initialize();
5     void processData(Data data);
6 }
7
8 // Plugin Factory
9 public abstract class PluginFactory {
10     public abstract Plugin createPlugin(
11         String pluginType);
12
13     protected void validatePlugin(Plugin plugin) {
14         // Validation logic
15     }
16 }
17
18 // Plugin Manager (Observer Pattern)
19 public class PluginManager {
20     private List<PluginObserver> observers =
21         new ArrayList<>();
22     private Map<String, Plugin> activePlugins =
23         new HashMap<>();
24
25     public void registerPlugin(Plugin plugin) {
26         plugin.initialize();
27         activePlugins.put(plugin.getName(), plugin);
28         notifyObservers(
29             new PluginEvent(
30                 PluginEventType.REGISTERED,
31                 plugin
32             )
33         );
34     }
35
36     public void processData(Data data) {
37         for (Plugin plugin : activePlugins.values()) {
38             try {
39                 plugin.processData(data);
40             } catch (Exception e) {
41                 handlePluginError(plugin, e);
42             }
43         }
44     }
45
46     private void notifyObservers(PluginEvent event) {
47         for (PluginObserver observer : observers) {
48             observer.onPluginEvent(event);
49         }
50     }
51 }
52
53 // Concrete Implementation
54 public class ImageProcessingPlugin implements Plugin {
55     private ProcessingStrategy strategy;
56
57     @Override
58     public void processData(Data data) {
59         if (data instanceof ImageData) {

```

```

60         strategy.process((ImageData) data);
61     }
62 }
63
64 // Usage
65 public class Application {
66     public void initializePlugins() {
67         PluginManager manager = new PluginManager();
68         PluginFactory factory = new
69             DefaultPluginFactory();
70
71         // Load plugins
72         Plugin imagePlugin =
73             factory.createPlugin("image-processing");
74         Plugin analyticsPlugin =
75             factory.createPlugin("analytics");
76
77         // Register plugins
78         manager.registerPlugin(imagePlugin);
79         manager.registerPlugin(analyticsPlugin);
80
81         // Process data
82         Data data = loadData();
83         manager.processData(data);
84     }
85 }

```

#### Pattern Zusammenspiel:

- **Factory Pattern:**
  - Erzeugt Plugin-Instanzen
  - Kapselt Erstellung
  - Validiert Plugins
- **Strategy Pattern:**
  - Plugin-Interface
  - Austauschbare Verarbeitung
  - Erweiterbarkeit
- **Observer Pattern:**
  - Plugin-Lifecycle Events
  - Losgelöste Beobachter
  - Flexible Reaktionen

## Weitere KR und Beispiele für Design Patterns

### Chain of Responsibility implementieren

#### 1. Handler Struktur

- **Handler Interface:**
  - Einheitliche Methode
  - Nächster Handler
  - Handlungslogik
- **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
  - Handlungslogik
- **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.

```

1 // Handler Interface
2 public interface AuthHandler {
3     void setNext(AuthHandler next);
4     void handle(AuthRequest request)
5         throws AuthException;
6 }
7
8 // Abstract Base Handler
9 public abstract class BaseAuthHandler
10     implements AuthHandler {
11     protected AuthHandler nextHandler;
12
13     @Override
14     public void setNext(AuthHandler next) {
15         this.nextHandler = next;
16     }
17
18     protected void handleNext(AuthRequest request)
19         throws AuthException {
20         if (nextHandler != null) {
21             nextHandler.handle(request);
22         }
23     }
24 }
25
26 // Concrete Handlers
27 public class TokenValidationHandler
28     extends BaseAuthHandler {
29     @Override
30     public void handle(AuthRequest request)
31         throws AuthException {
32         String token = request.getToken();
33         if (token == null || token.isEmpty()) {
34             throw new AuthException("Missing token");
35         }
36         if (!isValidToken(token)) {
37             throw new AuthException("Invalid token");
38         }
39         handleNext(request);
40     }

```

```

41 }
42
43 public class RoleCheckHandler extends BaseAuthHandler {
44     private Set<String> requiredRoles;
45
46     @Override
47     public void handle(AuthRequest request)
48         throws AuthException {
49         Set<String> userRoles = getUserRoles(
50             request.getUserId());
51         if (!userRoles.containsAll(requiredRoles)) {
52             throw new AuthException(
53                 "Insufficient privileges");
54         }
55         handleNext(request);
56     }
57 }
58
59 // Chain Builder
60 public class AuthChainBuilder {
61     public AuthHandler buildChain() {
62         AuthHandler tokenHandler =
63             new TokenValidationHandler();
64         AuthHandler roleHandler =
65             new RoleCheckHandler(ADMIN_ROLES);
66         AuthHandler ipHandler =
67             new IPValidationHandler();
68
69         tokenHandler.setNext(roleHandler);
70         roleHandler.setNext(ipHandler);
71
72         return tokenHandler;
73     }
74 }
75
76 // Usage
77 public class SecurityService {
78     private AuthHandler authChain;
79
80     public void authenticateRequest(AuthRequest
81         request) {
82         try {
83             authChain.handle(request);
84             // Request authenticated
85             processRequest(request);
86         } catch (AuthException e) {
87             handleAuthFailure(e);
88         }
89     }

```



## 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.

```
1 // State Interface
2 public interface DocumentState {
3     void review(Document document);
4     void approve(Document document);
5     void reject(Document document);
6     void publish(Document document);
7 }
8
9 // Concrete States
10 public class DraftState implements DocumentState {
11     @Override
12     public void review(Document document) {
13         // Validiere Review-Berechtigung
14         validateReviewPermission();
15
16         // Prüfe Dokument-Vollständigkeit
17         if (!document.isComplete()) {
18             throw new IllegalStateException(
19                 "Document incomplete");
20         }
21
22         // Wechsel zu Review-Zustand
23         document.setState(new ReviewState());
24     }
25
26     @Override
27     public void approve(Document document) {
28         throw new IllegalStateException(
29             "Draft cannot be approved");
30     }
31
32     @Override
33     public void reject(Document document) {
34         throw new IllegalStateException(
35             "Draft cannot be rejected");
36     }
37
38     @Override
39     public void publish(Document document) {
40         throw new IllegalStateException(
41             "Draft cannot be published");
42     }
43 }
```

```
43 }
44
45 public class ReviewState implements DocumentState {
46     @Override
47     public void approve(Document document) {
48         validateApprovalPermission();
49         document.setState(new ApprovedState());
50     }
51
52     @Override
53     public void reject(Document document) {
54         document.addComment(
55             "Rejected in Review. Needs revision.");
56         document.setState(new DraftState());
57     }
58 }
59
60 // Context
61 public class Document {
62     private DocumentState state;
63     private String content;
64     private List<String> comments;
65     private User author;
66
67     public Document() {
68         this.state = new DraftState();
69         this.comments = new ArrayList<>();
70     }
71
72     public void review() {
73         state.review(this);
74     }
75
76     public void approve() {
77         state.approve(this);
78     }
79
80     public void reject() {
81         state.reject(this);
82     }
83
84     public void publish() {
85         state.publish(this);
86     }
87
88     void setState(DocumentState state) {
89         this.state = state;
90         notifyStateChange();
91     }
92
93     public boolean isComplete() {
94         return content != null &&
95             !content.isEmpty() &&
96             author != null;
97     }
98
99     public void addComment(String comment) {
100         comments.add(comment);
101     }
102 }
103
104 // Usage
105 public class DocumentService {
106     public void processDocument(Document doc) {
107         try {
108             doc.review();
109             // Automatische Prüfungen
110             if (automaticChecksPass(doc)) {
111                 doc.approve();
112                 doc.publish();
113             }
114         } catch (IllegalStateException e) {
115             handleError(doc, e);
116         }
117     }
118 }
```

```
113     } else {
114         doc.reject();
115     }
116 } catch (IllegalStateException e) {
117     handleError(doc, e);
118 }
119 }
120 }
```

## State Pattern Benefits:

- **Klare Zustandsstrennung:**
  - Jeder Zustand in eigener Klasse
  - Spezifisches Verhalten gekapselt
  - Einfach erweiterbar
- **Validierung:**
  - Unmögliche Übergänge verhindert
  - Zustandsspezifische Prüfungen
  - Klare Fehlermeldungen
- **Wartbarkeit:**
  - Neue Zustände einfach hinzufügbär
  - Zustandslogik zentralisiert
  - Gute Testbarkeit

## 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:

```
1 public class DocumentStateTest {
2     private Document document;
3
4     @BeforeEach
5     void setUp() {
6         document = new Document();
7     }
8
9     @Test
10    void draftShouldTransitionToReviewWhenComplete() {
11        // Given
12        document.setContent("Test content");
13        document.setAuthor(new User("author"));
14
15        // When
16        document.review();
17
18        // Then
```

```
19        assertInstanceOf(ReviewState.class,
20            document.getState());
21    }
22
23    @Test
24    void draftShouldNotAllowApproval() {
25        // Given
26        document = new Document(); // In Draft state
27
28        // When/Then
29        assertThrows(IllegalStateException.class,
30            () -> document.approve());
31    }
32
33    @Test
34    void reviewStateShouldAllowApprovalOrRejection() {
35        // Given
36        document.setContent("Test content");
37        document.setAuthor(new User("author"));
38        document.review(); // Move to Review state
39
40        // When/Then
41        assertDoesNotThrow(() -> document.approve());
42
43        // Reset and test rejection
44        document.review();
45        assertDoesNotThrow(() -> document.reject());
46    }
47 }
48
49 // Integration Test
50 public class DocumentWorkflowTest {
51     private DocumentService service;
52     private Document document;
53
54     @Test
55     void shouldCompleteWorkflow() {
56         // Given
57         document = createValidDocument();
58
59         // When
60         service.processDocument(document);
61
62         // Then
63         assertInstanceOf(PublishedState.class,
64             document.getState());
65         assertTrue(document.isPublished());
66         assertFalse(document.getComments().isEmpty());
67     }
68
69     @Test
70     void shouldHandleInvalidDocument() {
71         // Given
72         document = new Document(); // Incomplete
73
74         // When
75         service.processDocument(document);
76
77         // Then
78         assertInstanceOf(DraftState.class,
79             document.getState());
80         assertFalse(document.isPublished());
81         assertTrue(document.hasRejectionComments());
82     }
83 }
```

## 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.

```
1 // Subject Interface
2 public interface MonitoringSubject {
3     void addObserver(MonitoringObserver observer);
4     void removeObserver(MonitoringObserver observer);
5     void notifyObservers(SystemStatus status);
6 }
7
8 // Observer Interface
9 public interface MonitoringObserver {
10     void update(SystemStatus status);
11 }
12
13 // Concrete Subject
14 public class SystemMonitor implements
15     MonitoringSubject {
16     private List<MonitoringObserver> observers =
17         Collections.synchronizedList(new
18             ArrayList<>());
19     private SystemStatus currentStatus;
20
21     @Override
```

```

20 public void addObserver(MonitoringObserver
21     observer) {
22     observers.add(observer);
23     // Send current status to new observer
24     if (currentStatus != null) {
25         observer.update(currentStatus);
26     }
27 }
28
29 @Override
30 public void notifyObservers(SystemStatus status) {
31     this.currentStatus = status;
32     observers.forEach(observer -> {
33         try {
34             observer.update(status);
35         } catch (Exception e) {
36             handleObserverError(observer, e);
37         }
38     });
39 }
40
41 public void checkSystem() {
42     SystemStatus status = calculateSystemStatus();
43     if (statusChanged(status)) {
44         notifyObservers(status);
45     }
46 }
47
48 // Concrete Observers
49 public class AlertSystem implements MonitoringObserver
50 {
51     @Override
52     public void update(SystemStatus status) {
53         if (status.isCritical()) {
54             sendAlerts(status);
55         }
56     }
57
58     private void sendAlerts(SystemStatus status) {
59         // Send SMS, Email, etc.
60     }
61 }
62
63 public class DashboardUpdater implements
64     MonitoringObserver {
65     private Dashboard dashboard;
66
67     @Override
68     public void update(SystemStatus status) {
69         dashboard.updateMetrics(
70             status.getMetrics());
71     }
72 }
73
74 // Usage
75 public class MonitoringService {
76     private SystemMonitor monitor;
77
78     @Scheduled(fixedRate = 5000)
79     public void monitorSystem() {
80         monitor.checkSystem();
81     }
82 }

```

## 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.

```

1 // Abstract Base Class
2 public abstract class ReportGenerator {
3
4     // Template Method
5     public final Report generateReport(ReportData
6         data) {
7         validateData(data);
8         Report report = new Report();
9
10        try {
11            // 1. Header
12            report.setHeader(createHeader(data));
13
14            // 2. Content
15            List<ReportSection> sections =
16                processData(data);
17            report.setSections(sections);
18
19            // 3. Optional Customization
20            if (shouldCustomize()) {
21                customizeReport(report);
22            }
23
24            // 4. Footer
25            report.setFooter(createFooter(data));
26
27            // 5. Optional Validation
28            validateReport(report);
29
30            return report;
31        } catch (Exception e) {
32            handleError(e);
33            throw new ReportGenerationException(e);
34        }
35    }
36
37    // Abstract Methods (must implement)
38    protected abstract ReportHeader createHeader(
39        ReportData data);
40    protected abstract List<ReportSection> processData(
41        ReportData data);
42    protected abstract ReportFooter createFooter(
43        ReportData data);
44
45    // Hook Methods (optional override)
46    protected boolean shouldCustomize() {
47        return false;
48    }
49
50    protected void customizeReport(Report report) {
51        // Default empty implementation
52    }
53 }

```

```

52     protected void validateReport(Report report) {
53         // Default validation
54         if (report.getSections().isEmpty()) {
55             throw new EmptyReportException();
56         }
57     }
58 }
59
60     protected void validateData(ReportData data) {
61         if (data == null) {
62             throw new InvalidDataException("No data");
63         }
64     }
65 }
66
67 // Concrete Implementation
68 public class SalesReport extends ReportGenerator {
69     private SalesDataAnalyzer analyzer;
70
71     @Override
72     protected ReportHeader createHeader(ReportData
73         data) {
74         SalesData salesData = (SalesData) data;
75         return new ReportHeader(
76             "Sales Report",
77             salesData.getPeriod(),
78             salesData.getDepartment()
79         );
80     }
81
82     @Override
83     protected List<ReportSection> processData(
84         ReportData data) {
85         SalesData salesData = (SalesData) data;
86
87         List<ReportSection> sections = new
88             ArrayList<>();
89
90         // Sales Summary
91         sections.add(
92             createSummarySection(
93                 analyzer.analyzeSales(salesData)));
94
95         // Top Products
96         sections.add(
97             createTopProductsSection(
98                 analyzer.getTopProducts(salesData)));
99
100        // Revenue Trends
101        sections.add(
102            createTrendsSection(
103                analyzer.analyzeTrends(salesData)));
104
105        return sections;
106    }
107
108    @Override
109    protected boolean shouldCustomize() {
110        return true;
111    }
112
113    @Override
114    protected void customizeReport(Report report) {
115        // Add company branding
116        report.setStyle(new CorporateStyle());
117
118        // Add charts
119        report.addCharts(createSalesCharts());
120    }

```

```

120 // Usage
121 public class ReportingService {
122     private Map<ReportType, ReportGenerator>
123         generators;
124
125     public Report createReport(
126         ReportType type,
127         ReportData data) {
128         ReportGenerator generator =
129             generators.get(type);
130         if (generator == null) {
131             throw new UnknownReportTypeException(type);
132         }
133         return generator.generateReport(data);
134     }

```

## 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:**
  - Strategiewahl
  - Parameter
  - Fallback

## Strategy Pattern: Payment Processing

**Aufgabe:** Implementieren Sie ein flexibles Zahlungssystem mit verschiedenen Strategien.

```

1 // Strategy Interface
2 public interface PaymentStrategy {
3     PaymentResult process(PaymentRequest request);
4 }
5
6 // Concrete Strategies
7 public class CreditCardPayment implements
8     PaymentStrategy {
9     private CreditCardValidator validator;
10    private PaymentGateway gateway;
11
12    @Override
13    public PaymentResult process(PaymentRequest
14        request) {
15        // Validate credit card
16        CreditCardDetails card =
17            request.getCardDetails();
18        if (!validator.isValid(card)) {
19            return PaymentResult.failed(
20                "Invalid card details");
21        }
22
23        // Process payment

```

```

21     try {
22         TransactionResult result =
23             gateway.processCard(
24                 card,
25                 request.getAmount()
26             );
27
28         return PaymentResult.success(
29             result.getTransactionId());
30     } catch (GatewayException e) {
31         return PaymentResult.failed(
32             e.getMessage());
33     }
34 }
35
36 public class PayPalPayment implements PaymentStrategy {
37     private PayPalClient paypalClient;
38
39     @Override
40     public PaymentResult process(PaymentRequest
41         request) {
42         try {
43             String paymentId =
44                 paypalClient.createPayment(
45                     request.getAmount(),
46                     request.getCurrency());
47
48             PayPalResponse response =
49                 paypalClient.executePayment(paymentId);
50
51             return PaymentResult.success(
52                 response.getTransactionId());
53         } catch (PayPalException e) {
54             return PaymentResult.failed(
55                 "PayPal error: " + e.getMessage());
56         }
57     }
58
59     // Context
60     public class PaymentProcessor {
61         private Map<PaymentType, PaymentStrategy>
62             strategies =
63                 new EnumMap<>(PaymentType.class);
64
65         public PaymentProcessor() {
66             strategies.put(
67                 PaymentType.CREDIT_CARD,
68                 new CreditCardPayment()
69             );
70             strategies.put(
71                 PaymentType.PAYPAL,
72                 new PayPalPayment()
73             );
74         }
75
76         public PaymentResult processPayment(
77             PaymentRequest request) {
78             PaymentStrategy strategy =
79                 strategies.get(request.getType());
80
81             if (strategy == null) {
82                 return PaymentResult.failed(
83                     "Unsupported payment type");
84             }
85
86             return strategy.process(request);
87         }
88     }

```

```

88
89 // Usage with Error Handling
90 public class CheckoutService {
91     private PaymentProcessor paymentProcessor;
92     private OrderRepository orderRepository;
93
94     @Transactional
95     public CheckoutResult checkout(Order order) {
96         try {
97             PaymentRequest request =
98                 createPaymentRequest(order);
99
100             PaymentResult result =
101                 paymentProcessor.processPayment(request);
102
103             if (result.isSuccessful()) {
104                 order.markAsPaid(result.getTransactionId());
105                 orderRepository.save(order);
106                 return CheckoutResult.success(order);
107             } else {
108                 return CheckoutResult.failed(
109                     result.getErrorMessage());
110             }
111         } catch (Exception e) {
112             order.markAsFailed(e.getMessage());
113             orderRepository.save(order);
114             throw new CheckoutException(
115                 "Checkout failed", e);
116         }
117     }
118 }

```

#### Tests:

```

1 @ExtendWith(MockitoExtension.class)
2 public class PaymentProcessorTest {
3     @Mock
4     private CreditCardPayment creditCardPayment;
5
6     @Mock
7     private PayPalPayment paypalPayment;
8
9     @InjectMocks
10    private PaymentProcessor processor;
11
12    @Test
13    void shouldProcessCreditCardPayment() {
14        // Given
15        PaymentRequest request = new PaymentRequest(
16            PaymentType.CREDIT_CARD,
17            Money.of(100, "USD")
18        );
19
20        PaymentResult expected =
21            PaymentResult.success("TX123");
22
23        when(creditCardPayment.process(request))
24            .thenReturn(expected);
25
26        // When
27        PaymentResult result =
28            processor.processPayment(request);
29
30        // Then
31        assertEquals(expected, result);
32        verify(creditCardPayment).process(request);
33        verifyNoInteractions(paypalPayment);
34    }
35
36    @Test

```

```

37    void shouldHandleUnsupportedPaymentType() {
38        // Given
39        PaymentRequest request = new PaymentRequest(
40            PaymentType.BITCOIN,
41            Money.of(100, "USD")
42        );
43
44        // When
45        PaymentResult result =
46            processor.processPayment(request);
47
48        // Then
49        assertFalse(result.isSuccessful());
50        assertEquals(
51            "Unsupported payment type",
52            result.getErrorMessage()
53        );
54    }
55 }

```

## 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.

```
1 // Product Interfaces
2 public interface Button {
3     void render();
4     void handleClick();
5 }
6
7 public interface TextField {
8     void render();
9     void handleInput(String text);
10 }
11
12 // Abstract Factory
13 public interface ComponentFactory {
14     Button createButton();
15     TextField createTextField();
16 }
17
18 // Concrete Products for Light Theme
```

```
19 public class LightButton implements Button {
20     @Override
21     public void render() {
22         // Render light themed button
23     }
24
25     @Override
26     public void handleClick() {
27         // Handle click with light theme feedback
28     }
29 }
30
31 public class LightTextField implements TextField {
32     @Override
33     public void render() {
34         // Render light themed text field
35     }
36
37     @Override
38     public void handleInput(String text) {
39         // Handle input with light theme styling
40     }
41 }
42
43 // Concrete Factory for Light Theme
44 public class LightThemeFactory implements
45     ComponentFactory {
46     @Override
47     public Button createButton() {
48         return new LightButton();
49     }
50
51     @Override
52     public TextField createTextField() {
53         return new LightTextField();
54     }
55 }
56
57 // Dark Theme Implementation
58 public class DarkThemeFactory implements
59     ComponentFactory {
60     @Override
61     public Button createButton() {
62         return new DarkButton();
63     }
64
65     @Override
66     public TextField createTextField() {
67         return new DarkTextField();
68     }
69 }
70
71 // Factory Provider
72 public class ThemeFactory {
73     private static Map<ThemeType, ComponentFactory>
74     factories = new EnumMap<>(ThemeType.class);
75
76     static {
77         factories.put(ThemeType.LIGHT,
78             new LightThemeFactory());
79         factories.put(ThemeType.DARK,
80             new DarkThemeFactory());
81     }
82
83     public static ComponentFactory getFactory(
84         ThemeType type) {
85         ComponentFactory factory = factories.get(type);
86         if (factory == null) {
87             throw new IllegalArgumentException(
88                 "Unknown theme: " + type);
89         }
90     }
91 }
```

```
87     }
88     return factory;
89 }
90
91 // Usage
92 public class Form {
93     private ComponentFactory factory;
94     private List<Button> buttons = new ArrayList<>();
95     private List<TextField> fields = new ArrayList<>();
96
97     public Form(ThemeType theme) {
98         this.factory = ThemeFactory.getFactory(theme);
99     }
100
101     public void addButton() {
102         Button button = factory.createButton();
103         buttons.add(button);
104     }
105
106     public void addTextField() {
107         TextField field = factory.createTextField();
108         fields.add(field);
109     }
110
111     public void render() {
112         buttons.forEach(Button::render);
113         fields.forEach(TextField::render);
114     }
115 }
116 }
```

### 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.

```
1 // Component Interface
2 public interface FileSystemItem {
3     String getName();
4     long getSize();
5     void accept(FileSystemVisitor visitor);
6 }
7
8 // Leaf
9 public class File implements FileSystemItem {
10     private String name;
11     private long size;
12
13     @Override
14     public String getName() {
15         return name;
16     }
17
18     @Override
19     public long getSize() {
20         return size;
21     }
22
23     @Override
24     public void accept(FileSystemVisitor visitor) {
25         visitor.visitFile(this);
26     }
27 }
```

```
27 }
28
29 // Composite
30 public class Directory implements FileSystemItem {
31     private String name;
32     private List<FileSystemItem> children =
33         new ArrayList<>();
34
35     @Override
36     public String getName() {
37         return name;
38     }
39
40     @Override
41     public long getSize() {
42         return children.stream()
43             .mapToLong(FileSystemItem::getSize)
44             .sum();
45     }
46
47     public void addItem(FileSystemItem item) {
48         children.add(item);
49     }
50
51     public void removeItem(FileSystemItem item) {
52         children.remove(item);
53     }
54
55     public List<FileSystemItem> getChildren() {
56         return Collections.unmodifiableList(children);
57     }
58
59     @Override
60     public void accept(FileSystemVisitor visitor) {
61         visitor.visitDirectory(this);
62         children.forEach(child ->
63             child.accept(visitor));
64     }
65
66 // Visitor Interface for Operations
67 public interface FileSystemVisitor {
68     void visitFile(File file);
69     void visitDirectory(Directory directory);
70 }
71
72 // Concrete Visitor for Size Calculation
73 public class SizeCalculatorVisitor
74     implements FileSystemVisitor {
75     private long totalSize = 0;
76
77     @Override
78     public void visitFile(File file) {
79         totalSize += file.getSize();
80     }
81
82     @Override
83     public void visitDirectory(Directory directory) {
84         // Optional: Add directory overhead
85         totalSize += 4096; // Directory entry size
86     }
87
88     public long getTotalSize() {
89         return totalSize;
90     }
91 }
92
93 // Usage
94 public class FileSystemManager {
95     private Directory root;
```

```
96
97     public long calculateTotalSize() {
98         SizeCalculatorVisitor visitor =
99             new SizeCalculatorVisitor();
100         root.accept(visitor);
101         return visitor.getTotalSize();
102     }
103
104     public void printStructure() {
105         printItem(root, 0);
106     }
107
108     private void printItem(
109         FileSystemItem item,
110         int level) {
111         String indent = " ".repeat(level);
112         System.out.println(indent + item.getName());
113
114         if (item instanceof Directory) {
115             Directory dir = (Directory) item;
116             dir.getChildren().forEach(child ->
117                 printItem(child, level + 1));
118         }
119     }
120 }
121
122 // Tests
123 public class FileSystemTest {
124     private Directory root;
125     private File file1;
126     private File file2;
127
128     @BeforeEach
129     void setUp() {
130         root = new Directory("root");
131         file1 = new File("file1", 1000);
132         file2 = new File("file2", 2000);
133
134         Directory subDir = new Directory("subDir");
135         subDir.addItem(file2);
136
137         root.addItem(file1);
138         root.addItem(subDir);
139     }
140
141     @Test
142     void shouldCalculateTotalSize() {
143         // When
144         SizeCalculatorVisitor visitor =
145             new SizeCalculatorVisitor();
146         root.accept(visitor);
147
148         // Then
149         assertEquals(7096, visitor.getTotalSize());
150         // 1000 + 2000 + 2*4096
151     }
152
153     @Test
154     void shouldTraverseStructure() {
155         // Given
156         List<String> visitedNames = new ArrayList<>();
157
158         FileSystemVisitor nameCollector =
159             new FileSystemVisitor() {
160                 @Override
161                 public void visitFile(File file) {
162                     visitedNames.add(file.getName());
163                 }
164
165                 @Override
```



```

166         public void visitDirectory(
167             Directory directory) {
168             visitedNames.add(directory.getName());
169         }
170     };
171
172     // When
173     root.accept(nameCollector);
174
175     // Then
176     assertEquals(Arrays.asList(
177         "root", "file1", "subDir", "file2"),
178         visitedNames);
179     }
180 }

```

#### Pattern Benefits:

- **Strukturelle Vorteile:**
  - Einheitliche Behandlung von Dateien und Verzeichnissen
  - Beliebige Verschachtelungstiefe
  - Einfache Erweiterung um neue Operationen durch Visitor
- **Wartbarkeit:**
  - Klare Trennung von Struktur und Operationen
  - Einfaches Hinzufügen neuer Komponententypen
  - Gute Testbarkeit
- **Flexibilität:**
  - Rekursive Verarbeitung
  - Polymorphes Verhalten
  - Erweiterbare Operationen

## 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

```
1 // Bestehende Schnittstelle
2 interface ModernPrinter {
3     void printDocument(String content);
4 }
5
6 // Alte Drittanbieter-Klasse
7 class LegacyPrinter {
8     public void print(String[] pages) {
9         for(String page : pages) {
10             System.out.println(page);
11         }
12     }
13 }
14
15 // Adapter
16 class PrinterAdapter implements ModernPrinter {
17     private LegacyPrinter legacyPrinter;
18
19     public PrinterAdapter(LegacyPrinter
20         printer) {
21         this.legacyPrinter = printer;
22     }
23
24     public void printDocument(String content) {
25         String[] pages = content.split("\n");
26         legacyPrinter.print(pages);
27     }
28 }
```

## Simple Factory

**Szenario:** Erzeugung von verschiedenen Datenbankverbindungen

```
1 class DatabaseFactory {
2     public static Database
3     createDatabase(String type) {
4         switch(type) {
5             case "MySQL":
6                 return new MySQLDatabase();
7             case "PostgreSQL":
8                 return new PostgreSQLDatabase();
9             default:
10                 throw new
11                     IllegalArgumentException("Unknown
12                     DB type");
13         }
14     }
15 }
16
17 // Verwendung
18 Database db =
19     DatabaseFactory.createDatabase("MySQL");
```

## Singleton

**Szenario:** Globale Konfigurationsverwaltung

```
1 public class Configuration {
2     private static Configuration instance;
3     private Map<String, String> config;
4
5     private Configuration() {
6         config = new HashMap<>();
7     }
8
9     public static Configuration getInstance() {
10         if(instance == null) {
11             instance = new Configuration();
12         }
13         return instance;
14     }
15 }
```

## Dependency Injection

**Szenario:** Flexible Logger-Implementation

```
1 interface Logger {
2     void log(String message);
3 }
4
5 class FileLogger implements Logger {
6     public void log(String message) {
7         // Log to file
8     }
9 }
10
11 class UserService {
12     private final Logger logger;
13
14     public UserService(Logger logger) { //
15         Dependency Injection
16         this.logger = logger;
17     }
18 }
```

## Proxy

**Szenario:** Verzögertes Laden eines großen Bildes

```
1 interface Image {
2     void display();
3 }
4
5 class RealImage implements Image {
6     private String filename;
7
8     public RealImage(String filename) {
9         this.filename = filename;
10         loadFromDisk();
11     }
12
13     private void loadFromDisk() {
14         System.out.println("Loading " +
15             filename);
16     }
17
18     public void display() {
19         System.out.println("Displaying " +
20             filename);
21     }
22 }
23
24 class ImageProxy implements Image {
25     private RealImage realImage;
26     private String filename;
27
28     public ImageProxy(String filename) {
29         this.filename = filename;
30     }
31
32     public void display() {
33         if(realImage == null) {
34             realImage = new RealImage(filename);
35         }
36         realImage.display();
37     }
38 }
```

## Chain of Responsibility

Szenario: Authentifizierungskette

```
1 abstract class AuthHandler {
2     protected AuthHandler next;
3
4     public void setNext(AuthHandler next) {
5         this.next = next;
6     }
7
8     public abstract boolean handle(String
9         username, String password);
10 }
11
12 class LocalAuthHandler extends AuthHandler {
13     public boolean handle(String username,
14         String password) {
15         if(checkLocalDB(username, password)) {
16             return true;
17         }
18         return next != null ?
19             next.handle(username, password) :
20             false;
21     }
22 }
23
24 class LDAPAuthHandler extends AuthHandler {
25     public boolean handle(String username,
26         String password) {
27         if(checkLDAP(username, password)) {
28             return true;
29         }
30         return next != null ?
31             next.handle(username, password) :
32             false;
33     }
34 }
```

## Decorator

Szenario: Dynamische Erweiterung eines Text-Editors

```
1 interface TextComponent {
2     String render();
3 }
4
5 class SimpleText implements TextComponent {
6     private String text;
7
8     public SimpleText(String text) {
9         this.text = text;
10    }
11
12    public String render() {
13        return text;
14    }
15 }
16
17 class BoldDecorator implements TextComponent {
18     private TextComponent component;
19
20     public BoldDecorator(TextComponent
21         component) {
22         this.component = component;
23     }
24
25     public String render() {
26         return "<b>" + component.render() +
27             "</b>";
28     }
29 }
```

## Observer

Szenario: News-Benachrichtigungssystem

```
1 interface NewsObserver {
2     void update(String news);
3 }
4
5 class NewsAgency {
6     private List<NewsObserver> observers = new
7         ArrayList<>();
8
9     public void addObserver(NewsObserver
10         observer) {
11         observers.add(observer);
12     }
13
14     public void notifyObservers(String news) {
15         for(NewsObserver observer : observers) {
16             observer.update(news);
17         }
18     }
19 }
20
21 class NewsChannel implements NewsObserver {
22     private String name;
23
24     public NewsChannel(String name) {
25         this.name = name;
26     }
27
28     public void update(String news) {
29         System.out.println(name + " received: "
30             + news);
31     }
32 }
```

## Strategy

Szenario: Verschiedene Zahlungsmethoden

```
1 interface PaymentStrategy {
2     void pay(int amount);
3 }
4
5 class CreditCardPayment implements
6     PaymentStrategy {
7     private String cardNumber;
8
9     public void pay(int amount) {
10         System.out.println("Paid " + amount + "
11             using Credit Card");
12     }
13 }
14
15 class PayPalPayment implements PaymentStrategy {
16     private String email;
17
18     public void pay(int amount) {
19         System.out.println("Paid " + amount + "
20             using PayPal");
21     }
22 }
```

## Strategy Pattern Implementation

```
1 public interface SortStrategy {
2     void sort(List<String> data);
3 }
4
5 public class QuickSort implements SortStrategy {
6     public void sort(List<String> data) {
7         // Implementierung
8     }
9 }
10
11 public class Context {
12     private SortStrategy strategy;
13
14     public void setStrategy(SortStrategy strategy) {
15         this.strategy = strategy;
16     }
17
18     public void executeStrategy(List<String> data) {
19         strategy.sort(data);
20     }
21 }
```

## Composite

Szenario: Dateisystem-Struktur

```
1 interface FileSystemComponent {
2     void list(String prefix);
3 }
4
5 class File implements FileSystemComponent {
6     private String name;
7
8     public void list(String prefix) {
9         System.out.println(prefix + name);
10    }
11 }
12
13 class Directory implements FileSystemComponent {
14     private String name;
15     private List<FileSystemComponent> children
16         = new ArrayList<>();
17
18     public void add(FileSystemComponent
19         component) {
20         children.add(component);
21     }
22
23     public void list(String prefix) {
24         System.out.println(prefix + name);
25         for(FileSystemComponent child :
26             children) {
27             child.list(prefix + " ");
28         }
29     }
30 }
```

## State

Szenario: Verkaufsautomat

```
1 interface VendingMachineState {
2     void insertCoin();
3     void ejectCoin();
4     void selectProduct();
5     void dispense();
6 }
7
8 class HasCoinState implements
9     VendingMachineState {
10     private VendingMachine machine;
11
12     public void selectProduct() {
13         System.out.println("Product selected");
14         machine.setState(machine.getSoldState());
15     }
16
17     public void insertCoin() {
18         System.out.println("Already have coin");
19     }
20 }
21
22 class VendingMachine {
23     private VendingMachineState currentState;
24
25     public void setState(VendingMachineState
26         state) {
27         this.currentState = state;
28     }
29
30     public void insertCoin() {
31         currentState.insertCoin();
32     }
33 }
```

### Visitor

**Szenario:** Dokumentstruktur mit verschiedenen Operationen

```
1 interface DocumentElement {
2     void accept(Visitor visitor);
3 }
4
5 interface Visitor {
6     void visit(Paragraph paragraph);
7     void visit(Heading heading);
8 }
9
10 class HTMLVisitor implements Visitor {
11     public void visit(Paragraph p) {
12         System.out.println("<p>" + p.getText()
13             + "</p>");
14     }
15
16     public void visit(Heading h) {
17         System.out.println("<h1>" + h.getText()
18             + "</h1>");
19     }
20 }
```

### Facade

**Szenario:** Vereinfachte Multimedia-Bibliothek

```
1 class MultimediaFacade {
2     private AudioSystem audio;
3     private VideoSystem video;
4     private SubtitleSystem subtitles;
5
6     public void playMovie(String movie) {
7         audio.initialize();
8         video.initialize();
9         subtitles.load(movie);
10        video.play(movie);
11        audio.play();
12    }
13 }
```

### Abstract Factory

**Szenario:** GUI-Elemente für verschiedene Betriebssysteme

```
1 interface GUIFactory {
2     Button createButton();
3     Checkbox createCheckbox();
4 }
5
6 class WindowsFactory implements GUIFactory {
7     public Button createButton() {
8         return new WindowsButton();
9     }
10
11     public Checkbox createCheckbox() {
12         return new WindowsCheckbox();
13     }
14 }
15
16 class MacFactory implements GUIFactory {
17     public Button createButton() {
18         return new MacButton();
19     }
20
21     public Checkbox createCheckbox() {
22         return new MacCheckbox();
23     }
24 }
```

### Factory Method Implementation

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

**Lösung:**

```
1 // Interface fuer Produkte
2 interface Document {
3     void open();
4     void save();
5 }
6
7 // Konkrete Produkte
8 class PdfDocument implements Document {
9     public void open() { /* ... */ }
10    public void save() { /* ... */ }
11 }
12
13 // Factory Method Pattern
14 abstract class DocumentCreator {
15     abstract Document createDocument();
16
17     // Template Method
18     final void processDocument() {
19         Document doc = createDocument();
20         doc.open();
21         doc.save();
22     }
23 }
24
25 // Konkrete Factory
26 class PdfDocumentCreator extends
27     DocumentCreator {
28     Document createDocument() {
29         return new PdfDocument();
30     }
31 }
```

## Observer Pattern Implementation

**Aufgabe:** Implementieren Sie ein Benachrichtigungssystem für Aktienkurse

**Lösung:**

```
1 interface StockObserver {
2     void update(String stock, double price);
3 }
4
5 class StockMarket {
6     private List<StockObserver> observers = new
        ArrayList<>();
7
8     public void attach(StockObserver observer) {
9         observers.add(observer);
10    }
11
12    public void notifyObservers(String stock,
        double price) {
13        for(StockObserver observer : observers)
14        {
15            observer.update(stock, price);
16        }
17    }
18 }
19 class StockDisplay implements StockObserver {
20     public void update(String stock, double
        price) {
21         System.out.println("Stock: " + stock +
22                             " updated to " +
23                             price);
24     }
25 }
```

## 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:** Refaktoren Sie den folgenden Code unter Anwendung geeigneter Patterns

## Vorher:

```

1 public class Order {
2     private List<OrderItem> items;
3     private double totalAmount;
4     private String status;
5
6     public void calculateTotal() {
7         totalAmount = 0;
8         for(OrderItem item : items) {
9             totalAmount += item.getPrice() *
10                item.getQuantity();
11            // Komplexe Rabattberechnung
12            if(item.getQuantity() > 10) {
13                totalAmount *= 0.9; // 10%
14                Rabatt
15            }
16            if(totalAmount > 1000) {
17                totalAmount *= 0.95; // 5%
18                Rabatt
19            }
20        }
21    }
22 }

```

```

19
20 public void processOrder() {
21     calculateTotal();
22     // Komplexe Statusberechnung
23     if(totalAmount < 100) {
24         status = "SMALL_ORDER";
25     } else if(totalAmount < 1000) {
26         status = "MEDIUM_ORDER";
27     } else {
28         status = "LARGE_ORDER";
29     }
30     // Weitere 20 Zeilen Status-Logik...
31 }
32 }

```

## Nachher:

```

1 public class Order {
2     private List<OrderItem> items;
3     private OrderStatus status;
4     private DiscountStrategy discountStrategy;
5
6     public Money calculateTotal() {
7         Money total = items.stream()
8             .map(OrderItem::getSubtotal)
9             .reduce(Money.ZERO, Money::add);
10
11         return discountStrategy.applyDiscount(total);
12     }
13
14     public void processOrder() {
15         Money total = calculateTotal();
16         status = OrderStatus.forAmount(total);
17         status.process(this);
18     }
19 }
20
21 public interface DiscountStrategy {
22     Money applyDiscount(Money amount);
23 }
24
25 public class QuantityBasedDiscount implements
26     DiscountStrategy {
27     public Money applyDiscount(Money amount) {
28         if(quantity > 10) {
29             return amount.multiply(0.9);
30         }
31         return amount;
32     }
33 }
34
35 public enum OrderStatus {
36     SMALL_ORDER(amount ->
37         amount.isLessThan(Money.of(100))),
38     MEDIUM_ORDER(amount ->
39         amount.isLessThan(Money.of(1000))),
40     LARGE_ORDER(amount -> true);

```

```

39 private final Predicate<Money> matcher;
40
41 public static OrderStatus forAmount(Money
42     amount) {
43     return Stream.of(values())
44         .filter(s -> s.matcher.test(amount))
45         .findFirst()
46         .orElseThrow();
47 }
48
49 public void process(Order order) {
50     // Status-spezifische Verarbeitung
51 }

```

## Angewendete Refactorings:

- Extract Method für Berechnungslogik
- Strategy Pattern für Rabattberechnung
- State Pattern für Orderstatus
- Replace Conditional with Polymorphism
- Value Objects für Geldbeträge

## 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

```
1 public class ShoppingCartTest {
2     private ShoppingCart cart;
3     private Product testProduct;
4
5     @BeforeEach
6     void setUp() {
7         cart = new ShoppingCart();
8         testProduct = new Product("Test",
9             Money.of(10));
10    }
11
12    @Test
13    void shouldCalculateTotal_whenEmpty() {
14        assertEquals(Money.ZERO,
15            cart.getTotal());
16    }
17
18    @Test
19    void shouldCalculateTotal_withOneItem() {
20        cart.addItem(testProduct, 1);
21        assertEquals(Money.of(10),
22            cart.getTotal());
23    }
24
25    @Test
26    void shouldApplyQuantityDiscount_whenOver10Items()
27    {
28        cart.addItem(testProduct, 11);
29        Money expected = Money.of(10 * 11 *
30            0.9);
31        assertEquals(expected, cart.getTotal());
32    }
33
34    @Test
35    void shouldThrowException_whenNegativeQuantity()
36    {
37        assertThrows(IllegalArgumentException.class,
38            () -> cart.addItem(testProduct,
39                -1));
40    }
41 }
```

## 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:

```
1 // Original Code
2 public class DataManager {
3     private static DataManager instance;
4     private Connection conn;
5
6     private DataManager() {
7         try {
8             conn =
9                 DriverManager.getConnection("db_url");
10        } catch (Exception e) {
11            e.printStackTrace();
12        }
13    }
14
15    public static DataManager getInstance() {
16        if (instance == null) {
17            instance = new DataManager();
18        }
19        return instance;
20    }
21
22    public void saveData(String data) {
23        try {
24            Statement stmt =
25                conn.createStatement();
26            stmt.execute("INSERT INTO data
27                VALUES('" + data + "')");
28        } catch (Exception e) {
29            System.out.println("Error: " +
30                e.getMessage());
31        }
32    }
33 }
```

### 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:

```
1 @Component
2 public class DataManager implements
   AutoCloseable {
3     private final DataSource dataSource;
4     private final Logger logger =
       LoggerFactory.getLogger(DataManager.class);
5
6     public DataManager(DataSource dataSource) {
7         this.dataSource = dataSource;
8     }
9
10    public void saveData(String data) {
11        String sql = "INSERT INTO data
12                      VALUES(?)";
13        try (Connection conn =
14              dataSource.getConnection();
15              PreparedStatement stmt =
16                  conn.prepareStatement(sql)) {
17
18            stmt.setString(1, data);
19            stmt.execute();
20
21        } catch (SQLException e) {
22            logger.error("Failed to save data",
23                        e);
24            throw new
25                DataAccessException("Could not
26                                    save data", e);
27        }
28    }
29
30    @Override
31    public void close() {
32        // Cleanup code
33    }
34 }
```

## 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:**

```

1 // Message Broker Interface
2 public interface MessageBroker {
3     void publish(String topic, Message message);
4     void subscribe(String topic, MessageHandler
5         handler);
6 }
7 // Chat Service
8 @Service
9 public class ChatService {
10     private final MessageBroker broker;
11     private final MessageRepository repository;
12
13     public void sendMessage(ChatMessage
14         message) {
15         // Persist message
16         repository.save(message);
17
18         // Publish to online users
19         broker.publish(
20             "chat." + message.getRoomId(),
21             message
22         );
23
24         // Handle offline users
25         message.getRecipients().stream()
26             .filter(user -> !userIsOnline(user))
27             .forEach(user ->
28                 queueOfflineMessage(user,
29                     message));
30     }
31
32     public void joinRoom(String userId, String
33         roomId) {
34         broker.subscribe("chat." + roomId,
35             message -> deliverToUser(userId,
36                 message));
37     }
38
39     private void queueOfflineMessage(
40         String userId, ChatMessage message)
41     {
42         {
43             broker.publish(
44                 "offline." + userId,
45                 message
46             );
47         }
48     }
49 }
50
51 // Load Balancer Configuration
52 @Configuration
53 public class LoadBalancerConfig {

```

## Message Oriented Middleware

**Aufgabe:** Implementieren Sie ein Nachrichtensystem mit JMS

```
1 // Message Producer
2 @Component
3 public class OrderProducer {
4     @Autowired
5     private JmsTemplate jmsTemplate;
6
7     public void sendOrder(Order order) {
8         try {
9             jmsTemplate.convertAndSend("orders",
10                                     order, message -> {
11                                         message.setStringProperty(
12                                             "orderType",
13                                             order.getType().toString()
14                                         );
15                                         return message;
16                                     });
17         } catch (JmsException e) {
18             handleMessageError(order, e);
19         }
20 }
21
22 // Message Consumer
23 @Component
24 public class OrderConsumer {
25     @JmsListener(
26         destination = "orders",
27         selector = "orderType = 'PREMIUM'"
28     )
29     public void processPremiumOrder(Order
30                                     order) {
31         try {
32             // Process order with high priority
33             processOrderWithPriority(order);
34         } catch (Exception e) {
35             // Dead Letter Queue
36             handleFailedOrder(order, e);
37         }
38     }
39
40     @JmsListener(
41         destination = "orders",
42         selector = "orderType = 'STANDARD'"
43     )
44     public void processStandardOrder(Order
45                                     order) {
46         // Process normal order
47     }
48 }
49
50 // Error Handling
51 @Component
52 public class DeadLetterQueueHandler {
53     @JmsListener(destination = "DLQ")
54     public void handleFailedMessages(Message
55                                     failedMessage) {
56         // Analyze failure
57         // Retry with backoff
58     }
59 }
```

## 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

```
1 public class CircuitBreaker {
2     private final long timeout;
3     private final int failureThreshold;
4     private final long resetTimeout;
5
6     private AtomicInteger failures = new
7         AtomicInteger();
8     private AtomicReference<State> state =
9         new AtomicReference<>(State.CLOSED);
10    private AtomicLong lastFailureTime = new
11        AtomicLong();
12
13    public enum State {
14        CLOSED, OPEN, HALF_OPEN
15    }
16
17    public <T> T execute(Supplier<T> action)
18        throws Exception {
19        if (shouldExecute()) {
20            try {
21                T result = action.get();
22                reset();
23                return result;
24            } catch (Exception e) {
25                handleFailure();
26                throw e;
27            }
28        }
29        throw new
30            CircuitBreakerException("Circuit
31                                    open");
32    }
33
34    private boolean shouldExecute() {
35        State currentState = state.get();
36        if (currentState == State.CLOSED) {
37            return true;
38        }
39        if (currentState == State.OPEN) {
40            if (hasResetTimeoutExpired()) {
41                state.compareAndSet(State.OPEN,
42                                    State.HALF_OPEN);
43                return true;
44            }
45            return false;
46        }
47        return true; // HALF_OPEN
48    }
49
50    private void handleFailure() {
51        lastFailureTime.set(System.currentTimeMillis);
52        if (failures.incrementAndGet() >=
53            failureThreshold) {
54            state.set(State.OPEN);
55        }
56    }
57 }
```

## 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

```
1 // Eureka Server
2 @SpringBootApplication
3 @EnableEurekaServer
4 public class ServiceRegistryApplication {
5     public static void main(String[] args) {
6         SpringApplication.run(
7             ServiceRegistryApplication.class,
8             args
9         );
10    }
11 }
12
13 // Service Registration
14 @SpringBootApplication
15 @EnableDiscoveryClient
16 public class UserServiceApplication {
17     public static void main(String[] args) {
18         SpringApplication.run(
19             UserServiceApplication.class,
20             args
21         );
22    }
23 }
24
25 // Service Discovery
26 @Service
27 public class UserServiceClient {
28     @Autowired
29     private DiscoveryClient discoveryClient;
30
31     @Autowired
32     private RestTemplate restTemplate;
33
34     public UserDTO getUser(String id) {
35         // Get service instance
36         List<ServiceInstance> instances =
37             discoveryClient.getInstances("user-service");
38
39         if (instances.isEmpty()) {
40             throw new ServiceNotFoundException(
41                 "user-service not available");
42         }
43
44         // Load balance
45         ServiceInstance instance =
46             loadBalance(instances);
47
48         // Make request
49         return restTemplate.getForObject(
50             instance.getUri() + "/api/users/" +
51                 id,
52             UserDTO.class
53         );
54
55     private ServiceInstance loadBalance(
56         List<ServiceInstance> instances) {
57         // Simple round-robin
```