Software Entwicklung 1

Jil Zerndt, Lucien Perret
January 2025

Einführung und Überblick

Software Engineering

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

Modellierung in der Softwareentwicklung

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

Modellierungsumfang bestimmen Folgende Fragen zur Bestimmung des notwendigen Modellierungsumfangs:

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

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

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

[Previous content continues...]

Charakteristiken iterativ-inkrementeller Prozesse

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

[Rest of previous content...]

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

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

Musterlösung:

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

[Previous formulas and diagrams remain as they were...]

Anforderungsanalyse

[Previous content remains unchanged until User-Centered Design]

User Research durchführen Systematisches Vorgehen für User und Domain Research:

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

[Previous content about UCD remains]

Persona erstellen **Aufgabe:** Erstellen Sie eine Persona für ein Online-Banking-System.
Lösung: Sarah Schmidt, 34, Projektmanagerin

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

[Previous content about Requirements Engineering]

Anforderungsarten **Funktionale Anforderungen:**

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

Nicht-funktionale Anforderungen:

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

Randbedingungen:

- Technische Einschränkungen
- Gesetzliche Vorgaben
- Budget und Zeitrahmen

[Previous content about Use Cases]

Use Case Verfeinerung Schritte zur Verfeinerung eines Use Cases:

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

Typische Prüfungsaufgabe: Use Case Analyse **Aufgabe:** Analysieren Sie den folgenden Use Case und identifizieren Sie mögliche Probleme:
Use Case: "Der Benutzer loggt sich ein und das System zeigt die Startseite. Er klickt auf den Button und die Daten werden in der Datenbank gespeichert."
Probleme:

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

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

[Previous content about SSD remains]

SSD Übungsaufgabe **Aufgabe:** Erstellen Sie ein Systemsequenzdiagramm für den Use Case "Geld abhebenän einem Bankautomaten.
Wichtige Aspekte:

- Kartenvalidierung
- PIN-Eingabe
- Betragseingabe
- Kontostandsprüfung
- Geldausgabe
- Belegdruck

Essentielle Systemoperationen:

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

Domänenmodellierung

[Previous content about Domänenmodell concept remains unchanged]

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

[Previous KR about Domänenmodell creation remains]

Prüfungsaufgabe: Konzeptidentifikation **Aufgabentext:** Ein Bibliothekssystem verwaltet Bücher, die von Mitgliedern ausgeliehen werden können. Jedes Buch hat eine ISBN und mehrere Exemplare. Mitglieder können maximal 5 Bücher gleichzeitig für 4 Wochen ausleihen. Bei Überschreitung wird eine Mahngebühr fällig."
Identifizierte Konzepte:
- Buch (Beschreibungsklasse)
- Exemplar (Physisches Objekt)
- Mitglied (Rolle)
- Ausleihe (Transaktion)
- Mahnung (Artefakt)

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

Analysemuster Anwendung Systematisches Vorgehen bei der Anwendung von Analysemustern:
1. **Muster identifizieren**
   - Beschreibungsklassen bei gleichartigen Objekten
   - Generalisierung bei ïst-einBeziehungen
   - Komposition bei existenzabhängigen Teilen
   - Zustände bei Objektlebenszyklen
   - Rollen bei verschiedenen Funktionen
   - Assoziationsklassen bei Beziehungsattributen
   - Wertobjekte bei komplexen Werten
2. **Muster anwenden**
   - Struktur des Musters übernehmen
   - An Kontext anpassen
   - Konsistenz prüfen
3. **Muster kombinieren**
   - Überschneidungen identifizieren
   - Konflikte auflösen
   - Gesamtmodell harmonisieren

Komplexes Domänenmodell: Reisebuchungssystem **Anforderung:** Modellieren Sie ein System für Pauschalreisen mit Flügen, Hotels und Aktivitäten.
Verwendete Analysemuster:
- **Beschreibungsklassen:**
  - Flugverbindung vs. konkreter Flug
  - Hotelkategorie vs. konkretes Zimmer
  - Aktivitätstyp vs. konkrete Durchführung
- **Zustände:**
  - Buchungszustände: angefragt, bestätigt, storniert
  - Zahlungszustände: offen, teilbezahlt, vollständig
- **Rollen:**
  - Person als: Kunde, Reiseleiter, Kontaktperson
- **Wertobjekte:**
  - Geldbetrag mit Währung
  - Zeitraum für Reisedauer

[Hier könnte ein Beispiel-Diagramm folgen]

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

[Previous content about common modeling errors remains]

Typische Prüfungsaufgabe: Modell verbessern **Fehlerhaftes Modell:**
- Klasse ÜserManager"mit CRUD-Operationen
- Attribute "customerIDünd örderIDßtatt Assoziationen
- Operation "calculateTotal()ïn Bestellung
- Technische Klasse "DatabaseConnection"

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

[Previous content remains unchanged until after the initial concepts]

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

Typische Prüfungsaufgabe: Architekturanalyse **Aufgabenstellung:** Analysieren Sie folgende Anforderungen und leiten Sie architektonische Konsequenzen ab:
- System muss 24/7 verfügbar sein
- 10.000 gleichzeitige Benutzer
- Reaktionszeit unter 1 Sekunde
- Jährliche Wartungsfenster maximal 4 Stunden

Lösung:
- **Architekturentscheidungen:**
  - Verteilte Architektur für Hochverfügbarkeit
  - Load Balancing für gleichzeitige Benutzer
  - Caching-Strategien für Performanz
  - Blue-Green Deployment für Wartung
- **Begründungen:**
  - Verteilung minimiert Single Points of Failure
  - Load Balancer verteilt Last gleichmäßig
  - Caching reduziert Datenbankzugriffe
  - Blue-Green erlaubt Updates ohne Downtime

[Previous content about N+1 View Model remains]

UML Diagrammauswahl Entscheidungshilfe für die Wahl des UML-Diagrammtyps:
1. **Strukturbeschreibung benötigt:**
   - Klassendiagramm für Typen und Beziehungen
   - Paketdiagramm für Modularisierung
   - Komponentendiagramm für Bausteinsicht
   - Verteilungsdiagramm für Deployment
2. **Verhaltensbeschreibung benötigt:**
   - Sequenzdiagramm für Interaktionsabläufe
   - Aktivitätsdiagramm für Workflows
   - Zustandsdiagramm für Objektlebenszyklen
   - Kommunikationsdiagramm für Objektkollaborationen
3. **Abstraktionsebene wählen:**
   - Analyse: Konzeptuelle Diagramme
   - Design: Detaillierte Spezifikation
   - Implementation: Codenahes Design

GRASP Anwendung **Szenario:** Online-Shop Warenkorb-Funktionalität
GRASP-Prinzipien angewandt:
- **Information Expert:**
  - Warenkorb kennt seine Positionen
  - Berechnet selbst Gesamtsumme
- **Creator:**
  - Warenkorb erstellt Warenkorbpositionen
  - Bestellung erstellt aus Warenkorb
- **Controller:**
  - ShoppingController koordiniert UI und Domain
  - Keine Geschäftslogik im Controller
- **Low Coupling:**
  - UI kennt nur Controller
  - Domain unabhängig von UI

Architektur-Review durchführen **Vorgehen:**

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

Prüfkriterien:

- Anforderungserfüllung
- Technische Machbarkeit
- Zukunftssicherheit
- Best Practices

[Previous examples and definitions remain]

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

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

Bewertungskriterien:

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

Use Case Realisation

[Previous concept about Use Case Realization remains]

Use Case Realization Ziele

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

[Previous definition about UML remains]

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

Systemoperationen identifizieren:

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

Operation Contract für withdrawMoney():

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

[Previous KR about procedure remains]

Design Class Diagram (DCD) erstellen **1. Klassen identifizieren**

- Aus Domänenmodell übernehmen
- Technische Klassen ergänzen
- Controller bestimmen

2. Attribute definieren

- Datentypen festlegen
- Sichtbarkeiten bestimmen
- Validierungen vorsehen

3. Methoden hinzufügen

- Systemoperationen verteilen
- GRASP-Prinzipien anwenden
- Signaturen definieren

4. Beziehungen modellieren

- Assoziationen aus Domänenmodell
- Navigierbarkeit festlegen
- Abhängigkeiten minimieren

Vollständige Use Case Realization **Use Case:** Bestellung aufgeben

1. Systemoperationen:

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

2. Design-Entscheidungen:

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

3. GRASP-Anwendung:

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

4. Implementierung:

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

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

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

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

Implementierung prüfen **1. Funktionale Prüfung**

- Use Case Szenarien durchspielen
- Randfälle testen
- Fehlersituationen prüfen

2. Strukturelle Prüfung

- Architekturkonformität
- GRASP-Prinzipien
- Clean Code Regeln

3. Qualitätsprüfung

- Testabdeckung
- Wartbarkeit
- Performance

Typische Prüfungsaufgabe **Aufgabe:** Gegeben ist folgender Use Case: "Kunde meldet sich an". Erstellen Sie:

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

Bewertungskriterien:

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

[Previous KR about implementation errors remains] [Previous content remains unchanged until after initial pattern definitions]

Pattern-Analyse für Prüfung **Systematisches Vorgehen:**

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

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

Aufgabe:

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

Musterlösung:

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

Pattern-Vergleich: Adapter vs. Facade **Gegeben sind zwei Patterns. Vergleichen Sie diese:**
Adapter:
- **Zweck:** Inkompatible Schnittstellen vereinen
- **Struktur:** Wrapper um einzelne Klasse
- **Anwendung:** Bei existierenden, inkompatiblen Klassen

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

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

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

Strategy Pattern Implementation

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

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

public class Context {
    private SortStrategy strategy;

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

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

[Continue with previous pattern definitions...] [Previous content until Implementation Strategies remains unchanged]

Prüfungsaufgabe: Entwicklungsansätze vergleichen **Szenario:** Ein Team soll eine neue Webanwendung entwickeln. Diskutieren Sie die Vor- und Nachteile von TDD gegenüber CDD für dieses Projekt.
Musterlösung:
- **TDD Vorteile:**
  - Testbare Architektur von Anfang an
  - Frühe Fehlererkennung
  - Dokumentation durch Tests
  - Sicherheit bei Refactoring
- **TDD Nachteile:**
  - Initial höherer Zeitaufwand
  - Lernkurve für das Team
  - Schwierig bei unklaren Anforderungen
- **Empfehlung:**
  - TDD für kritische Kernkomponenten
  - CDD für Prototypen und UI
  - Hybridansatz je nach Modulkritikalität

Code Review Durchführung **1. Vorbereitung**
- Code-Guidelines bereitstellen
- Checkliste erstellen
- Scope definieren

2. Review durchführen
- Lesbarkeit prüfen
- Naming Conventions
- Architekturkonformität
- Testabdeckung

3. Feedback geben
- Konstruktiv formulieren
- Priorisieren
- Lösungen vorschlagen

Typische Prüfungsaufgabe: Code Smells **Analysieren Sie folgenden Code auf Code Smells:**
Problematischer Code:
- Klasse ÜserManager"mit 1000 Zeilen
- Methode "processData"mit 200 Zeilen
- Variable "data"wird in 15 Methoden verwendet
- Duplizierte Validierungslogik in mehreren Klassen

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

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

[Previous content about Testing basics remains]

Prüfungsaufgabe: Teststrategie **Szenario:** Ein Onlineshop-System soll getestet werden. Entwickeln Sie eine Teststrategie.
Lösung:
- **Unit Tests:**
  - Warenkorb-Berechnungen
  - Preis-Kalkulationen
  - Validierungsfunktionen
- **Integrationstests:**
  - Bestellprozess
  - Zahlungsabwicklung
  - Lagerverwaltung
- **System Tests:**
  - Performance unter Last
  - Sicherheitsaspekte
  - Datenbankinteraktionen
- **Akzeptanztests:**
  - Benutzerszenarien
  - Geschäftsprozesse
  - Reporting

Testabdeckung optimieren **1. Analyse der Testabdeckung**
- Code Coverage messen
- Kritische Pfade identifizieren
- Lücken dokumentieren

2. Priorisierung
- Geschäftskritische Funktionen
- Fehleranfällige Bereiche
- Komplexe Algorithmen

3. Ergänzung der Tests
- Randfall-Tests
- Negativtests
- Performance-Tests

4. Wartung
- Regelmäßige Überprüfung
- Anpassung an Änderungen
- Entfernung veralteter Tests

Prüfungsaufgabe: Testfälle entwerfen **Aufgabe:** Entwickeln Sie Testfälle für eine Methode zur Validierung einer Email-Adresse.
Testfälle:
- **Positive Tests:**
  - Standard Email (user@domain.com)
  - Subdomain (user@sub.domain.com)
  - Mit Punkten (first.last@domain.com)
- **Negative Tests:**
  - Fehlende @ (userdomain.com)
  - Mehrere @ (user@@domain.com)
  - Ungültige Zeichen (user#@domain.com)
- **Randfälle:**
  - Leerer String
  - Nur Whitespace
  - Sehr lange Adressen

[Previous content until initial definitions remains unchanged]

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

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

Analysieren Sie die Architekturstile:

1. Client-Server

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

2. Peer-to-Peer

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

Empfehlung: Peer-to-Peer mit hybriden Elementen

Verteilungsprobleme analysieren **1. Probleme identifizieren**

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

2. Lösungsstrategien entwickeln

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

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

CAP-Theorem Komponenten:

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

Analyse der Trade-offs:

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

Verteilte System-Design **1. Anforderungsanalyse**

- **Funktional:**
  - Kernfunktionalitäten
  - Datenmodell
  - Schnittstellen
- **Nicht-funktional:**
  - Skalierbarkeit
  - Verfügbarkeit
  - Latenz

2. Architekturentscheidungen

- **Kommunikation:**
  - Synchron vs. Asynchron
  - Push vs. Pull
  - Protokollwahl
- **Datenmanagement:**
  - Sharding
  - Replikation
  - Caching

3. Implementierungsaspekte

- **Fehlerbehandlung:**
  - Retry-Strategien
  - Fallbacks
  - Monitoring
- **Sicherheit:**
  - Authentifizierung
  - Verschlüsselung
  - Autorisierung

[Previous content about middleware and error sources remains] [Previous content until initial concepts remains unchanged]

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

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

Lösung:

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

Persistenzstrategie wählen **1. Anforderungen analysieren**

- **Funktional:**
  - Datenmodell-Komplexität
  - Abfrageanforderungen
  - Transaktionsverhalten
- **Nicht-funktional:**
  - Performance
  - Skalierbarkeit
  - Wartbarkeit

2. Technologien evaluieren

- **JDBC:**
  - Direkte Kontrolle
  - Hohe Performance
  - Hoher Implementierungsaufwand
- **JPA:**
  - Standardisiert
  - Produktiv
  - Lernkurve
- **NoSQL:**
  - Flexibles Schema
  - Hohe Skalierbarkeit
  - Spezielle Anwendungsfälle

Prüfungsaufgabe: Design Pattern Vergleich **Aufgabe:** Vergleichen Sie Active Record, DAO und Repository Pattern.
Analysematrix:
- **Active Record:**
  - **Vorteile:**
    - Einfache Implementierung
    - Schnell zu entwickeln
  - **Nachteile:**
    - Keine Trennung der Belange
    - Schlechte Testbarkeit
    - Vermischung von Fachlogik und Persistenz
- **DAO:**
  - **Vorteile:**
    - Klare Trennung der Belange
    - Gute Testbarkeit
    - Austauschbare Implementierung
  - **Nachteile:**
    - Mehr Initialaufwand
    - Zusätzliche Abstraktionsebene
- **Repository:**
  - **Vorteile:**
    - Domänenorientierte Schnittstelle
    - Zentrale Abfragelogik
    - DDD-konform
  - **Nachteile:**
    - Komplexere Implementierung
    - Höhere Lernkurve

JPA Entity Design **1. Grundstruktur**
- **Basisanforderungen:**
  - Default Constructor
  - Serializable (optional)
  - Getter/Setter
- **Identifikation:**
  - Primary Key Strategie
  - Natural vs. Surrogate Key

2. Beziehungen
- **Kardinalität:**
  - OneToOne
  - OneToMany/ManyToOne
  - ManyToMany
- **Richtung:**
  - Unidirektional
  - Bidirektional
- **Lifecycle:**
  - Cascade-Operationen
  - Orphan Removal

3. Optimierungen
- **Lazy Loading:**
  - Fetch-Strategien
  - Join Fetching
- **Caching:**
  - First-Level Cache
  - Second-Level Cache

[Previous content about Repository Pattern remains] [Previous content about Framework basics remains]

Prüfungsaufgabe: Framework-Analyse **Szenario:** Ein Framework für die Verarbeitung verschiedener Dokumentformate (PDF, DOC, TXT) soll entwickelt werden.
Aufgabe: Analysieren Sie die Design-Entscheidungen.
Lösung:
- **Erweiterungspunkte:**
  - Dokumenttyp-Erkennung
  - Parser für Formate
  - Konvertierungslogik
- **Design Patterns:**
  - Factory für Parser-Erzeugung
  - Strategy für Verarbeitungsalgorithmen
  - Template Method für Konvertierung
- **Schnittstellen:**
  - DocumentParser Interface
  - ConversionStrategy Interface
  - DocumentMetadata Klasse

Framework Design Principles **1. Abstraktionsebenen definieren**
- **Core API:**
  - Zentrale Interfaces
  - Hauptfunktionalität
  - Erweiterungspunkte
- **Extensions:**
  - Plugin-Mechanismen
  - Callback-Interfaces
  - Event-Systeme
- **Implementierung:**
  - Standard-Implementierungen
  - Utility-Klassen
  - Helper-Funktionen

2. Erweiterungsmechanismen
- **Interface-basiert:**
  - Klare Verträge
  - Lose Kopplung
  - Einfache Erweiterung
- **Annotations:**
  - Deklarative Konfiguration
  - Metadaten-getrieben
  - Runtime-Processing
- **Composition:**
  - Plugin-System
  - Service-Loader
  - Dependency Injection

Framework Design Pattern Anwendung **Aufgabe:** Implementieren Sie ein Plugin-System mit verschiedenen Design Patterns.
Analyse der Pattern-Kombination:
- **Abstract Factory:**
  - Plugin-Familie erzeugen
  - Zusammengehörige Komponenten
  - Austauschbare Implementierungen
- **Template Method:**
  - Plugin-Lifecycle definieren
  - Standardablauf vorgeben
  - Erweiterungspunkte bieten
- **Command:**
  - Plugin-Aktionen kapseln
  - Asynchrone Ausführung
  - Undo-Funktionalität

Framework Evaluation **1. Qualitätskriterien**
- **Usability:**
  - Intuitive API
  - Gute Dokumentation
  - Beispiele/Templates
- **Flexibilität:**
  - Erweiterbarkeit
  - Konfigurierbarkeit
  - Modularität
- **Wartbarkeit:**
  - Klare Struktur
  - Testbarkeit
  - Versionierung

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

Typische Prüfungsaufgabe: Framework Migration **Szenario:** Ein bestehendes System soll von einem proprietären Framework auf ein Standard-Framework migriert werden.
Aufgabenstellung:
- Analysieren Sie die Herausforderungen
- Entwickeln Sie eine Migrationsstrategie
- Bewerten Sie Risiken

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

[Previous content about modern framework patterns remains]