

Probeklausur

Softwaretechnik und Programmierparadigmen

Universität: Technische Universität Berlin
Kurs/Modul: Softwaretechnik und Programmierparadigmen
Bearbeitungszeit: 180 Minuten
Erstellungsdatum: September 19, 2025



Zielorientierte Lerninhalte, kostenlos!
Entdecke zugeschnittene Materialien für deine Kurse:

<https://study.AllWeCanLearn.com>

Softwaretechnik und Programmierparadigmen

Aufgabe 1.

- (a) Beschreiben Sie die wesentlichen Unterschiede zwischen Wasserfall- und iterativen Vorgehensmodellen. Nennen Sie zwei Vorteile sowie zwei Nachteile beider Ansätze.
- (b) Welche Rolle spielt die Anforderungsanalyse in der frühen Phase eines Softwareprojekts? Nennen Sie zwei Techniken zur Anforderungserhebung und erläutern Sie deren Einsatzkontext.
- (c) Nennen Sie zwei Techniken der Projektorganisation und diskutieren Sie deren Auswirkungen auf Teamkommunikation und Transparenz. Geben Sie jeweils ein kurzes Beispiel.
- (d) Skizzieren Sie den Zusammenhang zwischen Anforderungsanalyse, Entwurf, Implementierung und Qualitätssicherung. Nennen Sie zwei typische Qualitätsmetriken und erläutern Sie deren Zweck.

Aufgabe 2.

- (a) Beschreiben Sie die Grundzüge der Programmierparadigmen Objektorientierte Programmierung, Funktionale Programmierung, Logische Programmierung und Modellgetriebene Programmierung. Nennen Sie jeweils zentrale Konzepte.
- (b) Geben Sie ein kurzes Pseudocode-Beispiel für eine Funktion, die eine Liste von Ganzzahlen quadriert. Erklären Sie, wie dieses Beispiel in funktionaler versus objektorientierter Perspektive umgesetzt werden könnte.
- (c) Erklären Sie, was Kapselung bedeutet, und wie sie in der objektorientierten Programmierung umgesetzt wird. Geben Sie ein kurzes Beispiel.
- (d) Welche Rolle spielt die Modellierung in der modellgetriebenen Programmierung? Nennen Sie zwei Vorteile.

Aufgabe 3.

(a) Definieren Sie eine einfache Anforderungsspezifikation für ein Bibliothekssystem. Formulieren Sie mindestens drei funktionale Anforderungen und zwei nicht-funktionale Anforderungen.

(b) Erstellen Sie eine kleine Nachverfolgbarkeits-Matrix, die drei Anforderungen A1, A2, A3 drei Modulen M1, M2 zuordnet. Verwenden Sie eine klare Tabellenstruktur.

(c) Geben Sie ein kurzes Hoare-Triple-Beispiel an, das eine einfache Zuweisungsoperation verifiziert:

$$\{x \geq 0\} \ x := x + 1 \ \{x \geq 1\}$$

(d) Nennen Sie drei Metriken zur Softwarequalität und erläutern Sie knapp deren Messansätze.

Aufgabe 4.

- (a) Vergleichen Sie Schichtenarchitektur und Mikroservice-Architektur hinsichtlich Skalierbarkeit, Wartbarkeit und Deployment-Strategien.
- (b) Geben Sie eine grobe Zerlegung eines Systems zur Verwaltung von Mediendaten in Module mit jeweiligen Verantwortlichkeiten an.
- (c) Diskutieren Sie Kopplung und Kohäsion innerhalb eines Beispielsmoduls und ordnen Sie das Modul einem Paradigma zu.
- (d) Beschreiben Sie eine einfache Abhängigkeitsbeziehung zwischen Modulen in Textform und erläutern Sie, wie sie durch klare Schnittstellen minimiert werden kann.

Aufgabe 5.

- (a) Welche Vor- und Nachteile ergeben sich aus der Auswahl bestimmter Programmierparadigmen für ein gegebenes Domänenproblem?
- (b) Entwerfen Sie eine klare Schnittstelle (Interface) für eine Komponente „Sortieren“ in einem hypothetischen System. Definieren Sie mindestens drei Methoden und deren Zweck.
- (c) Erläutern Sie kurz, wie modellgetriebene Programmierung in der Praxis angewandt werden könnte. Geben Sie zwei konkrete Anwendungsfelder an.
- (d) Welche Paradigmen eignen sich besonders gut für domänenspezifische Modellierung? Geben Sie jeweils ein Beispiel.

Lösungen

Aufgabe 1.

(a) Unterschiede zwischen Wasserfall- und iterativen Vorgehensmodellen sowie zwei Vor- bzw. Nachteilen beider Ansätze:

- Wasserfall-Modell
 - Unterschiede: lineare, sequentielle Phasen (Anforderungen, Design, Implementierung, Verifikation, Wartung) mit festen Abnahmekriterien. Änderungen sind aufwändig; Dokumentation und Planbarkeit stehen im Vordergrund.
 - Vorteile: (i) klare Planbarkeit und Budgetierung, (ii) einfache Nachverfolgung der Anforderungen und Abnahmekriterien, (iii) gut geeignet für regulierte Umgebungen mit stabilen Anforderungen.
 - Nachteile: (i) geringe Flexibilität bei sich ändernden Anforderungen, (ii) späte Fehlererkennung, (iii) lange Lieferzyklen.
- Iteratives/Vorgehensmodell (z.B. Evolution, Agile, Inkrementalisierung)
 - Unterschiede: schrittweise, inkrementelle Entwicklung mit frühen Prototypen und regelmäßigem Feedback von Stakeholdern; Anpassung der Anforderungen in kurzen Zyklen.
 - Vorteile: (i) frühzeitige Lieferung von Teilprodukten, (ii) Risikoreduzierung durch ständiges Feedback, (iii) erhöhte Flexibilität bei Änderungen.
 - Nachteile: (i) erfordert konstante Stakeholder-Beteiligung, (ii) potenziell unklare langfristige Planung, (iii) anfängliche Architektur muss robust genug sein, um spätere Änderungen zu tragen.

(b) Rolle der Anforderungsanalyse in der frühen Phase und zwei Techniken zur Anforderungserhebung:

- Rolle: Festlegung des Umfangs, Identifikation relevanter Stakeholder:innen, Formulierung verständlicher Ziele, Minimierung von Änderungswellen in späteren Phasen; Ausgangspunkt für Designentscheidungen und Tests.
- Techniken:
 - Interviews/Workshops mit Stakeholdern (z. B. Fachexperten, Endnutzer); Einsatzkontext: frühzeitiges Erfassen von Zielen, Einschränkungen und kritischen Szenarien.
 - Use-Case-Modellierung bzw. User Stories (ggf. mit Akteur:innen); Einsatzkontext: systematische Strukturierung von Anforderungen nach Nutzerperspektive, erleichtert die Priorisierung und die Validierung durch Tests.

(c) Zwei Techniken der Projektorganisation und deren Auswirkungen auf Teamkommunikation/Transparenz, jeweils mit kurzem Beispiel:

- Scrum (agiles Rahmenwerk)

- Auswirkungen: regelmäßige Synchronisation (Daily Scrum), definierte Sprints, produktorientierte Teams erhöhen Transparenz der Fortschritte; klare Rollen (Product Owner, Scrum Master, Entwicklungsteam).
- Beispiel: Ein Team plant einen zwei-wöchigen Sprint, besitzt ein Sprint Backlog; am Sprint-Ende wird potenziell auslieferbare Software präsentiert.
- Kanban (flussorientiert, WIP-Limits)
 - Auswirkungen: fokussiert auf den Fluss der Arbeit, begrenzte Parallelität (WIP-Limits) erhöht Sichtbarkeit von Engpässen; weniger formale Rituale als Scrum.
 - Beispiel: Ein Board zeigt Aufgaben in den Spalten „Zu erledigen“ – „In Arbeit“ – „Fertig“, wobei nur eine feste Anzahl von Tasks gleichzeitig in „In Arbeit“ sein darf, um Engpässe sichtbar zu machen.

(d) Zusammenhang zwischen Anforderungsanalyse, Entwurf, Implementierung und Qualitätssicherung; zwei typische Qualitätsmetriken:

- Zusammenhang: Anforderungsanalyse bildet Input für Entwurf (Architektur und Details), Entwurf leitet Implementierung; Qualitätssicherung (Tests/Qualitätssicherung) bewertet Erfüllung der Anforderungen, Robustheit und Zuverlässigkeit des Systems.
- Metriken:
 - Defect Density (Anzahl Fehler pro KLOC oder Funktionpunkte) – Zweck: Kontextbezogene Fehlerhäufigkeit zur Beurteilung der Codequalität und der Effektivität von Tests.
 - Code Coverage bzw. Testabdeckung – Zweck: Anteil der Codepfade oder Anforderungen, die durch Tests abgedeckt sind; Indikator für Testumfang und Risikoabdeckung.

Aufgabe 2.

(a) Grundzüge der Programmierparadigmen mit zentralen Konzepten:

- Objektorientierte Programmierung (OOP): Zentrale Konzepte sind Klassen, Objekte, Kapselung, Vererbung, Polymorphie; Fokus auf Modellierung von Entitäten als Objekte mit Verhalten.
- Funktionale Programmierung: Funktionen als zentrale Bausteine, Abstraktion durch Funktionskomposition, Unveränderlichkeit (Immutability), First-class/High-order Functions, Nebenläufigkeit als Kernelement.
- Logische Programmierung: Wissen wird in Form von Fakten und Regeln ausgedrückt; Abfragen (Queries) lösen sich durch Inferenz-/Beweisprozesse; Beispielhafte Einführung durch Logik-Programmiersprachen (z. B. Prolog).
- Modellgetriebene Programmierung (MDD/MBSD): Modelle dienen als primärer Entwicklungsausgangspunkt; Automatisierte Generierung von Code/Artefakten aus Modellen; Schwerpunkt auf Abstraktion, Konsistenz und Wiederverwendung durch Model-Transformationen.

(b) Kurzes Pseudocode-Beispiel für eine Funktion, die eine Liste von Ganzzahlen quadriert; Umsetzung in funktionaler vs. objektorientierter Perspektive:

```
# Funktionale Perspektive
function squareList(lst: List[Int]) -> List[Int]:
    return map(lambda x: x * x, lst)

# Objektorientierte Perspektive
class ListProcessor:
    method squareAll(self, xs: List[Int]) -> List[Int]:
        result = []
        for x in xs:
            result.append(x * x)
        return result
```

(c) Kapselung: Bedeutung und Umsetzung in der OOP; kurzes Beispiel:

- Bedeutung: Verstecken von Implementierungsdetails (Daten) hinter einer gut definierten Schnittstelle; Schutz von Invarianten und Erhöhung Wartbarkeit.
- Umsetzung (Beispiel in Pseudocode):

```
class BankKonto:
    private double saldo

    public void einzahlen(double betrag):
        saldo = saldo + betrag

    public bool auszahlen(double betrag):
```

```
if betrag <= saldo:
    saldo = saldo - betrag
    return true
else:
    return false
```

(d) Rolle der Modellierung in der modellgetriebenen Programmierung; zwei Vorteile:

- Abstraktion und domänenbezogene Abbildung: Modelle ermöglichen eine klare, domänennahe Repräsentation unabhängig von einer konkreten Implementierung.
- Automatisierte Codegenerierung und Konsistenz: Modelle dienen als Quelle für automatisch generierten Code, Tests und Dokumentation, wodurch Inkonsistenzen reduziert werden.

Aufgabe 3.

(a) Eine einfache Anforderungsspezifikation für ein Bibliothekssystem (mindestens drei funktionale, zwei nicht-funktionale Anforderungen):

- Funktional:
 - F1: Ausleihe eines Mediums durch registrierten Benutzer; Verfügbarkeit prüfen und Leihfrist setzen.
 - F2: Rückgabe eines Mediums; Aktualisierung der Verfügbarkeit und ggf. Mahnprozesse.
 - F3: Suche nach Medien nach Titel, Autor oder Schlagwort; Rückgabe relevanter Treffer.
- Nicht-funktional:
 - N1: Reaktionszeit bei Suchanfragen unter 2 Sekunden bei typischer Last.
 - N2: Verfügbarkeit des Systems von mindestens 99,9

(b) Nachverfolgbarkeits-Matrix (Zuordnung von Anforderungen A1, A2, A3 zu Modulen M1, M2):

	M1	M2
A1	✓	
A2		✓
A3	✓	✓

(c) Kurzes Hoare-Triple-Beispiel (bereits Teil der Aufgabenstellung). Beurteilung der Korrektheit:

$$\{x \geq 0\} \ x := x + 1 \ \{x \geq 1\}$$

Begründung: Vorbedingung $x \geq 0$ garantiert, dass nach der Zuweisung $x = x + 1 - 1$ gilt; damit erfüllt sich die Nachbedingung. Das Triple ist gültig, da die Transformation $x := x + 1$ die Bedingung $x \geq 0$ in $x \geq 1$ überführt.

(d) Drei Metriken zur Softwarequalität und knappe Erläuterung ihrer Messansätze:

- Defect Density (Fehlerdichte): Anzahl gefundener Fehler relativ zur Codebasis (z. B. Defekte pro KLOC oder Funktion Points); gibt Hinweis auf die Qualität des Codes und der Tests.
- Code Coverage (Testabdeckung): Anteil des Codes bzw. der Pfade, die durch Tests abgedeckt sind; hoher Wert signalisiert umfangreicheren Testumfang.
- Zyklomatische Komplexität (McCabe): Misst die Kontrollflusskomplexität einer Funktion/Prozedur; dient als Indikator für Wartbarkeit und Fehleranfälligkeit.

Aufgabe 4.

(a) Vergleich Schichtenarchitektur vs. Mikroservice-Architektur bezüglich Skalierbarkeit, Wartbarkeit und Deployment-Strategien:

- Schichtenarchitektur:
 - Skalierbarkeit: Typischerweise vertikal skalierbar; einzelne Schicht skaliert schwerer, Risiken bei Engpässen zwischen Schichten.
 - Wartbarkeit: Gute Trennung in Presentation/Business/Data; zentrale Änderungen betreffen oft mehrere Schichten.
 - Deployment: Ein monolithisches Deployment; Änderungen erfordern erneutes Build/Deploy der gesamten Anwendung.
- Mikroservice-Architektur:
 - Skalierbarkeit: Horizontale Skalierung einzelner Dienste basierend auf Bedarf; unabhängig skalierbar.
 - Wartbarkeit: Kleinschrittige, unabhängige Services; Teams können eigenständig an Diensten arbeiten.
 - Deployment: Unabhängige Deployments pro Dienst; Infrastruktur (Containerisierung, Orchestrierung) unterstützt Continuous Deployment.

(b) Grobe Zerlegung eines Systems zur Verwaltung von Mediendaten in Module mit jeweiligen Verantwortlichkeiten:

- IngestionService: Aufnahme neuer Mediendateien, Formatkonvertierung, Metadatenextraktion.
- MetadataService: Verwaltung von Metadaten (Titel, Tags, Beschreibungen), Suche/Indexierung.
- MediaStore: Speichern von Mediendateien (Objekt-/Dateispeicher) und Versionierung.
- TranscodingService: Transkodierung in verschiedene Formate/Qualitäten.
- API-Gateway/REST-Schnittstelle: Schnittstelle nach außen, Authentifizierung, Ratenbegrenzung.
- UserInterface/Frontend: Präsentation der Medien, Suchfunktionen, Wiedergabe-UI.
- AdminTools/Monitoring: Verwaltung, Logs, Telemetrie, Fehlerbehandlung.

(c) Diskussion von Kopplung und Kohäsion innerhalb eines Beispielsmoduls und Zuordnung zu einem Paradigma:

- Beispielsmodul: MediaMetadata (Kohäsion hoch)
 - Verantwortlichkeiten: Lesen/Schreiben von Metadaten, Validierung, Indizierung, Exposition von Suchfeldern.

- Kopplung: Geringe Kopplung zu anderen Modulen über definierte Schnittstellen (z. B. Repository-API); interne Datenstrukturen verborgen.
- Paradigma: Objektorientierte Programmierung mit encapsulated state und klaren Schnittstellen; unterstützt durch Domänenmodelle (Entities wie Media, Metadata).

(d) Beschreibung einer einfachen Abhängigkeitsbeziehung zwischen Modulen in Textform und wie sie durch klare Schnittstellen minimiert werden kann:

- Beispiel: Modul A (Benutzeroberfläche) hängt von Modul B (Datenprovider) ab.
- Minimierung durch: (i) klare API/Schnittstellen zwischen A und B; (ii) Programmierer gegen Abstraktionen (Interfaces) statt gegen konkrete Implementierungen; (iii) Implementierung der Abhängigkeitsinversion (Dependency Inversion Principle) durch Injizierung von Abhängigkeiten (z. B. via Konstruktor/Injektions-Framework).

Aufgabe 5.

(a) Vor- und Nachteile der Auswahl bestimmter Programmierparadigmen für ein gegebenes Domänenproblem:

- Objektorientierte Programmierung (OOP):
 - Vorteile: Modeling von realweltlichen Entitäten, gute Strukturierung durch Klassen/Objekte, Unterstützung durch Vererbung/Polymorphie.
 - Nachteile: Komplexität bei großen Hierarchien, potenziell hohe Kopplung; Overhead durch Objektorientierung.
- Funktionale Programmierung (FP):
 - Vorteile: Reine Funktionen erleichtern Tests, Immutabilität reduziert Nebenwirkungen, gut geeignet für fehlertolerante, parallele Berechnungen.
 - Nachteile: Lernkurve bei State- und Side-Effect-Management, oft weniger intuitiv bei UI-Interaktionen.
- Logische Programmierung:
 - Vorteile: Natürliche Darstellung von Regelwissen, einfache Implementierung von Informations- und Inferenzprozessen.
 - Nachteile: Performance-Overhead, Debugging kann schwieriger sein, geeignet vor allem für problemräume mit expliziten Regeln.
- Modellgetriebene Programmierung (MDP/MBSD):
 - Vorteile: Abstraktion, Konsistenz durch Modelle, automatische Generierung von Artefakten; erleichtert Domain-Spezialisierung.
 - Nachteile: Lernaufwand, Abhängigkeit von Tooling und Transformationslogik, potenzielle Übermodellierung.

(b) Entwurf einer klaren Schnittstelle (Interface) für eine Komponente „Sortieren“ in einem hypothetischen System; Definieren Sie mindestens drei Methoden und deren Zweck:

```
interface Sorter<T> {  
    // sortiert eine Liste in der Standardreihenfolge  
    List<T> sort(List<T> items);  
  
    // sortiert eine Liste direkt in-place  
    void sortInPlace(List<T> items);  
  
    // sortiert nach einem gegebenen Key/Comparator  
    List<T> sortBy(List<T> items, Comparator<T> cmp);  
}
```

(c) Kurze Erläuterung, wie modellgetriebene Programmierung praxisnah angewandt werden könnte; zwei konkrete Anwendungsfelder:

- Automobilindustrie: Modellbasierte Entwicklung von Steuergeräten (ECUs) mit automatischer Code-Generierung aus Verhaltens- und Architektursmodellen; Simulation von Systemverhalten vor der Implementierung.
- Gesundheitswesen/Clinical-Workflows: Modelle von Behandlungsabläufen, Entscheidungen und Richtlinien, gefolgt von Code-Generierung oder Validierung gegen Richtlinien-Engines.

(d) Welche Paradigmen sich besonders gut für domänenspezifische Modellierung eignen; jeweils ein Beispiel:

- Logische Programmierung (Regelbasierte DSLs): z. B. Regelwerke in Compliance-/Diagnose-Systemen; Prolog-ähnliche DSLs.
- Modellgetriebene Ansätze (MD/MBSD): Domänengetriebene Sprachen (DSLs) generieren Code und Tests aus Modellen, z. B. für Automatisierungs- oder Embedded-Systeme.
- Funktionale Programmierung mit internen DSLs: mathematische oder data-centric Modelle, die Transformationen auf Datenbeständen ausdrücken.